EBook Gratis

APRENDIZAJE Java Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#java

Tabla de contenido

Acerca de 1

Capítulo 1: Empezando con el lenguaje Java 2

Observaciones 2

Ediciones y versiones de Java 2

Instalando Java 3

Compilación y ejecución de programas Java. 3

¿Que sigue? 3

Pruebas 3

Otro 3

Versiones 4

Examples 4

Creando tu primer programa Java 4

Una mirada más cercana al programa Hello World. 6

Capítulo 2: Acceso nativo de Java 11

Examples 11

Introducción a JNA 11

¿Qué es JNA? 11

¿Como puedo usar lo? 11

¿A dónde ir ahora? 12

Capítulo 3: Afirmando 13

Sintaxis 13

Parámetros 13

Observaciones 13

Examples 13

Comprobando aritmética con aseverar. 13

Capítulo 4: Agentes de Java 14

Examples 14

Modificando clases con agentes. 14

Agregar un agente en tiempo de ejecución 15

Configuración de un agente básico 15
Capítulo 5: Ajuste de rendimiento de Java 17
Examples 17
Enfoque general 17
Reduciendo cantidad de cuerdas 17
Un enfoque basado en la evidencia para el ajuste de rendimiento de Java 18
Capítulo 6: Análisis XML utilizando las API de JAXP 20
Observaciones 20
Principios de la interfaz DOM 20
Principios de la interfaz SAX 20
Principios de la interfaz StAX 21
Examples 21
Analizar y navegar un documento utilizando la API DOM 21
Analizar un documento utilizando la API StAX 22
Capítulo 7: Anotaciones 25
Introducción 25
Sintaxis 25
Observaciones 25
Tipos de parametros 25
Examples 25
Anotaciones incorporadas 25
Comprobaciones de anotación en tiempo de ejecución a través de la reflexión 29
Definición de tipos de anotación 29
Valores predeterminados 30
Meta-Anotaciones 30
@Objetivo 30
Valores disponibles 30
@Retencion 31
Valores disponibles 32
@Documentado 32
@Heredado 32

@Repeable 33
Obtención de valores de anotación en tiempo de ejecución 33
Repetir anotaciones 34
Anotaciones heredadas 35
Ejemplo 35
Procesamiento de tiempo de compilación utilizando procesador de anotación 36
La anotacion 36
El procesador de anotaciones. 36
embalaje 38
Ejemplo de clase anotada 38
Usando el procesador de anotaciones con javac 39
Integración IDE 39
Netbeans 39
Resultado 40
La idea detrás de las anotaciones. 40
Anotaciones para 'este' y parámetros del receptor. 40
Añadir múltiples valores de anotación 42
Capítulo 8: Apache Commons Lang 43
Examples 43
Implementar el método equals () 43
Implementar el método hashCode () 43
Implementar el método toString () 44
Capítulo 9: API de apilación 46
Introducción 46
Examples 46
Imprimir todos los marcos de pila de la secuencia actual 46
Imprimir la clase de llamada actual 47
Mostrando la reflexión y otros marcos ocultos. 47
Capítulo 10: API de reflexión 49
Introducción 49
Observaciones 49

Actuación 49
Examples 49
Introducción 49
Invocando un método 51
Obtención y configuración de campos 51
Constructor de llamadas 53
Obtención del objeto constructor 53
Nueva Instancia usando Objeto Constructor 53
Obteniendo las constantes de una enumeración 53
Obtener clase dado su nombre (totalmente calificado) 54
Llamar a los constructores sobrecargados utilizando la reflexión 55
Uso incorrecto de la API de Reflection para cambiar variables privadas y finales 56
Call constructor de clase anidada 57
Proxies dinámicos 57
Evil hacks de Java con Reflection 59
Capítulo 11: AppDynamics y TIBCO BusinessWorks Instrumentation para una fácil integración 61
Introducción 61
Examples 61
Ejemplo de instrumentación de todas las aplicaciones BW en un solo paso para Appdynamics 61
*** Variables comunes. Modifica estos solo. *** 61
Capítulo 12: Applets 63
Introducción 63
Observaciones 63
Examples 63
Applet mínimo 63
Creando una GUI 64
Abrir enlaces desde dentro del applet 65
Cargando imágenes, audio y otros recursos 65
Cargar y mostrar una imagen 66
Cargar y reproducir un archivo de audio 66
Cargar y mostrar un archivo de texto 66
Capítulo 13: Archivo I / O 68

Introducción ... 68
Examples ... 68
Leyendo todos los bytes a un byte [] ... 68
Leyendo una imagen de un archivo. ... 68
Escribir un byte [] a un archivo ... 68
Stream vs Writer / Reader API ... 69
Leyendo un archivo completo a la vez ... 70
Leyendo un archivo con un escáner ... 71
Iterando sobre un directorio y filtrando por extensión de archivo ... 71
Migración de java.io.File a Java 7 NIO (java.nio.file.Path) ... 72
Apuntar a un camino ... 72
Caminos relativos a otro camino. ... 72
Convertir archivo de / a ruta para usar con bibliotecas ... 72
Compruebe si el archivo existe y elimínelo si existe. ... 72
Escribir en un archivo a través de un OutputStream ... 73
Iterando en cada archivo dentro de una carpeta ... 73
Iteración de la carpeta recursiva ... 74
Lectura / escritura de archivos usando FileInputStream / FileOutputStream ... 75
Leyendo de un archivo binario ... 76
Cierre ... 76
Copiando un archivo usando InputStream y OutputStream ... 77
Leyendo un archivo usando Channel y Buffer ... 77
Copiando un archivo usando el canal ... 79
Leyendo un archivo usando BufferedInputStream ... 79
Escribiendo un archivo usando Channel y Buffer ... 80
Escribiendo un archivo usando PrintStream ... 80
Iterar sobre un directorio de subdirectorios de impresión en él. ... 81
Agregando Directorios ... 81
Bloqueo o redireccionamiento de salida / error estándar ... 82
Accediendo a los contenidos de un archivo ZIP. ... 83
Leyendo de un archivo existente ... 83

Creando un nuevo archivo ... 83
Capítulo 14: Archivos JAR Multi-Release ... 84
Introducción ... 84
Examples ... 84
Ejemplo de contenido de un archivo Jar multi-release ... 84
Creando un Jar multi-release usando la herramienta jar ... 84
URL de una clase cargada dentro de un Jar multi-release ... 86
Capítulo 15: Arrays ... 87
Introducción ... 87
Sintaxis ... 87
Parámetros ... 87
Examples ... 87
Creación e inicialización de matrices ... 87
Casos basicos ... 87
Arrays, colecciones y corrientes ... 88
Introducción ... 88
Creación e inicialización de matrices de tipos primitivos. ... 90
Creando e inicializando arrays multidimensionales. ... 91
Representación de matriz multidimensional en Java ... 92
Creación e inicialización de matrices de tipos de referencia ... 92
Creando e inicializando arrays genéricos ... 93
Rellenar una matriz después de la inicialización ... 94
Declaración separada e inicialización de matrices. ... 94
Es posible que las matrices no se puedan reinicializar con la sintaxis de acceso directo d ... 95
Creando una matriz de una colección ... 95
Arrays a una cadena ... 97
Creación de una lista a partir de una matriz ... 97
Notas importantes relacionadas con el uso del método Arrays.asList () ... 98
Matrices multidimensionales y dentadas ... 99
Cómo se representan los arreglos multidimensionales en Java ... 100

ArrayIndexOutOfBoundsException 101
Obtener la longitud de una matriz 102
Comparando matrices para la igualdad 103
Arreglos para transmitir 104
Iterando sobre matrices 104
Copiando matrices 107
en bucle 107
Object.clone () 107
Arrays.copyOf () 108
System.arraycopy () 108
Arrays.copyOfRange () 108
Matrices de fundición 108
Eliminar un elemento de una matriz 109
Usando ArrayList 109
Usando System.arraycopy 109
Usando Apache Commons Lang 110
Array Covariance 110
¿Cómo cambias el tamaño de una matriz? 111
Una mejor alternativa para cambiar el tamaño de matriz 111
Encontrar un elemento en una matriz 112
Usando Arrays.binarySearch (solo para arreglos ordenados) 112
Uso de Arrays.asList (solo para matrices no primitivas) 112
Usando una Stream 113
Búsqueda lineal utilizando un bucle. 113
Búsqueda lineal utilizando bibliotecas de terceros como org.apache.commons 113
Probando si una matriz contiene un elemento 113
Ordenando matrices 114
Conversión de matrices entre primitivas y tipos de caja. 115
Capítulo 16: Audio 117
Observaciones 117
Examples 117

Reproducir un archivo de audio en bucle 117
Reproducir un archivo MIDI 117
Sonido de metal desnudo 119
Salida de audio básica 120
Capítulo 17: Autoboxing 121
Introducción 121
Observaciones 121
Examples 121
Usando int y entero indistintamente 121
Usando Boolean en la sentencia if 122
Auto-unboxing puede llevar a NullPointerException 123
Memoria y sobrecarga computacional de Autoboxing 123
Casos diferentes cuando Integer e int pueden usarse indistintamente 124
Capítulo 18: Banderas JVM 126
Observaciones 126
Examples 126
-XXaggresivo 126
-XXallocClearChunks 126
-XXallocClearChunkSize 127
-XXcallProfiling 127
-XXdisableFatSpin 127
-XXdisponibilidad a las características de la visión 128
-XXdumpSize 128
-XXexitOnOutOfMemory 128
Capítulo 19: BigDecimal 130
Introducción 130
Examples 130
Los objetos BigDecimal son inmutables. 130
Comparando BigDecimals 130
Operaciones matemáticas con BigDecimal. 130
1.Adición 130
2.Subtraction 131

3.Multiplicacion 131
4.Division 131
5.El resto o módulo 132
6. Poder 132
7.Max 133
8.Min 133
9. Mover punto a la izquierda 133
10. Mover punto a la derecha 133
Usando BigDecimal en lugar de flotar 134
BigDecimal.valueOf () 135
Inicialización de BigDecimals con valor cero, uno o diez. 135
Capítulo 20: BigInteger 136
Introducción 136
Sintaxis 136
Observaciones 136
Examples 137
Inicialización 137
Comparando BigIntegers 138
Ejemplos de operaciones matemáticas de BigInteger 139
Operaciones de lógica binaria en BigInteger 141
Generando BigIntegers aleatorios 142
Capítulo 21: ByteBuffer 144
Introducción 144
Sintaxis 144
Examples 144
Uso básico - Creación de un ByteBuffer 144
Uso básico - Escribir datos en el búfer 145
Uso básico - Uso de DirectByteBuffer 145
Capítulo 22: Calendario y sus subclases 147
Observaciones 147
Examples 147

Creando objetos de calendario ... 147
Aumentar / disminuir los campos del calendario ... 147
Encontrando AM / PM ... 148
Restando calendarios ... 148
Capítulo 23: Características de Java SE 7 ... 149
Introducción ... 149
Observaciones ... 149
Examples ... 149
Nuevas características del lenguaje de programación Java SE 7 ... 149
Literales binarios ... 149
La declaración de prueba con recursos ... 150
Subrayados en literales numéricos ... 150
Inferencia de tipos para la creación de instancias genéricas ... 151
Cadenas en las instrucciones del interruptor ... 151
Capítulo 24: Características de Java SE 8 ... 153
Introducción ... 153
Observaciones ... 153
Examples ... 153
Nuevas características del lenguaje de programación Java SE 8 ... 153
Capítulo 25: Cargadores de clases ... 155
Observaciones ... 155
Examples ... 155
Instalar y usar un cargador de clases ... 155
Implementando un classLoader personalizado ... 155
Cargando un archivo .class externo ... 156
Capítulo 26: Cifrado RSA ... 158
Examples ... 158
Un ejemplo que utiliza un sistema criptográfico híbrido que consiste en OAEP y GCM ... 158
Capítulo 27: Clase - Reflexión de Java ... 163
Introducción ... 163
Examples ... 163
Método getClass () de la clase Object ... 163

Capítulo 28: Clase de fecha 164
Sintaxis 164
Parámetros 164
Observaciones 164
Examples 165
Creando objetos de fecha 165
Comparando objetos de fecha 166
Calendario, fecha y fecha local 166
Antes, después, compare y métodos iguales. 166
isBefore, isAfter, compareTo y es igual a métodos 167
Comparación de fechas antes de Java 8 168
Desde Java 8 168
Convertir la fecha a un determinado formato de cadena 169
Convertir cadena en fecha 169
Una fecha básica de salida. 170
Convertir la representación de cadena formateada de la fecha en el objeto Fecha 170
Creando una fecha específica 171
Objetos Java 8 LocalDate y LocalDateTime 172
Zonas horarias y java.util.Date 173
Convertir java.util.Date a java.sql.Date 173
Hora local 174
Capítulo 29: Clase de objetos Métodos y constructor 176
Introducción 176
Sintaxis 176
Examples 176
método toString () 176
método igual () 177
Comparación de clases 179
método hashCode () 180
Usando Arrays.hashCode () como un atajo 181
Caché interno de códigos hash 182

Métodos de esperar () y notificar () 183
método getClass () 184
método clone () 185
finalizar () método 186
Constructor de objetos 187
Capítulo 30: Clase de propiedades 190
Introducción 190
Sintaxis 190
Observaciones 190
Examples 191
Cargando propiedades 191
Los archivos de propiedades advierten: espacios en blanco al final 191
Guardar propiedades como XML 194
Capítulo 31: Clase EnumSet 196
Introducción 196
Examples 196
Ejemplo de conjunto de enumeración 196
Capítulo 32: Clase inmutable 197
Introducción 197
Observaciones 197
Examples 197
Reglas para definir clases inmutables. 197
Ejemplo sin referencias mutables. 197
Ejemplo con referencias mutables 198
¿Cuál es la ventaja de la inmutabilidad? 199
Capítulo 33: Clase interna local 200
Introducción 200
Examples 200
Clase interna local 200
Capítulo 34: Clases anidadas e internas 201
Introducción 201
Sintaxis 201

Observaciones 201
Terminologia y clasificacion 201
Diferencias semanticas 202
Examples 202
Una pila simple usando una clase anidada 202
Clases anidadas estáticas y no estáticas 203
Modificadores de Acceso para Clases Internas 205
Clases internas anónimas 206
Constructores 207
Método de clases internas locales 207
Acceso a la clase externa desde una clase interna no estática 208
Crear instancia de clase interna no estática desde el exterior 209
Capítulo 35: Clases y objetos 210
Introducción 210
Sintaxis 210
Examples 210
Clase posible más simple 210
Miembro Objeto vs Miembro Estático 210
Métodos de sobrecarga 211
Construcción y uso de objetos básicos 212
Constructores 215
Inicializando campos finales estáticos usando un inicializador estático 216
Explicando qué es el método de sobrecarga y anulación 216
Capítulo 36: Clonación de objetos 220
Observaciones 220
Examples 220
Clonación utilizando un constructor de copia 220
Clonación implementando la interfaz clonable 221
Clonación realizando una copia superficial 221
Clonación realizando una copia profunda 222
Clonación utilizando una fábrica de copias 223
Capítulo 37: Codificación de caracteres 224

Examples 224
Leyendo texto de un archivo codificado en UTF-8 224
Escribiendo texto a un archivo en UTF-8 224
Obtención de la representación en byte de una cadena en UTF-8 225
Capítulo 38: Código oficial de Oracle estándar 226
Introducción 226
Observaciones 226
Examples 226
Convenciones de nombres 226
Nombres de paquetes 226
Clase, Interfaz y Nombres Enum 226
Nombres de los métodos 227
Variables 227
Variables de tipo 227
Constantes 227
Otras directrices sobre nombramiento 228
Archivos fuente de Java 228
Caracteres especiales 228
Declaración del paquete 228
Declaraciones de importación 229
Importaciones de comodines 229
Estructura de clase 229
Orden de los miembros de la clase 229
Agrupación de miembros de la clase 230
Modificadores 230
Sangría 231
Envolver declaraciones 231
Declaraciones del método de envoltura 232
Expresiones de envoltura 233
Espacio en blanco 233
Espacio en blanco vertical 233

Espacio en blanco horizontal 234
Declaraciones Variables 234
Anotaciones 234
Expresiones lambda 235
Paréntesis redundantes 236
Literales 236
Tirantes 236
Formas cortas 237
Capítulo 39: Colas y deques 238
Examples 238
El uso de la PriorityQueue 238
LinkedList como una cola FIFO 238
Pilas 239
¿Qué es una pila? 239
API de pila 239
Ejemplo 239
BlockingQueue 240
Interfaz de cola 241
Deque 242
Agregar y acceder a los elementos 243
Removiendo elementos 243
Capítulo 40: Colecciones 244
Introducción 244
Observaciones 244
Examples 245
Declarar una ArrayList y agregar objetos 245
Construyendo colecciones a partir de datos existentes 246
Colecciones estandar 246
Marco de Colecciones Java 246
Marco de colecciones de guayaba de Google 246
Mapeo de Colecciones 247

Marco de Colecciones Java....247
Marco de Apache Commons Collections....247
Marco de colecciones de guayaba de Google....247
Unir listas....248
Eliminar elementos de una lista dentro de un bucle....248
INCORRECTO....248
Extracción de iteración de for declaración Omite "Banana":....249
Eliminando en la excepción mejorada for tiros de instrucción :....249
CORRECTO....249
Eliminando en bucle while usando un Iterator....249
Iterando hacia atrás....250
Iterando hacia adelante, ajustando el índice de bucle....251
Usando una lista de "debería ser eliminado"....251
Filtrando una corriente....251
Utilizando removeIf....251
Colección no modificable....252
Iterando sobre colecciones....252
Iterando sobre la lista....252
Iterando sobre Set....253
Iterando sobre el mapa....254
Colecciones vacias inmutables....254
Colecciones y valores primitivos....255
Eliminar elementos coincidentes de las listas utilizando Iterator....255
Creando su propia estructura de Iterable para usar con Iterator o para cada bucle....256
Pitfall: excepciones de modificaciones concurrentes....258
Sub Colecciones....259
Lista de sublistas (int fromIndex, int toIndex)....259
Establecer subSet (fromIndex, toIndex)....259
Mapa de mapa secundario (fromKey, toKey)....260
Capítulo 41: Colecciones alternativas....261
Observaciones....261

Examples 261
Apache HashBag, Guava HashMultiset y Eclipse HashBag 261
1. Usando SynchronizedSortedBag de Apache : 261
2. Usando TreeBag de Eclipse (GC) : 262
3. Usando LinkedHashMultiset desde Guava : 262
Más ejemplos: 263
Multimap en colecciones de guayaba, apache y eclipse 263
Más ejemplos: 266
Comparar operación con colecciones - Crear colecciones 266
Comparar operación con colecciones - Crear colecciones 266
Capítulo 42: Colecciones concurrentes 272
Introducción 272
Examples 272
Colecciones a prueba de hilos 272
Colecciones concurrentes 272
Hilo seguro pero ejemplos no concurrentes 274
Inserción en el mapa de hash concurrente 274
Capítulo 43: Comandos de tiempo de ejecución 276
Examples 276
Añadiendo ganchos de apagado 276
Capítulo 44: Comparable y comparador 277
Sintaxis 277
Observaciones 277
Examples 277
Ordenar una lista usando Comparable o un comparador 278
Comparadores basados en expresiones Lambda 281
Métodos predeterminados del comparador 281
Invertir el orden de un comparador 281
Los métodos de comparar y comparar 281
Clasificación natural (comparable) vs explícita (comparativa) 282
Ordenar entradas de mapa 283

Creando un comparador usando el método de comparación 284
Capítulo 45: Comparación de C ++ 285
Introducción 285
Observaciones 285
Clases definidas dentro de otras construcciones # 285
Definido dentro de otra clase 285
C ++ 285
Java 285
Definido estáticamente dentro de otra clase 285
C ++ 286
Java 286
Definido dentro de un método 286
C ++ 286
Java 286
Anulando vs Sobrecarga 287
Polimorfismo 287
Orden de Construcción / Destrucción 287
Limpieza de objetos 287
Métodos y clases abstractas 288
Modificadores de accesibilidad 288
Ejemplo de C ++ Friend 289
El temido problema del diamante 289
clase java.lang.Object 289
Colecciones Java y Contenedores C ++ 289
Diagrama de flujo de colecciones Java 289
Diagrama de flujo de contenedores C ++ 289
Tipos enteros 289
Examples 290
Miembros de la clase estática 290
Ejemplo de C ++ 290

Ejemplo de Java 291
Clases definidas dentro de otras construcciones 291
Definido dentro de otra clase 291
C ++ 291
Java 291
Definido estáticamente dentro de otra clase 291
C ++ 292
Java 292
Definido dentro de un método 292
C ++ 292
Java 292
Pasar por valor y Pasar por referencia 293
Ejemplo de C ++ (código completo) 293
Ejemplo de Java (código completo) 293
Herencia vs Composición 294
Despreciación por desvalorización 294
Ejemplo de C ++ 294
Ejemplo de Java 294
Métodos y clases abstractas 294
Método abstracto 294
C ++ 294
Java 295
Clase abstracta 295
C ++ 295
Java 295
Interfaz 295
C ++ 295
Java 295
Capítulo 46: Compilador de Java - 'javac' 296
Observaciones 296

Examples ... 296
El comando 'javac' - empezando ... 296
Ejemplo simple ... 296
Ejemplo con paquetes ... 297
Compilando múltiples archivos a la vez con 'javac' ... 298
Opciones 'javac' usadas comúnmente ... 299
Referencias ... 299
Compilando para una versión diferente de Java ... 299
Compilando Java antiguo con un compilador más nuevo ... 299
Compilando para una plataforma de ejecución anterior ... 300
Capítulo 47: CompletableFuture ... 302
Introducción ... 302
Examples ... 302
Convertir el método de bloqueo a asíncrono ... 302
Ejemplo simple de CompletableFuture ... 303
Capítulo 48: Conjuntos ... 304
Examples ... 304
Declarando un HashSet con valores ... 304
Tipos y uso de conjuntos ... 304
HashSet - Clasificación aleatoria ... 304
LinkedHashSet - Orden de inserción ... 304
TreeSet - por compareTo() o Comparator ... 305
Inicialización ... 305
Fundamentos de set ... 306
Crear una lista de un conjunto existente ... 307
Eliminando duplicados utilizando Set ... 308
Capítulo 49: Constructores ... 309
Introducción ... 309
Observaciones ... 309
Examples ... 309
Constructor predeterminado ... 309

Constructor con Argumentos ... 310
Llamar al constructor padre ... 311
Capítulo 50: Convertir hacia y desde cuerdas ... 313
Examples ... 313
Convertir otros tipos de datos a String ... 313
Conversión a / desde bytes ... 313
Codificación / decodificación de Base64 ... 314
Analizar cadenas a un valor numérico ... 315
Obteniendo un `String` de un` InputStream` ... 316
Convertir cadena a otros tipos de datos ... 317
Capítulo 51: Corrientes ... 319
Introducción ... 319
Sintaxis ... 319
Examples ... 319
Usando Streams ... 319
Cierre de arroyos ... 320
Orden de procesamiento ... 321
Diferencias de Contenedores (o Colecciones) ... 322
Recoge los elementos de una corriente en una colección ... 322
Recopilar con toList() y toSet() ... 322
Control explícito sobre la implementación de List o Set ... 323
Hoja de trucos ... 325
Corrientes infinitas ... 325
Corrientes de consumo ... 326
h21 ... 327
Creando un Mapa de Frecuencia ... 328
Corriente paralela ... 328
Impacto en el rendimiento ... 329
Convertir un flujo de opcionales a un flujo de valores ... 329
Creando un Stream ... 329
Encontrar estadísticas sobre flujos numéricos ... 331
Obtener un trozo de un arroyo ... 331

Corrientes de concatenación....331
IntStream to String....332
Ordenar usando Stream....332
Arroyos de primitivos....333
Recopilar los resultados de una secuencia en una matriz....333
Encontrar el primer elemento que coincide con un predicado....334
Usando IntStream para iterar sobre los índices....334
Aplanar arroyos con mapa plano ()....335
Crear un mapa basado en una corriente....335
Generando cadenas aleatorias utilizando Streams....337
Usando Streams para Implementar Funciones Matemáticas....338
Uso de flujos y referencias de métodos para escribir procesos de autodocumentación....338
Uso de flujos de Map.Entry para conservar los valores iniciales después del mapeo....339
Categorías de operaciones de flujo....339
Operaciones intermedias:....339
Terminal de Operaciones....340
Operaciones sin Estado....340
Operaciones de estado....340
Convertir un iterador a un flujo....341
Reducción con arroyos....341
Unir un flujo a una sola cadena....344
Capítulo 52: Creando imágenes programáticamente....346
Observaciones....346
Examples....346
Creando una imagen simple programáticamente y mostrándola....346
Guardar una imagen en el disco....347
Especificando la calidad de renderizado de la imagen....347
Creando una imagen con la clase BufferedImage....349
Edición y reutilización de imágenes con BufferedImage....350
Configuración del color de píxel individual en BufferedImage....351
Cómo escalar una imagen almacenada....351
Capítulo 53: Desmontaje y Descompilación....353

Sintaxis 353
Parámetros 353
Examples 354
Viendo bytecode con javap 354
Capítulo 54: Despliegue de Java 361
Introducción 361
Observaciones 361
Examples 361
Haciendo un JAR ejecutable desde la línea de comando 361
Creando archivos JAR, WAR y EAR 362
Creando archivos JAR y WAR usando Maven 363
Creando archivos JAR, WAR y EAR usando Ant 363
Creando archivos JAR, WAR y EAR usando un IDE 363
Creando archivos JAR, WAR y EAR usando el comando jar 363
Introducción a Java Web Start 364
Prerrequisitos 364
Un ejemplo de archivo JNLP 365
Configurando el servidor web 365
Habilitar el lanzamiento a través de una página web. 365
Lanzar aplicaciones Web Start desde la línea de comandos. 366
Creando un UberJAR para una aplicación y sus dependencias 366
Creando un UberJAR usando el comando "jar" 366
Creando un UberJAR utilizando Maven 367
Las ventajas y desventajas de los UberJARs. 367
Capítulo 55: Dividir una cadena en partes de longitud fija 369
Observaciones 369
Examples 369
Rompe una cuerda en subcadenas, todas de una longitud conocida 369
Rompe una cadena en subcadenas todas de longitud variable 369
Capítulo 56: Documentando el código de Java 370
Introducción 370

Sintaxis ... 370
Observaciones ... 371
Examples ... 371
Documentación de la clase ... 371
Método de Documentación ... 372
Documentacion de campo ... 372
Documentación del paquete ... 373
Campo de golf ... 373
Construyendo Javadocs desde la línea de comando ... 375
Documentación del código en línea ... 375
Fragmentos de código dentro de la documentación ... 376
Capítulo 57: E / S de consola ... 378
Examples ... 378
Lectura de entrada de usuario desde la consola ... 378
Utilizando BufferedReader : ... 378
Utilizando Scanner : ... 378
Utilizando System.console : ... 379
Implementar el comportamiento básico de la línea de comandos ... 380
Alineación de cuerdas en consola ... 381
Ejemplos de cadenas de formato ... 382
Capítulo 58: Ediciones, versiones, lanzamientos y distribuciones de Java ... 383
Examples ... 383
Diferencias entre las distribuciones Java SE JRE o Java SE JDK ... 383
Java Runtime Environment ... 383
Kit de desarrollo de Java ... 383
¿Cuál es la diferencia entre Oracle Hotspot y OpenJDK? ... 384
Diferencias entre Java EE, Java SE, Java ME y JavaFX ... 385
Las plataformas de lenguaje de programación Java ... 385
Java SE ... 385
Java EE ... 385
Java ME ... 386

Java FX 386
Versiones de Java SE 386
Historial de versiones de Java SE 386
Aspectos destacados de la versión de Java SE 387
Capítulo 59: Ejecutor, ExecutorService y grupos de subprocesos 389
Introducción 389
Observaciones 389
Examples 389
Fuego y olvido - Tareas ejecutables 389
ThreadPoolExecutor 390
Recuperando valor de cómputo - Callable 391
Programar tareas para que se ejecuten a una hora determinada, después de un retraso o repe 392
Iniciar una tarea después de un retraso fijo 392
Comenzando tareas a una tasa fija 392
Comenzando tareas con un retraso fijo 393
Manejar Ejecución Rechazada 393
diferencias de manejo de excepciones de submit () vs execute () 394
Casos de uso para diferentes tipos de construcciones de concurrencia. 396
Espere a que se completen todas las tareas en ExecutorService 397
Casos de uso para diferentes tipos de servicios de ejecución 399
Uso de grupos de subprocesos 402
Capítulo 60: El classpath 403
Introducción 403
Observaciones 403
Examples 403
Diferentes formas de especificar el classpath. 403
Agregando todos los JARs en un directorio a la ruta de clase 404
Sintaxis de ruta de clase 405
Classpath dinámico 405
Cargar un recurso desde el classpath 405
Asignación de nombres de clases a rutas de acceso 406
Qué significa el classpath: cómo funcionan las búsquedas 407

La ruta de clases de arranque 407

Capítulo 61: El Comando de Java - 'java' y 'javaw' 409

Sintaxis 409

Observaciones 409

Examples 409

Ejecutando un archivo JAR ejecutable 409

Ejecutando aplicaciones Java a través de una clase "principal" 410

Ejecutando la clase HelloWorld 410

Especificando un classpath 410

Clases de punto de entrada 411

Puntos de entrada de JavaFX 411

Solución de problemas del comando 'java' 411

"Comando no encontrado" 411

"No se pudo encontrar o cargar la clase principal" 412

"El método principal no se encuentra en la clase <nombre>" 413

Otros recursos 413

Ejecutando una aplicación Java con dependencias de biblioteca 414

Espacios y otros caracteres especiales en argumentos 415

Soluciones usando un shell POSIX 415

Solución para Windows 416

Opciones de Java 416

Configurando las propiedades del sistema con -D 417

Opciones de memoria, pila y recolector de basura 417

Habilitar y deshabilitar aserciones 417

Seleccionando el tipo de máquina virtual 418

Capítulo 62: Encapsulacion 419

Introducción 419

Observaciones 419

Examples 419

Encapsulamiento para mantener invariantes 419

Encapsulamiento para reducir el acoplamiento 420

Capítulo 63: Enchufes 422
Introducción 422
Examples 422
Leer de socket 422
Capítulo 64: Enum a partir del número 423
Introducción 423
Examples 423
Enum con nombre al principio 423
Capítulo 65: Enums 424
Introducción 424
Sintaxis 424
Observaciones 424
Restricciones 424
consejos y trucos 424
Examples 425
Declarar y usar una enumeración básica 425
Enums con constructores 428
Utilizando métodos y bloques estáticos 430
Implementa interfaz 431
Patrón de polimorfismo enum 432
Enums con métodos abstractos 433
Documentando enumeraciones 433
Obtener los valores de una enumeración 434
Enum como un parámetro de tipo limitado 435
Obtener enumeración constante por nombre 435
Implementar el patrón Singleton con una enumeración de un solo elemento 436
Enumerar con propiedades (campos) 436
Convertir enum a cadena 437
Convertir usando name() 437
Convertir utilizando toString() 437
Por defecto: 438
Ejemplo de ser anulado 438

Enumeración específica del cuerpo....438
Enumeración cero instancia....440
Enums con campos estáticos....440
Comparar y Contiene para los valores Enum....441
Capítulo 66: Errores comunes de Java....443
Introducción....443
Examples....443
Pitfall: utilizando == para comparar objetos de envoltorios primitivos, como Integer....443
Pitfall: olvidarse de liberar recursos....444
Trampa: fugas de memoria....445
Pitfall: usando == para comparar cadenas....446
Pitfall: probar un archivo antes de intentar abrirlo....448
Pitfall: pensar las variables como objetos....449
Clase de ejemplo....450
Múltiples variables pueden apuntar al mismo objeto....450
El operador de igualdad NO prueba que dos objetos son iguales....451
Las llamadas a métodos NO pasan objetos en absoluto....452
Pitfall: combinación de asignación y efectos secundarios....453
Pitfall: no entender que String es una clase inmutable....453
Capítulo 67: Errores de Java - Nulls y NullPointerException....455
Observaciones....455
Examples....455
Trampa: el uso innecesario de envolturas primitivas puede llevar a NullPointerExceptions....455
Pitfall - Usar nulo para representar una matriz o colección vacía....456
Pitfall - "Haciendo buenos" nulos inesperados....457
¿Qué significa que "a" o "b" sean nulos?....458
¿El nulo vino de una variable no inicializada?....458
¿El nulo representa un "no se sabe" o "valor perdido"?....458
Si se trata de un error (o un error de diseño), ¿debemos "solucionarlo"?....458
¿Es esto eficiente / bueno para la calidad del código?....459
En resumen....459
Pitfall - Devolver nulo en lugar de lanzar una excepción....459

Pitfall: no se comprueba si un flujo de E / S ni siquiera se inicializa al cerrarlo ... 460
Pitfall: uso de la "notación Yoda" para evitar la excepción NullPointerException ... 460
Capítulo 68: Errores de Java - Problemas de rendimiento ... 462
Introducción ... 462
Observaciones ... 462
Examples ... 462
Pitfall - Los gastos generales de crear mensajes de registro ... 462
Solución ... 462
Pitfall - La concatenación de cadenas en un bucle no se escala ... 463
Pitfall: el uso de 'nuevo' para crear instancias de contenedor primitivas es ineficiente ... 464
Pitfall - Llamar 'nueva cadena (String)' es ineficiente ... 465
Pitfall - Calling System.gc () es ineficiente ... 465
Pitfall - El uso excesivo de tipos de envoltorios primitivos es ineficiente ... 466
Pitfall - Iterar las claves de un mapa puede ser ineficiente ... 467
Pitfall: el uso de size () para comprobar si una colección está vacía es ineficiente ... 468
Pitfall - Problemas de eficiencia con expresiones regulares ... 468
Las instancias de Pattern y Matcher deben ser reutilizadas ... 468
No uses match () cuando deberías usar find () ... 470
Usar alternativas más eficientes a las expresiones regulares ... 470
Retroceso catastrófico ... 471
Pitfall - Interning Strings para que puedas usar == es una mala idea ... 472
Fragilidad ... 472
Costos de usar 'intern ()' ... 472
El impacto en la recolección de basura ... 473
El tamaño de hashtable del grupo de cadenas ... 473
Interning como un potencial vector de denegación de servicio ... 473
Pitfall - Las lecturas / escrituras pequeñas en flujos no almacenados en búfer son inefici ... 474
¿Qué pasa con las corrientes basadas en caracteres? ... 475
¿Por qué los flujos amortiguados hacen tanta diferencia? ... 475
¿Las transmisiones amortiguadas son siempre una victoria? ... 476
¿Es esta la forma más rápida de copiar un archivo en Java? ... 476
Capítulo 69: Errores de Java - sintaxis de lenguaje ... 477

Introducción 477
Observaciones 477
Examples 477
Pitfall - Ignorar la visibilidad del método 477
Pitfall - Falta un 'break' en un caso de 'cambio' 477
Pitfall - punto y coma mal colocados y llaves faltantes 478
Trampa: omitir llaves: los problemas de "colgar si" y "colgar de otra manera" 480
Trampa: sobrecargar en lugar de anular 482
Pitfall - Octales literales 483
Pitfall - Declarar clases con los mismos nombres que las clases estándar 483
Pitfall - Usar '==' para probar un booleano 484
Pitfall: las importaciones de comodines pueden hacer que su código sea frágil 485
Pitfall: Usar 'assert' para validar argumentos o entradas de usuario 486
El escollo de los objetos nulos de autoenvasado en primitivos 487
Capítulo 70: Errores de Java - Uso de excepciones 488
Introducción 488
Examples 488
Pitfall - Ignorar o aplastar excepciones 488
Pitfall: captura de Throwable, Exception, Error o RuntimeException 489
Pitfall - Lanzar Throwable, Exception, Error o RuntimeException 490
Declarar Throwable o Exception en los "lanzamientos" de un método es problemático. 491
Pitfall - Catching InterruptedException 492
Pitfall - Uso de excepciones para el control de flujo normal 494
Pitfall - Stacktraces excesivos o inapropiados 495
Pitfall - Subclasificando directamente `Throwable` 495
Capítulo 71: Escáner 497
Sintaxis 497
Parámetros 497
Observaciones 497
Examples 497
Lectura de la entrada del sistema utilizando el escáner 497
Lectura de entrada de archivo utilizando escáner 497

Lee toda la entrada como una cadena usando un escáner ... 498
Usando delimitadores personalizados ... 498
Patrón general que realiza las tareas más frecuentes ... 499
Lee un int desde la línea de comando ... 501
Cerrar cuidadosamente un escáner ... 501
Capítulo 72: Escogiendo Colecciones ... 503
Introducción ... 503
Examples ... 503
Diagrama de flujo de colecciones Java ... 503
Capítulo 73: Escritor Buffered ... 504
Sintaxis ... 504
Observaciones ... 504
Examples ... 504
Escribe una línea de texto a archivo ... 504
Capítulo 74: Estructuras de control básicas ... 506
Observaciones ... 506
Examples ... 506
Si / Else Si / Else Control ... 506
Para loops ... 507
Mientras bucles ... 508
hacer ... mientras bucle ... 508
Para cada ... 509
Si / Else ... 510
Cambiar la declaración ... 510
Operador ternario ... 512
Descanso ... 512
Intenta ... Atrapa ... Finalmente ... 513
Pausa anidada / continuar ... 514
Continuar declaración en Java ... 514
Capítulo 75: Evaluación XML XPath ... 515
Observaciones ... 515
Examples ... 515

Evaluando un NodeList en un documento XML 515
Analizar múltiples expresiones XPath en un solo XML 516
Analizar una sola expresión XPath varias veces en un XML 516
Capítulo 76: Examen de la unidad 518
Introducción 518
Observaciones 518
Marcos de prueba unitaria 518
Herramientas de prueba unitaria 518
Examples 518
¿Qué es la prueba unitaria? 518
Las pruebas deben ser automatizadas 520
Las pruebas deben ser de grano fino 520
Entrar en la prueba de la unidad 521
Capítulo 77: Excepciones y manejo de excepciones. 522
Introducción 522
Sintaxis 522
Examples 522
Atrapando una excepción con try-catch 522
Prueba-captura con un bloque de captura 522
Prueba-captura con múltiples capturas 523
Bloques de captura multi-excepción 524
Lanzar una excepción 525
Encadenamiento de excepciones 525
Excepciones personalizadas 526
La declaración de prueba con recursos 528
¿Qué es un recurso? 528
La declaración básica de prueba con recursos 528
Las declaraciones mejoradas de prueba con recursos 529
Gestionando múltiples recursos 529
Equivalencia de try-with-resource y clásico try-catch-finally 530
Creación y lectura de stacktraces 531

Imprimiendo un stacktrace ... 531
Entendiendo un stacktrace ... 532
Excepción de encadenamiento y stacktraces anidados. ... 533
Capturando un stacktrace como una cadena ... 534
Manipulación InterruptedException ... 535
La jerarquía de excepciones de Java - Excepciones no verificadas y revisadas ... 536
Excepciones marcadas versus no verificadas ... 537
Ejemplos de excepciones verificadas ... 538
Introducción ... 539
Declaraciones de retorno en el bloque try catch ... 541
Funciones avanzadas de excepciones ... 542
Examinando la pila de llamadas programáticamente ... 542
Optimizando la construcción de excepciones. ... 543
Borrado o sustitución del trazo de pila. ... 544
Excepciones suprimidas ... 544
Las declaraciones try-finally y try-catch-finally ... 544
Intento-finalmente ... 545
prueba-captura-finalmente ... 545
La cláusula de 'tiros' en una declaración de método. ... 546
¿Cuál es el punto de declarar excepciones no marcadas como lanzadas? ... 547
Tiros y anulación de método. ... 547
Capítulo 78: Expresiones ... 548
Introducción ... 548
Observaciones ... 548
Examples ... 548
Precedencia del operador ... 548
Expresiones constantes ... 550
Usos para expresiones constantes ... 550
Orden de evaluación de expresiones ... 551
Ejemplo simple ... 552
Ejemplo con un operador que tiene un efecto secundario ... 552
Conceptos básicos de expresión ... 552

El tipo de expresión 553
El valor de una expresión 554
Declaraciones de Expresión 554
Capítulo 79: Expresiones lambda 555
Introducción 555
Sintaxis 555
Examples 555
Usando expresiones Lambda para ordenar una colección 555
Listas de clasificación 555
Clasificando mapas 556
Introducción a las lambdas de Java 557
Interfaces funcionales 557
Expresiones lambda 558
Devoluciones implícitas 559
Acceso a variables locales (cierres de valor) 559
Aceptando Lambdas 560
El tipo de expresión lambda 560
Referencias del método 561
Referencia del método de instancia (a una instancia arbitraria) 561
Referencia del método de instancia (a una instancia específica) 561
Referencia de método estático 562
Referencia a un constructor 562
Hoja de trucos 562
Implementando multiples interfaces 563
Lambdas y patrón de ejecución 563
Usando lambda expresión con su propia interfaz funcional 564
`return 'solo regresa de la lambda, no del método externo 565
Cierres de Java con expresiones lambda 566
Lambda - Ejemplo de oyente 568
Estilo tradicional al estilo Lambda 569
Lambdas y utilización de memoria 570

Usar expresiones y predicados lambda para obtener un determinado valor (es) de una lista ... 570

Capítulo 80: Expresiones regulares ... 572

Introducción ... 572

Sintaxis ... 572

Observaciones ... 572

Importaciones ... 572

Escollos ... 572

Símbolos importantes explicados ... 572

Otras lecturas ... 573

Examples ... 573

Utilizando grupos de captura ... 573

Usando expresiones regulares con comportamiento personalizado compilando el patrón con ban ... 574

Personajes de escape ... 574

Coincidencia con un regex literal ... 575

No coincide con una cadena dada ... 576

A juego con una barra invertida ... 576

Capítulo 81: Fechas y hora (java.time. *) ... 578

Examples ... 578

Manipulaciones de fecha simple ... 578

Fecha y hora ... 578

Operaciones en fechas y horarios ... 579

Instante ... 579

Uso de varias clases de Date Time API ... 579

Formato de fecha y hora ... 581

Calcular la diferencia entre 2 fechas locales ... 582

Capítulo 82: FileUpload a AWS ... 583

Introducción ... 583

Examples ... 583

Subir archivo a s3 bucket ... 583

Capítulo 83: Formato numérico ... 586

Examples ... 586

Formato numérico ... 586

Capítulo 84: FTP (Protocolo de transferencia de archivos) ... 587
Sintaxis ... 587
Parámetros ... 587
Examples ... 587
Conexión e inicio de sesión en un servidor FTP ... 587
Capítulo 85: Generación de números aleatorios ... 593
Observaciones ... 593
Examples ... 593
Números pseudoaleatorios ... 593
Números pseudoaleatorios en rango específico ... 593
Generación de números pseudoaleatorios criptográficamente seguros ... 594
Seleccionar números aleatorios sin duplicados ... 595
Generando números aleatorios con una semilla especificada ... 596
Generando números aleatorios usando apache-common lang3 ... 596
Capítulo 86: Generando Código Java ... 598
Examples ... 598
Generar POJO desde JSON ... 598
Capítulo 87: Genéricos ... 599
Introducción ... 599
Sintaxis ... 599
Observaciones ... 599
Examples ... 599
Creando una clase genérica ... 599
Extendiendo una clase genérica ... 600
Parámetros de tipo múltiple ... 602
Declarar un método genérico ... 602
El diamante ... 603
Requerir múltiples límites superiores ("extiende A y B") ... 604
Creando una clase genérica limitada ... 605
Decidir entre 'T', `? super T`, y `? extiende T` ... 606
Beneficios de la clase genérica y la interfaz ... 607

Controles de tipo más fuertes en tiempo de compilación 608
Eliminación de moldes 608
Permitiendo a los programadores implementar algoritmos genéricos 608
Vinculación de parámetro genérico a más de 1 tipo 608
Nota: 609
Creando un tipo genérico 609
Soluciones 609
Refiriéndose al tipo genérico declarado dentro de su propia declaración. 610
Uso de instanceof con Genéricos. 611
Diferentes formas de implementar una interfaz genérica (o extender una clase genérica) 613
Usando genéricos para auto-cast 614
Obtenga una clase que satisfaga el parámetro genérico en tiempo de ejecución 615
Capítulo 88: Gerente de seguridad 617
Examples 617
Habilitando el SecurityManager 617
Clases de sandboxing cargadas por un ClassLoader 617
Implementando reglas de negación de políticas. 618
La clase de DeniedPermission 619
La clase DenyingPolicy 623
Manifestación 625
Capítulo 89: Gestión de memoria de Java 627
Observaciones 627
Examples 627
Finalización 627
Los finalizadores solo se ejecutan una vez. 628
Activación manual de GC 628
Recolección de basura 628
El enfoque de C ++ - nuevo y eliminar 628
El enfoque de Java - recolección de basura 629
¿Qué sucede cuando un objeto se vuelve inalcanzable? 630
Ejemplos de objetos alcanzables e inalcanzables 630
Configuración de los tamaños Heap, PermGen y Stack 631

Fugas de memoria en Java 632
Los objetos alcanzables pueden fugarse 632
Los cachés pueden ser fugas de memoria 633
Capítulo 90: Gráficos 2D en Java 635
Introducción 635
Examples 635
Ejemplo 1: dibujar y rellenar un rectángulo utilizando Java 635
Ejemplo 2: Dibujo y relleno oval 637
Capítulo 91: Hechiceros y Setters 638
Introducción 638
Examples 638
Adición de Getters y Setters 638
Usar un setter o getter para implementar una restricción 639
¿Por qué usar Getters y Setters? 639
Capítulo 92: Herencia 642
Introducción 642
Sintaxis 642
Observaciones 642
Examples 642
Clases abstractas 642
Herencia estática 644
Usando 'final' para restringir la herencia y anular 645
Clases finales 645
Casos de uso para las clases finales 646
Métodos finales 646
El principio de sustitución de Liskov 647
Herencia 647
Herencia y métodos estáticos 649
Sombreado variable 650
Reducción y ampliación de referencias de objetos 650
Programación a una interfaz 651
Clase abstracta y uso de la interfaz: "Is-a" relationship vs "Has-a" capacity 654

Anulando en herencia ... 657
Capítulo 93: Hora local ... 659
Sintaxis ... 659
Parámetros ... 659
Observaciones ... 659
Examples ... 659
Modificación de tiempo ... 659
Zonas horarias y su diferencia horaria ... 660
Cantidad de tiempo entre dos LocalTime ... 660
Introducción ... 661
Capítulo 94: HttpURLConnection ... 663
Observaciones ... 663
Examples ... 663
Obtener el cuerpo de respuesta de una URL como una cadena ... 663
Datos POST ... 664
Cómo funciona ... 665
Eliminar recurso ... 665
Cómo funciona ... 665
Compruebe si el recurso existe ... 666
Explicación: ... 666
Ejemplo: ... 666
Capítulo 95: Implementaciones del sistema de plugin Java ... 667
Observaciones ... 667
Examples ... 667
Utilizando URLClassLoader ... 667
Capítulo 96: InputStreams y OutputStreams ... 672
Sintaxis ... 672
Observaciones ... 672
Examples ... 672
Leyendo InputStream en una cadena ... 672
Escritura de bytes en un OutputStream ... 672

Cierre de arroyos ... 673
Copiar el flujo de entrada al flujo de salida ... 674
Envolviendo flujos de entrada / salida ... 674
Combinaciones utiles ... 674
Lista de envoltorios de flujo de entrada / salida ... 675
Ejemplo de DataInputStream ... 676
Capítulo 97: Instalando Java (Edición Estándar) ... 677
Introducción ... 677
Examples ... 677
Configurando% PATH% y% JAVA_HOME% después de instalar en Windows ... 677
Suposiciones ... 677
Pasos de configuración ... 677
Revisa tu trabajo ... 678
Selección de una versión adecuada de Java SE ... 678
Lanzamiento de Java y nombre de versión ... 679
¿Qué necesito para el desarrollo de Java? ... 680
Instalación de un JDK de Java en Linux ... 680
Usando el Administrador de paquetes ... 680
Instalación desde un archivo de Oracle Java RPM. ... 682
Instalación de un JDK o JRE de Java en Windows ... 682
Instalando un Java JDK en macOS ... 683
Configurando y cambiando versiones de Java en Linux usando alternativas ... 685
Usando Alternativas ... 685
Instalaciones basadas en arco ... 686
Listado de entornos instalados ... 686
Cambio de entorno actual ... 686
Comprobación y configuración posterior a la instalación en Linux ... 686
Instalando Oracle Java en Linux con el último archivo tar ... 688
Rendimiento esperado: ... 689
Capítulo 98: Instrumentos de cuerda ... 690
Introducción ... 690
Observaciones ... 690

Examples ... 691
Comparando cuerdas ... 691
No use el operador == para comparar cadenas ... 692
Comparando cadenas en una instrucción switch ... 692
Comparando cadenas con valores constantes ... 693
Ordenaciones de cuerdas ... 693
Comparando con cuerdas internadas ... 694
Cambiando el caso de los personajes dentro de una cadena ... 695
Encontrar una cadena dentro de otra cadena ... 696
Obtener la longitud de una cadena ... 697
Subcadenas ... 698
Obtención del personaje nth en una cadena ... 698
Plataforma independiente de nueva línea separadora. ... 699
Añadiendo el método toString () para objetos personalizados ... 699
Dividir cuerdas ... 700
Uniendo cuerdas con un delimitador ... 703
Cuerdas de inversión ... 704
Contar las apariciones de una subcadena o carácter en una cadena ... 704
Concatenación de cadenas y StringBuilders ... 705
Reemplazo de partes de cuerdas ... 706
Coincidencia exacta ... 707
Reemplazar el carácter único con otro carácter único: ... 707
Reemplace la secuencia de caracteres con otra secuencia de caracteres: ... 707
Regex ... 707
Reemplazar todos los partidos: ... 707
Reemplazar solo el primer partido: ... 708
Eliminar espacios en blanco desde el principio y el final de una cadena ... 708
Cadena de almacenamiento y almacenamiento en montón ... 708
Interruptor insensible a la caja ... 711
Capítulo 99: Interfaces ... 712
Introducción ... 712

Sintaxis 712
Examples 712
Declarar e implementar una interfaz 712
Implementando multiples interfaces 713
Extendiendo una interfaz 714
Usando interfaces con genéricos 715
Utilidad de las interfaces 717
Implementando interfaces en una clase abstracta 719
Métodos predeterminados 719
Implementación del patrón observador 719
Problema del diamante 720
Utilice métodos predeterminados para resolver problemas de compatibilidad 721
Modificadores en interfaces 722
Variables 722
Métodos 722
Reforzar los parámetros de tipo acotado 723
Capítulo 100: Interfaces funcionales 725
Introducción 725
Examples 725
Lista de interfaces funcionales estándar de Java Runtime Library por firma 725
Capítulo 101: Interfaz de herramientas JVM 728
Observaciones 728
Examples 728
Iterar sobre objetos accesibles desde objeto (Heap 1.0) 728
Obtener entorno JVMTI 730
Ejemplo de inicialización dentro del método Agent_OnLoad 731
Capítulo 102: Interfaz de salida de cola 732
Introducción 732
Observaciones 732
Examples 732
Añadiendo elementos a Deque 732
Eliminando Elementos de Deque 732

Recuperando elemento sin quitar ... 733
Iterando a través de Deque ... 733
Capítulo 103: Interfaz fluida ... 734
Observaciones ... 734
Examples ... 734
Verdad - Marco de prueba de fluidez ... 734
Estilo de programación fluido ... 734
Capítulo 104: Interfaz nativa de Java ... 737
Parámetros ... 737
Observaciones ... 737
Examples ... 737
Llamando a los métodos de C ++ desde Java ... 737
Código Java ... 738
Código C ++ ... 738
Salida ... 739
Llamando a métodos Java desde C ++ (callback) ... 739
Código Java ... 739
Código C ++ ... 740
Salida ... 740
Obteniendo el descriptor ... 740
Cargando bibliotecas nativas ... 741
Búsqueda de archivos de destino ... 741
Capítulo 105: Invocación de método remoto (RMI) ... 743
Observaciones ... 743
Examples ... 743
Cliente-Servidor: invocando métodos en una JVM desde otra ... 743
Devolución de llamada: invocar métodos en un "cliente" ... 745
Visión general ... 745
Las interfaces remotas compartidas ... 745
Las implementaciones ... 746
Ejemplo de RMI simple con implementación de cliente y servidor ... 749
Paquete de servidor ... 749

Paquete de cliente ... 751
Prueba tu aplicación ... 752
Capítulo 106: Iterador e iterable ... 753
Introducción ... 753
Observaciones ... 753
Examples ... 753
Usando Iterable en for loop ... 753
Usando el iterador crudo ... 753
Creando tu propio iterable ... 754
Eliminar elementos utilizando un iterador ... 755
Capítulo 107: JavaBean ... 757
Introducción ... 757
Sintaxis ... 757
Observaciones ... 757
Examples ... 758
Bean Java Básico ... 758
Capítulo 108: JAXB ... 759
Introducción ... 759
Sintaxis ... 759
Parámetros ... 759
Observaciones ... 759
Examples ... 759
Escribir un archivo XML (ordenando un objeto) ... 759
Lectura de un archivo XML (no riguroso) ... 760
Usando XmlAdapter para generar el formato xml deseado ... 761
Configuración automática de asignación de campos / propiedades XML (@XmlAccessorType) ... 762
Configuración manual de asignación de campos / propiedades XML ... 764
Especificando una instancia de XmlAdapter para (re) usar datos existentes ... 764
Ejemplo ... 765
Clase de usuario ... 765
Adaptador ... 765
Ejemplos de XML ... 767

Usando el adaptador ... 767
Vincular un espacio de nombres XML a una clase Java serializable ... 767
Usando XmlAdapter para recortar la cadena ... 768
Capítulo 109: JAX-WS ... 769
Examples ... 769
Autenticación básica ... 769
Capítulo 110: JMX ... 770
Introducción ... 770
Examples ... 770
Ejemplo simple con Platform MBean Server ... 770
Capítulo 111: JNDI ... 775
Examples ... 775
RMI a través de JNDI ... 775
Capítulo 112: JShell ... 780
Introducción ... 780
Sintaxis ... 780
Observaciones ... 780
Importaciones por defecto ... 780
Examples ... 781
Entrando y saliendo de JShell ... 781
A partir de JShell ... 781
Saliendo de JShell ... 781
Expresiones ... 781
Variables ... 781
Métodos y Clases ... 782
Editando Fragmentos ... 782
Capítulo 113: JSON en Java ... 784
Introducción ... 784
Observaciones ... 784
Examples ... 784
Codificación de datos como JSON ... 784
Decodificación de datos JSON ... 785

Métodos optXXX vs getXXX 785
Objeto a JSON (Gson Library) 786
JSON a Objeto (Biblioteca Gson) 786
Extraer un solo elemento de JSON 786
Usando Jackson Object Mapper 787
Detalles 787
Instancia de ObjectMapper 787
Deserialización: 787
Método para la serialización: 788
Iteración json 788
JSON Builder - métodos de encadenamiento 788
JSONObject.NULL 789
JsonArray a la lista de Java (Gson Library) 789
Deserializar la colección JSON a la colección de Objetos usando Jackson 790
Deserialización de matriz JSON 790
Enfoque de TypeFactory 791
Tipo de enfoque de referencia 791
Deserialización del mapa JSON 791
Enfoque de TypeFactory 791
Tipo de enfoque de referencia 791
Detalles 791
Nota 792
Capítulo 114: Just in Time (JIT) compilador 793
Observaciones 793
Historia 793
Examples 793
Visión general 793
Capítulo 115: La clase java.util.Objects 796
Examples 796
Uso básico para la comprobación del objeto nulo. 796
Para el método de verificación nulo 796

Para el método de check in no nulo 796
Uso de referencia del método Objects.nonNull () en la API de flujo 796
Capítulo 116: Las trampas de Java - Hilos y concurrencia 797
Examples 797
Trampa: uso incorrecto de esperar () / notificar () 797
El problema de "notificación perdida" 797
El error "Estado del monitor ilegal" 797
Esperar / notificar es de muy bajo nivel 798
Pitfall - Extendiendo 'java.lang.Thread' 798
Pitfall - Demasiados hilos hace que una aplicación sea más lenta 799
Pitfall - La creación de hilos es relativamente cara 800
Pitfall: las variables compartidas requieren una sincronización adecuada 801
¿Funcionará según lo previsto? 802
¿Cómo solucionamos el problema? 802
¿Pero no es la asignación atómica? 803
¿Por qué hicieron esto? 803
¿Por qué no puedo reproducir esto? 805
Capítulo 117: Lectores y escritores 806
Introducción 806
Examples 806
BufferedReader 806
Introducción 806
Conceptos básicos sobre el uso de un BufferedReader 806
El tamaño del búfer BufferedReader 807
El método BufferedReader.readLine () 807
Ejemplo: leer todas las líneas de un archivo en una lista 807
Ejemplo de StringWriter 807
Capítulo 118: LinkedHashMap 809
Introducción 809
Examples 809
Clase LinkedHashMap de Java 809

Capítulo 119: Lista vs SET 811
Introducción 811
Examples 811
Lista vs conjunto 811
Capítulo 120: Literales 812
Introducción 812
Examples 812
Literales hexadecimales, octales y binarios 812
Usando el guión bajo para mejorar la legibilidad 812
Secuencias de escape en literales 813
Unicode se escapa 814
Escapar en expresiones regulares 814
Decimales enteros literales 814
Literales enteros ordinarios 814
Literales enteros largos 815
Literales booleanos 815
Literales de cuerda 816
Cuerdas largas 816
Interning de literales de cuerda 816
El literal nulo 817
Literales de punto flotante 817
Formas decimales simples 817
Formas decimales escaladas 818
Formas hexadecimales 818
Guiones bajos 819
Casos especiales 819
Literales de personajes 819
Capítulo 121: Liza 821
Introducción 821
Sintaxis 821
Observaciones 821

Examples ... 822
Ordenar una lista genérica ... 822
Creando una lista ... 824
Operaciones de acceso posicional ... 825
Iterando sobre elementos en una lista ... 827
Eliminar elementos de la lista B que están presentes en la lista A ... 827
Encontrando elementos comunes entre 2 listas ... 828
Convertir una lista de enteros en una lista de cadenas ... 828
Creación, adición y eliminación de elementos de un ArrayList ... 829
Sustitución in situ de un elemento de lista ... 829
Haciendo una lista no modificable ... 830
Mover objetos alrededor de la lista ... 830
Clases implementando Lista - Pros y Contras ... 831
Clases implementando la lista ... 831
Pros y contras de cada implementación en términos de complejidad de tiempo ... 832
Lista de arreglo ... 832
Lista de atributos ... 833
CopyOnWriteArrayList ... 833
Lista enlazada ... 833
Lista de roles ... 833
RoleUnresolvedList ... 834
Apilar ... 834
Vector ... 834
Capítulo 122: Localización e internacionalización. ... 835
Observaciones ... 835
Recursos Generales ... 835
Recursos de Java ... 835
Examples ... 835
Fechas formateadas automáticamente usando "locale" ... 835
Deja que Java haga el trabajo por ti. ... 836
Comparación de cuerdas ... 836
Lugar ... 837

Idioma ... 837
Creando una Localidad ... 837
Java ResourceBundle ... 837
Configuración regional ... 838
Capítulo 123: log4j / log4j2 ... 839
Introducción ... 839
Sintaxis ... 839
Observaciones ... 839
Final de vida para Log4j 1 alcanzado ... 839
Examples ... 840
Como obtener Log4j ... 840
Cómo usar Log4j en código Java ... 841
Configuración de archivo de propiedades ... 841
Archivo de configuración log4j2.xml básico ... 842
Migración de log4j 1.x a 2.x ... 842
Propiedades-Archivo para iniciar sesión en DB ... 843
Filtrar Logoutput por nivel (log4j 1.x) ... 844
Capítulo 124: Los operadores ... 846
Introducción ... 846
Observaciones ... 846
Examples ... 846
El operador de concatenación de cuerdas (+) ... 846
Optimización y eficiencia ... 847
Los operadores aritméticos (+, -, *, /,%) ... 848
Operandos y tipos de resultados, y promoción numérica ... 849
El significado de división ... 850
El significado del resto ... 850
Desbordamiento de enteros ... 851
Valores de punto flotante INF y NAN ... 851
Los operadores de igualdad (==,! =) ... 852
Los operadores numéricos == y != ... 852

Los operadores booleanos == y != 853
Los operadores Reference == y != 853
Acerca de los casos de borde de NaN 854
Los operadores de incremento / decremento (++ / -) 854
El Operador Condicional (? :) 855
Sintaxis 855
Uso común 856
Los operadores bitwise y lógicos (~, &, |, ^) 857
Tipos de operandos y tipos de resultados. 857
La instancia del operador 858
Los operadores de asignación (=, + =, - =, * =, / =,% =, << =, >> =, >>> =, & =, | = y ^ = 859
Los operadores condicionales y condicionales u (&& y ||) 861
Ejemplo: usar && como guardia en una expresión 861
Ejemplo: usar && para evitar un cálculo costoso 862
Los operadores de turno (<<, >> y >>>) 862
El operador Lambda (->) 863
Los operadores relacionales (<, <=,>,> =) 864
Capítulo 125: Manipulación de bits 866
Observaciones 866
Examples 866
Empaquetar / desempaquetar valores como fragmentos de bits 866
Comprobación, configuración, borrado y conmutación de bits individuales. Utilizando mascar 867
Expresando el poder de 2 867
Comprobando si un número es una potencia de 2 868
clase java.util.BitSet 870
Cambio firmado / no firmado 871
Capítulo 126: Mapa débil 872
Introducción 872
Examples 872
Conceptos de HashHashmap 872
Capítulo 127: Mapa Enum 874
Introducción 874

Examples ... 874
Ejemplo de libro de mapas Enum ... 874
Capítulo 128: Mapas ... 875
Introducción ... 875
Observaciones ... 875
Examples ... 875
Agregar un elemento ... 875
Añadir varios elementos ... 876
Usando métodos de mapa predeterminados desde Java 8 ... 877
Borrar el mapa ... 879
Iterando a través de los contenidos de un Mapa ... 880
Fusionar, combinar y componer mapas ... 881
Componiendo el Mapa <X, Y> y el Mapa <Y, Z> para obtener el Mapa <X, Z> ... 882
Comprobar si existe la clave ... 882
Los mapas pueden contener valores nulos ... 882
Iterando entradas de mapa de manera eficiente ... 883
Usar objeto personalizado como clave ... 886
Uso de HashMap ... 886
Creación e inicialización de mapas ... 887
Introducción ... 887
Capítulo 129: Máquina virtual de Java (JVM) ... 890
Examples ... 890
Estos son los fundamentos ... 890
Capítulo 130: Método dinámico de envío ... 891
Introducción ... 891
Observaciones ... 891
Examples ... 891
Método dinámico de envío - Código de ejemplo ... 891
Capítulo 131: Métodos de la fábrica de recolección ... 894
Introducción ... 894
Sintaxis ... 894

Parámetros ... 894
Examples ... 895
Lista Ejemplos de métodos de fábrica ... 895
Conjunto Ejemplos de métodos de fábrica ... 895
Mapa Ejemplos de métodos de fábrica ... 895
Capítulo 132: Métodos predeterminados ... 896
Introducción ... 896
Sintaxis ... 896
Observaciones ... 896
Métodos predeterminados ... 896
Métodos estáticos ... 896
Referencias: ... 897
Examples ... 897
Uso básico de los métodos por defecto ... 897
Acceso a otros métodos de interfaz dentro del método predeterminado ... 898
Accediendo a métodos predeterminados sobrescritos desde la implementación de la clase ... 899
¿Por qué usar métodos predeterminados? ... 899
Clase, clase abstracta y prioridad de método de interfaz ... 900
Método predeterminado colisión de herencia múltiple ... 901
Capítulo 133: Modelo de memoria de Java ... 903
Observaciones ... 903
Examples ... 903
Motivación para el modelo de memoria ... 903
Reordenación de asignaciones ... 904
Efectos de cachés de memoria ... 905
Sincronización adecuada ... 905
El modelo de memoria ... 905
Relaciones de antes-antes ... 906
Comportamiento ... 906
Orden del programa y orden de sincronización ... 906
Sucede antes de la orden ... 907
Razonamiento antes del razonamiento aplicado a algunos ejemplos ... 908

Código de un solo hilo ... 908
Comportamiento de 'volátil' en un ejemplo con 2 hilos ... 908
Volátil con tres hilos ... 909
Cómo evitar tener que entender el modelo de memoria ... 910
Capítulo 134: Modificación Bytecode ... 912
Examples ... 912
¿Qué es Bytecode? ... 912
¿Cuál es la lógica detrás de esto? ... 912
Bueno, tiene que haber más derecho? ... 912
¿Cómo puedo escribir / editar el bytecode? ... 912
¡Me gustaría aprender más sobre el bytecode! ... 913
Cómo editar archivos jar con ASM ... 913
Cómo cargar un ClassNode como una clase ... 916
Cómo cambiar el nombre de las clases en un archivo jar ... 916
Javassist Basic ... 917
Capítulo 135: Modificadores sin acceso ... 919
Introducción ... 919
Examples ... 919
final ... 919
volátil ... 920
estático ... 921
resumen ... 922
sincronizado ... 923
transitorio ... 924
strictfp ... 924
Capítulo 136: Módulos ... 925
Sintaxis ... 925
Observaciones ... 925
Examples ... 925
Definiendo un módulo básico ... 925
Capítulo 137: Moneda y dinero ... 927

Examples ... 927
Añadir moneda personalizada ... 927
Capítulo 138: Motor de JavaScript Nashorn ... 928
Introducción ... 928
Sintaxis ... 928
Observaciones ... 928
Examples ... 928
Establecer variables globales ... 928
Hola Nashorn ... 929
Ejecutar archivo JavaScript ... 929
Salida de script de intercepción ... 930
Evaluar cadenas aritméticas ... 930
Uso de objetos Java en JavaScript en Nashorn ... 930
Implementando una interfaz desde script ... 931
Establecer y obtener variables globales ... 932
Capítulo 139: NIO - Redes ... 933
Observaciones ... 933
Examples ... 933
Usando Selector para esperar eventos (ejemplo con OP_CONNECT) ... 933
Capítulo 140: Nuevo archivo I / O ... 935
Sintaxis ... 935
Examples ... 935
Creando caminos ... 935
Recuperar información sobre una ruta ... 935
Manipulando caminos ... 936
Unirse a dos caminos ... 936
Normalizando un camino ... 936
Recuperando información utilizando el sistema de archivos ... 936
Comprobando la existencia ... 936
Comprobando si una ruta apunta a un archivo o directorio ... 937
Obteniendo propiedades ... 937

Obteniendo el tipo MIME 938
Archivos de lectura 938
Escribiendo archivos 938
Capítulo 141: Objetos inmutables 939
Observaciones 939
Examples 939
Creación de una versión inmutable de un tipo mediante copia defensiva. 939
La receta para una clase inmutable. 940
Fallas típicas de diseño que evitan que una clase sea inmutable. 941
Capítulo 142: Objetos seguros 945
Sintaxis 945
Examples 945
SealedObject (javax.crypto.SealedObject) 945
SignedObject (java.security.SignedObject) 945
Capítulo 143: Opcional 947
Introducción 947
Sintaxis 947
Examples 947
Devolver valor predeterminado si Opcional está vacío 947
Mapa 948
Lanzar una excepción, si no hay valor. 949
Filtrar 949
Usando contenedores opcionales para tipos de números primitivos 950
Ejecutar código solo si hay un valor presente 950
Proporcionar perezosamente un valor predeterminado utilizando un Proveedor 950
Mapa plano 951
Capítulo 144: Operaciones de punto flotante de Java 952
Introducción 952
Examples 952
Comparando valores de punto flotante 952
OverFlow y UnderFlow 954
Formato de los valores de punto flotante 955

Adherencia estricta a la especificación IEEE ... 956

Capítulo 145: Paquetes ... 958

Introducción ... 958

Observaciones ... 958

Examples ... 958

Usando paquetes para crear clases con el mismo nombre ... 958

Uso del alcance del paquete protegido ... 958

Capítulo 146: Polimorfismo ... 960

Introducción ... 960

Observaciones ... 960

Examples ... 960

Método de sobrecarga ... 960

Método Anulando ... 962

Agregar comportamiento agregando clases sin tocar el código existente ... 963

Funciones virtuales ... 964

Polimorfismo y diferentes tipos de anulación ... 965

Capítulo 147: Preferencias ... 970

Examples ... 970

Añadiendo oyentes de eventos ... 970

PreferenceChangeEvent ... 970

NodeChangeEvent ... 970

Obtención de subnodos de preferencias ... 971

Coordinar las preferencias de acceso a través de múltiples instancias de aplicaciones ... 972

Preferencias de exportación ... 972

Importando preferencias ... 973

Eliminar oyentes de eventos ... 974

Obteniendo valores de preferencias ... 975

Configuración de valores de preferencias ... 975

Usando preferencias ... 975

Capítulo 148: Procesamiento de argumentos de línea de comando ... 977

Sintaxis ... 977

Parámetros ... 977

Observaciones....977
Examples....977
Procesamiento de argumentos utilizando GWT ToolBase....977
Procesando argumentos a mano....978
Un comando sin argumentos....978
Un comando con dos argumentos....978
Un comando con opciones de "bandera" y al menos un argumento....979
Capítulo 149: Proceso....980
Observaciones....980
Examples....980
Ejemplo simple (versión de Java <1.5)....980
Usando la clase ProcessBuilder....980
Bloqueo frente a llamadas no bloqueadas....981
ch.vorburger.exec....981
Pitfall: Runtime.exec, Process y ProcessBuilder no entienden la sintaxis de shell....982
Espacios en las rutas....982
Redireccionamiento, tuberías y otra sintaxis de shell....983
Los comandos de shell incorporados no funcionan....983
Capítulo 150: Programación Concurrente (Hilos)....985
Introducción....985
Observaciones....985
Examples....985
Multihilo básico....985
Productor-consumidor....986
Usando ThreadLocal....987
CountDownLatch....988
Sincronización....989
Operaciones atómicas....991
Creación de un sistema básico de punto muerto....992
Pausa de ejecucion....993
Visualizar barreras de lectura / escritura mientras se usa sincronizado / volátil....994
Creando una instancia java.lang.Thread....995

Interrupción de hilo / hilos de parada ... 997
Ejemplo de productor / consumidor múltiple con cola global compartida ... 999
Acceso exclusivo de lectura / lectura simultánea ... 1001
Objeto ejecutable ... 1002
Semáforo ... 1003
Agregue dos arreglos `int` usando un Threadpool ... 1004
Obtenga el estado de todas las hebras iniciadas por su programa, excluyendo las hebras del ... 1004
Callable y Futuro ... 1005
Bloqueos como ayudas de sincronización ... 1007
Capítulo 151: Programación paralela con el framework Fork / Join. ... 1009
Examples ... 1009
Horquilla / Unir tareas en Java ... 1009
Capítulo 152: Publicación por entregas ... 1011
Introducción ... 1011
Examples ... 1011
Serialización básica en Java ... 1011
Serialización con Gson ... 1013
Serialización con Jackson 2 ... 1013
Serialización personalizada ... 1014
Versiones y serialVersionUID ... 1017
Cambios compatibles ... 1017
Cambios incompatibles ... 1018
Deserialización JSON personalizada con Jackson ... 1018
Capítulo 153: Puntos de referencia ... 1021
Introducción ... 1021
Examples ... 1021
Ejemplo simple de JMH ... 1021
Capítulo 154: Recursion ... 1024
Introducción ... 1024
Observaciones ... 1024
Diseñando un Método Recursivo ... 1024
Salida ... 1024

Eliminación de Java y Tail-Call 1024
Examples 1024
La idea básica de la recursión. 1024
Cálculo del Número N. de Fibonacci 1025
Cálculo de la suma de enteros de 1 a N 1026
Cálculo de la enésima potencia de un número 1026
Invertir una cadena usando Recursión 1026
Atravesando una estructura de datos de árbol con recursión 1027
Tipos de recursion 1027
StackOverflowError & recursion to loop 1028
Ejemplo 1028
Solución 1028
Ejemplo 1028
La recursión profunda es problemática en Java 1030
Por qué la eliminación de la llamada de cola no está implementada en Java (todavía) 1031
Capítulo 155: Recursos (en classpath) 1032
Introducción 1032
Observaciones 1032
Examples 1033
Cargando una imagen de un recurso 1033
Cargando la configuración por defecto 1033
Cargando recursos del mismo nombre desde múltiples archivos JAR 1034
Encontrar y leer recursos usando un cargador de clases 1034
Rutas de recursos absolutos y relativos 1034
Obtención de una Clase o Classloader 1034
Los metodos de get 1035
Capítulo 156: Redes 1036
Sintaxis 1036
Examples 1036
Comunicación básica de cliente y servidor mediante un socket 1036
Servidor: Iniciar y esperar las conexiones entrantes. 1036
Servidor: Manejo de clientes. 1036

Cliente: Conectarse al servidor y enviar un mensaje....1036
Cierre de enchufes y manejo de excepciones....1037
Servidor básico y cliente - ejemplos completos....1037
Cargando TrustStore y KeyStore desde InputStream....1038
Ejemplo de socket: leer una página web utilizando un socket simple....1039
Comunicación básica cliente / servidor mediante UDP (datagrama)....1040
Multidifusión....1041
Deshabilite temporalmente la verificación SSL (para propósitos de prueba)....1043
Descargando un archivo usando el canal....1043
Notas....1044
Capítulo 157: Referencias de objetos....1045
Observaciones....1045
Examples....1045
Referencias de objetos como parámetros del método....1045
Capítulo 158: Registro (java.util.logging)....1048
Examples....1048
Usando el registrador predeterminado....1048
Niveles de registro....1048
Registro de mensajes complejos (eficientemente)....1049
Capítulo 159: Seguridad y criptografía....1052
Examples....1052
Calcular los hash criptográficos....1052
Generar datos criptográficamente aleatorios....1052
Generar pares de claves públicas / privadas....1053
Calcular y verificar firmas digitales....1053
Cifrar y descifrar datos con claves públicas / privadas....1054
Capítulo 160: Seguridad y criptografía....1055
Introducción....1055
Observaciones....1055
Examples....1055
La jce....1055
Claves y gestión de claves....1055

Vulnerabilidades comunes de Java ... 1055
Preocupaciones de redes ... 1055
Aleatoriedad y tu ... 1055
Hashing y Validación ... 1056
Capítulo 161: ServiceLoader ... 1057
Observaciones ... 1057
Examples ... 1057
Servicio de registrador ... 1057
Servicio ... 1057
Implementaciones del servicio. ... 1057
META-INF / services / servicetest.Logger ... 1058
Uso ... 1058
Ejemplo simple de ServiceLoader ... 1058
Capítulo 162: Servicio de impresión de Java ... 1061
Introducción ... 1061
Examples ... 1061
Descubriendo los servicios de impresión disponibles. ... 1061
Descubriendo el servicio de impresión predeterminado ... 1061
Creación de un trabajo de impresión desde un servicio de impresión ... 1062
Construyendo el Doc que será impreso. ... 1062
Definiendo atributos de solicitud de impresión ... 1063
Escuchar cambio de estado de solicitud de trabajo de impresión ... 1063
El argumento PrintJobEvent pje ... 1064
Otra forma de lograr el mismo objetivo. ... 1064
Capítulo 163: Singletons ... 1066
Introducción ... 1066
Examples ... 1066
Enum Singleton ... 1066
Hilo seguro Singleton con doble control de bloqueo ... 1066
Singleton sin uso de Enum (inicialización impaciente) ... 1067
Inicialización perezosa segura para subprocesos utilizando la clase de soporte | Implement ... 1067
Extendiendo singleton (herencia singleton) ... 1068

Capítulo 164: Sockets de Java 1071
Introducción 1071
Observaciones 1071
Examples 1071
Un simple servidor TCP con respaldo 1071
Capítulo 165: SortedMap 1075
Introducción 1075
Examples 1075
Introducción al mapa ordenado 1075
Capítulo 166: StringBuffer 1076
Introducción 1076
Examples 1076
Clase de búfer de cadena 1076
Capítulo 167: StringBuilder 1078
Introducción 1078
Sintaxis 1078
Observaciones 1078
Examples 1078
Repetir una cadena n veces 1078
Comparando StringBuffer, StringBuilder, Formatter y StringJoiner 1079
Capítulo 168: sun.misc.Unsafe 1081
Observaciones 1081
Examples 1081
Instalar sun.misc.Unsafe a través de la reflexión 1081
Creación de una instancia de sun.misc.Unsafe a través de bootclasspath 1081
Obtención de instancia de inseguro 1081
Usos de inseguros 1082
Capítulo 169: súper palabra clave 1084
Examples 1084
Uso de palabras clave super con ejemplos 1084
Nivel de constructor 1084

Nivel de método 1085
Nivel variable 1085
Capítulo 170: Tabla de picadillo 1087
Introducción 1087
Examples 1087
Tabla de picadillo 1087
Capítulo 171: ThreadLocal 1088
Observaciones 1088
Examples 1088
ThreadLocal Java 8 inicialización funcional 1088
Uso básico de ThreadLocal 1088
Múltiples hilos con un objeto compartido 1090
Capítulo 172: Tipo de conversión 1092
Sintaxis 1092
Examples 1092
Fundición primitiva no numérica 1092
Fundición primitiva numérica 1092
Fundición de objetos 1093
Promoción Numérica Básica 1093
Probar si un objeto puede ser lanzado usando instanceof 1093
Capítulo 173: Tipos atómicos 1094
Introducción 1094
Parámetros 1094
Observaciones 1094
Examples 1094
Creando Tipos Atómicos 1094
Motivación para los tipos atómicos 1095
¿Cómo se implementan los tipos atómicos? 1096
¿Cómo funcionan los tipos atómicos? 1096
Capítulo 174: Tipos de datos de referencia 1098
Examples 1098

Creando un tipo de referencia 1098
Desreferenciación 1098
Capítulo 175: Tipos de datos primitivos 1099
Introducción 1099
Sintaxis 1099
Observaciones 1099
Examples 1100
El primitivo int 1100
El primitivo corto 1100
El primitivo largo 1101
El primitivo booleano 1102
El byte primitivo 1102
El flotador primitivo 1103
El doble primitivo 1104
El primitivo char 1104
Representación de valor negativo 1105
Consumo de memoria de primitivas vs. primitivas en caja 1106
Cachés de valor en caja 1107
Primitivas de conversión 1107
Tipos primitivos Cheatsheet 1108
Capítulo 176: Tipos de referencia 1110
Examples 1110
Diferentes tipos de referencia 1110
Capítulo 177: Tokenizador de cuerdas 1112
Introducción 1112
Examples 1112
StringTokenizer dividido por el espacio 1112
StringTokenizer Dividir por comas ',' 1112
Capítulo 178: TreeMap y TreeSet 1113
Introducción 1113
Examples 1113
TreeMap de un tipo de Java simple 1113

TreeSet de un tipo de Java simple ... 1113
TreeMap / TreeSet de un tipo de Java personalizado ... 1114
TreeMap y TreeSet Thread Safety ... 1116
Capítulo 179: Usando la palabra clave estática ... 1118
Sintaxis ... 1118
Examples ... 1118
Usando static para declarar constantes ... 1118
Usando estática con esto ... 1118
Referencia a miembro no estático del contexto estático ... 1119
Capítulo 180: Usando otros lenguajes de scripting en Java ... 1121
Introducción ... 1121
Observaciones ... 1121
Examples ... 1121
Evaluando un archivo javascript en modo de script de nashorn ... 1121
Capítulo 181: Usando ThreadPoolExecutor en aplicaciones MultiThreaded. ... 1124
Introducción ... 1124
Examples ... 1124
Realización de tareas asíncronas donde no se necesita un valor de retorno utilizando una i ... 1124
Realización de tareas asíncronas donde se necesita un valor de retorno utilizando una inst ... 1125
Definición de tareas asíncronas en línea usando Lambdas ... 1128
Capítulo 182: Varargs (Argumento Variable) ... 1130
Observaciones ... 1130
Examples ... 1130
Especificando un parámetro varargs ... 1130
Trabajando con los parámetros de Varargs ... 1130
Capítulo 183: Visibilidad (control de acceso a los miembros de una clase) ... 1132
Sintaxis ... 1132
Observaciones ... 1132
Examples ... 1132
Miembros de la interfaz ... 1132
Visibilidad pública ... 1133
Visibilidad privada ... 1133

Visibilidad del paquete 1134
Visibilidad Protegida 1134
Resumen de los Modificadores de Acceso de los Miembros de la Clase 1135
Capítulo 184: XJC 1136
Introducción 1136
Sintaxis 1136
Parámetros 1136
Observaciones 1136
Examples 1136
Generando código Java desde un simple archivo XSD 1136
Esquema XSD (schema.xsd) 1136
Usando xjc 1137
Archivos de resultados 1137
package-info.java 1138
Capítulo 185: XOM - Modelo de objetos XML 1139
Examples 1139
Leyendo un archivo XML 1139
Escribir en un archivo XML 1141
Creditos 1145

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: java-language

It is an unofficial and free Java Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Java Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con el lenguaje Java

Observaciones

El lenguaje de programación de Java es ...

- **Uso general** : está diseñado para ser utilizado para escribir software en una amplia variedad de dominios de aplicación y carece de funciones especializadas para cualquier dominio específico.
- **Basado en clase** : su estructura de objeto se define en clases. Las instancias de clase siempre tienen esos campos y métodos especificados en sus definiciones de clase (ver Clases y Objetos). Esto contrasta con lenguajes no basados en clases como JavaScript.
- **Escritura estática** : el compilador verifica en el momento de la compilación que se respetan los tipos de variables. Por ejemplo, si un método espera un argumento de tipo `String` , ese argumento debe ser de hecho una cadena cuando se llama al método.
- **Orientado a objetos** : la mayoría de las cosas en un programa Java son instancias de clase, es decir, paquetes de estado (campos) y comportamiento (métodos que operan sobre datos y forman la *interfaz* del objeto con el mundo exterior).
- **Portátil** : se puede compilar en cualquier plataforma con `javac` y los archivos de clase resultantes se pueden ejecutar en cualquier plataforma que tenga una JVM.

El objetivo de Java es permitir que los desarrolladores de aplicaciones "escriban una vez, se ejecuten en cualquier lugar" (WORA), lo que significa que el código compilado de Java puede ejecutarse en todas las plataformas que admiten Java sin la necesidad de una recompilación.

El código Java se compila a un código de bytes (los archivos `.class`) que, a su vez, son interpretados por la Máquina Virtual de Java (JVM). En teoría, el código de bytes creado por un compilador de Java debería ejecutarse de la misma manera en cualquier JVM, incluso en un tipo diferente de computadora. La JVM podría (y en los programas del mundo real) elegir compilar en comandos de máquina nativos las partes del código de bytes que se ejecutan a menudo. Esto se denomina "compilación Just-in-time (JIT)".

Ediciones y versiones de Java

Hay tres "ediciones" de Java definidas por Sun / Oracle:

- *Java Standard Edition (SE)* es la edición que está diseñada para uso general.
- *Java Enterprise Edition (EE)* agrega una gama de facilidades para crear servicios de "grado empresarial" en Java. Java EE se cubre por separado .
- *Java Micro Edition (ME)* se basa en un subconjunto de *Java SE* y está diseñado para su uso en dispositivos pequeños con recursos limitados.

Hay un tema separado en las ediciones de Java SE / EE / ME .

Cada edición tiene múltiples versiones. Las versiones de Java SE se enumeran a continuación.

Instalando Java

Hay un tema separado sobre la instalación de Java (edición estándar) .

Compilación y ejecución de programas Java.

Hay temas separados sobre:

- Compilando el código fuente de Java
- Despliegue de Java incluyendo la creación de archivos JAR
- Ejecutando aplicaciones Java
- El classpath

¿Que sigue?

Aquí hay enlaces a temas para continuar aprendiendo y entendiendo el lenguaje de programación Java. Estos temas son los conceptos básicos de la programación de Java para comenzar.

- Tipos de datos primitivos en Java
- Operadores en Java
- Cuerdas en java
- Estructuras de control básicas en Java
- Clases y objetos en Java
- Arrays en Java
- Estándares de código de Java

Pruebas

Si bien Java no tiene soporte para realizar pruebas en la biblioteca estándar, existen bibliotecas de terceros que están diseñadas para admitir pruebas. Las dos bibliotecas de pruebas de unidades más populares son:

- JUnit (Sitio Oficial)
- TestNG (Sitio Oficial)

Otro

- Los patrones de diseño para Java están cubiertos en los patrones de diseño .

- La programación para Android está cubierta en Android .
- Las tecnologías Java Enterprise Edition están cubiertas en Java EE .
- Las tecnologías Oracle JavaFX están cubiertas en JavaFX .

1. En la sección **Versiones** , la fecha de *finalización de la vida útil (gratuita)* es cuando Oracle dejará de publicar nuevas actualizaciones de Java SE en sus sitios públicos de descarga. Los clientes que necesitan acceso continuo a correcciones de errores críticos y correcciones de seguridad, así como el mantenimiento general de Java SE pueden obtener soporte a largo plazo a través del soporte de Oracle Java SE .

Versiones

Versión de Java SE	Nombre clave	Fin de la vida (libre [1])	Fecha de lanzamiento
Java SE 9 (acceso temprano)	*Ninguna*	futuro	2017-07-27
Java SE 8	Araña	futuro	2014-03-18
Java SE 7	Delfín	2015-04-14	2011-07-28
Java SE 6	Mustango	2013-04-16	2006-12-23
Java SE 5	Tigre	2009-11-04	2004-10-04
Java SE 1.4	Esmerejón	antes del 2009-11-04	2002-02-06
Java SE 1.3	Cernícalo	antes del 2009-11-04	2000-05-08
Java SE 1.2	Patio de recreo	antes del 2009-11-04	1998-12-08
Java SE 1.1	*Ninguna*	antes del 2009-11-04	1997-02-19
Java SE 1.0	Roble	antes del 2009-11-04	1996-01-21

Examples

Creando tu primer programa Java

Cree un nuevo archivo en su editor de texto o IDE llamado `HelloWorld.java` . Luego pegue este bloque de código en el archivo y guarde:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Corre en vivo en Ideone

Nota: Para que Java reconozca esto como una `public class` (y no arroje un error de tiempo de compilación), el nombre del archivo debe ser el mismo que el nombre de la clase (`HelloWorld` en este ejemplo) con una extensión `.java` . También debe haber un modificador de acceso `public` antes de él.

Las convenciones de nomenclatura recomiendan que las clases de Java comiencen con un carácter en mayúscula y estén en formato de caja de camello (en el que la primera letra de cada palabra se escribe con mayúscula). Las convenciones recomiendan contra guiones bajos (_) y signos de dólar (`$`).

Para compilar, abra una ventana de terminal y navegue al directorio de `HelloWorld.java` :

```
cd /path/to/containing/folder/
```

Nota: cd es el comando terminal para cambiar el directorio.

Ingrese `javac` seguido del nombre del archivo y la extensión de la siguiente manera:

```
$ javac HelloWorld.java
```

Es bastante común que el error `'javac' is not recognized as an internal or external command, operable program or batch file.` incluso cuando haya instalado el `JDK` y pueda ejecutar el programa desde `IDE` ej. `eclipse` etc. Dado que la ruta no se agrega al entorno de forma predeterminada.

En caso de que obtenga esto en Windows, para resolverlo, primero intente `javac.exe` su ruta `javac.exe` , es muy probable que esté en su `C:\Program Files\Java\jdk(version number)\bin` . A continuación, intente ejecutarlo con a continuación.

```
$ C:\Program Files\Java\jdk(version number)\bin\javac HelloWorld.java
```

Anteriormente, cuando llamábamos a `javac` , era igual que el comando anterior. Solo en ese caso, su `OS` sabía dónde residía `javac` . Así que vamos a decirlo ahora, de esta manera no tienes que escribir todo el camino cada vez. Necesitaríamos agregar esto a nuestro `PATH`

Para editar la `PATH` entorno `PATH` en Windows XP / Vista / 7/8/10:

- Panel de control ⇒ Sistema ⇒ Configuración avanzada del sistema
- Cambie a la pestaña "Avanzado" ⇒ Variables de entorno
- En "Variables del sistema", desplácese hacia abajo para seleccionar "RUTA" ⇒ Editar

No puedes deshacer esto así que ten cuidado. Primero copia tu ruta existente al bloc de notas. Luego, para obtener la ruta de acceso exacta a su `javac` busque manualmente la carpeta donde reside `javac` haga clic en la barra de direcciones y luego cópiela. Debería verse como `c:\Program Files\Java\jdk1.8.0_xx\bin`

En el campo "Valor variable", pegue este **EN FRENTE** de todos los directorios existentes, seguido

de un punto y coma (;). **NO BORRAR** ninguna entrada existente.

```
Variable name  : PATH
Variable value : c:\Program Files\Java\jdk1.8.0_xx\bin;[Existing Entries...]
```

Ahora esto debería resolverse.

Para sistemas basados en Linux intente aquí .

Nota: *`javac` comando `javac` invoca el compilador de Java.*

El compilador generará entonces un código de bytes archivo llamado `HelloWorld.class` que puede ser ejecutado en la máquina virtual de Java (JVM) . El compilador del lenguaje de programación Java, `javac` , lee los archivos de origen escritos en el lenguaje de programación Java y los compila en archivos de clase de código de `bytecode` . Opcionalmente, el compilador también puede procesar las anotaciones encontradas en los archivos de origen y de clase utilizando la API de Procesamiento de Anotación `Pluggable` . El compilador es una herramienta de línea de comandos, pero también se puede invocar utilizando la API del compilador de Java.

Para ejecutar su programa, ingrese `java` seguido del nombre de la clase que contiene el método `main` (`HelloWorld` en nuestro ejemplo). Observe cómo se omite `.class` :

```
$ java HelloWorld
```

Nota: *el comando `java` ejecuta una aplicación Java.*

Esto saldrá a su consola:

> ¡Hola Mundo!

¡Has codificado y construido exitosamente tu primer programa Java!

Nota: Para que los comandos de Java (`java` , `javac` , etc.) sean reconocidos, deberá asegurarse de que:

- Se instala un JDK (por ejemplo, Oracle , OpenJDK y otras fuentes)
- Sus variables de entorno están configuradas correctamente

Deberá usar un compilador (`javac`) y un ejecutor (`java`) proporcionado por su JVM. Para saber qué versiones tiene instaladas, ingrese `java -version` y `javac -version` en la línea de comandos. El número de versión de su programa se imprimirá en el terminal (por ejemplo, `1.8.0_73`).

Una mirada más cercana al programa Hello World.

El programa "Hello World" contiene un solo archivo, que consiste en una definición de clase

`HelloWorld` , un método `main` y una declaración dentro del método `main` .

```
public class HelloWorld {
```

La palabra clave de `class` comienza la definición de clase para una clase llamada `HelloWorld` . Cada aplicación Java contiene al menos una definición de clase (Más información sobre clases).

```
    public static void main(String[] args)  {
```

Este es un método de punto de entrada (definido por su nombre y firma de `public static void main(String[])`) desde el cual `JVM` puede ejecutar su programa. Cada programa de Java debería tener uno. Es:

- `public` : lo que significa que el método también se puede llamar desde cualquier lugar desde fuera del programa. Ver Visibilidad para más información sobre esto.
- `static` : significa que existe y se puede ejecutar por sí mismo (a nivel de clase sin crear un objeto).
- `void` : significa que no devuelve ningún valor. ***Nota:*** *Esto es diferente a C y C ++ donde se espera un código de retorno como* `int` *(la forma de Java es* `System.exit()` *).*

Este método principal acepta:

- Una matriz (normalmente llamada `args`) de `String` s pasa como argumentos a la función principal (por ejemplo, desde los argumentos de la línea de comando)

Casi todo esto es necesario para un método de punto de entrada de Java.

Piezas no requeridas:

- El nombre `args` es un nombre de variable, por lo que puede llamarse como quieras, aunque normalmente se llama `args` .
- Si su tipo de parámetro es una matriz (`String[] args`) o Varargs (`String... args`) no importa porque las matrices se pueden pasar a varargs.

Nota: una sola aplicación puede tener varias clases que contengan un método de punto de entrada (`main`). El punto de entrada de la aplicación está determinado por el nombre de clase pasado como un argumento al comando `java` .

Dentro del método principal, vemos la siguiente declaración:

```
        System.out.println("Hello, World!");
```

Vamos a desglosar esta declaración elemento por elemento:

Elemento	Propósito
`System`	esto denota que la siguiente expresión llamará a la clase `System` , desde el paquete `java.lang` .

Elemento	Propósito
`.`	este es un "operador de puntos". Los operadores de puntos le brindan acceso a una clase de miembros [1] ; Es decir, sus campos (variables) y sus métodos. En este caso, este operador de punto le permite hacer referencia al campo estático de `out` dentro de la clase `System` .
`out`	este es el nombre del campo estático del tipo `PrintStream` dentro de la clase `System` contiene la funcionalidad de salida estándar.
`.`	este es otro operador de puntos. Este operador de puntos proporciona acceso al método `println` dentro de la variable `out` .
`println`	este es el nombre de un método dentro de la clase PrintStream. Este método, en particular, imprime el contenido de los parámetros en la consola e inserta una nueva línea después.
`(`	este paréntesis indica que se está accediendo a un método (y no a un campo) y comienza a pasar los parámetros al método `println` .
`"Hello, World!"`	este es el literal de cadena que se pasa como parámetro al método `println` . Las comillas dobles en cada extremo delimitan el texto como una cadena.
`)`	este paréntesis significa el cierre de los parámetros que se pasan al método `println` .
`;`	este punto y coma marca el final de la declaración.

Nota: *Cada declaración en Java debe terminar con un punto y coma (`;`).*

El cuerpo del método y el cuerpo de la clase se cierran.

```
    }  // end of main function scope
}      // end of class HelloWorld scope
```

Aquí hay otro ejemplo que demuestra el paradigma OO. Vamos a modelar un equipo de fútbol con un miembro (¡sí, uno!). Puede haber más, pero lo discutiremos cuando lleguemos a los arreglos.

Primero, definamos nuestra clase de `Team` :

```
public class Team {
    Member member;
    public Team(Member member) {  // who is in this Team?
        this.member = member;  //  one 'member' is in this Team!
    }
}
```

Ahora, definamos nuestra clase de `Member` :

```
class Member {
    private String name;
    private String type;
    private int level; // note the data type here
    private int rank; // note the data type here as well

    public Member(String name, String type, int level, int rank) {
        this.name = name;
        this.type = type;
        this.level = level;
        this.rank = rank;
    }
}
```

¿Por qué usamos `private` aquí? Bueno, si alguien desea saber su nombre, debe preguntarle directamente, en lugar de buscar en su bolsillo y sacar su tarjeta de Seguro Social. Este `private` hace algo así: impide que las entidades externas accedan a sus variables. Solo puede devolver miembros `private` través de las funciones de obtención (que se muestran a continuación).

Después de ponerlo todo junto, y de agregar los métodos de obtención y el método principal, como se mencionó anteriormente, tenemos:

```
public class Team {
    Member member;
    public Team(Member member) {
        this.member = member;
    }

    // here's our main method
    public static void main(String[] args) {
       Member myMember = new Member("Aurieel", "light", 10, 1);
       Team myTeam = new Team(myMember);
       System.out.println(myTeam.member.getName());
       System.out.println(myTeam.member.getType());
       System.out.println(myTeam.member.getLevel());
       System.out.println(myTeam.member.getRank());
    }
}

class Member {
    private String name;
    private String type;
    private int level;
    private int rank;

    public Member(String name, String type, int level, int rank) {
        this.name = name;
        this.type = type;
        this.level = level;
        this.rank = rank;
    }

    /* let's define our getter functions here */
    public String getName() { // what is your name?
        return this.name; // my name is ...
    }

    public String getType() { // what is your type?
```

```
        return this.type; // my type is ...
    }

    public int getLevel() { // what is your level?
        return this.level; // my level is ...
    }

    public int getRank() { // what is your rank?
        return this.rank; // my rank is
    }
}
```

Salida:

```
Aurieel
light
10
1
```

Corre en ideone

Una vez más, el método `main` dentro de la clase de `Test` es el punto de entrada a nuestro programa. Sin el método `main` , no podemos decirle a la Máquina Virtual Java (JVM) desde dónde comenzar la ejecución del programa.

---

1 - Debido a que la clase `HelloWorld` tiene poca relación con la clase `System` , solo puede acceder a datos `public` .

Lea Empezando con el lenguaje Java en línea: https://riptutorial.com/es/java/topic/84/empezando-con-el-lenguaje-java

Capítulo 2: Acceso nativo de Java

Examples

Introducción a JNA

¿Qué es JNA?

Java Native Access (JNA) es una biblioteca desarrollada por la comunidad que proporciona a los programas Java un acceso fácil a las bibliotecas compartidas nativas (archivos `.dll` en Windows, archivos `.so` en Unix ...)

¿Como puedo usar lo?

- En primer lugar, descargue la última versión de JNA y haga referencia a su jna.jar en CLASSPATH de su proyecto.
- En segundo lugar, copie, compile y ejecute el siguiente código Java

  Para el propósito de esta introducción, suponemos que la plataforma nativa en uso es Windows. Si está ejecutando en otra plataforma, simplemente reemplace la cadena `"msvcrt"` con la cadena `"c"` en el código a continuación.

El pequeño programa Java a continuación imprimirá un mensaje en la consola llamando a la `printf` C `printf`.

CRuntimeLibrary.java

```
package jna.introduction;

import com.sun.jna.Library;
import com.sun.jna.Native;

// We declare the printf function we need and the library containing it (msvcrt)...
public interface CRuntimeLibrary extends Library {

   CRuntimeLibrary INSTANCE =
       (CRuntimeLibrary) Native.loadLibrary("msvcrt", CRuntimeLibrary.class);

   void printf(String format, Object... args);
}
```

MyFirstJNAProgram.java

```
package jna.introduction;

// Now we call the printf function...
```

```
public class MyFirstJNAProgram {
    public static void main(String args[]) {
         CRuntimeLibrary.INSTANCE.printf("Hello World from JNA !");
    }
}
```

¿A dónde ir ahora?

Salta a otro tema aquí o salta al sitio oficial .

Lea Acceso nativo de Java en línea: https://riptutorial.com/es/java/topic/5244/acceso-nativo-de-java

Capítulo 3: Afirmando

Sintaxis

- afirmar *expresión1* ;
- afirmar *expresión1* : *expresión2* ;

Parámetros

Parámetro	Detalles
expresión1	La declaración de aserción lanza un `AssertionError` si esta expresión se evalúa como `false` .
expresión2	Opcional. Cuando se utiliza, `AssertionError` s arrojado por la declaración de aserción tiene este mensaje.

Observaciones

Por defecto, las aserciones están deshabilitadas en tiempo de ejecución.

Para habilitar aserciones, debe ejecutar java con el indicador `-ea` .

```
java -ea com.example.AssertionExample
```

Las aserciones son declaraciones que arrojarán un error si su expresión se evalúa como `false` . Las afirmaciones solo deben usarse para *probar el* código; nunca deben ser utilizados en la producción.

Examples

Comprobando aritmética con aseverar.

```
a = 1 - Math.abs(1 - a % 2);

// This will throw an error if my arithmetic above is wrong.
assert a >= 0 && a <= 1 : "Calculated value of " + a + " is outside of expected bounds";

return a;
```

Lea Afirmando en línea: https://riptutorial.com/es/java/topic/407/afirmando

Capítulo 4: Agentes de Java

Examples

Modificando clases con agentes.

En primer lugar, asegúrese de que el agente que se está utilizando tenga los siguientes atributos en el archivo Manifest.mf:

```
Can-Redefine-Classes: true
Can-Retransform-Classes: true
```

Iniciar un agente java permitirá que el agente acceda a la clase de Instrumentación. Con Instrumentation puede llamar a *addTransformer (transformador ClassFileTransformer)* . ClassFileTransformers te permitirá reescribir los bytes de las clases. La clase tiene un solo método que suministra el ClassLoader que carga la clase, el nombre de la clase, una instancia de java.lang.Class, es ProtectionDomain y, por último, los bytes de la propia clase.

Se parece a esto:

```
byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,
          ProtectionDomain protectionDomain, byte[] classfileBuffer)
```

Modificar una clase puramente de bytes puede llevar siglos. Para remediar esto, hay bibliotecas que se pueden usar para convertir los bytes de clase en algo más utilizable.

En este ejemplo, usaré ASM, pero otras alternativas como Javassist y BCEL tienen características similares.

```
ClassNode getNode(byte[] bytes) {
    // Create a ClassReader that will parse the byte array into a ClassNode
    ClassReader cr = new ClassReader(bytes);
    ClassNode cn = new ClassNode();
    try {
        // This populates the ClassNode
        cr.accept(cn, ClassReader.EXPAND_FRAMES);
        cr = null;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return cn;
}
```

Desde aquí se pueden hacer cambios al objeto ClassNode. Esto hace que cambiar el acceso al campo / método sea increíblemente fácil. Además, con la API del árbol de ASM, la modificación del código de bytes de los métodos es muy sencilla.

Una vez que las ediciones hayan finalizado, puede volver a convertir el ClassNode en bytes con el siguiente método y devolverlos en el método de *transformación* :

```
public static byte[] getNodeBytes(ClassNode cn, boolean useMaxs) {
    ClassWriter cw = new ClassWriter(useMaxs ? ClassWriter.COMPUTE_MAXS :
ClassWriter.COMPUTE_FRAMES);
    cn.accept(cw);
    byte[] b = cw.toByteArray();
    return b;
}
```

Agregar un agente en tiempo de ejecución

Los agentes se pueden agregar a una JVM en tiempo de ejecución. Para cargar un agente, deberá usar *VirtualMachine.attatch (Id* . De *cadena) de* Attach API. A continuación, puede cargar un jar de agente compilado con el siguiente método:

```
public static void loadAgent(String agentPath) {
    String vmName = ManagementFactory.getRuntimeMXBean().getName();
    int index = vmName.indexOf('@');
    String pid = vmName.substring(0, index);
    try {
        File agentFile = new File(agentPath);
        VirtualMachine vm = VirtualMachine.attach(pid);
        vm.loadAgent(agentFile.getAbsolutePath(), "");
        VirtualMachine.attach(vm.id());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Esto no llamará *premain ((String agentArgs, Instrumentation inst)* en el agente cargado, sino que llamará a *agentmain (String agentArgs, Instrumentation inst)* . Esto requiere que *Agent-Class* esté configurado en el agente Manifest.mf.

Configuración de un agente básico

La clase Premain contendrá el método *"premain (String agentArgs Instrumentation inst)"*

Aquí hay un ejemplo:

```
import java.lang.instrument.Instrumentation;

public class PremainExample {
    public static void premain(String agentArgs, Instrumentation inst) {
        System.out.println(agentArgs);
    }
}
```

Cuando esté compilado en un archivo jar, abra el Manifiesto y asegúrese de que tenga el atributo de clase principal.

Aquí hay un ejemplo:

```
Premain-Class: PremainExample
```

Para usar el agente con otro programa java "myProgram", debe definir el agente en los argumentos de JVM:

```
java -javaagent:PremainAgent.jar -jar myProgram.jar
```

Lea Agentes de Java en línea: https://riptutorial.com/es/java/topic/1265/agentes-de-java

Capítulo 5: Ajuste de rendimiento de Java

Examples

Enfoque general

Internet está repleto de consejos para mejorar el rendimiento de los programas Java. Quizás el consejo número uno sea la conciencia. Eso significa:

- Identificar posibles problemas de rendimiento y cuellos de botella.
- Utilizar herramientas de análisis y pruebas.
- Conocer buenas prácticas y malas prácticas.

El primer punto se debe hacer durante la etapa de diseño si se habla de un nuevo sistema o módulo. Si se habla de código heredado, las herramientas de análisis y prueba entran en escena. La herramienta más básica para analizar el rendimiento de su JVM es JVisualVM, que se incluye en el JDK.

El tercer punto es principalmente sobre la experiencia y la investigación extensa, y por supuesto, consejos crudos que se mostrarán en esta página y otras, como esta .

Reduciendo cantidad de cuerdas

En Java, es demasiado "fácil" crear muchas instancias de String que no son necesarias. Esa y otras razones pueden hacer que su programa tenga muchas cadenas que el GC está ocupado limpiando.

Algunas formas en las que podrías estar creando instancias de String:

```
myString += "foo";
```

O peor aún, en un bucle o recursión:

```
for (int i = 0; i < N; i++) {
    myString += "foo" + i;
}
```

El problema es que cada `+` crea una nueva Cadena (generalmente, ya que los nuevos compiladores optimizan algunos casos). Se puede hacer una optimización posible utilizando `StringBuilder` o `StringBuffer` :

```
StringBuffer sb = new StringBuffer(myString);
for (int i = 0; i < N; i++) {
    sb.append("foo").append(i);
}
myString = sb.toString();
```

Si crea cadenas largas con frecuencia (por ejemplo, SQL), use una API de construcción de cadenas.

Otras cosas a considerar:

- Reducir el uso de `replace` , `substring` , etc.
- Evite `String.toArray()` , especialmente en el código de acceso frecuente.
- Las impresiones de registro que están destinadas a ser filtradas (debido al nivel de registro, por ejemplo) no deben generarse (el nivel de registro debe verificarse de antemano).
- Utilice bibliotecas como esta si es necesario.
- StringBuilder es mejor si la variable se utiliza de manera no compartida (a través de hilos).

Un enfoque basado en la evidencia para el ajuste de rendimiento de Java

Donald Knuth a menudo es citado diciendo esto:

> "Los programadores pierden enormes cantidades de tiempo pensando o preocupándose por la velocidad de las partes no críticas de sus programas, y estos intentos de eficiencia en realidad tienen un fuerte impacto negativo cuando se consideran la depuración y el mantenimiento. *Debemos olvidar las pequeñas eficiencias, por ejemplo 97% del tiempo* : la optimización prematura es la raíz de todo mal. Sin embargo, no debemos dejar pasar nuestras oportunidades en ese 3% crítico ".

fuente

Teniendo en cuenta este sabio consejo, este es el procedimiento recomendado para optimizar programas:

1. En primer lugar, diseñe y codifique su programa o biblioteca con un enfoque en la simplicidad y la corrección. Para empezar, no dediques mucho esfuerzo al rendimiento.

2. Llegue a un estado de trabajo y (idealmente) desarrolle pruebas unitarias para las partes clave del código base.

3. Desarrollar una prueba de rendimiento de nivel de aplicación. El punto de referencia debe cubrir los aspectos críticos de rendimiento de su aplicación y debe realizar una serie de tareas que son típicas de cómo se usará la aplicación en producción.

4. Medir el rendimiento.

5. Compare el rendimiento medido con sus criterios para determinar qué tan rápido debe ser la aplicación. (Evite criterios poco realistas, inalcanzables o no cuantificables como "lo más rápido posible".)

6. Si ha cumplido con los criterios, PARE. Tu trabajo está hecho. (Cualquier esfuerzo adicional es probablemente una pérdida de tiempo.)

7. Perfile la aplicación mientras ejecuta su prueba de rendimiento.

8. Examine los resultados de la creación de perfiles y elija los "hotspots de rendimiento" más grandes (no optimizados); Es decir, las secciones del código donde la aplicación parece estar gastando más tiempo.

9. Analice la sección del código del hotspot para tratar de entender por qué es un cuello de botella y piense en una manera de hacerlo más rápido.

10. Implementar eso como un cambio de código propuesto, probar y depurar.

11. Vuelva a ejecutar el punto de referencia para ver si el cambio de código ha mejorado el rendimiento:

    - En caso afirmativo, vuelva al paso 4.
    - Si No, abandone el cambio y vuelva al paso 9. Si no está progresando, elija un punto de acceso diferente para su atención.

Eventualmente, llegará a un punto en el que la aplicación es lo suficientemente rápida o ha considerado todos los hotspots importantes. En este punto es necesario detener este enfoque. Si una sección del código consume (por ejemplo) el 1% del tiempo total, incluso una mejora del 50% solo hará que la aplicación sea un 0,5% más rápida en general.

Claramente, hay un punto más allá del cual la optimización del punto de acceso es una pérdida de esfuerzo. Si llegas a ese punto, debes adoptar un enfoque más radical. Por ejemplo:

- Mira la complejidad algorítmica de tus algoritmos centrales.
- Si la aplicación pasa mucho tiempo recolectando basura, busque formas de reducir la tasa de creación de objetos.
- Si las partes clave de la aplicación son CPU intensivas y de un solo hilo, busque oportunidades para el paralelismo.
- Si la aplicación ya tiene varios subprocesos, busque cuellos de botella de concurrencia.

Pero siempre que sea posible, confíe en las herramientas y la medición en lugar del instinto para dirigir su esfuerzo de optimización.

Lea Ajuste de rendimiento de Java en línea: https://riptutorial.com/es/java/topic/4160/ajuste-de-rendimiento-de-java

Capítulo 6: Análisis XML utilizando las API de JAXP

Observaciones

El análisis XML es la interpretación de documentos XML para manipular su contenido mediante construcciones sensibles, ya sean "nodos", "atributos", "documentos", "espacios de nombres" o eventos relacionados con estas construcciones.

Java tiene una API nativa para el manejo de documentos XML, llamada JAXP, o API de Java para el procesamiento XML . JAXP y una implementación de referencia se han incluido en todas las versiones de Java desde Java 1.4 (JAXP v1.1) y han evolucionado desde entonces. Java 8 enviado con la versión 1.6 de JAXP.

La API proporciona diferentes formas de interactuar con documentos XML, que son:

- La interfaz DOM (Modelo de Objeto de Documento)
- La interfaz SAX (API simple para XML)
- La interfaz StAX (Streaming API para XML)

Principios de la interfaz DOM

La interfaz DOM apunta a proporcionar una forma compatible con W3C DOM de interpretar XML. Varias versiones de JAXP han admitido varios niveles de especificación de DOM (hasta el nivel 3).

Bajo la interfaz del Modelo de objetos de documento, un documento XML se representa como un árbol, comenzando con el "Elemento del documento". El tipo base de la API es el tipo de `Node` , permite navegar de un `Node` a su padre, sus hijos o sus hermanos (aunque, no todos los `Node` pueden tener hijos, por ejemplo, los nodos de `Text` son finales en el árbol, y nunca tener hijos). Las etiquetas XML se representan como `Element` s, que amplían notablemente el `Node` con métodos relacionados con atributos.

La interfaz DOM es muy útil ya que permite un análisis de una línea de documentos XML como árboles y permite una modificación fácil del árbol construido (adición de nodo, supresión, copia, ...) y, finalmente, su serialización (de nuevo al disco).) modificaciones posteriores. Sin embargo, esto tiene un precio: el árbol reside en la memoria, por lo tanto, los árboles DOM no siempre son prácticos para grandes documentos XML. Además, la construcción del árbol no siempre es la forma más rápida de tratar con el contenido XML, especialmente si uno no está interesado en todas las partes del documento XML.

Principios de la interfaz SAX

La API de SAX es una API orientada a eventos para tratar con documentos XML. Bajo este modelo, los componentes de un documento XML se interpretan como eventos (por ejemplo, "se ha abierto una etiqueta", "se ha cerrado una etiqueta", "se ha encontrado un nodo de texto", "se ha encontrado un comentario"). ..

La API de SAX utiliza un enfoque de "análisis de inserción", donde un `Parser` SAX es responsable de interpretar el documento XML e invoca métodos en un delegado (un `ContentHandler`) para tratar cualquier evento que se encuentre en el documento XML. Por lo general, uno nunca escribe un analizador, pero proporciona un controlador para recopilar todas las informaciones necesarias del documento XML.

La interfaz SAX supera las limitaciones de la interfaz DOM manteniendo solo los datos mínimos necesarios en el nivel del analizador (por ejemplo, contextos de espacios de nombres, estado de validación), por lo tanto, solo las informaciones que guarda `ContentHandler` , de las que usted, el desarrollador, es responsable, son guardado en la memoria. La desventaja es que no hay manera de "retroceder en el tiempo / el documento XML" con este enfoque: mientras que DOM permite que un `Node` regrese a su padre, no existe tal posibilidad en SAX.

Principios de la interfaz StAX

La API StAX adopta un enfoque similar para procesar XML como la API SAX (es decir, impulsada por eventos), la única diferencia muy significativa es que StAX es un analizador de extracción (donde SAX era un analizador de inserción). En SAX, el `Parser` está en control y utiliza devoluciones de llamada en el `ContentHandler` . En Stax, usted llama al analizador y controla cuándo / si desea obtener el siguiente "evento" XML.

La API comienza con XMLStreamReader (o XMLEventReader), que son las puertas de acceso a través de las cuales el desarrollador puede preguntar a `nextEvent()` , de forma `nextEvent()` .

Examples

Analizar y navegar un documento utilizando la API DOM

Teniendo en cuenta el siguiente documento:

```
<?xml version='1.0' encoding='UTF-8' ?>
<library>
   <book id='1'>Effective Java</book>
   <book id='2'>Java Concurrency In Practice</book>
</library>
```

Uno puede usar el siguiente código para construir un árbol DOM a partir de una `String` :

```
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;
```

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.StringReader;

public class DOMDemo {

public static void main(String[] args) throws Exception {
    String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>"
            + "<library>"
            + "<book id='1'>Effective Java</book>"
            + "<book id='2'>Java Concurrency In Practice</book>"
            + "</library>";

    DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();
    // This is useless here, because the XML does not have namespaces, but this option is
usefull to know in cas
    documentBuilderFactory.setNamespaceAware(true);
    DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder();
    // There are various options here, to read from an InputStream, from a file, ...
    Document document = documentBuilder.parse(new InputSource(new StringReader(xmlDocument)));

    // Root of the document
    System.out.println("Root of the XML Document: " +
document.getDocumentElement().getLocalName());

    // Iterate the contents
    NodeList firstLevelChildren = document.getDocumentElement().getChildNodes();
    for (int i = 0; i < firstLevelChildren.getLength(); i++) {
        Node item = firstLevelChildren.item(i);
        System.out.println("First level child found, XML tag name is: " +
item.getLocalName());
        System.out.println("\tid attribute of this tag is : " +
item.getAttributes().getNamedItem("id").getTextContent());
    }

    // Another way would have been
    NodeList allBooks = document.getDocumentElement().getElementsByTagName("book");
}
}
```

El código produce lo siguiente:

```
Root of the XML Document: library
First level child found, XML tag name is: book
id attribute of this tag is : 1
First level child found, XML tag name is: book
id attribute of this tag is : 2
```

Analizar un documento utilizando la API StAX

Teniendo en cuenta el siguiente documento:

```
<?xml version='1.0' encoding='UTF-8' ?>
<library>
   <book id='1'>Effective Java</book>
   <book id='2'>Java Concurrency In Practice</book>
   <notABook id='3'>This is not a book element</notABook>
```

```
</library>
```

Uno puede usar el siguiente código para analizarlo y construir un mapa de títulos de libros por ID de libro.

```
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamReader;
import java.io.StringReader;
import java.util.HashMap;
import java.util.Map;

public class StaxDemo {

public static void main(String[] args) throws Exception {
    String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>"
            + "<library>"
                + "<book id='1'>Effective Java</book>"
                + "<book id='2'>Java Concurrency In Practice</book>"
                + "<notABook id='3'>This is not a book element </notABook>"
            + "</library>";

    XMLInputFactory xmlInputFactory = XMLInputFactory.newFactory();
    // Various flavors are possible, e.g. from an InputStream, a Source, ...
    XMLStreamReader xmlStreamReader = xmlInputFactory.createXMLStreamReader(new
StringReader(xmlDocument));

    Map<Integer, String> bookTitlesById = new HashMap<>();

    // We go through each event using a loop
    while (xmlStreamReader.hasNext()) {
        switch (xmlStreamReader.getEventType()) {
            case XMLStreamConstants.START_ELEMENT:
                System.out.println("Found start of element: " +
xmlStreamReader.getLocalName());
                // Check if we are at the start of a <book> element
                if ("book".equals(xmlStreamReader.getLocalName())) {
                    int bookId = Integer.parseInt(xmlStreamReader.getAttributeValue("",
"id"));
                    String bookTitle = xmlStreamReader.getElementText();
                    bookTitlesById.put(bookId, bookTitle);
                }
                break;
            // A bunch of other things are possible : comments, processing instructions,
Whitespace...
            default:
                break;
        }
        xmlStreamReader.next();
    }

    System.out.println(bookTitlesById);
}
```

Esto produce:

```
Found start of element: library
Found start of element: book
```

```
Found start of element: book
Found start of element: notABook
{1=Effective Java, 2=Java Concurrency In Practice}
```

En esta muestra, uno debe tener cuidado con algunas cosas:

1. El uso de `xmlStreamReader.getAttributeValue` funciona porque hemos verificado primero que el analizador está en el estado `START_ELEMENT` . En todos los demás estados (excepto `ATTRIBUTES`), el analizador está obligado a lanzar la `IllegalStateException` , porque los atributos solo pueden aparecer al principio de los elementos.

2. Lo mismo ocurre con `xmlStreamReader.getTextContent()` , funciona porque estamos en un `START_ELEMENT` y sabemos en este documento que el elemento `<book>` no tiene nodos secundarios que no sean de texto.

Para el análisis de documentos más complejos (elementos más profundos, anidados, ...), es una buena práctica "delegar" el analizador a sub-métodos u otros objetos, por ejemplo, tener una clase o método `BookParser` , y hacer que se ocupe de cada elemento de START_ELEMENT a END_ELEMENT de la etiqueta XML del libro.

También se puede usar un objeto de `Stack` para mantener datos importantes arriba y abajo del árbol.

Lea Análisis XML utilizando las API de JAXP en línea:
https://riptutorial.com/es/java/topic/3943/analisis-xml-utilizando-las-api-de-jaxp

Capítulo 7: Anotaciones

Introducción

En Java, una anotación es una forma de metadatos sintácticos que se pueden agregar al código fuente de Java. Proporciona datos sobre un programa que no forma parte del programa en sí. Las anotaciones no tienen ningún efecto directo en el funcionamiento del código que anotan. Clases, métodos, variables, parámetros y paquetes pueden ser anotados.

Sintaxis

- @AnnotationName // 'Anotación de marcador' (sin parámetros)
- @AnnotationName (someValue) // establece el parámetro con el nombre 'valor'
- @AnnotationName (param1 = value1) // parámetro con nombre
- @AnnotationName (param1 = value1, param2 = value2) // múltiples parámetros nombrados
- @AnnotationName (param1 = {1, 2, 3}) // parámetro de matriz con nombre
- @AnnotationName ({value1}) // array con un solo elemento como parámetro con el nombre 'value'

Observaciones

Tipos de parametros

Solo se permiten expresiones constantes de los siguientes tipos para los parámetros, así como matrices de estos tipos:

- `String`
- `Class`
- tipos primitivos
- Tipos de enumeración
- Tipos de anotaciones

Examples

Anotaciones incorporadas

La edición estándar de Java viene con algunas anotaciones predefinidas. No es necesario que los defina por sí mismo y puede usarlos inmediatamente. Permiten al compilador habilitar algunas comprobaciones fundamentales de métodos, clases y código.

@Anular

Esta anotación se aplica a un método y dice que este método debe anular un método de

superclase o implementar una definición de método de superclase abstracta. Si esta anotación se utiliza con cualquier otro tipo de método, el compilador arrojará un error.

Superclase de concreto

```
public class Vehicle {
   public void drive() {
        System.out.println("I am driving");
   }
}

class Car extends Vehicle {
    // Fine
    @Override
    public void drive() {
        System.out.prinln("Brrrm, brrm");
    }
}
```

Clase abstracta

```
abstract class Animal  {
   public abstract void makeNoise();
}

class Dog extends Animal {
    // Fine
    @Override
    public void makeNoise() {
        System.out.prinln("Woof");
    }
}
```

No funciona

```
class Logger1 {
    public void log(String logString) {
        System.out.prinln(logString);
    }
}

class Logger2 {
    // This will throw compile-time error. Logger2 is not a subclass of Logger1.
    // log method is not overriding anything
    @Override
    public void log(String logString) {
        System.out.println("Log 2" + logString);
    }
}
```

El propósito principal es detectar errores, donde crees que estás anulando un método, pero en realidad estás definiendo uno nuevo.

```
class Vehicle {
   public void drive() {
        System.out.println("I am driving");
```

```
    }
}

class Car extends Vehicle {
    // Compiler error. "dirve" is not the correct method name to override.
    @Override
    public void dirve() {
        System.out.prinln("Brrrm, brrm");
    }
}
```

Tenga en cuenta que el significado de `@Override` ha cambiado con el tiempo:

- En Java 5, significaba que el método anotado tenía que anular un método no abstracto declarado en la cadena de superclase.
- Desde Java 6 en adelante, *también se* satisface si el método anotado implementa un método abstracto declarado en la jerarquía de superclase / interfaz de clases.

(Ocasionalmente, esto puede causar problemas al realizar un back-port del código a Java 5.)

@Obsoleto

Esto marca el método como obsoleto. Puede haber varias razones para esto:

- la API es defectuosa y no es práctico de arreglar,
- el uso de la API puede llevar a errores,
- la API ha sido sustituida por otra API,
- la API está obsoleta,
- la API es experimental y está sujeta a cambios incompatibles,
- o cualquier combinación de los anteriores.

El motivo específico de la desaprobación se puede encontrar en la documentación de la API.

La anotación hará que el compilador emita un error si lo usa. Los IDE también pueden resaltar este método de alguna manera como desaprobado

```
class ComplexAlgorithm {
    @Deprecated
    public void oldSlowUnthreadSafeMethod() {
        // stuff here
    }

    public void quickThreadSafeMethod() {
        // client code should use this instead
    }
}
```

@SuppressWarnings

En casi todos los casos, cuando el compilador emite una advertencia, la acción más apropiada es corregir la causa. En algunos casos (por ejemplo, el código genérico que usa un código pre-genérico seguro para los tipos) puede que no sea posible y es mejor suprimir las advertencias que espera y no puede corregir, por lo que puede ver más claramente las advertencias inesperadas.

Esta anotación se puede aplicar a toda una clase, método o línea. Toma la categoría de advertencia como parámetro.

```
@SuppressWarnings("deprecation")
public class RiddledWithWarnings {
    // several methods calling deprecated code here
}

@SuppressWarning("finally")
public boolean checkData() {
    // method calling return from within finally block
}
```

Es mejor limitar el alcance de la anotación tanto como sea posible, para evitar que también se supriman las advertencias inesperadas. Por ejemplo, al limitar el alcance de la anotación a una sola línea:

```
ComplexAlgorithm algorithm = new ComplexAlgorithm();
@SuppressWarnings("deprecation") algoritm.slowUnthreadSafeMethod();
// we marked this method deprecated in an example above

@SuppressWarnings("unsafe") List<Integer> list = getUntypeSafeList();
// old library returns, non-generic List containing only integers
```

Las advertencias admitidas por esta anotación pueden variar de compilador a compilador. Solo las advertencias `unchecked` y de `deprecation` se mencionan específicamente en el JLS. Se ignorarán los tipos de advertencia no reconocidos.

@SafeVarargs

Debido al borrado de tipo, el `void method(T... t)` se convertirá al `void method(Object[] t)` lo que significa que el compilador no siempre puede verificar que el uso de varargs es de tipo seguro. Por ejemplo:

```
private static <T> void generatesVarargsWarning(T... lists) {
```

Hay casos en que el uso es seguro, en cuyo caso puede anotar el método con la anotación `SafeVarargs` para suprimir la advertencia. Esto obviamente oculta la advertencia si su uso no es seguro también.

@FunctionalInterface

Esta es una anotación opcional utilizada para marcar un FunctionalInterface. Hará que el compilador se queje si no cumple con la especificación FunctionalInterface (tiene un solo método abstracto)

```
@FunctionalInterface
public interface ITrade {
  public boolean check(Trade t);
}

@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
}
```

Comprobaciones de anotación en tiempo de ejecución a través de la reflexión.

La API de Reflection de Java permite al programador realizar varias comprobaciones y operaciones en campos de clase, métodos y anotaciones durante el tiempo de ejecución. Sin embargo, para que una anotación sea visible en el tiempo de ejecución, la `RetentionPolicy` debe cambiarse a `RUNTIME` , como se muestra en el siguiente ejemplo:

```
@interface MyDefaultAnnotation {

}

@Retention(RetentionPolicy.RUNTIME)
@interface MyRuntimeVisibleAnnotation {

}

public class AnnotationAtRuntimeTest {

    @MyDefaultAnnotation
    static class RuntimeCheck1 {
    }

    @MyRuntimeVisibleAnnotation
    static class RuntimeCheck2 {
    }

    public static void main(String[] args) {
        Annotation[] annotationsByType = RuntimeCheck1.class.getAnnotations();
        Annotation[] annotationsByType2 = RuntimeCheck2.class.getAnnotations();

        System.out.println("default retention: " + Arrays.toString(annotationsByType));
        System.out.println("runtime retention: " + Arrays.toString(annotationsByType2));
    }
}
```

Definición de tipos de anotación.

Los tipos de anotación se definen con `@interface` . Los parámetros se definen de manera similar a los métodos de una interfaz regular.

```
@interface MyAnnotation {
    String param1();
    boolean param2();
    int[] param3();  // array parameter
```

```
}
```

Valores predeterminados

```
@interface MyAnnotation {
    String param1() default "someValue";
    boolean param2() default true;
    int[] param3() default {};
}
```

Meta-Anotaciones

Las meta-anotaciones son anotaciones que se pueden aplicar a los tipos de anotación. La meta-anotación predefinida especial define cómo se pueden usar los tipos de anotación.

@Objetivo

La meta-anotación de `@Target` restringe los tipos a los que se puede aplicar la anotación.

```
@Target(ElementType.METHOD)
@interface MyAnnotation {
    // this annotation can only be applied to methods
}
```

Se pueden agregar varios valores utilizando la notación de matriz, por ejemplo, `@Target({ElementType.FIELD, ElementType.TYPE})`

Valores disponibles

Tipo de elemento	objetivo	ejemplo de uso en el elemento de destino
ANNOTATION_TYPE	tipos de anotaciones	`@Retention(RetentionPolicy.RUNTIME)` `@interface MyAnnotation`
CONSTRUCTOR	constructores	`@MyAnnotation` `public MyClass() {}`
CAMPO	campos, constantes de enumeración	`@XmlAttribute` `private int count;`

Tipo de elemento	objetivo	ejemplo de uso en el elemento de destino
VARIABLE LOCAL	Declaraciones variables dentro de los métodos.	`for (@LoopVariable int i = 0; i < 100; i++) { @Unused String resultVariable; }`
PAQUETE	paquete (en `package-info.java`)	`@Deprecated package very.old;`
MÉTODO	metodos	`@XmlElement public int getCount() {...}`
PARÁMETRO	Parámetros de método / constructor	`public Rectangle( @NamedArg("width") double width, @NamedArg("height") double height) { ... }`
TIPO	clases, interfaces, enumeraciones	`@XmlRootElement public class Report {}`

Java SE 8

Tipo de elemento	objetivo	ejemplo de uso en el elemento de destino
TYPE_PARAMETER	Declaraciones de parámetros de tipo	`public <@MyAnnotation T> void f(T t) {}`
TYPE_USE	Uso de un tipo	`Object o = "42"; String s = (@MyAnnotation String) o;`

@Retencion

La meta-anotación de `@Retention` define la visibilidad de la anotación durante el proceso de

compilación o ejecución de las aplicaciones. De forma predeterminada, las anotaciones se incluyen en los archivos `.class` , pero no son visibles en tiempo de ejecución. Para hacer que una anotación sea accesible en tiempo de ejecución, `RetentionPolicy.RUNTIME` debe establecerse en esa anotación.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    // this annotation can be accessed with reflections at runtime
}
```

Valores disponibles

Política de retención	Efecto
CLASE	La anotación está disponible en el archivo `.class` , pero no en tiempo de ejecución
RUNTIME	La anotación está disponible en tiempo de ejecución y se puede acceder a través de la reflexión
FUENTE	La anotación está disponible en tiempo de compilación, pero no se agrega a los archivos `.class` . La anotación se puede utilizar, por ejemplo, por un procesador de anotaciones.

@Documentado

La meta-anotación `@Documented` se usa para marcar anotaciones cuyo uso debe ser documentado por los generadores de documentación API como javadoc . No tiene valores. Con `@Documented` , todas las clases que usan la anotación la incluirán en su página de documentación generada. Sin `@Documented` , no es posible ver qué clases usan la anotación en la documentación.

@Heredado

La meta-anotación `@Inherited` es relevante para las anotaciones que se aplican a las clases. No tiene valores. Marcar una anotación como `@Inherited` altera la forma en que funciona la consulta de anotación.

- Para una anotación no heredada, la consulta solo examina la clase que se está examinando.
- Para una anotación heredada, la consulta también verificará la cadena de superclase (recursivamente) hasta que se encuentre una instancia de la anotación.

Tenga en cuenta que solo se consultan las superclases: se ignorarán todas las anotaciones asociadas a las interfaces en la jerarquía de clases.

@Repeable

La meta-anotación `@Repeatable` se agregó en Java 8. Indica que se pueden adjuntar múltiples instancias de la anotación al destino de la anotación. Esta meta-anotación no tiene valores.

Obtención de valores de anotación en tiempo de ejecución

Puede obtener las propiedades actuales de la anotación utilizando Reflexión para obtener el Método o el Campo o la Clase que tiene aplicada una anotación, y luego obtener las propiedades deseadas.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    String key() default "foo";
    String value() default "bar";
}


class AnnotationExample {
    // Put the Annotation on the method, but leave the defaults
    @MyAnnotation
    public void testDefaults() throws Exception {
        // Using reflection, get the public method "testDefaults", which is this method with
no args
        Method method = AnnotationExample.class.getMethod("testDefaults", null);

        // Fetch the Annotation that is of type MyAnnotation from the Method
        MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

        // Print out the settings of the Annotation
        print(annotation);
    }

    //Put the Annotation on the method, but override the settings
    @MyAnnotation(key="baz", value="buzz")
    public void testValues() throws Exception {
        // Using reflection, get the public method "testValues", which is this method with no
args
        Method method = AnnotationExample.class.getMethod("testValues", null);

        // Fetch the Annotation that is of type MyAnnotation from the Method
        MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

        // Print out the settings of the Annotation
        print(annotation);
    }

    public void print(MyAnnotation annotation) {
        // Fetch the MyAnnotation 'key' & 'value' properties, and print them out
        System.out.println(annotation.key() + " = " + annotation.value());
    }

    public static void main(String[] args) {
        AnnotationExample example = new AnnotationExample();
        try {
            example.testDefaults();
            example.testValues();
```

```
        } catch( Exception e ) {
            // Shouldn't throw any Exceptions
            System.err.println("Exception [" + e.getClass().getName() + "] - " +
e.getMessage());
            e.printStackTrace(System.err);
        }
    }
}
```

La salida será

```
foo = bar
baz = buzz
```

Repetir anotaciones

Hasta Java 8, dos instancias de la misma anotación no se podían aplicar a un solo elemento. La solución estándar era usar una anotación de contenedor que contenga una matriz de alguna otra anotación:

```
// Author.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
    String value();
}

// Authors.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Authors {
    Author[] value();
}

// Test.java
@Authors({
    @Author("Mary"),
    @Author("Sam")
})
public class Test {
    public static void main(String[] args) {
        Author[] authors = Test.class.getAnnotation(Authors.class).value();
        for (Author author : authors) {
            System.out.println(author.value());
            // Output:
            // Mary
            // Sam
        }
    }
}
```

Java SE 8

Java 8 proporciona una forma más limpia y transparente de usar anotaciones de contenedor, utilizando la anotación `@Repeatable` . Primero agregamos esto a la clase de `Author` :

```
@Repeatable(Authors.class)
```

Esto le dice a Java que trate las múltiples anotaciones de `@Author` como si estuvieran rodeadas por el contenedor `@Authors` . También podemos usar `Class.getAnnotationsByType()` para acceder a la matriz `@Author` por su propia clase, en lugar de a través de su contenedor:

```
@Author("Mary")
@Author("Sam")
public class Test {
    public static void main(String[] args) {
        Author[] authors = Test.class.getAnnotationsByType(Author.class);
        for (Author author : authors) {
            System.out.println(author.value());
            // Output:
            // Mary
            // Sam
        }
    }
}
```

Anotaciones heredadas

Por defecto, las anotaciones de clase no se aplican a los tipos que las extienden. Esto se puede cambiar agregando la anotación `@Inherited` a la definición de anotación

Ejemplo

Considere las siguientes 2 anotaciones:

```
@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface InheritedAnnotationType {
}
```

y

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface UninheritedAnnotationType {
}
```

Si se anotan tres clases así:

```
@UninheritedAnnotationType
class A {
}

@InheritedAnnotationType
class B extends A {
}

class C extends B {
}
```

ejecutando este código

```
System.out.println(new A().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new B().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println("__________________________________");
System.out.println(new A().getClass().getAnnotation(UninheritedAnnotationType.class));
System.out.println(new B().getClass().getAnnotation(UninheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(UninheritedAnnotationType.class));
```

imprimirá un resultado similar a este (dependiendo de los paquetes de la anotación):

```
null
@InheritedAnnotationType()
@InheritedAnnotationType()
__________________________________
@UninheritedAnnotationType()
null
null
```

Tenga en cuenta que las anotaciones solo pueden heredarse de las clases, no de las interfaces.

Procesamiento de tiempo de compilación utilizando procesador de anotación

Este ejemplo muestra cómo realizar la comprobación de tiempo de compilación de un elemento anotado.

La anotacion

La anotación `@Setter` es un marcador que se puede aplicar a los métodos. La anotación se descartará durante la compilación y no estará disponible posteriormente.

```
package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.METHOD)
public @interface Setter {
}
```

El procesador de anotaciones.

El `SetterProcessor` utiliza la clase `SetterProcessor` para procesar las anotaciones. Comprueba, si los métodos anotados con la anotación `@Setter` son `public` , `static` métodos no `static` con un nombre que comienza con `set` y que tienen una letra mayúscula como cuarta letra. Si una de

estas condiciones no se cumple, se escribe un error en el `Messager` . El compilador escribe esto en stderr, pero otras herramientas podrían usar esta información de manera diferente. Por ejemplo, el IDE de NetBeans permite al usuario especificar procesadores de anotación que se utilizan para mostrar mensajes de error en el editor.

```
package annotation.processor;

import annotation.Setter;
import java.util.Set;
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Messager;
import javax.annotation.processing.ProcessingEnvironment;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.ElementKind;
import javax.lang.model.element.ExecutableElement;
import javax.lang.model.element.Modifier;
import javax.lang.model.element.TypeElement;
import javax.tools.Diagnostic;

@SupportedAnnotationTypes({"annotation.Setter"})
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SetterProcessor extends AbstractProcessor {

    private Messager messager;

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv)
{
        // get elements annotated with the @Setter annotation
        Set<? extends Element> annotatedElements =
roundEnv.getElementsAnnotatedWith(Setter.class);

        for (Element element : annotatedElements) {
            if (element.getKind() == ElementKind.METHOD) {
                // only handle methods as targets
                checkMethod((ExecutableElement) element);
            }
        }

        // don't claim annotations to allow other processors to process them
        return false;
    }

    private void checkMethod(ExecutableElement method) {
        // check for valid name
        String name = method.getSimpleName().toString();
        if (!name.startsWith("set")) {
            printError(method, "setter name must start with \"set\"");
        } else if (name.length() == 3) {
            printError(method, "the method name must contain more than just \"set\"");
        } else if (Character.isLowerCase(name.charAt(3))) {
            if (method.getParameters().size() != 1) {
                printError(method, "character following \"set\" must be upper case");
            }
        }
```

```
        // check, if setter is public
        if (!method.getModifiers().contains(Modifier.PUBLIC)) {
            printError(method, "setter must be public");
        }

        // check, if method is static
        if (method.getModifiers().contains(Modifier.STATIC)) {
            printError(method, "setter must not be static");
        }
    }

    private void printError(Element element, String message) {
        messager.printMessage(Diagnostic.Kind.ERROR, message, element);
    }

    @Override
    public void init(ProcessingEnvironment processingEnvironment) {
        super.init(processingEnvironment);

        // get messager for printing errors
        messager = processingEnvironment.getMessager();
    }

}
```

embalaje

Para ser aplicado por el compilador, el procesador de anotaciones debe estar disponible para el SPI (ver ServiceLoader).

Para hacer esto, se debe agregar un archivo de texto `META-INF/services/javax.annotation.processing.Processor` al archivo jar que contiene el procesador de anotaciones y la anotación además de los otros archivos. El archivo debe incluir el nombre completo del procesador de anotaciones, es decir, debe tener este aspecto

```
annotation.processor.SetterProcessor
```

Asumiremos que el archivo jar se llama `AnnotationProcessor.jar` continuación.

Ejemplo de clase anotada

La siguiente clase es una clase de ejemplo en el paquete predeterminado con las anotaciones aplicadas a los elementos correctos de acuerdo con la política de retención. Sin embargo, solo el procesador de anotaciones solo considera el segundo método como un objetivo de anotación válido.

```
import annotation.Setter;

public class AnnotationProcessorTest {

    @Setter
```

```
    private void setValue(String value) {}

    @Setter
    public void setString(String value) {}

    @Setter
    public static void main(String[] args) {}

}
```

Usando el procesador de anotaciones con javac

Si el procesador de anotaciones se descubre utilizando el SPI, se utiliza automáticamente para procesar elementos anotados. Ej. Compilar la clase `AnnotationProcessorTest` usando

```
javac -cp AnnotationProcessor.jar AnnotationProcessorTest.java
```

produce la siguiente salida

```
AnnotationProcessorTest.java:6: error: setter must be public
    private void setValue(String value) {}
                 ^
AnnotationProcessorTest.java:12: error: setter name must start with "set"
    public static void main(String[] args) {}
                       ^
2 errors
```

En lugar de compilar normalmente. No se crea ningún archivo `.class` .

Esto se podría evitar especificando la opción `-proc:none` para `javac` . También puede renunciar a la compilación habitual especificando `-proc:only` lugar.

Integración IDE

Netbeans

Los procesadores de anotación se pueden utilizar en el editor de NetBeans. Para hacer esto, el procesador de anotaciones debe especificarse en la configuración del proyecto:

1. `Build` a `Project Properties` > `Compiling` > `Compiling`
2. agregar marcas de verificación para `Enable Annotation Processing` y `Enable Annotation Processing in Editor`
3. haga clic en `Add` junto a la lista de procesadores de anotaciones

4. en la ventana emergente que aparece, ingrese el nombre de clase completamente calificado del procesador de anotaciones y haga clic en `Ok` .

Resultado

```
import annotation.Setter;

public class Annotation

    @Setter
    private void setValue(String value) {}

    @Setter
    public void setString(String value) {}

    @Setter
    public static void main(String[] args) {}

}
```

setter must be public

(Alt-Enter shows hints)

La idea detrás de las anotaciones.

La especificación del lenguaje Java describe las anotaciones de la siguiente manera:

> Una anotación es un marcador que asocia información con una construcción de programa, pero no tiene efecto en el tiempo de ejecución.

Las anotaciones pueden aparecer antes de los tipos o declaraciones. Es posible que aparezcan en un lugar donde se puedan aplicar tanto a un tipo como a una declaración.
A qué se aplica exactamente una anotación se rige por la "meta-anotación" `@Target` . Consulte "Definición de tipos de anotación" para obtener más información.

Las anotaciones se utilizan para una multitud de propósitos. Los marcos como Spring y Spring-MVC hacen uso de anotaciones para definir dónde se deben inyectar las dependencias o dónde se deben enrutar las solicitudes.

Otros marcos usan anotaciones para la generación de código. Lombok y JPA son ejemplos principales, que usan anotaciones para generar código Java (y SQL).

Este tema tiene como objetivo proporcionar una visión global de:

- ¿Cómo definir tus propias anotaciones?
- ¿Qué anotaciones proporciona el lenguaje Java?
- ¿Cómo se utilizan las anotaciones en la práctica?

Anotaciones para 'este' y parámetros del receptor.

Cuando se introdujeron las anotaciones de Java por primera vez, no había ninguna disposición

para anotar el objetivo de un método de instancia o el parámetro de constructor oculto para un constructor de clases internas. Esto se solucionó en Java 8 con la adición de declaraciones de *parámetros* del *receptor* ; ver JLS 8.4.1 .

> El parámetro receptor es un dispositivo sintáctico opcional para un método de instancia o un constructor de clase interno. Para un método de instancia, el parámetro receptor representa el objeto para el que se invoca el método. Para el constructor de una clase interna, el parámetro receptor representa la instancia inmediatamente adjunta del objeto recién construido. De cualquier manera, el parámetro del receptor existe únicamente para permitir que el tipo del objeto representado se indique en el código fuente, de modo que el tipo pueda ser anotado. El parámetro receptor no es un parámetro formal; más precisamente, no es una declaración de ningún tipo de variable (§4.12.3), nunca se vincula a ningún valor pasado como un argumento en una expresión de invocación de método o expresión de creación de instancia de clase calificada, y no tiene ningún efecto en absoluto correr tiempo

El siguiente ejemplo ilustra la sintaxis para ambos tipos de parámetros del receptor:

```
public class Outer {
    public class Inner {
        public Inner (Outer this) {
           // ...
        }
        public void doIt(Inner this) {
           // ...
        }
    }
}
```

El único propósito de los parámetros del receptor es permitirle agregar anotaciones. Por ejemplo, puede tener una anotación personalizada `@IsOpen` cuyo propósito es afirmar que un objeto `Closeable` no se ha cerrado cuando se llama a un método. Por ejemplo:

```
public class MyResource extends Closeable {
    public void update(@IsOpen MyResource this, int value) {
        // ...
    }

    public void close() {
        // ...
    }
}
```

En un nivel, la anotación `@IsOpen` sobre `this` podría simplemente servir como documentación. Sin embargo, potencialmente podríamos hacer más. Por ejemplo:

- Un procesador de anotaciones podría insertar una verificación de tiempo de ejecución de que `this` no está en estado cerrado cuando se llama a la `update` .
- Un verificador de código podría realizar un análisis de código estático para encontrar casos donde `this` *podría* cerrarse cuando se llama a la `update` .

Añadir múltiples valores de anotación

Un parámetro de anotación puede aceptar múltiples valores si se define como una matriz. Por ejemplo, la anotación estándar `@SuppressWarnings` se define así:

```
public @interface SuppressWarnings {
    String[] value();
}
```

El parámetro de `value` es una matriz de cadenas. Puede establecer varios valores utilizando una notación similar a los inicializadores de matriz:

```
@SuppressWarnings({"unused"})
@SuppressWarnings({"unused", "javadoc"})
```

Si solo necesita establecer un único valor, se pueden omitir los paréntesis:

```
@SuppressWarnings("unused")
```

Lea Anotaciones en línea: https://riptutorial.com/es/java/topic/157/anotaciones

Capítulo 8: Apache Commons Lang

Examples

Implementar el método equals ()

Para implementar fácilmente el método `equals` de un objeto, puede usar la clase `EqualsBuilder` .

Seleccionando los campos:

```
@Override
public boolean equals(Object obj) {

    if(!(obj instanceof MyClass)) {
        return false;
    }
    MyClass theOther = (MyClass) obj;

    EqualsBuilder builder = new EqualsBuilder();
    builder.append(field1, theOther.field1);
    builder.append(field2, theOther.field2);
    builder.append(field3, theOther.field3);

    return builder.isEquals();
}
```

Usando la reflexión:

```
@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj, false);
}
```

el parámetro booleano es para indicar si los iguales deben verificar los campos transitorios.

Usando la reflexión evitando algunos campos:

```
@Override
public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj, "field1", "field2");
}
```

Implementar el método hashCode ()

Para implementar `hashCode` método `hashCode` de un objeto, puede usar la clase `HashCodeBuilder` .

Seleccionando los campos:

```
@Override
public int hashCode() {
```

```
    HashCodeBuilder builder = new HashCodeBuilder();
    builder.append(field1);
    builder.append(field2);
    builder.append(field3);

    return builder.hashCode();
}
```

Usando la reflexión:

```
@Override
public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this, false);
}
```

el parámetro booleano indica si debe usar campos transitorios.

Usando la reflexión evitando algunos campos:

```
@Override
public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this, "field1", "field2");
}
```

Implementar el método toString ()

Para implementar fácilmente el método `toString` de un objeto, puede usar la clase `ToStringBuilder` .

Seleccionando los campos:

```
@Override
public String toString() {

    ToStringBuilder builder = new ToStringBuilder(this);
    builder.append(field1);
    builder.append(field2);
    builder.append(field3);

    return builder.toString();
}
```

Resultado de ejemplo:

```
ar.com.jonat.lang.MyClass@dd7123[<null>,0,false]
```

Dando nombres explícitos a los campos:

```
@Override
public String toString() {

    ToStringBuilder builder = new ToStringBuilder(this);
```

```
    builder.append("field1",field1);
    builder.append("field2",field2);
    builder.append("field3",field3);

    return builder.toString();
}
```

Resultado de ejemplo:

```
ar.com.jonat.lang.MyClass@dd7404[field1=<null>,field2=0,field3=false]
```

Podrías cambiar el estilo vía parámetro:

```
@Override
public String toString() {

    ToStringBuilder builder = new ToStringBuilder(this,
            ToStringStyle.MULTI_LINE_STYLE);
    builder.append("field1", field1);
    builder.append("field2", field2);
    builder.append("field3", field3);

    return builder.toString();
}
```

Resultado de ejemplo:

```
ar.com.bna.lang.MyClass@ebbf5c[
  field1=<null>
  field2=0
  field3=false
]
```

Hay algunos estilos, por ejemplo JSON, sin nombre de clase, cortos, etc.

Vía la reflexión:

```
@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this);
}
```

También puedes indicar el estilo:

```
@Override
public String toString() {
    return ToStringBuilder.reflectionToString(this, ToStringStyle.JSON_STYLE);
}
```

Lea Apache Commons Lang en línea: https://riptutorial.com/es/java/topic/3338/apache-commons-lang

Capítulo 9: API de apilación

Introducción

Antes de Java 9, el acceso a los marcos de la pila de hilos estaba limitado a una clase interna `sun.reflect.Reflection` . Específicamente el método `sun.reflect.Reflection::getCallerClass` . Algunas bibliotecas se basan en este método que está en desuso.

Una API estándar alternativo ahora se proporciona en el JDK 9 a través de la `java.lang.StackWalker` clase, y está diseñado para ser eficiente al permitir el acceso perezoso para los marcos de pila. Algunas aplicaciones pueden usar esta API para atravesar la pila de ejecución y filtrar en las clases.

Examples

Imprimir todos los marcos de pila de la secuencia actual

Lo siguiente imprime todos los marcos de pila del hilo actual:

```
package test;

import java.lang.StackWalker.StackFrame;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.List;
import java.util.stream.Collectors;

public class StackWalkerExample {

   public static void main(String[] args) throws NoSuchMethodException, SecurityException, IllegalAccessException, IllegalArgumentException, InvocationTargetException {
       Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);
       fooMethod.invoke(null, (Object[]) null);
   }
}

class FooHelper {
   protected static void foo() {
       BarHelper.bar();
   }
}

class BarHelper {
   protected static void bar() {
       List<StackFrame> stack = StackWalker.getInstance()
               .walk((s) -> s.collect(Collectors.toList()));
       for(StackFrame frame : stack) {
           System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " + frame.getMethodName());
       }
   }
}
```

Salida:

```
test.BarHelper 26 bar
test.FooHelper 19 foo
test.StackWalkerExample 13 main
```

Imprimir la clase de llamada actual

A continuación se imprime la clase de llamada actual. Tenga en cuenta que, en este caso, `StackWalker` debe crearse con la opción `RETAIN_CLASS_REFERENCE` , para que `Class` instancias de la `Class` se conserven en los objetos `StackFrame` . De lo contrario se produciría una excepción.

```
public class StackWalkerExample {

    public static void main(String[] args) {
        FooHelper.foo();
    }

}

class FooHelper {
    protected static void foo() {
        BarHelper.bar();
    }
}

class BarHelper {
    protected static void bar() {

System.out.println(StackWalker.getInstance(Option.RETAIN_CLASS_REFERENCE).getCallerClass());
    }
}
```

Salida:

```
class test.FooHelper
```

Mostrando la reflexión y otros marcos ocultos.

Un par de otras opciones permiten que los seguimientos de pila incluyan marcos de implementación y / o reflexión. Esto puede ser útil para propósitos de depuración. Por ejemplo, podemos agregar la opción `SHOW_REFLECT_FRAMES` a la instancia de `StackWalker` momento de la creación, para que también se impriman los marcos para los métodos reflectivos:

```
package test;

import java.lang.StackWalker.Option;
import java.lang.StackWalker.StackFrame;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.List;
import java.util.stream.Collectors;
```

```
public class StackWalkerExample {

    public static void main(String[] args) throws NoSuchMethodException, SecurityException,
IllegalAccessException, IllegalArgumentException, InvocationTargetException {
        Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);
        fooMethod.invoke(null, (Object[]) null);
    }
}

class FooHelper {
    protected static void foo() {
        BarHelper.bar();
    }
}

class BarHelper {
    protected static void bar() {
        // show reflection methods
        List<StackFrame> stack = StackWalker.getInstance(Option.SHOW_REFLECT_FRAMES)
                .walk((s) -> s.collect(Collectors.toList()));
        for(StackFrame frame : stack) {
            System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
frame.getMethodName());
        }
    }
}
```

Salida:

```
test.BarHelper 27 bar
test.FooHelper 20 foo
jdk.internal.reflect.NativeMethodAccessorImpl -2 invoke0
jdk.internal.reflect.NativeMethodAccessorImpl 62 invoke
jdk.internal.reflect.DelegatingMethodAccessorImpl 43 invoke
java.lang.reflect.Method 563 invoke
test.StackWalkerExample 14 main
```

Tenga en cuenta que los números de línea para algunos métodos de reflexión pueden no estar disponibles, por lo que `StackFrame.getLineNumber()` puede devolver valores negativos.

Lea API de apilación en línea: https://riptutorial.com/es/java/topic/9868/api-de-apilacion

Capítulo 10: API de reflexión

Introducción

Reflection es comúnmente utilizado por programas que requieren la capacidad de examinar o modificar el comportamiento en tiempo de ejecución de las aplicaciones que se ejecutan en la JVM. Java Reflection API se usa para ese propósito donde permite inspeccionar clases, interfaces, campos y métodos en tiempo de ejecución, sin saber sus nombres en el momento de la compilación. Y también hace posible instanciar nuevos objetos, e invocar métodos utilizando la reflexión.

Observaciones

Actuación

Tenga en cuenta que la reflexión puede disminuir el rendimiento, solo úselo cuando su tarea no pueda completarse sin reflexión.

Desde el tutorial de Java La API de reflexión :

> Debido a que la reflexión involucra tipos que se resuelven dinámicamente, ciertas optimizaciones de máquinas virtuales Java no se pueden realizar. En consecuencia, las operaciones de reflexión tienen un rendimiento más lento que sus contrapartes no reflexivas, y deben evitarse en las secciones de código que se llaman con frecuencia en aplicaciones sensibles al rendimiento.

Examples

Introducción

Lo esencial

La API de reflexión permite verificar la estructura de clase del código en tiempo de ejecución e invocar el código dinámicamente. Esto es muy poderoso, pero también es peligroso ya que el compilador no es capaz de determinar estáticamente si las invocaciones dinámicas son válidas.

Un ejemplo simple sería obtener los constructores públicos y los métodos de una clase determinada:

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

// This is a object representing the String class (not an instance of String!)
Class<String> clazz = String.class;
```

```
Constructor<?>[] constructors = clazz.getConstructors(); // returns all public constructors of
String
Method[] methods = clazz.getMethods(); // returns all public methods from String and parents
```

Con esta información es posible instanciar el objeto y llamar a diferentes métodos dinámicamente.

Reflexión y tipos genéricos.

La información de tipo genérico está disponible para:

- Parámetros del método, utilizando `getGenericParameterTypes()` .
- Método devuelve los tipos, usando `getGenericReturnType()` .
- Campos **públicos** , utilizando `getGenericType` .

El siguiente ejemplo muestra cómo extraer la información de tipo genérico en los tres casos:

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.List;
import java.util.Map;

public class GenericTest {

    public static void main(final String[] args) throws Exception {
        final Method method = GenericTest.class.getMethod("testMethod", Map.class);
        final Field field = GenericTest.class.getField("testField");

        System.out.println("Method parameter:");
        final Type parameterType = method.getGenericParameterTypes()[0];
        displayGenericType(parameterType, "\t");

        System.out.println("Method return type:");
        final Type returnType = method.getGenericReturnType();
        displayGenericType(returnType, "\t");

        System.out.println("Field type:");
        final Type fieldType = field.getGenericType();
        displayGenericType(fieldType, "\t");

    }

    private static void displayGenericType(final Type type, final String prefix) {
        System.out.println(prefix + type.getTypeName());
        if (type instanceof ParameterizedType) {
            for (final Type subtype : ((ParameterizedType) type).getActualTypeArguments()) {
                displayGenericType(subtype, prefix + "\t");
            }
        }

    }

    public Map<String, Map<Integer, List<String>>> testField;

    public List<Number> testMethod(final Map<String, Double> arg) {
        return null;
    }
```

```
}
```

Esto resulta en el siguiente resultado:

```
Method parameter:
    java.util.Map<java.lang.String, java.lang.Double>
        java.lang.String
        java.lang.Double
Method return type:
    java.util.List<java.lang.Number>
        java.lang.Number
Field type:
    java.util.Map<java.lang.String, java.util.Map<java.lang.Integer,
java.util.List<java.lang.String>>>
        java.lang.String
        java.util.Map<java.lang.Integer, java.util.List<java.lang.String>>
            java.lang.Integer
            java.util.List<java.lang.String>
                java.lang.String
```

Invocando un método

Usando la reflexión, se puede invocar un método de un objeto durante el tiempo de ejecución.

El ejemplo muestra cómo invocar los métodos de un objeto `String` .

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

String s = "Hello World!";

// method without parameters
// invoke s.length()
Method method1 = String.class.getMethod("length");
int length = (int) method1.invoke(s); // variable length contains "12"

// method with parameters
// invoke s.substring(6)
Method method2 = String.class.getMethod("substring", int.class);
String substring = (String) method2.invoke(s, 6); // variable substring contains "World!"
```

Obtención y configuración de campos

Usando la API de Reflection, es posible cambiar u obtener el valor de un campo en tiempo de ejecución. Por ejemplo, podría usarlo en una API para recuperar diferentes campos según un factor, como el sistema operativo. También puede eliminar modificadores como `final` para permitir que los campos de modificación sean finales.

Para hacerlo, deberá usar el método Clase # getField () de una manera como la que se muestra a continuación:

```
// Get the field in class SomeClass "NAME".
```

```
Field nameField = SomeClass.class.getDeclaredField("NAME");

// Get the field in class Field "modifiers". Note that it does not
// need to be static
Field modifiersField = Field.class.getDeclaredField("modifiers");

// Allow access from anyone even if it's declared private
modifiersField.setAccessible(true);

// Get the modifiers on the "NAME" field as an int.
int existingModifiersOnNameField = nameField.getModifiers();

// Bitwise AND NOT Modifier.FINAL (16) on the existing modifiers
// Readup here https://en.wikipedia.org/wiki/Bitwise_operations_in_C
// if you're unsure what bitwise operations are.
int newModifiersOnNameField = existingModifiersOnNameField & ~Modifier.FINAL;

// Set the value of the modifiers field under an object for non-static fields
modifiersField.setInt(nameField, newModifiersOnNameField);

// Set it to be accessible. This overrides normal Java
// private/protected/package/etc access control checks.
nameField.setAccessible(true);

 // Set the value of "NAME" here. Note the null argument.
 // Pass null when modifying static fields, as there is no instance object
nameField.set(null, "Hacked by reflection...");

// Here I can directly access it. If needed, use reflection to get it. (Below)
System.out.println(SomeClass.NAME);
```

Obtener campos es mucho más fácil. Podemos usar Field # get () y sus variantes para obtener su valor:

```
// Get the field in class SomeClass "NAME".
Field nameField = SomeClass.class.getDeclaredField("NAME");

// Set accessible for private fields
nameField.setAccessible(true);

// Pass null as there is no instance, remember?
String name = (String) nameField.get(null);
```

Tenga en cuenta esto:

Cuando use Class # getDeclaredField , utilícelo para obtener un campo en la clase en sí:

```
class HackMe extends Hacked {
    public String iAmDeclared;
}

class Hacked {
    public String someState;
}
```

Aquí, `HackMe#iAmDeclared` es campo declarado. Sin embargo, `HackMe#someState` no es un campo declarado, ya que se hereda de su superclase, hackeado.

Constructor de llamadas

Obtención del objeto constructor

Puedes obtener la clase `Constructor` del objeto `Class` esta manera:

```
Class myClass = ... // get a class object
Constructor[] constructors = myClass.getConstructors();
```

Donde la variable de `constructors` tendrá una instancia de `Constructor` para cada constructor público declarado en la clase.

Si conoce los tipos de parámetros precisos del constructor al que desea acceder, puede filtrar el constructor específico. El siguiente ejemplo devuelve el constructor público de la clase dada que toma un `Integer` como parámetro:

```
Class myClass = ... // get a class object
Constructor constructor = myClass.getConstructor(new Class[]{Integer.class});
```

Si ningún constructor coincide con los argumentos de constructor `NoSuchMethodException` se lanza una `NoSuchMethodException` .

Nueva Instancia usando Objeto Constructor

```
Class myClass = MyObj.class // get a class object
Constructor constructor = myClass.getConstructor(Integer.class);
MyObj myObj = (MyObj) constructor.newInstance(Integer.valueOf(123));
```

Obteniendo las constantes de una enumeración

Dando esta enumeración como ejemplo:

```
enum Compass {
    NORTH(0),
    EAST(90),
    SOUTH(180),
    WEST(270);
    private int degree;
    Compass(int deg){
        degree = deg;
    }
    public int getDegree(){
        return degree;
    }
}
```

En Java, una clase de enumeración es como cualquier otra clase, pero tiene algunas constantes definidas para los valores de enumeración. Además, tiene un campo que es una matriz que contiene todos los valores y dos métodos estáticos con `values()` nombre `values()` y

`valueOf(String)` .
Podemos ver esto si usamos Reflection para imprimir todos los campos de esta clase

```
for(Field f : Compass.class.getDeclaredFields())
    System.out.println(f.getName());
```

La salida será:

NORTE
ESTE
SUR
OESTE
la licenciatura
ENUM $ VALORES

Así que podríamos examinar clases de enumeración con Reflexión como cualquier otra clase. Pero la API de Reflection ofrece tres métodos específicos de enumeración.

control de enumeración

```
Compass.class.isEnum();
```

Devuelve true para las clases que representan un tipo de enumeración.

recuperando valores

```
Object[] values = Compass.class.getEnumConstants();
```

Devuelve una matriz de todos los valores de enumeración como Compass.values () pero sin la necesidad de una instancia.

enumeración constante

```
for(Field f : Compass.class.getDeclaredFields()){
    if(f.isEnumConstant())
        System.out.println(f.getName());
}
```

Enumera todos los campos de clase que son valores de enumeración.

Obtener clase dado su nombre (totalmente calificado)

Dada una `String` contiene el nombre de una clase, se puede acceder a su objeto de `Class` usando `Class.forName` :

```
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer");
} catch (ClassNotFoundException ex) {
```

```
    throw new IllegalStateException(ex);
}
```

Java SE 1.2

Puede especificarse, si la clase debería inicializarse (segundo parámetro de `forName`) y qué `ClassLoader` debería usarse (tercer parámetro):

```
ClassLoader classLoader = ...
boolean initialize = ...
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer", initialize, classLoader);
} catch (ClassNotFoundException ex) {
    throw new IllegalStateException(ex);
}
```

Llamar a los constructores sobrecargados utilizando la reflexión.

Ejemplo: invocar diferentes constructores pasando parámetros relevantes

```
import java.lang.reflect.*;

class NewInstanceWithReflection{
    public NewInstanceWithReflection(){
        System.out.println("Default constructor");
    }
    public NewInstanceWithReflection( String a){
        System.out.println("Constructor :String => "+a);
    }
    public static void main(String args[]) throws Exception {

        NewInstanceWithReflection object =
(NewInstanceWithReflection)Class.forName("NewInstanceWithReflection").newInstance();
        Constructor constructor = NewInstanceWithReflection.class.getDeclaredConstructor( new
Class[] {String.class});
        NewInstanceWithReflection object1 =
(NewInstanceWithReflection)constructor.newInstance(new Object[]{"StackOverFlow"});

    }
}
```

salida:

```
Default constructor
Constructor :String => StackOverFlow
```

Explicación:

1. Crear instancia de clase usando `Class.forName` : llama al constructor predeterminado
2. Invoque `getDeclaredConstructor` de la clase pasando tipos de parámetros como `Class array`
3. Después de obtener el constructor, cree `newInstance` pasando un valor de parámetro como `Object array`

Uso incorrecto de la API de Reflection para cambiar variables privadas y finales

La reflexión es útil cuando se usa correctamente para el propósito correcto. Al utilizar la reflexión, puede acceder a las variables privadas y reinicializar las variables finales.

A continuación se muestra el fragmento de código, que **no se** recomienda.

```
import java.lang.reflect.*;

public class ReflectionDemo{
    public static void main(String args[]){
        try{
            Field[] fields = A.class.getDeclaredFields();
            A a = new A();
            for ( Field field:fields ) {
                if(field.getName().equalsIgnoreCase("name")){
                    field.setAccessible(true);
                    field.set(a, "StackOverFlow");
                    System.out.println("A.name="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("age")){
                    field.set(a, 20);
                    System.out.println("A.age="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("rep")){
                    field.setAccessible(true);
                    field.set(a,"New Reputation");
                    System.out.println("A.rep="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("count")){
                    field.set(a,25);
                    System.out.println("A.count="+field.get(a));
                }
            }
        }catch(Exception err){
            err.printStackTrace();
        }
    }
}

class A {
    private String name;
    public int age;
    public final String rep;
    public static int count=0;

    public A(){
        name = "Unset";
        age = 0;
        rep = "Reputation";
        count++;
    }
}
```

Salida:

```
A.name=StackOverFlow
A.age=20
A.rep=New Reputation
A.count=25
```

Explicación:

En el escenario normal, no se puede acceder a `private` variables `private` fuera de la clase declarada (sin los métodos get y set). `final` variables `final` no pueden ser reasignadas después de la inicialización.

`Reflection` rompe ambas barreras se puede utilizar para cambiar las variables privadas y finales como se explicó anteriormente.

`field.setAccessible(true)` es la clave para lograr la funcionalidad deseada.

Call constructor de clase anidada

Si desea crear una instancia de una clase anidada interna, debe proporcionar un objeto de clase de la clase envolvente como un parámetro adicional con la Clase # getDeclaredConstructor .

```
public class Enclosing{
    public class Nested{
    public Nested(String a){
            System.out.println("Constructor :String => "+a);
        }
    }
    public static void main(String args[]) throws Exception {
        Class<?> clazzEnclosing = Class.forName("Enclosing");
        Class<?> clazzNested = Class.forName("Enclosing$Nested");
        Enclosing objEnclosing = (Enclosing)clazzEnclosing.newInstance();
        Constructor<?> constructor = clazzNested.getDeclaredConstructor(new
Class[]{Enclosing.class, String.class});
        Nested objInner = (Nested)constructor.newInstance(new Object[]{objEnclosing,
"StackOverFlow"});
    }
}
```

Si la clase anidada es estática, no necesitará esta instancia adjunta.

Proxies dinámicos

Los proxies dinámicos realmente no tienen mucho que ver con Reflection, pero son parte de la API. Es básicamente una forma de crear una implementación dinámica de una interfaz. Esto podría ser útil al crear servicios de maqueta.
Un proxy dinámico es una instancia de una interfaz que se crea con el llamado controlador de invocación que intercepta todas las llamadas de método y permite el manejo de su invocación manualmente.

```
public class DynamicProxyTest {

    public interface MyInterface1{
```

```
        public void someMethod1();
        public int someMethod2(String s);
    }

    public interface MyInterface2{
        public void anotherMethod();
    }

    public static void main(String args[]) throws Exception {
        // the dynamic proxy class
        Class<?> proxyClass = Proxy.getProxyClass(
                ClassLoader.getSystemClassLoader(),
                new Class[] {MyInterface1.class, MyInterface2.class});
        // the dynamic proxy class constructor
        Constructor<?> proxyConstructor =
            proxyClass.getConstructor(InvocationHandler.class);

        // the invocation handler
        InvocationHandler handler = new InvocationHandler(){
            // this method is invoked for every proxy method call
            // method is the invoked method, args holds the method parameters
            // it must return the method result
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{
                String methodName = method.getName();

                if(methodName.equals("someMethod1")){
                    System.out.println("someMethod1 was invoked!");
                    return null;
                }
                if(methodName.equals("someMethod2")){
                    System.out.println("someMethod2 was invoked!");
                    System.out.println("Parameter: " + args[0]);
                    return 42;
                }
                if(methodName.equals("anotherMethod")){
                    System.out.println("anotherMethod was invoked!");
                    return null;
                }
                System.out.println("Unkown method!");
                return null;
            }
        };

        // create the dynamic proxy instances
        MyInterface1 i1 = (MyInterface1) proxyConstructor.newInstance(handler);
        MyInterface2 i2 = (MyInterface2) proxyConstructor.newInstance(handler);

        // and invoke some methods
        i1.someMethod1();
        i1.someMethod2("stackoverflow");
        i2.anotherMethod();
    }
}
```

El resultado de este código es este:

```
someMethod1 was invoked!
someMethod2 was invoked!
```

```
Parameter: stackoverflow
anotherMethod was invoked!
```

Evil hacks de Java con Reflection

La API de Reflection podría usarse para cambiar los valores de los campos privados y finales incluso en la biblioteca predeterminada de JDK. Esto podría usarse para manipular el comportamiento de algunas clases bien conocidas como veremos.

Lo que no es posible

Comencemos primero con la única limitación significa que el único campo que no podemos cambiar con Reflexión. Ese es el Java `SecurityManager` . Se declara en java.lang.System as

```
private static volatile SecurityManager security = null;
```

Pero no se incluirá en la clase del sistema si ejecutamos este código

```
for(Field f : System.class.getDeclaredFields())
    System.out.println(f);
```

`fieldFilterMap` debe al `fieldFilterMap` en `sun.reflect.Reflection` que contiene el mapa en sí y el campo de seguridad en `System.class` y los protege contra cualquier acceso con Reflection. Por eso no pudimos desactivar el `SecurityManager` .

Cuerdas locas

Cada cadena Java está representada por la JVM como una instancia de la clase de `String` . Sin embargo, en algunas situaciones, la JVM ahorra espacio de almacenamiento utilizando la misma instancia para las cadenas que son. Esto sucede para los literales de cadena, y también para las cadenas que han sido "internadas" llamando a `String.intern()` . Entonces, si tiene `"hello"` en su código varias veces, siempre será la misma instancia de objeto.

Se supone que las cuerdas son inmutables, pero es posible usar la reflexión "malvada" para cambiarlas. El siguiente ejemplo muestra cómo podemos cambiar los caracteres en una cadena reemplazando su campo de `value` .

```
public class CrazyStrings {
    static {
        try {
            Field f = String.class.getDeclaredField("value");
            f.setAccessible(true);
            f.set("hello", "you stink!".toCharArray());
        } catch (Exception e) {
        }
    }
    public static void main(String args[])  {
        System.out.println("hello");
    }
}
```

Así que este código imprimirá "apestas!"

1 = 42

La misma idea podría ser usada con la clase entera.

```
public class CrazyMath {
    static {
        try {
            Field value = Integer.class.getDeclaredField("value");
            value.setAccessible(true);
            value.setInt(Integer.valueOf(1), 42);
        } catch (Exception e) {
        }
    }
    public static void main(String args[])  {
        System.out.println(Integer.valueOf(1));
    }
}
```

Todo es verdad

Y de acuerdo con este post de stackoverflow podemos usar la reflexión para hacer algo realmente malo.

```
public class Evil {
    static {
        try {
            Field field = Boolean.class.getField("FALSE");
            field.setAccessible(true);
            Field modifiersField = Field.class.getDeclaredField("modifiers");
            modifiersField.setAccessible(true);
            modifiersField.setInt(field, field.getModifiers() & ~Modifier.FINAL);
            field.set(null, true);
        } catch (Exception e) {
        }
    }
    public static void main(String args[]){
        System.out.format("Everything is %s", false);
    }
}
```

Tenga en cuenta que lo que estamos haciendo aquí hará que la JVM se comporte de forma inexplicable. Esto es muy peligroso.

Lea API de reflexión en línea: https://riptutorial.com/es/java/topic/629/api-de-reflexion

Capítulo 11: AppDynamics y TIBCO BusinessWorks Instrumentation para una fácil integración

Introducción

Como AppDynamics pretende proporcionar una manera de medir el rendimiento de las aplicaciones, la velocidad de desarrollo, la entrega (implementación) de las aplicaciones es un factor esencial para que los esfuerzos de DevOps sean un verdadero éxito. La supervisión de una aplicación TIBCO BW con AppD es generalmente simple y no requiere mucho tiempo, pero cuando se implementan grandes conjuntos de aplicaciones, la instrumentación rápida es la clave. Esta guía muestra cómo instrumentar todas sus aplicaciones BW en un solo paso sin modificar cada aplicación antes de implementar.

Examples

Ejemplo de instrumentación de todas las aplicaciones BW en un solo paso para Appdynamics

1. Localice y abra su archivo TIBCO BW bwengine.tra tícicamente bajo TIBCO_HOME / bw / 5.12 / bin / bwengine.tra (entorno Linux)

2. Busca la línea que dice:

*** Variables comunes. Modifica estos solo. ***

3. Agregue la siguiente línea justo después de esa sección tibco.deployment =% tibco.deployment%

4. Vaya al final del archivo y agregue (reemplace? Con sus propios valores según sea necesario o elimine el indicador que no se aplica): java.extended.properties = -javaagent: /opt/appd/current/appagent/javaagent.jar - Dappdynamics.http.proxyHost =? - Dappdynamics.http.proxyPort =? -Dappdynamics.agent.applicationName =? - Dappdynamics.agent.tierName =? -Dappdynamics.agent.nodeName =% tibco.deployment% -Dappdynamics.controller.ssl.enabled =? -Dappdynamics.controller.sslPort =? - Dappdynamics.agent.logs.dir =? -Dappdynamics.agent.runtime.dir =? - Dappdynamics.controller.hostName =? -Dappdynamics.controller.port =? - Dappdynamics.agent.accountName =? -Dappdynamics.agent.accountAccessKey =?

5. Guardar archivo y volver a desplegar. Todas sus aplicaciones ahora deben ser

instrumentadas automáticamente en el momento del despliegue.

Lea AppDynamics y TIBCO BusinessWorks Instrumentation para una fácil integración en línea: https://riptutorial.com/es/java/topic/10602/appdynamics-y-tibco-businessworks-instrumentation-para-una-facil-integracion

Capítulo 12: Applets

Introducción

Los applets han sido parte de Java desde su lanzamiento oficial y se han utilizado para enseñar Java y programación durante varios años.

Los últimos años han visto un impulso activo para alejarse de los applets y otros complementos del navegador, con algunos navegadores que los bloquean o no los apoyan activamente.

En 2016, Oracle anunció sus planes de dejar de usar el complemento, pasando a una web sin complementos

Ya están disponibles las API más nuevas y mejores

Observaciones

Un applet es una aplicación Java que normalmente se ejecuta dentro de un navegador web. La idea básica es interactuar con el usuario sin la necesidad de interactuar con el servidor y transferir información. Este concepto fue muy exitoso alrededor del año 2000 cuando la comunicación por Internet era lenta y costosa.

Un applet ofrece cinco métodos para controlar su ciclo de vida.

nombre del método	descripción
`init()`	se llama una vez cuando se carga el applet
`destroy()`	se llama una vez cuando el applet se elimina de la memoria
`start()`	se llama cada vez que el applet se hace visible
`stop()`	se llama cada vez que el applet se superpone con otras ventanas
`paint()`	se llama cuando es necesario o se activa manualmente llamando a `repaint()`

Examples

Applet mínimo

Un applet muy simple dibuja un rectángulo e imprime una cadena en la pantalla.

```
public class MyApplet extends JApplet{
```

```
    private String str = "StackOverflow";

    @Override
    public void init() {
        setBackground(Color.gray);
    }
    @Override
    public void destroy() {}
    @Override
    public void start() {}
    @Override
    public void stop() {}
    @Override
    public void paint(Graphics g) {
        g.setColor(Color.yellow);
        g.fillRect(1,1,300,150);
        g.setColor(Color.red);
        g.setFont(new Font("TimesRoman", Font.PLAIN, 48));
        g.drawString(str, 10, 80);
    }
}
```

La clase principal de un applet se extiende desde `javax.swing.JApplet` .

Java SE 1.2

Antes de Java 1.2 y la introducción de los applets de API swing se extendió desde `java.applet.Applet` .

Los applets no requieren un método principal. El punto de entrada está controlado por el ciclo de vida. Para usarlos, necesitan estar incrustados en un documento HTML. Este es también el punto donde se define su tamaño.

```
<html>
  <head></head>
  <body>
     <applet code="MyApplet.class" width="400" height="200"></applet>
  </body>
</html>
```

Creando una GUI

Los applets podrían usarse fácilmente para crear una GUI. Actúan como un `Container` y tienen un método `add()` que toma cualquier componente `awt` o `swing` .

```
public class MyGUIApplet extends JApplet{

    private JPanel panel;
    private JButton button;
    private JComboBox<String> cmbBox;
    private JTextField textField;

    @Override
    public void init(){
```

```
        panel = new JPanel();
        button = new JButton("ClickMe!");
        button.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent ae) {
                if(((String)cmbBox.getSelectedItem()).equals("greet")) {
                    JOptionPane.showMessageDialog(null,"Hello " + textField.getText());
                } else {
                    JOptionPane.showMessageDialog(null,textField.getText() + " stinks!");
                }
            }
        });
        cmbBox = new JComboBox<>(new String[]{"greet", "offend"});
        textField = new JTextField("John Doe");
        panel.add(cmbBox);
        panel.add(textField);
        panel.add(button);
        add(panel);
    }
}
```

Abrir enlaces desde dentro del applet

Puede usar el método `getAppletContext()` para obtener un objeto `AppletContext` que le permite solicitar al navegador que abra un enlace. Para ello utiliza el método `showDocument()` . Su segundo parámetro le dice al navegador que use una nueva ventana `_blank` o la que muestra el applet `_self` .

```
public class MyLinkApplet extends JApplet{
    @Override
    public void init(){
        JButton button = new JButton("ClickMe!");
        button.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent ae) {
                AppletContext a = getAppletContext();
                try {
                    URL url = new URL("http://stackoverflow.com/");
                    a.showDocument(url,"_blank");
                } catch (Exception e) { /* omitted for brevity */ }
            }
        });
        add(button);
    }
}
```

Cargando imágenes, audio y otros recursos.

Los applets de Java son capaces de cargar diferentes recursos. Pero como se ejecutan en el navegador web del cliente, debe asegurarse de que estos recursos sean accesibles. Los applets no pueden acceder a los recursos del cliente como el sistema de archivos local.

Si desea cargar recursos desde la misma URL en la que está almacenado el Applet, puede usar el método `getCodeBase()` para recuperar la URL base. Para cargar recursos, los applets ofrecen los métodos `getImage()` y `getAudioClip()` para cargar imágenes o archivos de audio.

Cargar y mostrar una imagen.

```
public class MyImgApplet extends JApplet{

    private Image img;

    @Override
    public void init(){
        try {
            img = getImage(new URL("http://cdn.sstatic.net/stackexchange/img/logos/so/so-
logo.png"));
        } catch (MalformedURLException e) { /* omitted for brevity */ }
    }
    @Override
        public void paint(Graphics g) {
            g.drawImage(img, 0, 0, this);
        }
}
```

Cargar y reproducir un archivo de audio.

```
public class MyAudioApplet extends JApplet{

    private AudioClip audioClip;

    @Override
    public void init(){
        try {
            audioClip = getAudioClip(new URL("URL/TO/AN/AUDIO/FILE.WAV"));
        } catch (MalformedURLException e) { /* omitted for brevity */ }
    }
    @Override
    public void start() {
        audioClip.play();
    }
    @Override
    public void stop(){
        audioClip.stop();
    }
}
```

Cargar y mostrar un archivo de texto

```
public class MyTextApplet extends JApplet{
    @Override
    public void init(){
        JTextArea textArea = new JTextArea();
        JScrollPane sp = new JScrollPane(textArea);
```

```
        add(sp);
        // load text
        try {
            URL url = new URL("http://www.textfiles.com/fun/quotes.txt");
            InputStream in = url.openStream();
            BufferedReader bf = new BufferedReader(new InputStreamReader(in));
            String line = "";
            while((line = bf.readLine()) != null) {
                textArea.append(line + "\n");
            }
        } catch(Exception e) { /* omitted for brevity */ }
    }
}
```

Lea Applets en línea: https://riptutorial.com/es/java/topic/5503/applets

Capítulo 13: Archivo I / O

Introducción

La E / S de Java (entrada y salida) se utiliza para procesar la entrada y producir la salida. Java utiliza el concepto de flujo para agilizar la operación de E / S. El paquete java.io contiene todas las clases necesarias para las operaciones de entrada y salida. El manejo de archivos también se realiza en java mediante la API de E / S de Java.

Examples

Leyendo todos los bytes a un byte []

Java 7 introdujo la clase de archivos muy útil

Java SE 7

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.Path;

Path path = Paths.get("path/to/file");

try {
    byte[] data = Files.readAllBytes(path);
} catch(IOException e) {
    e.printStackTrace();
}
```

Leyendo una imagen de un archivo.

```
import java.awt.Image;
import javax.imageio.ImageIO;

...

try {
    Image img = ImageIO.read(new File("~/Desktop/cat.png"));
} catch (IOException e) {
    e.printStackTrace();
}
```

Escribir un byte [] a un archivo

Java SE 7

```
byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };

try(FileOutputStream stream = new FileOutputStream("Hello world.txt")) {
    stream.write(bytes);
```

```
} catch (IOException ioe) {
    // Handle I/O Exception
    ioe.printStackTrace();
}
```

Java SE 7

```
byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };

FileOutputStream stream = null;
try {
    stream = new FileOutputStream("Hello world.txt");
    stream.write(bytes);
} catch (IOException ioe) {
    // Handle I/O Exception
    ioe.printStackTrace();
} finally {
    if (stream != null) {
        try {
            stream.close();
        } catch (IOException ignored) {}
    }
}
```

La mayoría de las API de archivos java.io aceptan tanto `String` como `File` como argumentos, por lo que también podría usar

```
File file = new File("Hello world.txt");
FileOutputStream stream = new FileOutputStream(file);
```

Stream vs Writer / Reader API

Los flujos proporcionan el acceso más directo al contenido binario, por lo que cualquier implementación de `InputStream` / `OutputStream` siempre opera en `int` s y `byte` s.

```
// Read a single byte from the stream
int b = inputStream.read();
if (b >= 0) { // A negative value represents the end of the stream, normal values are in the
range 0 - 255
    // Write the byte to another stream
    outputStream.write(b);
}

// Read a chunk
byte[] data = new byte[1024];
int nBytesRead = inputStream.read(data);
if (nBytesRead >= 0) { // A negative value represents end of stream
    // Write the chunk to another stream
    outputStream.write(data, 0, nBytesRead);
}
```

Hay algunas excepciones, probablemente la más notable, `PrintStream` que agrega la "capacidad de imprimir representaciones de varios valores de datos de manera conveniente". Esto permite utilizar `System.out` como `InputStream` binario y como salida textual utilizando métodos como `System.out.println()` .

Además, algunas implementaciones de flujo funcionan como una interfaz para contenidos de alto nivel, como objetos Java (consulte Serialización) o tipos nativos, por ejemplo, `DataOutputStream` / `DataInputStream` .

Con las clases `Writer` y `Reader` , Java también proporciona una API para flujos de caracteres explícitos. Aunque la mayoría de las aplicaciones basarán estas implementaciones en flujos, la API de flujo de caracteres no expone ningún método para contenido binario.

```
// This example uses the platform's default charset, see below
// for a better implementation.

Writer writer = new OutputStreamWriter(System.out);
writer.write("Hello world!");

Reader reader = new InputStreamReader(System.in);
char singleCharacter = reader.read();
```

Siempre que sea necesario codificar caracteres en datos binarios (por ejemplo, al usar las clases `InputStreamWriter` / `OutputStreamWriter`), debe especificar un conjunto de caracteres si no desea depender del conjunto de caracteres predeterminado de la plataforma. En caso de duda, utilice una codificación compatible con Unicode, por ejemplo, UTF-8 que sea compatible con todas las plataformas Java. Por lo tanto, probablemente debería alejarse de clases como `FileWriter` y `FileReader` ya que siempre usan el conjunto de caracteres predeterminado de la plataforma. Una mejor manera de acceder a los archivos utilizando secuencias de caracteres es esta:

```
Charset myCharset = StandardCharsets.UTF_8;

Writer writer = new OutputStreamWriter( new FileOutputStream("test.txt"), myCharset );
writer.write('Ä');
writer.flush();
writer.close();

Reader reader = new InputStreamReader( new FileInputStream("test.txt"), myCharset );
char someUnicodeCharacter = reader.read();
reader.close();
```

Uno de los `Reader` s más utilizados es `BufferedReader` que proporciona un método para leer líneas enteras de texto de otro lector y es probablemente la forma más sencilla de leer una línea de caracteres en línea de secuencia:

```
// Read from baseReader, one line at a time
BufferedReader reader = new BufferedReader( baseReader );
String line;
while((line = reader.readLine()) != null) {
  // Remember: System.out is a stream, not a writer!
  System.out.println(line);
}
```

Leyendo un archivo completo a la vez

```
File f = new File(path);
String content = new Scanner(f).useDelimiter("\\Z").next();
```

\ Z es el símbolo EOF (fin de archivo). Cuando se establece como delimitador, el Escáner leerá el relleno hasta que se alcance el indicador EOF.

Leyendo un archivo con un escáner

Leyendo un archivo línea por línea

```
public class Main {

    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(new File("example.txt"));
            while(scanner.hasNextLine())
            {
                String line = scanner.nextLine();
                //do stuff
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

palabra por palabra

```
public class Main {

    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(new File("example.txt"));
            while(scanner.hasNext())
            {
                String line = scanner.next();
                //do stuff
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

y también puede cambiar el delimitador utilizando el método scanner.useDelimeter ()

Iterando sobre un directorio y filtrando por extensión de archivo

```
    public void iterateAndFilter() throws IOException {
        Path dir = Paths.get("C:/foo/bar");
        PathMatcher imageFileMatcher =
            FileSystems.getDefault().getPathMatcher(
                "regex:.*(?i:jpg|jpeg|png|gif|bmp|jpe|jfif)");

        try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir,
                entry -> imageFileMatcher.matches(entry.getFileName()))) {

            for (Path path : stream) {
                System.out.println(path.getFileName());
```

```
            }
        }
    }
```

Migración de java.io.File a Java 7 NIO (java.nio.file.Path)

Estos ejemplos asumen que ya sabes lo que es NIO de Java 7 en general, y estás acostumbrado a escribir código usando `java.io.File` . Utilice estos ejemplos como un medio para encontrar rápidamente más documentación centrada en NIO para la migración.

Hay mucho más en el NIO de Java 7, como los archivos asignados en memoria o la apertura de un archivo ZIP o JAR utilizando FileSystem . Estos ejemplos solo cubrirán un número limitado de casos de uso básicos.

Como regla básica, si está acostumbrado a realizar una operación de lectura / escritura del sistema de archivos utilizando un método de instancia `java.io.File` , lo encontrará como un método estático dentro de `java.nio.file.Files` .

Apuntar a un camino

```
// -> IO
File file = new File("io.txt");

// -> NIO
Path path = Paths.get("nio.txt");
```

Caminos relativos a otro camino.

```
// Forward slashes can be used in place of backslashes even on a Windows operating system
// -> IO
File folder = new File("C:/");
File fileInFolder = new File(folder, "io.txt");

// -> NIO
Path directory = Paths.get("C:/");
Path pathInDirectory = directory.resolve("nio.txt");
```

Convertir archivo de / a ruta para usar con bibliotecas

```
// -> IO to NIO
Path pathFromFile = new File("io.txt").toPath();

// -> NIO to IO
File fileFromPath = Paths.get("nio.txt").toFile();
```

Compruebe si el archivo existe y elimínelo si existe.

```
// -> IO
if (file.exists()) {
    boolean deleted = file.delete();
    if (!deleted) {
        throw new IOException("Unable to delete file");
    }
}

// -> NIO
Files.deleteIfExists(path);
```

Escribir en un archivo a través de un OutputStream

Hay varias formas de escribir y leer desde un archivo utilizando NIO para diferentes limitaciones de rendimiento y memoria, legibilidad y casos de uso, como `FileChannel` , `Files.write(Path path, byte\[\] bytes, OpenOption... options)` . .. En este ejemplo, solo se cubre `OutputStream` , pero se recomienda encarecidamente que conozca los archivos asignados en memoria y los diversos métodos estáticos disponibles en `java.nio.file.Files` .

```
List<String> lines = Arrays.asList(
        String.valueOf(Calendar.getInstance().getTimeInMillis()),
        "line one",
        "line two");

// -> IO
if (file.exists()) {
    // Note: Not atomic
    throw new IOException("File already exists");
}
try (FileOutputStream outputStream = new FileOutputStream(file)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}

// -> NIO
try (OutputStream outputStream = Files.newOutputStream(path, StandardOpenOption.CREATE_NEW)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}
```

Iterando en cada archivo dentro de una

carpeta

```
// -> IO
for (File selectedFile : folder.listFiles()) {
    // Note: Depending on the number of files in the directory folder.listFiles() may take a
long time to return
    System.out.println((selectedFile.isDirectory() ? "d" : "f") + " " +
selectedFile.getAbsolutePath());
}

// -> NIO
Files.walkFileTree(directory, EnumSet.noneOf(FileVisitOption.class), 1, new
SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult preVisitDirectory(Path selectedPath, BasicFileAttributes attrs)
throws IOException {
        System.out.println("d " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("f " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }
});
```

Iteración de la carpeta recursiva

```
// -> IO
recurseFolder(folder);

// -> NIO
// Note: Symbolic links are NOT followed unless explicitly passed as an argument to
Files.walkFileTree
Files.walkFileTree(directory, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws
IOException {
        System.out.println("d " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("f " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }
});


private static void recurseFolder(File folder) {
    for (File selectedFile : folder.listFiles()) {
```

```
        System.out.println((selectedFile.isDirectory() ? "d" : "f") + " " +
selectedFile.getAbsolutePath());
        if (selectedFile.isDirectory()) {
            // Note: Symbolic links are followed
            recurseFolder(selectedFile);
        }
    }
}
```

Lectura / escritura de archivos usando FileInputStream / FileOutputStream

Escribir en un archivo test.txt:

```
String filepath ="C:\\test.txt";
FileOutputStream fos = null;
try {
      fos = new FileOutputStream(filepath);
      byte[] buffer = "This will be written in test.txt".getBytes();
      fos.write(buffer, 0, buffer.length);
      fos.close();
} catch (FileNotFoundException e) {
      e.printStackTrace();
} catch (IOException e) {
      e.printStackTrace();
} finally{
      if(fos != null)
        fos.close();
}
```

Leer del archivo test.txt:

```
String filepath ="C:\\test.txt";
FileInputStream fis = null;
try {
   fis = new FileInputStream(filepath);
   int length = (int) new File(filepath).length();
   byte[] buffer = new byte[length];
   fis.read(buffer, 0, length);
} catch (FileNotFoundException e) {
     e.printStackTrace();
} catch (IOException e) {
     e.printStackTrace();
} finally{
   if(fis != null)
     fis.close();
}
```

Tenga en cuenta que desde Java 1.7 se introdujo la declaración try-with-resources lo que hizo que la implementación de la operación de lectura / escritura sea mucho más sencilla:

Escribir en un archivo test.txt:

```
String filepath ="C:\\test.txt";
try (FileOutputStream fos = new FileOutputStream(filepath)){
    byte[] buffer = "This will be written in test.txt".getBytes();
    fos.write(buffer, 0, buffer.length);
```

```
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Leer del archivo test.txt:

```
String filepath ="C:\\test.txt";
try (FileInputStream fis = new FileInputStream(filepath)){
    int length = (int) new File(filepath).length();
    byte[] buffer = new byte[length];
    fis.read(buffer, 0, length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Leyendo de un archivo binario

Puede leer un archivo binario utilizando este fragmento de código en todas las versiones recientes de Java:

Java SE 1.4

```
File file = new File("path_to_the_file");
byte[] data = new byte[(int) file.length()];
DataInputStream stream = new DataInputStream(new FileInputStream(file));
stream.readFully(data);
stream.close();
```

Si está utilizando Java 7 o posterior, hay una forma más sencilla de usar la `nio API` :

Java SE 7

```
Path path = Paths.get("path_to_the_file");
byte [] data = Files.readAllBytes(path);
```

Cierre

Un archivo puede ser bloqueado usando la API `FileChannel` que se puede adquirir de los `streams` y `readers` de Input Input.

Ejemplo con `streams`

// Abrir una secuencia de archivos FileInputStream ios = new FileInputStream (nombre de archivo);

```
    // get underlying channel
    FileChannel channel = ios.getChannel();
```

```
    /*
     * try to lock the file. true means whether the lock is shared or not i.e. multiple
processes can acquire a
     * shared lock (for reading only) Using false with readable channel only will generate an
exception. You should
     * use a writable channel (taken from FileOutputStream) when using false. tryLock will
always return immediately
     */
    FileLock lock = channel.tryLock(0, Long.MAX_VALUE, true);

    if (lock == null) {
        System.out.println("Unable to acquire lock");
    } else {
        System.out.println("Lock acquired successfully");
    }

    // you can also use blocking call which will block until a lock is acquired.
    channel.lock();

    // Once you have completed desired operations of file. release the lock
    if (lock != null) {
        lock.release();
    }

    // close the file stream afterwards
    // Example with reader
    RandomAccessFile randomAccessFile = new RandomAccessFile(filename,  "rw");
    FileChannel channel = randomAccessFile.getChannel();
    //repeat the same steps as above but now you can use shared as true or false as the
channel is in read write mode
```

Copiando un archivo usando InputStream y OutputStream

Podemos copiar directamente los datos de una fuente a un sumidero de datos utilizando un bucle. En este ejemplo, estamos leyendo datos de un InputStream y, al mismo tiempo, escribimos en un OutputStream. Una vez que hayamos terminado de leer y escribir, tenemos que cerrar el recurso.

```
public void copy(InputStream source, OutputStream destination) throws IOException {
    try {
        int c;
        while ((c = source.read()) != -1) {
            destination.write(c);
        }
    } finally {
        if (source != null) {
            source.close();
        }
        if (destination != null) {
            destination.close();
        }
    }
}
```

Leyendo un archivo usando Channel y Buffer

`Channel` utiliza un `Buffer` para leer / escribir datos. Un búfer es un contenedor de tamaño fijo donde podemos escribir un bloque de datos a la vez. `Channel` es bastante más rápido que la E / S basada

en flujo.

Para leer los datos de un archivo usando `Channel` necesitamos los siguientes pasos:

1. Necesitamos una instancia de `FileInputStream` . `FileInputStream` tiene un método llamado `getChannel()` que devuelve un canal.
2. Llame al método `getChannel()` de FileInputStream y adquiera Channel.
3. Crear un ByteBuffer. ByteBuffer es un contenedor de tamaño fijo de bytes.
4. El canal tiene un método de lectura y tenemos que proporcionar un ByteBuffer como argumento para este método de lectura. ByteBuffer tiene dos modos: modo de solo lectura y modo de solo escritura. Podemos cambiar el modo usando el método de llamada `flip()` . El búfer tiene una posición, límite y capacidad. Una vez que se crea un búfer con un tamaño fijo, su límite y capacidad son los mismos que el tamaño y la posición comienza desde cero. Mientras se escribe un búfer con datos, su posición aumenta gradualmente. Cambio de modo significa, cambiando la posición. Para leer datos desde el principio de un búfer, tenemos que establecer la posición a cero. método flip () cambiar la posición
5. Cuando llamamos al método de lectura del `Channel` , se llena el búfer usando datos.
6. Si necesitamos leer los datos de `ByteBuffer` , debemos voltear el búfer para cambiar su modo de solo escritura al modo de solo lectura y luego seguir leyendo los datos del búfer.
7. Cuando ya no hay datos para leer, el método de `read()` del canal devuelve 0 o -1.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class FileChannelRead {

public static void main(String[] args) {

   File inputFile = new File("hello.txt");

   if (!inputFile.exists()) {
    System.out.println("The input file doesn't exit.");
    return;
   }

  try {
   FileInputStream fis = new FileInputStream(inputFile);
   FileChannel fileChannel = fis.getChannel();
   ByteBuffer buffer = ByteBuffer.allocate(1024);

   while (fileChannel.read(buffer) > 0) {
    buffer.flip();
    while (buffer.hasRemaining()) {
     byte b = buffer.get();
     System.out.print((char) b);
    }
    buffer.clear();
   }

   fileChannel.close();
  } catch (IOException e) {
   e.printStackTrace();
```

```
  }
 }
}
```

Copiando un archivo usando el canal

Podemos usar el `Channel` para copiar el contenido del archivo más rápido. Para hacerlo, podemos usar el método `transferTo()` de `FileChannel` .

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.channels.FileChannel;

public class FileCopier {

    public static void main(String[] args) {
        File sourceFile = new File("hello.txt");
        File sinkFile = new File("hello2.txt");
        copy(sourceFile, sinkFile);
    }

    public static void copy(File sourceFile, File destFile) {
        if (!sourceFile.exists() || !destFile.exists()) {
            System.out.println("Source or destination file doesn't exist");
            return;
        }


        try (FileChannel srcChannel = new FileInputStream(sourceFile).getChannel();
             FileChannel sinkChanel = new FileOutputStream(destFile).getChannel()) {

            srcChannel.transferTo(0, srcChannel.size(), sinkChanel);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Leyendo un archivo usando BufferedInputStream

La lectura de archivos utilizando un `BufferedInputStream` generalmente es más rápida que `FileInputStream` porque mantiene un búfer interno para almacenar bytes leídos desde la secuencia de entrada subyacente.

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;


public class FileReadingDemo {

    public static void main(String[] args) {
        String source = "hello.txt";
```

```
        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(source))) {
            byte data;
            while ((data = (byte) bis.read()) != -1) {
                System.out.println((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

Escribiendo un archivo usando Channel y Buffer

Para escribir datos en un archivo usando `Channel` necesitamos los siguientes pasos:

1. Primero, necesitamos obtener un objeto de `FileOutputStream`
2. Adquiera `FileChannel` llamando al método `getChannel()` desde `FileOutputStream`
3. Crea un `ByteBuffer` y luego llénalo con datos
4. Luego tenemos que llamar al método `flip()` del `ByteBuffer` y pasarlo como un argumento del método `write()` del `FileChannel`
5. Una vez que hayamos terminado de escribir, tenemos que cerrar el recurso.

```
import java.io.*;
import java.nio.*;
public class FileChannelWrite {

 public static void main(String[] args) {

  File outputFile = new File("hello.txt");
  String text = "I love Bangladesh.";

  try {
   FileOutputStream fos = new FileOutputStream(outputFile);
   FileChannel fileChannel = fos.getChannel();
   byte[] bytes = text.getBytes();
   ByteBuffer buffer = ByteBuffer.wrap(bytes);
   fileChannel.write(buffer);
   fileChannel.close();
  } catch (java.io.IOException e) {
   e.printStackTrace();
  }
 }
}
```

Escribiendo un archivo usando PrintStream

Podemos usar la clase `PrintStream` para escribir un archivo. Tiene varios métodos que le permiten imprimir cualquier valor de tipo de datos. `println()` método `println()` agrega una nueva línea. Una vez que hayamos terminado de imprimir, tenemos que vaciar el `PrintStream` .

```
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.time.LocalDate;
```

```
public class FileWritingDemo {
    public static void main(String[] args) {
        String destination = "file1.txt";

        try(PrintStream ps = new PrintStream(destination)){
            ps.println("Stackoverflow documentation seems fun.");
            ps.println();
            ps.println("I love Java!");
            ps.printf("Today is: %1$tm/%1$td/%1$tY", LocalDate.now());

            ps.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

    }
}
```

Iterar sobre un directorio de subdirectorios de impresión en él.

```
  public void iterate(final String dirPath) throws IOException {
    final DirectoryStream<Path> paths = Files.newDirectoryStream(Paths.get(dirPath));
    for (final Path path : paths) {
      if (Files.isDirectory(path)) {
        System.out.println(path.getFileName());
      }
    }
  }
```

Agregando Directorios

Para crear un nuevo directorio a partir de una instancia de un `File` , deberá utilizar uno de los dos métodos siguientes: `mkdirs()` o `mkdir()` .

- `mkdir()` : crea el directorio nombrado por este nombre de ruta abstracto. (fuente)
- `mkdirs()` : crea el directorio nombrado por esta ruta de acceso abstracta, incluidos los directorios primarios necesarios pero no existentes. Tenga en cuenta que si esta operación falla, es posible que haya logrado crear algunos de los directorios principales necesarios. (fuente)

Nota: `createNewFile()` no creará un nuevo directorio solo un archivo.

```
File singleDir = new File("C:/Users/SomeUser/Desktop/A New Folder/");

File multiDir = new File("C:/Users/SomeUser/Desktop/A New Folder 2/Another Folder/");

// assume that neither "A New Folder" or "A New Folder 2" exist

singleDir.createNewFile(); // will make a new file called "A New Folder.file"
singleDir.mkdir(); // will make the directory
singleDir.mkdirs(); // will make the directory

multiDir.createNewFile(); // will throw a IOException
multiDir.mkdir(); // will not work
```

```
multiDir.mkdirs(); // will make the directory
```

Bloqueo o redireccionamiento de salida / error estándar

A veces, una biblioteca de terceros mal diseñada escribirá diagnósticos no deseados en las `System.err` `System.out` o `System.err` . Las soluciones recomendadas para esto serían encontrar una mejor biblioteca o (en el caso de código abierto) solucionar el problema y contribuir con un parche para los desarrolladores.

Si las soluciones anteriores no son factibles, entonces debería considerar redirigir los flujos.

Redireccionamiento en la línea de comando.

En un sistema UNIX, Linux o MacOSX se puede hacer desde el shell usando `>` redirección. Por ejemplo:

```
$ java -jar app.jar arg1 arg2 > /dev/null 2>&1
$ java -jar app.jar arg1 arg2 > out.log 2> error.log
```

El primero redirige la salida estándar y el error estándar a "/ dev / null", que desecha todo lo escrito en esas secuencias. El segundo redirige la salida estándar a "out.log" y el error estándar a "error.log".

(Para obtener más información sobre la redirección, consulte la documentación del shell de comandos que está utilizando. Se aplican consejos similares a Windows).

Alternativamente, puede implementar la redirección en un script de envoltura o un archivo por lotes que inicie la aplicación Java.

Redirección dentro de una aplicación Java

También es posible redirigir las secuencias *dentro de* una aplicación Java usando `System.setOut()` y `System.setErr()` . Por ejemplo, el siguiente fragmento de código redirige la salida estándar y el error estándar a 2 archivos de registro:

```
System.setOut(new PrintStream(new FileOutputStream(new File("out.log"))));
System.setErr(new PrintStream(new FileOutputStream(new File("err.log"))));
```

Si desea deshacerse de la salida por completo, puede crear una secuencia de salida que "escribe" en un descriptor de archivo no válido. Esto es funcionalmente equivalente a escribir en "/ dev / null" en UNIX.

```
System.setOut(new PrintStream(new FileOutputStream(new FileDescriptor())));
System.setErr(new PrintStream(new FileOutputStream(new FileDescriptor())));
```

Precaución: tenga cuidado con el uso de `setOut` y `setErr` :

1. La redirección afectará a toda la JVM.
2. Al hacer esto, está quitando la capacidad del usuario para redirigir los flujos desde la línea

de comandos.

Accediendo a los contenidos de un archivo ZIP.

La API del sistema de archivos de Java 7 permite leer y agregar entradas desde o hacia un archivo Zip usando la API del archivo NIO de Java de la misma manera que opera en cualquier otro sistema de archivos.

FileSystem es un recurso que debe cerrarse correctamente después de su uso, por lo tanto, debe usarse el bloque try-with-resources.

Leyendo de un archivo existente

```
Path pathToZip = Paths.get("path/to/file.zip");
try(FileSystem zipFs = FileSystems.newFileSystem(pathToZip, null)) {
  Path root = zipFs.getPath("/");
  ... //access the content of the zip file same as ordinary files
} catch(IOException ex) {
  ex.printStackTrace();
}
```

Creando un nuevo archivo

```
Map<String, String> env = new HashMap<>();
env.put("create", "true"); //required for creating a new zip file
env.put("encoding", "UTF-8"); //optional: default is UTF-8
URI uri = URI.create("jar:file:/path/to/file.zip");
try (FileSystem zipfs = FileSystems.newFileSystem(uri, env)) {
  Path newFile = zipFs.getPath("/newFile.txt");
  //writing to file
  Files.write(newFile, "Hello world".getBytes());
} catch(IOException ex) {
  ex.printStackTrace();
}
```

Lea Archivo I / O en línea: https://riptutorial.com/es/java/topic/93/archivo-i---o

Capítulo 14: Archivos JAR Multi-Release

Introducción

Una de las características introducidas en Java 9 es el Jar de lanzamiento múltiple (MRJAR), que permite agrupar código dirigido a múltiples lanzamientos de Java dentro del mismo archivo Jar. La característica se especifica en JEP 238 .

Examples

Ejemplo de contenido de un archivo Jar multi-release.

Al establecer `Multi-Release: true` en el archivo MANIFEST.MF, el archivo Jar se convierte en un Jar multi-release y el tiempo de ejecución de Java (siempre que sea compatible con el formato MRJAR) seleccionará las versiones apropiadas de las clases dependiendo de la versión principal actual .

La estructura de dicho Jar es la siguiente:

```
jar root
  - A.class
  - B.class
  - C.class
  - D.class
  - META-INF
     - versions
        - 9
           - A.class
           - B.class
        - 10
           - A.class
```

- En JDKs <9, solo las clases en la entrada raíz son visibles para el tiempo de ejecución de Java.
- En un JDK 9, las clases A y B se cargarán desde el directorio `root/META-INF/versions/9` , mientras que C y D se cargarán desde la entrada base.
- En un JDK 10, la clase A se cargaría desde el directorio `root/META-INF/versions/10` .

Creando un Jar multi-release usando la herramienta jar

El comando `jar` se puede usar para crear un Jar de varias versiones que contiene dos versiones de la misma clase compilada para Java 8 y Java 9, aunque con una advertencia que indica que las clases son idénticas:

```
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo
Warning: entry META-INF/versions/9/demo/SampleClass.class contains a class that
is identical to an entry already in the jar
```

La opción `--release 9` le dice a `jar` que incluya todo lo que sigue (el paquete de `demo` dentro del directorio `sampleproject-9`) dentro de una entrada versionada en MRJAR, es decir, bajo `root/META-INF/versions/9` . El resultado son los siguientes contenidos:

```
jar root
  - demo
     - SampleClass.class
  - META-INF
     - versions
        - 9
           - demo
              - SampleClass.class
```

Ahora creamos una clase llamada Main que imprime la URL de `SampleClass` y la agregamos para la versión Java 9:

```
package demo;

import java.net.URL;

public class Main {

    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}
```

Si compilamos esta clase y volvemos a ejecutar el comando jar, obtenemos un error:

```
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demoentry: META-INF/versions/9/demo/Main.class, contains a new public class
not found in base entries
Warning: entry META-INF/versions/9/demo/Main.java, multiple resources with same name
Warning: entry META-INF/versions/9/demo/SampleClass.class contains a class that
is identical to an entry already in the jar
invalid multi-release jar file MR.jar deleted
```

La razón es que la herramienta `jar` impide agregar clases públicas a las entradas versionadas si no se agregan a las entradas base también. Esto se hace para que MRJAR exponga la misma API pública para las diferentes versiones de Java. Tenga en cuenta que en tiempo de ejecución, esta regla no es necesaria. Solo se puede aplicar con herramientas como el `jar` . En este caso particular, el propósito de `Main` es ejecutar código de ejemplo, por lo que simplemente podemos agregar una copia en la entrada base. Si la clase formara parte de una implementación más nueva que solo necesitamos para Java 9, se podría hacer no pública.

Para agregar `Main` a la entrada raíz, primero debemos compilarlo para apuntar a una versión anterior a Java 9. Esto se puede hacer usando la nueva opción `--release` de `javac` :

```
C:\Users\manouti\sampleproject-base\demo>javac --release 8 Main.java
C:\Users\manouti\sampleproject-base\demo>cd ../..
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo
```

La ejecución de la clase Main muestra que SampleClass se carga desde el directorio versionado:

```
C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class
```

URL de una clase cargada dentro de un Jar multi-release

Teniendo en cuenta el siguiente Jar multi-lanzamiento:

```
jar root
  - demo
     - SampleClass.class
  - META-INF
     - versions
        - 9
           - demo
              - SampleClass.class
```

La siguiente clase imprime la URL de `SampleClass` :

```
package demo;

import java.net.URL;

public class Main {

    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}
```

Si la clase se compila y se agrega en la entrada versionada para Java 9 en MRJAR, ejecutarla resultaría en:

```
C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class
```

Lea Archivos JAR Multi-Release en línea: https://riptutorial.com/es/java/topic/9866/archivos-jar-multi-release

Capítulo 15: Arrays

Introducción

Las matrices permiten el almacenamiento y la recuperación de una cantidad arbitraria de valores. Son análogos a los vectores en matemáticas. Las matrices de matrices son análogas a las matrices y actúan como matrices multidimensionales. Las matrices pueden almacenar cualquier tipo de datos: primitivas como `int` o tipos de referencia como `Object` .

Sintaxis

- `ArrayType[] myArray;` // Declarando matrices
- `ArrayType myArray[];` // Otra sintaxis válida (menos utilizada y desaconsejada)
- `ArrayType[][][] myArray;` // Declaración de matrices irregulares multidimensionales (repetición [] s)
- `ArrayType myVar = myArray[index];` // Elemento de acceso (lectura) en el índice
- `myArray[index] = value;` // Asignar valor al `index` de posición de la matriz
- `ArrayType[] myArray = new ArrayType[arrayLength];` // Sintaxis de inicialización de matriz
- `int[] ints = {1, 2, 3};` // Sintaxis de inicialización de la matriz con los valores proporcionados, la longitud se deduce del número de valores proporcionados: {[valor1 [, valor2] *]}
- `new int[]{4, -5, 6} // Can be used as argument, without a local variable`
- `int[] ints = new int[3]; // same as {0, 0, 0}`
- `int[][] ints = {{1, 2}, {3}, null};` // Inicialización de matriz multidimensional. int [] extiende Object (y también lo hace anyType []), por lo que null es un valor válido.

Parámetros

Parámetro	Detalles
`ArrayType`	Tipo de la matriz. Puede ser primitivo (`int` , `long` , `byte`) u Objetos (`String` , `MyObject` , etc.).
índice	El índice se refiere a la posición de un determinado objeto en una matriz.
longitud	Cada matriz, cuando se crea, necesita una longitud determinada especificada. Esto se hace al crear una matriz vacía (`new int[3]`) o implícito al especificar valores (`{1, 2, 3}`).

Examples

Creación e inicialización de matrices

Casos basicos

```
int[]   numbers1 = new int[3];                  // Array for 3 int values, default value is 0
int[]   numbers2 = { 1, 2, 3 };                 // Array literal of 3 int values
int[]   numbers3 = new int[] { 1, 2, 3 };       // Array of 3 int values initialized
int[][] numbers4 = { { 1, 2 }, { 3, 4, 5 } };   // Jagged array literal
int[][] numbers5 = new int[5][];                // Jagged array, one dimension 5 long
int[][] numbers6 = new int[5][4];               // Multidimensional array: 5x4
```

Las matrices se pueden crear utilizando cualquier tipo de referencia o primitivo.

```
float[]  boats = new float[5];          // Array of five 32-bit floating point numbers.
double[] header = new double[] { 4.56, 332.267, 7.0, 0.3367, 10.0 };
                                        // Array of five 64-bit floating point numbers.
String[] theory = new String[] { "a", "b", "c" };
                                        // Array of three strings (reference type).
Object[] dArt = new Object[] { new Object(), "We love Stack Overflow.", new Integer(3) };
                                        // Array of three Objects (reference type).
```

Para el último ejemplo, tenga en cuenta que los subtipos del tipo de matriz declarada están permitidos en la matriz.

Las matrices para tipos definidos por el usuario también pueden construirse de manera similar a los tipos primitivos

```
UserDefinedClass[] udType = new UserDefinedClass[5];
```

Arrays, colecciones y corrientes

Java SE 1.2

```
// Parameters require objects, not primitives

// Auto-boxing happening for int 127 here
Integer[]       initial        = { 127, Integer.valueOf( 42 ) };
List<Integer>   toList         = Arrays.asList( initial );  // Fixed size!

// Note: Works with all collections
Integer[]       fromCollection = toList.toArray( new Integer[toList.size()] );

//Java doesn't allow you to create an array of a parameterized type
List<String>[]  list = new ArrayList<String>[2];  // Compilation error!
```

Java SE 8

```
// Streams - JDK 8+
Stream<Integer> toStream       = Arrays.stream( initial );
Integer[]       fromStream     = toStream.toArray( Integer[]::new );
```

Introducción

Una *matriz* es una estructura de datos que contiene un número fijo de valores primitivos **o** referencias a instancias de objetos.

Cada elemento de una matriz se denomina elemento, y se accede a cada elemento por su índice numérico. La longitud de una matriz se establece cuando se crea la matriz:

```
int size = 42;
int[] array = new int[size];
```

El tamaño de una matriz se fija en el tiempo de ejecución cuando se inicializa. No se puede cambiar después de la inicialización. Si el tamaño debe ser mutable en el tiempo de ejecución, se debe usar una clase de `Collection` como `ArrayList` su lugar. `ArrayList` almacena elementos en una matriz y admite el cambio de tamaño asignando una nueva matriz y copiando elementos de la matriz antigua.

Si la matriz es de un tipo primitivo, es decir

```
int[] array1 = { 1,2,3 };
int[] array2 = new int[10];
```

Los valores se almacenan en la propia matriz. En ausencia de un inicializador (como en la `array2` anterior), el valor predeterminado asignado a cada elemento es `0` (cero).

Si el tipo de matriz es una referencia de objeto, como en

```
SomeClassOrInterface[] array = new SomeClassOrInterface[10];
```

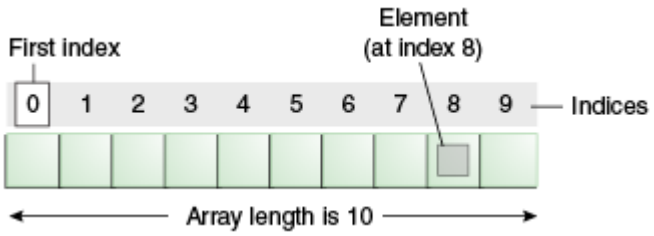
entonces la matriz contiene *referencias* a objetos de tipo `SomeClassOrInterface` . Esas referencias pueden referirse a una instancia de `SomeClassOrInterface` *o cualquier subclase (para clases) o clase de implementación (para interfaces) de* `SomeClassOrInterface` . Si la declaración de la matriz no tiene inicializador, el valor predeterminado de `null` se asigna a cada elemento.

Debido a que todas las matrices están `int` indexadas, el tamaño de una matriz debe ser especificado por una `int` . El tamaño de la matriz no se puede especificar como un `long` :

```
long size = 23L;
int[] array = new int[size]; // Compile-time error:
                             // incompatible types: possible lossy conversion from
                             // long to int
```

Las matrices utilizan un sistema de **índice de base cero** , lo que significa que la indexación comienza en `0` y termina en `length - 1` .

Por ejemplo, la siguiente imagen representa una matriz con tamaño `10` . Aquí, el primer elemento está en el índice `0` y el último elemento está en el índice `9` , en lugar de que el primer elemento esté en el índice `1` y el último elemento en el índice `10` (consulte la figura a continuación).



Los accesos a elementos de matrices se realizan en **tiempo constante** . Eso significa que acceder al primer elemento de la matriz tiene el mismo costo (en el tiempo) de acceder al segundo elemento, al tercer elemento y así sucesivamente.

Java ofrece varias formas de definir e inicializar matrices, incluidas **las** notaciones **literales** y de **constructor** . Al declarar matrices utilizando el `new Type[length]` constructor `new Type[length]` , cada elemento se inicializará con los siguientes valores predeterminados:

- `0` para tipos numéricos primitivos : `byte` , `short` , `int` , `long` , `float` y `double` .
- `'\u0000'` (carácter nulo) para el tipo `char` .
- `false` para el tipo `boolean` .
- `null` para los tipos de referencia .

Creación e inicialización de matrices de tipos primitivos.

```
int[] array1 = new int[] { 1, 2, 3 }; // Create an array with new operator and
                                      // array initializer.
int[] array2 = { 1, 2, 3 };           // Shortcut syntax with array initializer.
int[] array3 = new int[3];            // Equivalent to { 0, 0, 0 }
int[] array4 = null;                  // The array itself is an object, so it
                                      // can be set as null.
```

Al declarar una matriz, `[]` aparecerá como parte del tipo al comienzo de la declaración (después del nombre del tipo), o como parte del declarador para una variable en particular (después del nombre de la variable), o ambos:

```
int array5[];        /* equivalent to */  int[] array5;
int a, b[], c[][];   /* equivalent to */  int a; int[] b; int[][] c;
int[] a, b[];        /* equivalent to */  int[] a; int[][] b;
int a, []b, c[][];   /* Compilation Error, because [] is not part of the type at beginning
                        of the declaration, rather it is before 'b'. */
// The same rules apply when declaring a method that returns an array:
int foo()[] { ... } /* equivalent to */  int[] foo() { ... }
```

En el siguiente ejemplo, ambas declaraciones son correctas y pueden compilarse y ejecutarse sin ningún problema. Sin embargo, tanto la Convención de codificación de Java como la Guía de estilo de Google Java desalientan el formulario entre corchetes después del nombre de la variable: los corchetes identifican el tipo de matriz y deben aparecer con la designación de tipo . Lo mismo se debe utilizar para las firmas de retorno de método.

```
float array[]; /* and */ int foo()[] { ... } /* are discouraged */
float[] array; /* and */ int[] foo() { ... } /* are encouraged */
```

El tipo desalentado está pensado para adaptarse a los usuarios de C en transición , que están familiarizados con la sintaxis de C que tiene los corchetes después del nombre de la variable.

En Java, es posible tener matrices de tamaño `0` :

```
int[] array = new int[0]; // Compiles and runs fine.
int[] array2 = {};        // Equivalent syntax.
```

Sin embargo, como se trata de una matriz vacía, no se pueden leer ni asignar elementos de ella:

```
array[0] = 1;     // Throws java.lang.ArrayIndexOutOfBoundsException.
int i = array2[0]; // Also throws ArrayIndexOutOfBoundsException.
```

Estas matrices vacías suelen ser útiles como valores de retorno, por lo que el código de llamada solo tiene que preocuparse de tratar con una matriz, en lugar de un valor `null` potencial que puede llevar a una `NullPointerException` .

La longitud de una matriz debe ser un entero no negativo:

```
int[] array = new int[-1]; // Throws java.lang.NegativeArraySizeException
```

El tamaño de la matriz se puede determinar utilizando un campo final público llamado `length` :

```
System.out.println(array.length); // Prints 0 in this case.
```

Nota : `array.length` devuelve el tamaño real de la matriz y no el número de elementos de la matriz a los que se asignó un valor, a diferencia de `ArrayList.size()` que devuelve la cantidad de elementos de la matriz a los que se asignó un valor.

Creando e inicializando arrays multidimensionales.

La forma más sencilla de crear una matriz multidimensional es la siguiente:

```
int[][] a = new int[2][3];
```

Creará dos matrices `int` tres longitudes: `a[0]` y `a[1]` . Esto es muy similar a la inicialización clásica de estilo C de matrices multidimensionales rectangulares.

Puedes crear e inicializar al mismo tiempo:

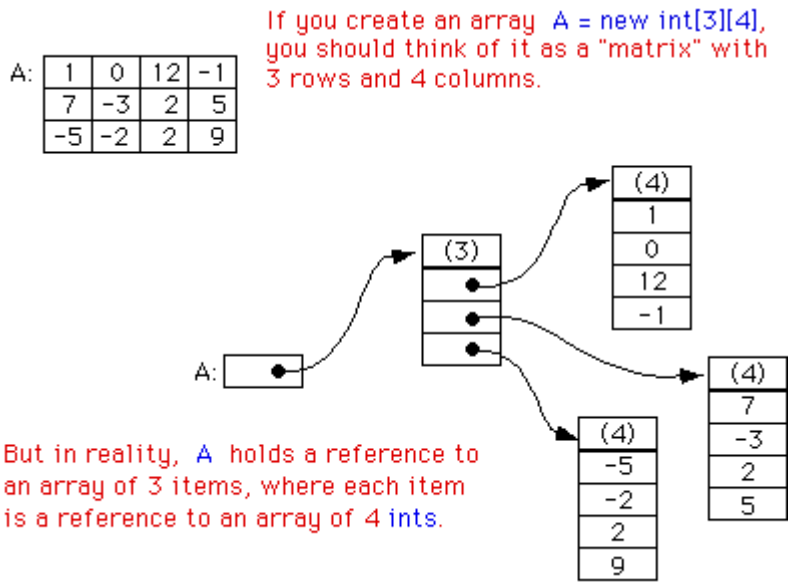
```
int[][] a = { {1, 2}, {3, 4}, {5, 6} };
```

A diferencia de C , donde solo se admiten matrices multidimensionales rectangulares, las matrices internas no tienen que ser de la misma longitud, ni siquiera definidas:

```
int[][] a = { {1}, {2, 3}, null };
```

Aquí, `a[0]` es una matriz `int` una longitud, mientras que `a[1]` es una matriz `int` dos longitudes y `a[2]` es `null` . Las matrices de este tipo se denominan matrices irregulares o matrices irregulares , es decir, son matrices de matrices. Las matrices multidimensionales en Java se implementan como matrices de matrices, es decir, `array[i][j][k]` es equivalente a `((array[i])[j])[k]` . A diferencia de C # , la `array[i,j]` sintaxis `array[i,j]` no es compatible con Java.

Representación de matriz multidimensional en Java



Fuente - Live en Ideone

Creación e inicialización de matrices de tipos de referencia

```
String[] array6 = new String[] { "Laurel", "Hardy" }; // Create an array with new
                                                       // operator and array initializer.
String[] array7 = { "Laurel", "Hardy" };               // Shortcut syntax with array
                                                       // initializer.
String[] array8 = new String[3];                       // { null, null, null }
String[] array9 = null;                                // null
```

Vivir en Ideone

Además de los `String` literales y primitivas mostradas anteriormente, la sintaxis de acceso directo

para array de inicialización también trabaja con canónicas `Object` tipos:

```
Object[] array10 = { new Object(), new Object() };
```

Debido a que las matrices son covariantes, una matriz de tipo de referencia se puede inicializar como una matriz de una subclase, aunque se `ArrayStoreException` una `ArrayStoreException` si intenta establecer un elemento en algo diferente a una `String` :

```
Object[] array11 = new String[] { "foo", "bar", "baz" };
array11[1] = "qux"; // fine
array11[1] = new StringBuilder(); // throws ArrayStoreException
```

La sintaxis de acceso directo no se puede usar para esto porque la sintaxis de acceso directo tendría un tipo implícito de `Object[]` .

Una matriz se puede inicializar con cero elementos utilizando `String[] emptyArray = new String[0]` . Por ejemplo, una matriz con una longitud cero como esta se usa para crear una `Array` partir de una `Collection` cuando el método necesita el tipo de tiempo de ejecución de un objeto.

Tanto en los tipos primitivos como en los de referencia, una inicialización de matriz vacía (por ejemplo, `String[] array8 = new String[3]`) inicializará la matriz con el valor predeterminado para cada tipo de datos .

Creando e inicializando arrays genéricos

En las clases genéricas, las matrices de tipos genéricos **no se** pueden inicializar de esta manera debido al borrado de tipos :

```
public class MyGenericClass<T> {
    private T[] a;

    public MyGenericClass() {
        a = new T[5]; // Compile time error: generic array creation
    }
}
```

En su lugar, se pueden crear utilizando uno de los siguientes métodos: (tenga en cuenta que estos generarán advertencias sin marcar)

1. Al crear una matriz de `Object` y convertirla en el tipo genérico:

   ```
   a = (T[]) new Object[5];
   ```

   Este es el método más simple, pero como la matriz subyacente aún es del tipo `Object[]` , este método no proporciona seguridad de tipo. Por lo tanto, este método de crear una matriz se usa mejor solo dentro de la clase genérica, no se expone públicamente.

2. Al usar `Array.newInstance` con un parámetro de clase:

```
public MyGenericClass(Class<T> clazz) {
    a = (T[]) Array.newInstance(clazz, 5);
}
```

Aquí la clase de `T` tiene que pasarse explícitamente al constructor. El tipo de retorno de `Array.newInstance` es siempre `Object` . Sin embargo, este método es más seguro porque la matriz recién creada siempre es del tipo `T[]` y, por lo tanto, se puede externalizar de manera segura.

Rellenar una matriz después de la inicialización

Java SE 1.2

`Arrays.fill()` se puede usar para llenar una matriz con **el mismo valor** después de la inicialización:

```
Arrays.fill(array8, "abc");        // { "abc", "abc", "abc" }
```

Vivir en Ideone

`fill()` también puede asignar un valor a cada elemento del rango especificado de la matriz:

```
Arrays.fill(array8, 1, 2, "aaa");  // Placing "aaa" from index 1 to 2.
```

Vivir en Ideone

Java SE 8

Desde la versión 8 de Java, el método `setAll` y su `parallelSetAll` `Concurrent` `setAll` se pueden usar para establecer cada elemento de una matriz en valores generados. A estos métodos se les pasa una función de generador que acepta un índice y devuelve el valor deseado para esa posición.

El siguiente ejemplo crea una matriz de enteros y establece todos sus elementos en su valor de índice respectivo:

```
int[] array = new int[5];
Arrays.setAll(array, i -> i); // The array becomes { 0, 1, 2, 3, 4 }.
```

Vivir en Ideone

Declaración separada e inicialización de matrices.

El valor de un índice para un elemento de matriz debe ser un número entero (0, 1, 2, 3, 4, ...) y menor que la longitud de la matriz (los índices se basan en cero). De lo contrario, se lanzará una ArrayIndexOutOfBoundsException :

```
int[] array9;              // Array declaration - uninitialized
array9 = new int[3];       // Initialize array  - { 0, 0, 0 }
array9[0] = 10;            // Set index 0 value - { 10, 0, 0 }
array9[1] = 20;            // Set index 1 value - { 10, 20, 0 }
array9[2] = 30;            // Set index 2 value - { 10, 20, 30 }
```

Es posible que las matrices no se puedan reinicializar con la sintaxis de acceso directo del inicializador de matriz

No es posible reinicializar una matriz a través de una sintaxis de acceso directo con un inicializador de matriz, ya que un inicializador de matriz solo se puede especificar en una declaración de campo o declaración de variable local, o como parte de una expresión de creación de matriz.

Sin embargo, es posible crear una nueva matriz y asignarla a la variable que se utiliza para hacer referencia a la antigua matriz. Si bien esto hace que la matriz referenciada por esa variable se reinicialice, el contenido de la variable es una matriz completamente nueva. Para hacer esto, el `new` operador se puede usar con un inicializador de matriz y asignarse a la variable de matriz:

```
// First initialization of array
int[] array = new int[] { 1, 2, 3 };

// Prints "1 2 3 ".
for (int i : array) {
    System.out.print(i + " ");
}

// Re-initializes array to a new int[] array.
array = new int[] { 4, 5, 6 };

// Prints "4 5 6 ".
for (int i : array) {
    System.out.print(i + " ");
}

array = { 1, 2, 3, 4 }; // Compile-time error! Can't re-initialize an array via shortcut
                        // syntax with array initializer.
```

Vivir en Ideone

Creando una matriz de una colección

Dos métodos en `java.util.Collection` crean una matriz de una colección:

- `Object[] toArray()`
- `<T> T[] toArray(T[] a)`

`Object[] toArray()` se puede utilizar de la siguiente manera:

Java SE 5

```
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// although set is a Set<String>, toArray() returns an Object[] not a String[]
Object[] objectArray = set.toArray();
```

`<T> T[] toArray(T[] a)` se puede utilizar de la siguiente manera:

Java SE 5

```
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// The array does not need to be created up front with the correct size.
// Only the array type matters. (If the size is wrong, a new array will
// be created with the same type.)
String[] stringArray = set.toArray(new String[0]);

// If you supply an array of the same size as collection or bigger, it
// will be populated with collection values and returned (new array
// won't be allocated)
String[] stringArray2 = set.toArray(new String[set.size()]);
```

La diferencia entre ellos es más que tener resultados tipificados frente a los no tipificados. Su rendimiento también puede diferir (para más detalles, lea esta sección de análisis de rendimiento):

- `Object[] toArray()` utiliza una `arraycopy` vectorizada, que es mucho más rápida que la `arraycopy` comprobación de `arraycopy` utilizada en `T[] toArray(T[] a)` .
- `T[] toArray(new T[non-zero-size])` necesita poner a cero la matriz en el tiempo de ejecución, mientras que `T[] toArray(new T[0])` no. Tal evitación hace que la última llamada sea más rápida que la anterior. Análisis detallado aquí: Arrays of Wisdom of the Ancients .

Java SE 8

A partir de Java SE 8+, donde se introdujo el concepto de `Stream` , es posible utilizar el `Stream` producido por la colección para crear una nueva matriz utilizando el método `Stream.toArray` .

```
String[] strings = list.stream().toArray(String[]::new);
```

Ejemplos tomados de dos respuestas (1 , 2) a Convertir 'ArrayList a' String [] 'en Java en desbordamiento de pila.

Arrays a una cadena

Java SE 5

Desde Java 1.5, puede obtener una representación en `String` del contenido de la matriz especificada sin iterar sobre todos sus elementos. Simplemente use `Arrays.toString(Object[])` o `Arrays.deepToString(Object[])` para arreglos multidimensionales:

```
int[] arr = {1, 2, 3, 4, 5};
System.out.println(Arrays.toString(arr));      // [1, 2, 3, 4, 5]

int[][] arr = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
System.out.println(Arrays.deepToString(arr));  // [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`Arrays.toString()` método `Arrays.toString()` utiliza el método `Object.toString()` para producir valores de `String` de cada elemento de la matriz, además de la matriz de tipo primitiva, puede usarse para todo tipo de matrices. Por ejemplo:

```
public class Cat { /* implicitly extends Object */
    @Override
    public String toString() {
      return "CAT!";
    }
}

Cat[] arr = { new Cat(), new Cat() };
System.out.println(Arrays.toString(arr));        // [CAT!, CAT!]
```

Si no existe una `toString()` anulada para la clase, se utilizará la `toString()` de `Object` heredada. Por lo general, la salida no es muy útil, por ejemplo:

```
public class Dog {
    /* implicitly extends Object */
}

Dog[] arr = { new Dog() };
System.out.println(Arrays.toString(arr));        // [Dog@17ed40e0]
```

Creación de una lista a partir de una matriz

El método `Arrays.asList()` se puede usar para devolver una `List` tamaño fijo que contiene los elementos de la matriz dada. La `List` resultante será del mismo tipo de parámetro que el tipo base de la matriz.

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = Arrays.asList(stringArray);
```

Nota : esta lista está respaldada por (*una vista* de) la matriz original, lo que significa que

cualquier cambio en la lista cambiará la matriz y viceversa. Sin embargo, los cambios en la lista que cambiarían su tamaño (y por lo tanto la longitud de la matriz) generarán una excepción.

Para crear una copia de la lista, use el constructor de `java.util.ArrayList` tomando una `Collection` como un argumento:

Java SE 5

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<String>(Arrays.asList(stringArray));
```

Java SE 7

En Java SE 7 y versiones posteriores, se pueden usar un par de corchetes angulares `<>` (conjunto vacío de argumentos de tipo), lo que se denomina Diamante . El compilador puede determinar los argumentos de tipo del contexto. Esto significa que la información de tipo se puede omitir al llamar al constructor de `ArrayList` y se deducirá automáticamente durante la compilación. Esto se denomina Inferencia de tipos, que forma parte de Java Generics .

```
// Using Arrays.asList()

String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<>(Arrays.asList(stringArray));

// Using ArrayList.addAll()

String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
list.addAll(Arrays.asList(stringArray));

// Using Collections.addAll()

String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, stringArray);
```

Un punto que vale la pena mencionar sobre el Diamante es que no se puede usar con Clases Anónimas .

Java SE 8

```
// Using Streams

int[] ints = {1, 2, 3};
List<Integer> list = Arrays.stream(ints).boxed().collect(Collectors.toList());

String[] stringArray = {"foo", "bar", "baz"};
List<Object> list = Arrays.stream(stringArray).collect(Collectors.toList());
```

Notas importantes relacionadas con el uso del método Arrays.asList ()

- Este método devuelve `List` , que es una instancia de `Arrays$ArrayList` (clase interna estática de `Arrays`) y no `java.util.ArrayList` . La `List` resultante es de tamaño fijo. Eso significa que agregar o quitar elementos no es compatible y emitirá una `UnsupportedOperationException` :

```
stringList.add("something"); // throws java.lang.UnsupportedOperationException
```

- Se puede crear una nueva `List` pasando una `List` respaldada por una matriz al constructor de una Nueva `List` . Esto crea una nueva copia de los datos, que tiene un tamaño variable y que no está respaldado por la matriz original:

```
List<String> modifiableList = new ArrayList<>(Arrays.asList("foo", "bar"));
```

- Llamar a `<T> List<T> asList(T... a)` en una matriz primitiva, como `int[]` , producirá una `List<int[]>` cuyo único elemento es la matriz primitiva de origen en lugar de los elementos reales de la matriz de origen.

  El motivo de este comportamiento es que los tipos primitivos no pueden utilizarse en lugar de los parámetros de tipo genérico, por lo que la matriz primitiva completa reemplaza el parámetro de tipo genérico en este caso. Para convertir una matriz primitiva en una `List` , en primer lugar, convierta la matriz primitiva en una matriz del tipo de envoltura correspondiente (es decir, llame a `Arrays.asList` en un `Integer[]` lugar de a `int[]`).

  Por lo tanto, esto imprimirá `false` :

```
int[] arr = {1, 2, 3};       // primitive array of int
System.out.println(Arrays.asList(arr).contains(1));
```

  Ver demostración

  Por otro lado, esto se imprimirá `true` :

```
Integer[] arr = {1, 2, 3};  // object array of Integer (wrapper for int)
System.out.println(Arrays.asList(arr).contains(1));
```

  Ver demostración

  Esto también se imprimirá `true` , porque la matriz se interpretará como un `Integer[]`):

```
System.out.println(Arrays.asList(1,2,3).contains(1));
```

  Ver demostración

Matrices multidimensionales y dentadas

Es posible definir una matriz con más de una dimensión. En lugar de acceder al proporcionar un índice único, se accede a una matriz multidimensional especificando un índice para cada dimensión.

La declaración de matriz multidimensional se puede hacer agregando `[]` para cada dimensión a una decoración de matriz regular. Por ejemplo, para hacer una matriz `int` bidimensional, agregue otro conjunto de corchetes a la declaración, como `int[][]` . Esto continúa para matrices tridimensionales (`int[][][]`) y así sucesivamente.

Para definir una matriz bidimensional con tres filas y tres columnas:

```
int rows = 3;
int columns = 3;
int[][] table = new int[rows][columns];
```

La matriz se puede indexar y asignarle valores con esta construcción. Tenga en cuenta que los valores no asignados son los valores predeterminados para el tipo de una matriz, en este caso `0` para `int` .

```
table[0][0] = 0;
table[0][1] = 1;
table[0][2] = 2;
```

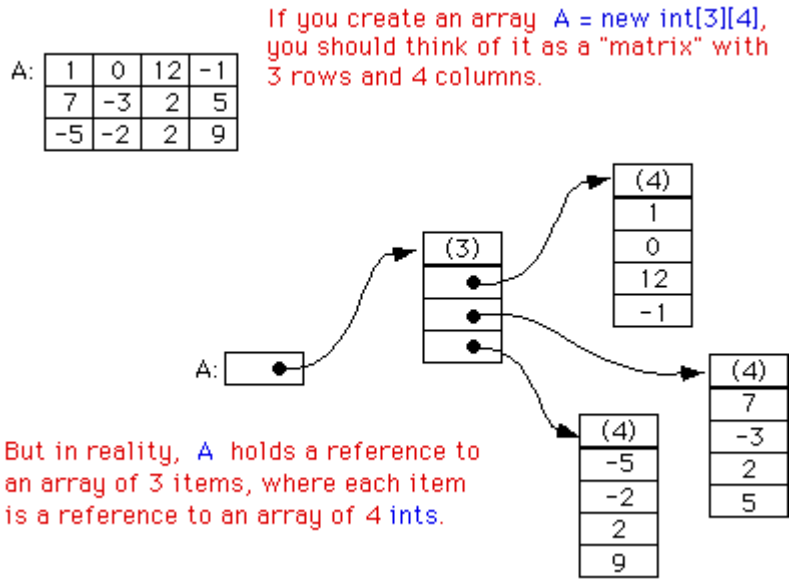
También es posible crear una instancia de una dimensión a la vez, e incluso hacer matrices no rectangulares. Estos se conocen más comúnmente como **matrices dentadas** .

```
int[][] nonRect = new int[4][];
```

Es importante tener en cuenta que aunque es posible definir cualquier dimensión de la matriz dentada, se ***debe*** definir su nivel anterior.

```
// valid
String[][] employeeGraph = new String[30][];

// invalid
int[][] unshapenMatrix = new int[][10];

// also invalid
int[][][] misshapenGrid = new int[100][][10];
```

Cómo se representan los arreglos multidimensionales en Java



Fuente de la imagen: http://math.hws.edu/eck/cs124/javanotes3/c8/s5.html

---

Intialización literal de matriz dentada

Las matrices multidimensionales y matrices irregulares también se pueden inicializar con una expresión literal. Lo siguiente declara y llena una matriz `int` 2x3:

```
int[][] table = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Nota : los subarreglos irregulares también pueden ser `null` . Por ejemplo, el siguiente código declara y llena una matriz `int` bidimensional cuyo primer subarreglo es `null` , el segundo subarreglo es de longitud cero, el tercer subarreglo es de una longitud y el último subarreglo es una matriz de dos longitudes:

```
int[][] table = {
    null,
    {},
    {1},
    {1,2}
};
```

Para matrices multidimensionales es posible extraer matrices de dimensión de nivel inferior por sus índices:

```
int[][][] arr = new int[3][3][3];
int[][] arr1 = arr[0]; // get first 3x3-dimensional array from arr
int[] arr2 = arr1[0]; // get first 3-dimensional array from arr1
int[] arr3 = arr[0]; // error: cannot convert from int[][] to int[]
```

ArrayIndexOutOfBoundsException

La `ArrayIndexOutOfBoundsException` produce cuando se accede a un índice no existente de una matriz.

Las matrices están indexadas en base a cero, por lo que el índice del primer elemento es `0` y el índice del último elemento es la capacidad de la matriz menos `1` (es decir, `array.length - 1`).

Por lo tanto, cualquier solicitud de un elemento de matriz por el índice `i` tiene que satisfacer la condición `0 <= i < array.length` , de lo contrario se `ArrayIndexOutOfBoundsException` la `ArrayIndexOutOfBoundsException` .

El siguiente código es un ejemplo simple donde se lanza una `ArrayIndexOutOfBoundsException` .

```
String[] people = new String[] { "Carol", "Andy" };

// An array will be created:
// people[0]: "Carol"
// people[1]: "Andy"

// Notice: no item on index 2. Trying to access it triggers the exception:
System.out.println(people[2]);  // throws an ArrayIndexOutOfBoundsException.
```

Salida:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
    at your.package.path.method(YourClass.java:15)
```

Tenga en cuenta que el índice ilegal al que se accede también se incluye en la excepción (`2` en el ejemplo); esta información podría ser útil para encontrar la causa de la excepción.

Para evitar esto, simplemente verifique que el índice esté dentro de los límites de la matriz:

```
int index = 2;
if (index >= 0 && index < people.length) {
    System.out.println(people[index]);
}
```

Obtener la longitud de una matriz

Las matrices son objetos que proporcionan espacio para almacenar hasta su tamaño de elementos del tipo especificado. El tamaño de una matriz no se puede modificar después de crear la matriz.

```
int[] arr1 = new int[0];
int[] arr2 = new int[2];
int[] arr3 = new int[]{1, 2, 3, 4};
int[] arr4 = {1, 2, 3, 4, 5, 6, 7};

int len1 = arr1.length; // 0
int len2 = arr2.length; // 2
int len3 = arr3.length; // 4
```

```
int len4 = arr4.length; // 7
```

El campo de `length` en una matriz almacena el tamaño de una matriz. Es un campo `final` y no puede ser modificado.

Este código muestra la diferencia entre la `length` de una matriz y la cantidad de objetos que almacena una matriz.

```
public static void main(String[] args) {
    Integer arr[] = new Integer[] {1,2,3,null,5,null,7,null,null,null,11,null,13};

    int arrayLength = arr.length;
    int nonEmptyElementsCount = 0;

    for (int i=0; i<arrayLength; i++) {
        Integer arrElt = arr[i];
        if (arrElt != null) {
            nonEmptyElementsCount++;
        }
    }

    System.out.println("Array 'arr' has a length of "+arrayLength+"\n"
                            + "and it contains "+nonEmptyElementsCount+" non-empty values");
}
```

Resultado:

```
Array 'arr' has a length of 13
and it contains 7 non-empty values
```

Comparando matrices para la igualdad

Los tipos de arreglos heredan las implementaciones de `equals()` (y `hashCode()`) de java.lang.Object , por lo que `equals()` solo devolverá true cuando se compare con el mismo objeto de arreglo. Para comparar arreglos de igualdad en función de sus valores, use `java.util.Arrays.equals` , que está sobrecargado para todos los tipos de arreglos.

```
int[] a = new int[]{1, 2, 3};
int[] b = new int[]{1, 2, 3};
System.out.println(a.equals(b)); //prints "false" because a and b refer to different objects
System.out.println(Arrays.equals(a, b)); //prints "true" because the elements of a and b have
the same values
```

Cuando el tipo de elemento es un tipo de referencia, `Arrays.equals()` llama a `equals()` en los elementos de la matriz para determinar la igualdad. En particular, si el tipo de elemento es en sí mismo un tipo de matriz, se utilizará la comparación de identidad. Para comparar matrices multidimensionales para la igualdad, use `Arrays.deepEquals()` lugar como se muestra a continuación:

```
int a[] = { 1, 2, 3 };
int b[] = { 1, 2, 3 };
```

```
Object[] aObject = { a }; // aObject contains one element
Object[] bObject = { b }; // bObject contains one element

System.out.println(Arrays.equals(aObject, bObject)); // false
System.out.println(Arrays.deepEquals(aObject, bObject));// true
```

Debido a que los conjuntos y mapas utilizan `equals()` y `hashCode()` , los arreglos generalmente no son útiles como elementos de conjunto o claves de mapa. `hashCode()` en una clase auxiliar que implemente `equals()` y `hashCode()` en términos de los elementos de la matriz, o `hashCode()` a instancias de `List` y almacene las listas.

Arreglos para transmitir

Java SE 8

Convertir una matriz de objetos a `Stream` :

```
String[] arr = new String[] {"str1", "str2", "str3"};
Stream<String> stream = Arrays.stream(arr);
```

La conversión de una matriz de primitivas a `Stream` usando `Arrays.stream()` transformará la matriz a una especialización primitiva de Stream:

```
int[] intArr = {1, 2, 3};
IntStream intStream = Arrays.stream(intArr);
```

También puede limitar el `Stream` a un rango de elementos en la matriz. El índice de inicio es inclusivo y el índice de finalización es exclusivo:

```
int[] values = {1, 2, 3, 4};
IntStream intStream = Arrays.stream(values, 2, 4);
```

Un método similar a `Arrays.stream()` aparece en la clase `Stream` : `Stream.of()` . La diferencia es que `Stream.of()` usa un parámetro varargs, por lo que puede escribir algo como:

```
Stream<Integer> intStream = Stream.of(1, 2, 3);
Stream<String> stringStream = Stream.of("1", "2", "3");
Stream<Double> doubleStream = Stream.of(new Double[]{1.0, 2.0});
```

Iterando sobre matrices

Puede iterar sobre matrices usando un bucle mejorado para (también conocido como foreach) o usando índices de matrices:

```
int[] array = new int[10];

// using indices: read and write
for (int i = 0; i < array.length; i++) {
    array[i] = i;
}
```

Java SE 5

```
// extended for: read only
for (int e : array) {
    System.out.println(e);
}
```

Vale la pena señalar aquí que no hay una forma directa de usar un iterador en un Array, pero a través de la biblioteca de Arrays se puede convertir fácilmente en una lista para obtener un objeto `Iterable` .

Para arrays en caja use Arrays.asList :

```
Integer[] boxed = {1, 2, 3};
Iterable<Integer> boxedIt = Arrays.asList(boxed); // list-backed iterable
Iterator<Integer> fromBoxed1 = boxedIt.iterator();
```

Para matrices primitivas (usando java 8) use flujos (específicamente en este ejemplo - Arrays.stream -> IntStream):

```
int[] primitives = {1, 2, 3};
IntStream primitiveStream = Arrays.stream(primitives); // list-backed iterable
PrimitiveIterator.OfInt fromPrimitive1 = primitiveStream.iterator();
```

Si no puede usar secuencias (no java 8), puede elegir usar la biblioteca de guayabas de google:

```
Iterable<Integer> fromPrimitive2 = Ints.asList(primitives);
```

En arreglos bidimensionales o más, ambas técnicas pueden usarse de una manera un poco más compleja.

Ejemplo:

```
int[][] array = new int[10][10];

for (int indexOuter = 0; indexOuter < array.length; indexOuter++) {
    for (int indexInner = 0; indexInner < array[indexOuter].length; indexInner++) {
        array[indexOuter][indexInner] = indexOuter + indexInner;
    }
}
```

Java SE 5

```
for (int[] numbers : array) {
    for (int value : numbers) {
        System.out.println(value);
    }
}
```

Es imposible establecer un Array en un valor no uniforme sin usar un bucle basado en índices.

Por supuesto, también puede usar bucles `while` o `do-while` while cuando se iteran utilizando índices.

Una nota de precaución: cuando utilice índices de matriz, asegúrese de que el índice esté entre `0` y `array.length - 1` (ambos inclusive). No haga suposiciones de código rígido en la longitud de la matriz, de lo contrario podría romper su código si la longitud de la matriz cambia pero sus valores codificados no lo hacen.

Ejemplo:

```
int[] numbers = {1, 2, 3, 4};

public void incrementNumbers() {
    // DO THIS :
    for (int i = 0; i < numbers.length; i++) {
        numbers[i] += 1; //or this: numbers[i] = numbers[i] + 1; or numbers[i]++;
    }

    // DON'T DO THIS :
    for (int i = 0; i < 4; i++) {
        numbers[i] += 1;
    }
}
```

También es mejor si no usa cálculos sofisticados para obtener el índice, pero si lo usa para iterar y si necesita valores diferentes, calcúlelos.

Ejemplo:

```
public void fillArrayWithDoubleIndex(int[] array) {
    // DO THIS :
    for (int i = 0; i < array.length; i++) {
        array[i] = i * 2;
    }

    // DON'T DO THIS :
    int doubleLength = array.length * 2;
    for (int i = 0; i < doubleLength; i += 2) {
        array[i / 2] = i;
    }
}
```

Acceso a matrices en orden inverso

```
int[] array = {0, 1, 1, 2, 3, 5, 8, 13};
for (int i = array.length - 1; i >= 0; i--) {
   System.out.println(array[i]);
}
```

Uso de matrices temporales para reducir la repetición de código

Iterar sobre una matriz temporal en lugar de repetir el código puede hacer que su código sea más limpio. Se puede utilizar donde se realiza la misma operación en múltiples variables.

```
// we want to print out all of these
String name = "Margaret";
int eyeCount = 16;
double height = 50.2;
int legs = 9;
int arms = 5;


// copy-paste approach:
System.out.println(name);
System.out.println(eyeCount);
System.out.println(height);
System.out.println(legs);
System.out.println(arms);


// temporary array approach:
for(Object attribute : new Object[]{name, eyeCount, height, legs, arms})
    System.out.println(attribute);

// using only numbers
for(double number : new double[]{eyeCount, legs, arms, height})
    System.out.println(Math.sqrt(number));
```

Tenga en cuenta que este código no debe usarse en secciones críticas para el rendimiento, ya que se crea una matriz cada vez que se ingresa al bucle, y las variables primitivas se copiarán en la matriz y, por lo tanto, no se podrán modificar.

Copiando matrices

Java proporciona varias formas de copiar una matriz.

en bucle

```
int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

Tenga en cuenta que el uso de esta opción con una matriz de objetos en lugar de una matriz primitiva llenará la copia con referencia al contenido original en lugar de una copia de la misma.

Object.clone ()

Dado que las matrices son `Object` en Java, puede usar `Object.clone()` .

```
int[] a = { 4, 1, 3, 2 };
int[] b = a.clone(); // [4, 1, 3, 2]
```

Tenga en cuenta que el método `Object.clone` para una matriz realiza una **copia superficial** , es

decir, devuelve una referencia a una nueva matriz que hace referencia a los **mismos** elementos que la matriz de origen.

Arrays.copyOf ()

`java.util.Arrays` proporciona una manera fácil de realizar la copia de una matriz a otra. Aquí está el uso básico:

```
int[] a = {4, 1, 3, 2};
int[] b = Arrays.copyOf(a, a.length); // [4, 1, 3, 2]
```

Tenga en cuenta que `Arrays.copyOf` también proporciona una sobrecarga que le permite cambiar el tipo de la matriz:

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```

System.arraycopy ()

> `public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)` Copia una matriz de la matriz de origen especificada, comenzando en la posición especificada, a la posición especificada de la matriz de destino.

A continuación un ejemplo de uso.

```
int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
System.arraycopy(a, 0, b, 0, a.length); // [4, 1, 3, 2]
```

Arrays.copyOfRange ()

Utilizado principalmente para copiar una parte de una matriz, también puede usarla para copiar una matriz completa a otra como se muestra a continuación

```
int[] a = { 4, 1, 3, 2 };
int[] b = Arrays.copyOfRange(a, 0, a.length); // [4, 1, 3, 2]
```

Matrices de fundición

Las matrices son objetos, pero su tipo está definido por el tipo de los objetos contenidos. Por lo tanto, uno no puede simplemente lanzar `A[]` a `T[]` , pero cada miembro A de `A[]` específico debe ser lanzado a un objeto `T` Ejemplo genérico:

```
public static <T, A> T[] castArray(T[] target, A[] array) {
    for (int i = 0; i < array.length; i++) {
        target[i] = (T) array[i];
    }
    return target;
}
```

Por lo tanto, dada una matriz `A[]` :

```
T[] target = new T[array.Length];
target = castArray(target, array);
```

Java SE proporciona el método `Arrays.copyOf(original, newLength, newType)` para este propósito:

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```

Eliminar un elemento de una matriz

Java no proporciona un método directo en `java.util.Arrays` para eliminar un elemento de una matriz. Para realizarlo, puede copiar la matriz original en una nueva sin el elemento para eliminar o convertir su matriz en otra estructura que permita la eliminación.

Usando ArrayList

Puede convertir la matriz en una `java.util.List` , eliminar el elemento y volver a convertir la lista en una matriz de la siguiente manera:

```
String[] array = new String[]{"foo", "bar", "baz"};

List<String> list = new ArrayList<>(Arrays.asList(array));
list.remove("foo");

// Creates a new array with the same size as the list and copies the list
// elements to it.
array = list.toArray(new String[list.size()]);

System.out.println(Arrays.toString(array)); //[bar, baz]
```

Usando System.arraycopy

`System.arraycopy()` se puede usar para hacer una copia de la matriz original y eliminar el elemento que desee. A continuación un ejemplo:

```
int[] array = new int[] { 1, 2, 3, 4 }; // Original array.
int[] result = new int[array.length - 1]; // Array which will contain the result.
int index = 1; // Remove the value "2".

// Copy the elements at the left of the index.
System.arraycopy(array, 0, result, 0, index);
```

```
// Copy the elements at the right of the index.
System.arraycopy(array, index + 1, result, index, array.length - index - 1);

System.out.println(Arrays.toString(result)); //[1, 3, 4]
```

Usando Apache Commons Lang

Para eliminar fácilmente un elemento, puede usar la biblioteca Lang de Apache Commons y especialmente el método estático `removeElement()` de la clase `ArrayUtils` . A continuación un ejemplo:

```
int[] array = new int[]{1,2,3,4};
array = ArrayUtils.removeElement(array, 2); //remove first occurrence of 2
System.out.println(Arrays.toString(array)); //[1, 3, 4]
```

Array Covariance

Las matrices de objetos son covariantes, lo que significa que así como `Integer` es una subclase de `Number` , `Integer[]` es una subclase de `Number[]` . Esto puede parecer intuitivo, pero puede resultar en un comportamiento sorprendente:

```
Integer[] integerArray = {1, 2, 3};
Number[] numberArray = integerArray;  // valid
Number firstElement = numberArray[0]; // valid
numberArray[0] = 4L;                  // throws ArrayStoreException at runtime
```

Aunque `Integer[]` es una subclase de `Number[]` , solo puede contener `Integer` s, y al intentar asignar un elemento `Long` se genera una excepción de tiempo de ejecución.

Tenga en cuenta que este comportamiento es exclusivo de las matrices y que se puede evitar utilizando una `List` genérica en su lugar:

```
List<Integer> integerList = Arrays.asList(1, 2, 3);
//List<Number> numberList = integerList;  // compile error
List<? extends Number> numberList = integerList;
Number firstElement = numberList.get(0);
//numberList.set(0, 4L);                  // compile error
```

No es necesario que todos los elementos de la matriz compartan el mismo tipo, siempre que sean una subclase del tipo de la matriz:

```
interface I {}

class A implements I {}
class B implements I {}
class C implements I {}

I[] array10 = new I[] { new A(), new B(), new C() }; // Create an array with new
                                                     // operator and array initializer.
```

```
I[] array11 = { new A(), new B(), new C() };         // Shortcut syntax with array
                                                     // initializer.

I[] array12 = new I[3];                              // { null, null, null }

I[] array13 = new A[] { new A(), new A() };          // Works because A implements I.

Object[] array14 = new Object[] { "Hello, World!", 3.14159, 42 }; // Create an array with
                                                     // new operator and array initializer.

Object[] array15 = { new A(), 64, "My String" };     // Shortcut syntax
                                                     // with array initializer.
```

¿Cómo cambias el tamaño de una matriz?

La respuesta simple es que no puedes hacer esto. Una vez que se ha creado una matriz, su tamaño no se puede cambiar. En cambio, una matriz solo puede "redimensionarse" creando una nueva matriz con el tamaño apropiado y copiando los elementos de la matriz existente a la nueva.

```
String[] listOfCities = new String[3];   // array created with size 3.
listOfCities[0] = "New York";
listOfCities[1] = "London";
listOfCities[2] = "Berlin";
```

Supongamos (por ejemplo) que es necesario agregar un nuevo elemento a la matriz `listOfCities` definida como se `listOfCities` anteriormente. Para hacer esto, necesitarás:

1. crear una nueva matriz con tamaño 4,
2. copie los 3 elementos existentes de la matriz antigua a la nueva matriz en las compensaciones 0, 1 y 2, y
3. agregue el nuevo elemento a la nueva matriz en el desplazamiento 3.

Hay varias formas de hacer lo anterior. Antes de Java 6, la forma más concisa era:

```
String[] newArray = new String[listOfCities.length + 1];
System.arraycopy(listOfCities, 0, newArray, 0, listOfCities.length);
newArray[listOfCities.length] = "Sydney";
```

Desde Java 6 en adelante, los métodos `Arrays.copyOf` y `Arrays.copyOfRange` pueden hacer esto de manera más simple:

```
String[] newArray = Arrays.copyOf(listOfCities, listOfCities.length + 1);
newArray[listOfCities.length] = "Sydney";
```

Para otras formas de copiar una matriz, consulte el siguiente ejemplo. Tenga en cuenta que necesita una copia de matriz con una longitud diferente a la original al cambiar el tamaño.

- Copiando matrices

Una mejor alternativa para cambiar el tamaño de matriz

Hay dos inconvenientes principales con el cambio de tamaño de una matriz como se describe anteriormente:

- Es ineficiente. Hacer una matriz más grande (o más pequeña) implica copiar muchos o todos los elementos de la matriz existente, y asignar un nuevo objeto de matriz. Cuanto mayor sea la matriz, más caro se vuelve.
- Debe poder actualizar cualquier variable "viva" que contenga referencias a la matriz anterior.

Una alternativa es crear la matriz con un tamaño suficientemente grande para comenzar. Esto solo es viable si puede determinar ese tamaño con precisión *antes de asignar la matriz* . Si no puede hacer eso, entonces surge nuevamente el problema de cambiar el tamaño de la matriz.

La otra alternativa es utilizar una clase de estructura de datos proporcionada por la biblioteca de clases Java SE o una biblioteca de terceros. Por ejemplo, el marco de "colecciones" de Java SE proporciona varias implementaciones de las API de `List` , `Set` y `Map` con diferentes propiedades de tiempo de ejecución. La clase `ArrayList` es la más cercana a las características de rendimiento de una matriz simple (p. Ej., Búsqueda O (N), obtención y configuración O (1), inserción y eliminación aleatoria O (N) al tiempo que proporciona un cambio de tamaño más eficiente sin el problema de actualización de referencia.

(La eficiencia de cambio de tamaño para `ArrayList` proviene de su estrategia de duplicar el tamaño de la matriz de respaldo en cada cambio de tamaño. Para un caso de uso típico, esto significa que solo cambia el tamaño ocasionalmente. Cuando amortiza durante la vida útil de la lista, el costo del cambio de tamaño por inserción es `O(1)` . Puede ser posible usar la misma estrategia al cambiar el tamaño de una matriz simple).

Encontrar un elemento en una matriz

Hay muchas maneras de encontrar la ubicación de un valor en una matriz. Los siguientes fragmentos de ejemplo suponen que la matriz es una de las siguientes:

```
String[] strings = new String[] { "A", "B", "C" };
int[] ints = new int[] { 1, 2, 3, 4 };
```

Además, cada uno establece el `index` o el `index index2` en el índice del elemento requerido, o `-1` si el elemento no está presente.

Usando `Arrays.binarySearch` (solo para arreglos ordenados)

```
int index = Arrays.binarySearch(strings, "A");
int index2 = Arrays.binarySearch(ints, 1);
```

Uso de `Arrays.asList` (solo para matrices no primitivas)

```
int index = Arrays.asList(strings).indexOf("A");
int index2 = Arrays.asList(ints).indexOf(1);  // compilation error
```

Usando una `Stream`

Java SE 8

```
int index = IntStream.range(0, strings.length)
                     .filter(i -> "A".equals(strings[i]))
                     .findFirst()
                     .orElse(-1); // If not present, gives us -1.
// Similar for an array of primitives
```

Búsqueda lineal utilizando un bucle.

```
int index = -1;
for (int i = 0; i < array.length; i++) {
    if ("A".equals(array[i])) {
        index = i;
        break;
    }
}
// Similar for an array of primitives
```

Búsqueda lineal utilizando bibliotecas de terceros como org.apache.commons

```
int index = org.apache.commons.lang3.ArrayUtils.contains(strings, "A");
int index2 = org.apache.commons.lang3.ArrayUtils.contains(ints, 1);
```

Nota: Usar una búsqueda lineal directa es más eficiente que envolver en una lista.

Probando si una matriz contiene un elemento

Los ejemplos anteriores se pueden adaptar para probar si la matriz contiene un elemento simplemente probando para ver si el índice calculado es mayor o igual a cero.

Alternativamente, también hay algunas variaciones más concisas:

```
boolean isPresent = Arrays.asList(strings).contains("A");
```

Java SE 8

```
boolean isPresent = Stream<String>.of(strings).anyMatch(x -> "A".equals(x));
```

```
boolean isPresent = false;
for (String s : strings) {
    if ("A".equals(s)) {
        isPresent = true;
        break;
    }
```

```
}

boolean isPresent = org.apache.commons.lang3.ArrayUtils.contains(ints, 4);
```

Ordenando matrices

La clasificación de matrices se puede hacer fácilmente con la matriz de matrices .

```
import java.util.Arrays;

// creating an array with integers
int[] array = {7, 4, 2, 1, 19};
// this is the sorting part just one function ready to be used
Arrays.sort(array);
// prints [1, 2, 4, 7, 19]
System.out.println(Arrays.toString(array));
```

Clasificación de matrices de cuerdas:

`String` no es un dato numérico, define su propio orden, que se denomina orden lexicográfico, también conocido como orden alfabético. Cuando ordena una matriz de String usando el método `sort()` , la matriz se ordena en el orden natural definido por la interfaz comparable, como se muestra a continuación:

Orden creciente

```
String[] names = {"John", "Steve", "Shane", "Adam", "Ben"};
System.out.println("String array before sorting : " + Arrays.toString(names));
Arrays.sort(names);
System.out.println("String array after sorting in ascending order : " +
Arrays.toString(names));
```

Salida:

```
String array before sorting : [John, Steve, Shane, Adam, Ben]
String array after sorting in ascending order : [Adam, Ben, John, Shane, Steve]
```

Orden decreciente

```
Arrays.sort(names, 0, names.length, Collections.reverseOrder());
System.out.println("String array after sorting in descending order : " +
Arrays.toString(names));
```

Salida:

```
String array after sorting in descending order : [Steve, Shane, John, Ben, Adam]
```

Ordenar una matriz de objetos

Para ordenar una matriz de objetos, todos los elementos deben implementar una interfaz `Comparable` o `Comparator` para definir el orden de la clasificación.

Podemos usar cualquiera de los métodos de `sort(Object[])` para ordenar una matriz de objetos en su orden natural, pero debe asegurarse de que todos los elementos de la matriz deben implementar `Comparable` .

Además, también deben ser mutuamente comparables, por ejemplo, `e1.compareTo(e2)` no debe lanzar una *`ClassCastException`* para ningún elemento e1 y e2 en la matriz. Alternativamente, puede ordenar una matriz de objetos en orden personalizado usando el método de `sort(T[], Comparator)` como se muestra en el siguiente ejemplo.

```
// How to Sort Object Array in Java using Comparator and Comparable
Course[] courses = new Course[4];
courses[0] = new Course(101, "Java", 200);
courses[1] = new Course(201, "Ruby", 300);
courses[2] = new Course(301, "Python", 400);
courses[3] = new Course(401, "Scala", 500);

System.out.println("Object array before sorting : " + Arrays.toString(courses));

Arrays.sort(courses);
System.out.println("Object array after sorting in natural order : " +
Arrays.toString(courses));

Arrays.sort(courses, new Course.PriceComparator());
System.out.println("Object array after sorting by price : " + Arrays.toString(courses));

Arrays.sort(courses, new Course.NameComparator());
System.out.println("Object array after sorting by name : " + Arrays.toString(courses));
```

Salida:

```
Object array before sorting : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 , #401
Scala@500 ]
Object array after sorting in natural order : [#101 Java@200 , #201 Ruby@300 , #301 Python@400
, #401 Scala@500 ]
Object array after sorting by price : [#101 Java@200 , #201 Ruby@300 , #301 Python@400 , #401
Scala@500 ]
Object array after sorting by name : [#101 Java@200 , #301 Python@400 , #201 Ruby@300 , #401
Scala@500 ]
```

Conversión de matrices entre primitivas y tipos de caja.

A veces es necesaria la conversión de tipos primitivos a tipos en caja .

Para convertir la matriz, es posible usar flujos (en Java 8 y superior):

Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray =
    Arrays.stream(primitiveArray).boxed().toArray(Integer[]::new);
```

Con versiones inferiores, puede ser iterando la matriz primitiva y copiándola explícitamente a la matriz en caja:

Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray = new Integer[primitiveArray.length];
for (int i = 0; i < primitiveArray.length; ++i) {
    boxedArray[i] = primitiveArray[i]; // Each element is autoboxed here
}
```

De manera similar, una matriz en caja se puede convertir en una matriz de su contraparte primitiva:

Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};
int[] primitiveArray =
    Arrays.stream(boxedArray).mapToInt(Integer::intValue).toArray();
```

Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};
int[] primitiveArray = new int[boxedArray.length];
for (int i = 0; i < boxedArray.length; ++i) {
    primitiveArray[i] = boxedArray[i]; // Each element is outboxed here
}
```

Lea Arrays en línea: https://riptutorial.com/es/java/topic/99/arrays

Capítulo 16: Audio

Observaciones

En lugar de usar el `Clip` muestra javax.sound.sampled, también puede usar el `AudioClip` que es de la API del applet. Sin embargo, se recomienda usar `Clip` ya que `AudioClip` es más antiguo y presenta funcionalidades limitadas.

Examples

Reproducir un archivo de audio en bucle

Importaciones necesarias:

```
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
```

Este código creará un clip y lo reproducirá continuamente una vez iniciado:

```
Clip clip = AudioSystem.getClip();
clip.open(AudioSystem.getAudioInputStream(new URL(filename)));
clip.start();
clip.loop(Clip.LOOP_CONTINUOUSLY);
```

Obtenga una matriz con todos los tipos de archivos compatibles:

```
AudioFileFormat.Type [] audioFileTypes = AudioSystem.getAudioFileTypes();
```

Reproducir un archivo MIDI

Los archivos MIDI se pueden reproducir utilizando varias clases del paquete `javax.sound.midi` . Un `Sequencer` realiza la reproducción del archivo MIDI, y muchos de sus métodos pueden usarse para configurar controles de reproducción como el conteo de bucles, el tempo, el silenciamiento de pistas y otros.

La reproducción general de datos MIDI se puede hacer de esta manera:

```
import java.io.File;
import java.io.IOException;
import javax.sound.midi.InvalidMidiDataException;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Sequence;
import javax.sound.midi.Sequencer;

public class MidiPlayback {
    public static void main(String[] args) {
        try {
```

```
            Sequencer sequencer = MidiSystem.getSequencer(); // Get the default Sequencer
            if (sequencer==null) {
                System.err.println("Sequencer device not supported");
                return;
            }
            sequencer.open(); // Open device
            // Create sequence, the File must contain MIDI file data.
            Sequence sequence = MidiSystem.getSequence(new File(args[0]));
            sequencer.setSequence(sequence); // load it into sequencer
            sequencer.start();  // start the playback
        } catch (MidiUnavailableException | InvalidMidiDataException | IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Para detener la reproducción usa:

```
sequencer.stop(); // Stop the playback
```

Se puede configurar un secuenciador para silenciar una o más de las pistas de la secuencia durante la reproducción, por lo que ninguno de los instrumentos en esos juegos especificados. El siguiente ejemplo configura la primera pista en la secuencia a silenciar:

```
import javax.sound.midi.Track;
// ...

Track[] track = sequence.getTracks();
sequencer.setTrackMute(track[0]);
```

Un secuenciador puede reproducir una secuencia repetidamente si se da el recuento de bucles. A continuación se establece el secuenciador para reproducir una secuencia cuatro veces e indefinidamente:

```
sequencer.setLoopCount(3);
sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
```

El secuenciador no siempre tiene que reproducir la secuencia desde el principio, ni tiene que reproducir la secuencia hasta el final. Puede comenzar y terminar en cualquier punto especificando la *marca* en la secuencia para comenzar y terminar en. También es posible especificar manualmente qué tic en la secuencia debe jugar el secuenciador desde:

```
sequencer.setLoopStartPoint(512);
sequencer.setLoopEndPoint(32768);
sequencer.setTickPosition(8192);
```

Los secuenciadores también pueden reproducir un archivo MIDI a un determinado tempo, que puede controlarse especificando el tempo en tiempos por minuto (BPM) o microsegundos por cuarto de nota (MPQ). El factor en el que se reproduce la secuencia también se puede ajustar.

```
sequencer.setTempoInBPM(1250f);
sequencer.setTempoInMPQ(4750f);
```

```
sequencer.setTempoFactor(1.5f);
```

Cuando termine de usar el `Sequencer` , recuerde cerrarla

```
sequencer.close();
```

Sonido de metal desnudo

También puedes usar el metal casi al desnudo al producir sonido con java. Este código escribirá datos binarios sin procesar en el búfer de audio del sistema operativo para generar sonido. Es extremadamente importante comprender las limitaciones y los cálculos necesarios para generar un sonido como este. Dado que la reproducción es básicamente instantánea, los cálculos deben realizarse casi en tiempo real.

Como tal, este método es inutilizable para un muestreo de sonido más complicado. Para tales fines, el uso de herramientas especializadas es el mejor enfoque.

El siguiente método genera y emite directamente una onda rectangular de una frecuencia dada en un volumen dado para una duración determinada.

```
public void rectangleWave(byte volume, int hertz, int msecs) {
    final SourceDataLine dataLine;
    // 24 kHz x 8bit, single-channel, signed little endian AudioFormat
    AudioFormat af = new AudioFormat(24_000, 8, 1, true, false);
    try {
        dataLine = AudioSystem.getSourceDataLine(af);
        dataLine.open(af, 10_000); // audio buffer size: 10k samples
    } catch (LineUnavailableException e) {
        throw new RuntimeException(e);
    }

    int waveHalf = 24_000 / hertz; // samples for half a period
    byte[] buffer = new byte[waveHalf * 20];
    int samples = msecs * (24_000 / 1000); // 24k (samples / sec) / 1000 (ms/sec) * time(ms)

    dataLine.start(); // starts playback
    int sign = 1;

    for (int i = 0; i < samples; i += buffer.length) {
        for (int j = 0; j < 20; j++) { // generate 10 waves into buffer
            sign *= -1;
            // fill from the jth wave-half to the j+1th wave-half with volume
            Arrays.fill(buffer, waveHalf * j, waveHalf * (j+1), (byte) (volume * sign));
        }
        dataLine.write(buffer, 0, buffer.length); //
    }
    dataLine.drain(); // forces buffer drain to hardware
    dataLine.stop();  // ends playback
}
```

Para una forma más diferenciada de generar diferentes ondas de sonido, son necesarios cálculos sinusales y posiblemente tamaños de muestra más grandes. Esto da como resultado un código significativamente más complejo y, por consiguiente, se omite aquí.

Salida de audio básica

El Hola Audio! de Java que reproduce un archivo de sonido desde el almacenamiento local o de Internet tiene el siguiente aspecto. Funciona para archivos .wav sin comprimir y no debe usarse para reproducir archivos mp3 o comprimidos.

```
import java.io.*;
import java.net.URL;
import javax.sound.sampled.*;

public class SoundClipTest {

   // Constructor
   public SoundClipTest() {
      try {
         // Open an audio input stream.
          File soundFile = new File("/usr/share/sounds/alsa/Front_Center.wav"); //you could
also get the sound file with an URL
          AudioInputStream audioIn = AudioSystem.getAudioInputStream(soundFile);
          AudioFormat format = audioIn.getFormat();
         // Get a sound clip resource.
         DataLine.Info info = new DataLine.Info(Clip.class, format);
         Clip clip = (Clip)AudioSystem.getLine(info);
         // Open audio clip and load samples from the audio input stream.
         clip.open(audioIn);
         clip.start();
      } catch (UnsupportedAudioFileException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      } catch (LineUnavailableException e) {
         e.printStackTrace();
      }
   }

   public static void main(String[] args) {
      new SoundClipTest();
   }
}
```

Lea Audio en línea: https://riptutorial.com/es/java/topic/160/audio

Capítulo 17: Autoboxing

Introducción

Autoboxing es la conversión automática que realiza el compilador de Java entre los tipos primitivos y sus correspondientes clases de envoltorios de objetos. Ejemplo, conversión de int -> Entero, doble -> Doble ... Si la conversión es a la inversa, esto se llama unboxing. Por lo general, esto se usa en Colecciones que no pueden contener más que Objetos, donde se necesitan tipos primitivos de boxeo antes de configurarlos en la colección.

Observaciones

Autoboxing puede tener problemas de rendimiento cuando se utiliza con frecuencia en su código.

- http://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html
- ¿El autoempaquetamiento integral y el boxeo automático dan problemas de rendimiento?

Examples

Usando int y entero indistintamente

A medida que usa tipos genéricos con clases de utilidad, a menudo puede encontrar que los tipos de números no son muy útiles cuando se especifican como tipos de objetos, ya que no son iguales a sus contrapartes primitivas.

```
List<Integer> ints = new ArrayList<Integer>();
```

Java SE 7

```
List<Integer> ints = new ArrayList<>();
```

Afortunadamente, las expresiones que evalúan a `int` pueden usarse en lugar de un `Integer` cuando sea necesario.

```
for (int i = 0; i < 10; i++)
    ints.add(i);
```

El `ints.add(i);` declaración es equivalente a:

```
ints.add(Integer.valueOf(i));
```

Y retiene las propiedades de `Integer#valueOf` , como tener los mismos objetos de `Integer` almacenados en caché por la JVM cuando está dentro del rango de almacenamiento en caché de números.

Esto también se aplica a:

- `byte` y `Byte`
- `short` y `Short`
- `float` y `Float`
- `double` y `Double`
- `long` y `Long`
- `char` y `Character`
- `boolean` y `Boolean`

Se debe tener cuidado, sin embargo, en situaciones ambiguas. Considere el siguiente código:

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
ints.add(3);
ints.remove(1); // ints is now [1, 3]
```

La interfaz `java.util.List` contiene tanto un `remove(int index)` (Método de interfaz de `List`) como un `remove(Object o)` (método heredado de `java.util.Collection`). En este caso no se lleva a cabo ningún boxeo y se llama a la `remove(int index)` .

Un ejemplo más del extraño comportamiento del código Java causado por los enteros automoxing con valores en el rango de `-128` a `127` :

```
Integer a = 127;
Integer b = 127;
Integer c = 128;
Integer d = 128;
System.out.println(a == b); // true
System.out.println(c <= d); // true
System.out.println(c >= d); // true
System.out.println(c == d); // false
```

Esto sucede porque el operador `>=` llama implícitamente a `intValue()` que devuelve `int` mientras que `==` compara las **referencias** , no los valores de `int` .

De forma predeterminada, Java almacena en caché los valores en el rango `[-128, 127]` , por lo que el operador `==` funciona porque los `Integers` en este rango hacen referencia a los mismos objetos si sus valores son los mismos. El valor máximo del rango `-XX:AutoBoxCacheMax` se puede definir con la `-XX:AutoBoxCacheMax` JVM. Por lo tanto, si ejecuta el programa con `-XX:AutoBoxCacheMax=1000` , el siguiente código se imprimirá `true` :

```
Integer a = 1000;
Integer b = 1000;
System.out.println(a == b); // true
```

Usando Boolean en la sentencia if

Debido al auto-boxeo, uno puede usar un `Boolean` en una sentencia `if` :

```
Boolean a = Boolean.TRUE;
if (a) { // a gets converted to boolean
    System.out.println("It works!");
}
```

Eso funciona para `while` , `do while` y la condición en las declaraciones `for` también.

Tenga en cuenta que, si el valor `Boolean` es `null` , se lanzará una `NullPointerException` en la conversión.

Auto-unboxing puede llevar a NullPointerException

Este código compila:

```
Integer arg = null;
int x = arg;
```

Pero se bloqueará en tiempo de ejecución con una `java.lang.NullPointerException` en la segunda línea.

El problema es que un `int` primitivo no puede tener un valor `null` .

Este es un ejemplo minimalista, pero en la práctica a menudo se manifiesta en formas más sofisticadas. La `NullPointerException` no es muy intuitiva y suele ser de poca ayuda para localizar estos errores.

Confíe en el autoboxing y el auto-boxeo con cuidado, asegúrese de que los valores unboxed no tengan valores `null` en el tiempo de ejecución.

Memoria y sobrecarga computacional de Autoboxing

Autoboxing puede venir en una sobrecarga de memoria sustancial. Por ejemplo:

```
Map<Integer, Integer> square = new HashMap<Integer, Integer>();
for(int i = 256; i < 1024; i++) {
    square.put(i, i * i); // Autoboxing of large integers
}
```

normalmente consumirá una cantidad sustancial de memoria (alrededor de 60kb para 6k de datos reales).

Además, los enteros en caja generalmente requieren viajes de ida y vuelta adicionales en la memoria, y por lo tanto hacen que los cachés de CPU sean menos efectivos. En el ejemplo anterior, la memoria a la que se accede se extiende a cinco ubicaciones diferentes que pueden estar en regiones completamente diferentes de la memoria: 1. el objeto `HashMap` , 2. el objeto de `Entry[] table` del mapa, 3. el objeto `Entry` , 4. el objeto objeto `key` entradas (boxeo de la clave primitiva), 5. objeto de `value` entradas (boxeo del valor primitivo).

```
class Example {
  int primitive; // Stored directly in the class `Example`
```

```
  Integer boxed; // Reference to another memory location
}
```

La lectura en `boxed` requiere dos accesos de memoria, accediendo solo a uno `primitive` .

Al obtener datos de este mapa, el código aparentemente inocente

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(i);
}
```

es equivalente a:

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(Integer.valueOf(i)).intValue();
}
```

Normalmente, el código anterior causa la *creación y recolección* de elementos no *utilizados* de un objeto `Integer` para cada operación `Map#get(Integer)` . (Vea la Nota a continuación para más detalles).

Para reducir esta sobrecarga, varias bibliotecas ofrecen colecciones optimizadas para tipos primitivos que *no* requieren boxeo. Además de evitar la sobrecarga del boxeo, esta colección requerirá aproximadamente 4 veces menos memoria por entrada. Si bien Java Hotspot *puede* optimizar el autoboxing al trabajar con objetos en la pila en lugar del montón, no es posible optimizar la sobrecarga de memoria y la indirección de la memoria resultante.

Las transmisiones Java 8 también tienen interfaces optimizadas para tipos de datos primitivos, como `IntStream` que no requieren boxeo.

Nota: un tiempo de ejecución de Java típico mantiene un caché simple de `Integer` y otro objeto de envoltorio primitivo que utilizan los métodos de fábrica `valueOf` , y mediante autofijación. Para `Integer` , el rango predeterminado de este caché es de -128 a +127. Algunas JVM proporcionan una opción de línea de comandos JVM para cambiar el tamaño / rango de la memoria caché.

Casos diferentes cuando Integer e int pueden usarse indistintamente

Caso 1: Mientras se usa en lugar de argumentos de método.

Si un método requiere un objeto de la clase envoltura como argumento. Luego, indistintamente, se le puede pasar una variable del tipo primitivo respectivo y viceversa.

Ejemplo:

```
int i;
Integer j;
void ex_method(Integer i)//Is a valid statement
void ex_method1(int j)//Is a valid statement
```

Caso 2: Mientras se pasan valores de retorno:

Cuando un método devuelve una variable de tipo primitivo, entonces se puede pasar un objeto de la clase envoltura correspondiente como valor de retorno de manera intercambiable y viceversa.

Ejemplo:

```
int i;
Integer j;
int ex_method()
{...
return j;}//Is a valid statement
Integer ex_method1()
{...
return i;//Is a valid statement
}
```

Caso 3: Al realizar operaciones.

Siempre que se realicen operaciones en números, la variable de tipo primitivo y el objeto de la respectiva clase envoltura se pueden usar indistintamente.

```
int i=5;
Integer j=new Integer(7);
int k=i+j;//Is a valid statement
Integer m=i+j;//Is also a valid statement
```

Trampa : recuerde inicializar o asignar un valor a un objeto de la clase contenedora.

Si bien el uso de un objeto de la clase de envoltorio y la variable primitiva no se olvida, nunca se olvida o se pierde para inicializar o asignar un valor al objeto de la clase de envoltorio, de lo contrario, puede provocar una excepción de puntero nulo en el tiempo de ejecución.

Ejemplo:

```
public class Test{
    Integer i;
    int j;
    public void met()
    {j=i;//Null pointer exception
    SOP(j);
    SOP(i);}
    public static void main(String[] args)
    {Test t=new Test();
    t.go();//Null pointer exception
    }
```

En el ejemplo anterior, el valor del objeto no está asignado ni inicializado y, por lo tanto, en tiempo de ejecución, el programa ejecutará una excepción de puntero nulo. Por lo tanto, el valor del objeto nunca debe dejarse sin inicializar ni asignar.

Lea Autoboxing en línea: https://riptutorial.com/es/java/topic/138/autoboxing

Capítulo 18: Banderas JVM

Observaciones

Se recomienda encarecidamente que utilice estas opciones solamente:

- Si tienes una comprensión profunda de tu sistema.
- Tenga en cuenta que, si se usa incorrectamente, estas opciones pueden tener un efecto negativo en la estabilidad o el rendimiento de su sistema.

Información recogida de documentación oficial de Java .

Examples

-XXaggresivo

`-XXaggressive` es una colección de configuraciones que hacen que la JVM funcione a alta velocidad y alcance un estado estable lo antes posible. Para lograr este objetivo, la JVM utiliza más recursos internos al inicio; sin embargo, requiere una optimización menos adaptable una vez que se alcanza el objetivo. Le recomendamos que utilice esta opción para aplicaciones de larga duración que requieren un uso intensivo de la memoria y que funcionan solas.

Uso:

```
-XXaggressive:<param>
```

<param>	Descripción
`opt`	Programa optimizaciones adaptativas antes y permite nuevas optimizaciones, que se espera sean las predeterminadas en futuras versiones.
`memory`	Configura el sistema de memoria para cargas de trabajo intensivas en memoria y establece una expectativa para habilitar grandes cantidades de recursos de memoria para garantizar un alto rendimiento. JRockit JVM también utilizará páginas grandes, si están disponibles.

-XXallocClearChunks

Esta opción le permite borrar un TLA para referencias y valores en el tiempo de asignación de TLA y obtener previamente el siguiente fragmento. Cuando se declara un entero, una referencia o cualquier otra cosa, tiene un valor predeterminado de 0 o nulo (según el tipo). En el momento adecuado, deberá borrar estas referencias y valores para liberar la memoria del montón para que Java pueda usarla o reutilizarla. Puede hacerlo cuando se asigna el objeto o, al usar esta opción, cuando solicita un nuevo TLA.

Uso:

```
-XXallocClearChunks
```

```
-XXallocClearChunks=<true | false>
```

Lo anterior es una opción booleana y generalmente se recomienda en sistemas IA64; En última instancia, su uso depende de la aplicación. Si desea establecer el tamaño de los trozos borrados, combine esta opción con `-XXallocClearChunkSize` . Si usa este indicador pero no especifica un valor booleano, el valor predeterminado es `true` .

-XXallocClearChunkSize

Cuando se usa con `-XXallocClearChunkSize` , esta opción establece el tamaño de los trozos que se borrarán. Si se utiliza este indicador pero no se especifica ningún valor, el valor predeterminado es 512 bytes.

Uso:

```
-XXallocClearChunks -XXallocClearChunkSize=<size>[k|K][m|M][g|G]
```

-XXcallProfiling

Esta opción permite el uso de perfiles de llamadas para optimizaciones de código. El perfil registra estadísticas útiles de tiempo de ejecución específicas para la aplicación y puede, en muchos casos, aumentar el rendimiento porque JVM puede actuar sobre esas estadísticas.

> Nota: esta opción es compatible con JRockit JVM R27.3.0 y la versión posterior. Puede convertirse en predeterminado en futuras versiones.

Uso:

```
java -XXcallProfiling myApp
```

Esta opción está deshabilitada por defecto. Debes habilitarlo para usarlo.

-XXdisableFatSpin

Esta opción deshabilita el código de giro de bloqueo de grasa en Java, permitiendo que los hilos que bloquean el intento de adquirir un bloqueo de grasa se duerman directamente.

Los objetos en Java se convierten en un bloqueo tan pronto como cualquier hilo entra en un bloque sincronizado en ese objeto. Todos los bloqueos se mantienen (es decir, permanecen bloqueados) hasta que se liberan por el hilo de bloqueo. Si el bloqueo no se va a liberar muy rápido, se puede inflar a un "bloqueo grueso". El "giro" se produce cuando un hilo que quiere un bloqueo específico comprueba continuamente ese bloqueo para ver si aún está en uso. Bucle apretado ya que hace el cheque. Girar contra un bloqueo de grasa es generalmente beneficioso,

aunque, en algunos casos, puede ser costoso y puede afectar el rendimiento. `-XXdisableFatSpin` permite desactivar el giro contra un bloqueo grueso y eliminar el impacto potencial de rendimiento.

Uso:

```
-XXdisableFatSpin
```

-XXdisponibilidad a las características de la visión

Esta opción deshabilita los cambios de estrategia del recolector de basura. Esta opción no afecta a las heurísticas de compactación ni a las de tamaño de vivero. Por defecto, las heurísticas de recolección de basura están habilitadas.

Uso:

```
-XXdisableFatSpin
```

-XXdumpSize

Esta opción hace que se genere un archivo de volcado y le permite especificar el tamaño relativo de ese archivo (es decir, pequeño, mediano o grande).

Uso:

```
-XXdumpsize:<size>
```

<tamaño>	Descripción
`none`	No genera un archivo de volcado.
`small`	En Windows, se genera un pequeño archivo de volcado (en Linux se genera un volcado de núcleo completo). Un pequeño volcado solo incluye las pilas de hilos, incluidas sus huellas y muy poco más. Este fue el valor predeterminado en JRockit JVM 8.1 con los paquetes de servicio 1 y 2, así como 7.0 con el paquete de servicio 3 y superior).
`normal`	Hace que se genere un volcado normal en todas las plataformas. Este archivo de volcado incluye toda la memoria excepto el montón java. Este es el valor predeterminado para JRockit JVM 1.4.2 y versiones posteriores.
`large`	Incluye todo lo que está en la memoria, incluido el montón de Java. Esta opción hace que `-XXdumpSize` sea equivalente a `-XXdumpFullState` .

-XXexitOnOutOfMemory

Esta opción hace que JRockit JVM salga en la primera aparición de un error de falta de memoria.

Se puede usar si prefiere reiniciar una instancia de JRockit JVM en lugar de eliminar los errores de memoria. Ingrese este comando al inicio para forzar a JRockit JVM a salir en la primera vez que ocurra un error de falta de memoria.

Uso:

```
-XXexitOnOutOfMemory
```

Lea Banderas JVM en línea: https://riptutorial.com/es/java/topic/2500/banderas-jvm

Capítulo 19: BigDecimal

Introducción

La clase BigDecimal proporciona operaciones para aritmética (sumar, restar, multiplicar, dividir), manipulación de escala, redondeo, comparación, hash y conversión de formato. El BigDecimal representa números decimales firmados de precisión arbitraria e inmutables. Esta clase se utilizará en una necesidad de cálculo de alta precisión.

Examples

Los objetos BigDecimal son inmutables.

Si desea calcular con BigDecimal, tiene que usar el valor devuelto porque los objetos BigDecimal son inmutables:

```
BigDecimal a = new BigDecimal("42.23");
BigDecimal b = new BigDecimal("10.001");

a.add(b); // a will still be 42.23

BigDecimal c = a.add(b); // c will be 52.231
```

Comparando BigDecimals

El método `compareTo` debe usarse para comparar `BigDecimals` :

```
BigDecimal a = new BigDecimal(5);
a.compareTo(new BigDecimal(0));    // a is greater, returns 1
a.compareTo(new BigDecimal(5));    // a is equal, returns 0
a.compareTo(new BigDecimal(10));   // a is less, returns -1
```

Comúnmente **no se** debe utilizar la `equals` método ya que considera dos `BigDecimals` iguales sólo si son iguales en valor y también **de la escala:**

```
BigDecimal a = new BigDecimal(5);
a.equals(new BigDecimal(5));       // value and scale are equal, returns true
a.equals(new BigDecimal(5.00));    // value is equal but scale is not, returns false
```

Operaciones matemáticas con BigDecimal.

Este ejemplo muestra cómo realizar operaciones matemáticas básicas con BigDecimals.

1.Adición

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a + b
BigDecimal result = a.add(b);
System.out.println(result);
```

Resultado: 12

2.Subtraction

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a - b
BigDecimal result = a.subtract(b);
System.out.println(result);
```

Resultado: -2

3.Multiplicacion

Al multiplicar dos `BigDecimal` s, el resultado tendrá una escala igual a la suma de las escalas de los operandos.

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

//Equivalent to result = a * b
BigDecimal result = a.multiply(b);
System.out.println(result);
```

Resultado: 36.89931

Para cambiar la escala del resultado, utilice el método de multiplicación sobrecargada que permite pasar `MathContext` , un objeto que describe las reglas para los operadores, en particular la precisión y el modo de redondeo del resultado. Para obtener más información sobre los modos de redondeo disponibles, consulte la documentación de Oracle.

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

MathContext returnRules = new MathContext(4, RoundingMode.HALF_DOWN);

//Equivalent to result = a * b
BigDecimal result = a.multiply(b, returnRules);
System.out.println(result);
```

Resultado: 36.90

4.Division

La división es un poco más complicada que las otras operaciones aritméticas, por ejemplo, considere el siguiente ejemplo:

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

BigDecimal result = a.divide(b);
System.out.println(result);
```

Esperamos que esto dé algo similar a: 0.7142857142857143, pero obtendríamos:

> Resultado: **java.lang.ArithmeticException: Expansión decimal sin terminación; No hay un resultado decimal representable exacto.**

Esto funcionaría perfectamente bien cuando el resultado fuera un decimal de terminación, por ejemplo, si quisiera dividir 5 por 2, pero para aquellos números que al dividir darían un resultado no final, obtendríamos una `ArithmeticException` . En el escenario del mundo real, uno no puede predecir los valores que se encontrarían durante la división, por lo que necesitamos especificar la **Escala** y el **Modo de redondeo** para la división BigDecimal. Para obtener más información sobre la escala y el modo de redondeo, consulte la documentación de Oracle .

Por ejemplo, podría hacer:

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a / b (Upto 10 Decimal places and Round HALF_UP)
BigDecimal result = a.divide(b,10,RoundingMode.HALF_UP);
System.out.println(result);
```

> Resultado: 0.7142857143

5.El resto o módulo

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a % b
BigDecimal result = a.remainder(b);
System.out.println(result);
```

> Resultado: 5

6. Poder

```
BigDecimal a = new BigDecimal("5");

//Equivalent to result = a^10
BigDecimal result = a.pow(10);
System.out.println(result);
```

Resultado: 9765625

7.Max

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = MAX(a,b)
BigDecimal result = a.max(b);
System.out.println(result);
```

Resultado: 7

8.Min

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = MIN(a,b)
BigDecimal result = a.min(b);
System.out.println(result);
```

Resultado: 5

9. Mover punto a la izquierda

```
BigDecimal a = new BigDecimal("5234.49843776");

//Moves the decimal point to 2 places left of current position
BigDecimal result = a.movePointLeft(2);
System.out.println(result);
```

Resultado: 52.3449843776

10. Mover punto a la derecha

```
BigDecimal a = new BigDecimal("5234.49843776");

//Moves the decimal point to 3 places right of current position
BigDecimal result = a.movePointRight(3);
```

```
System.out.println(result);
```

Resultado: 5234498.43776

Existen muchas más opciones y una combinación de parámetros para los ejemplos mencionados anteriormente (por ejemplo, hay 6 variaciones del método de división), este conjunto no es una lista exhaustiva y cubre algunos ejemplos básicos.

Usando BigDecimal en lugar de flotar

Debido a la forma en que el tipo flotante se representa en la memoria de la computadora, los resultados de las operaciones que utilizan este tipo pueden ser inexactos, algunos valores se almacenan como aproximaciones. Buenos ejemplos de esto son los cálculos monetarios. Si es necesaria una alta precisión, deben usarse otros tipos. por ejemplo, Java 7 proporciona BigDecimal.

```
import java.math.BigDecimal;

public class FloatTest {

public static void main(String[] args) {
    float accountBalance = 10000.00f;
    System.out.println("Operations using float:");
    System.out.println("1000 operations for 1.99");
    for(int i = 0; i<1000; i++){
        accountBalance -= 1.99f;
    }
    System.out.println(String.format("Account balance after float operations: %f",
accountBalance));

    BigDecimal accountBalanceTwo = new BigDecimal("10000.00");
    System.out.println("Operations using BigDecimal:");
    System.out.println("1000 operations for 1.99");
    BigDecimal operation = new BigDecimal("1.99");
    for(int i = 0; i<1000; i++){
        accountBalanceTwo = accountBalanceTwo.subtract(operation);
    }
    System.out.println(String.format("Account balance after BigDecimal operations: %f",
accountBalanceTwo));
}
```

La salida de este programa es:

```
Operations using float:
1000 operations for 1.99
Account balance after float operations: 8009,765625
Operations using BigDecimal:
1000 operations for 1.99
Account balance after BigDecimal operations: 8010,000000
```

Para un saldo inicial de 10000.00, después de 1000 operaciones por 1.99, esperamos que el saldo sea 8010.00. El uso del tipo de coma flotante nos da una respuesta en torno al 8009.77, lo que es inaceptablemente impreciso en el caso de los cálculos monetarios. Usar BigDecimal nos

da el resultado adecuado.

BigDecimal.valueOf ()

La clase BigDecimal contiene un caché interno de los números utilizados con frecuencia, por ejemplo, de 0 a 10. Los métodos BigDecimal.valueOf () se proporcionan con preferencia a los constructores con parámetros de tipo similar, es decir, en el ejemplo siguiente se prefiere a b.

```
BigDecimal a = BigDecimal.valueOf(10L); //Returns cached Object reference
BigDecimal b = new BigDecimal(10L); //Does not return cached Object reference

BigDecimal a = BigDecimal.valueOf(20L); //Does not return cached Object reference
BigDecimal b = new BigDecimal(20L); //Does not return cached Object reference


BigDecimal a = BigDecimal.valueOf(15.15); //Preferred way to convert a double (or float) into
a BigDecimal, as the value returned is equal to that resulting from constructing a BigDecimal
from the result of using Double.toString(double)
BigDecimal b = new BigDecimal(15.15); //Return unpredictable result
```

Inicialización de BigDecimals con valor cero, uno o diez.

BigDecimal proporciona propiedades estáticas para los números cero, uno y diez. Es una buena práctica usar estos en lugar de usar los números reales:

- `BigDecimal.ZERO`
- `BigDecimal.ONE`
- `BigDecimal.TEN`

Al usar las propiedades estáticas, evitas una creación de instancias innecesaria, también tienes un literal en tu código en lugar de un 'número mágico'.

```
//Bad example:
BigDecimal bad0 = new BigDecimal(0);
BigDecimal bad1 = new BigDecimal(1);
BigDecimal bad10 = new BigDecimal(10);

//Good Example:
BigDecimal good0 = BigDecimal.ZERO;
BigDecimal good1 = BigDecimal.ONE;
BigDecimal good10 = BigDecimal.TEN;
```

Lea BigDecimal en línea: https://riptutorial.com/es/java/topic/1667/bigdecimal

Capítulo 20: BigInteger

Introducción

La clase `BigInteger` se usa para operaciones matemáticas que involucran enteros grandes con magnitudes demasiado grandes para tipos de datos primitivos. Por ejemplo, 100 factorial es 158 dígitos, mucho más grande de lo que puede representar un `long` . `BigInteger` proporciona análogos a todos los operadores enteros primitivos de Java y todos los métodos relevantes de `java.lang.Math` , así como algunas otras operaciones.

Sintaxis

- BigInteger variable_name = new BigInteger ("12345678901234567890"); // un entero decimal como una cadena
- BigInteger variable_name = new BigInteger ("1010101101010101010100110001100111010110001111100001011010010", 2) // un entero binario como una cadena
- BigInteger variable_name = new BigInteger ("ab54a98ceb1f0800", 16) // un entero hexadecimal como una cadena
- BigInteger variable_name = new BigInteger (64, new Random ()); // un generador de números pseudoaleatorios que suministra 64 bits para construir un entero
- BigInteger variable_name = new BigInteger (new byte [] {0, -85, 84, -87, -116, -21, 31, 10, -46}); // firmó la representación complementaria de dos de un entero (big endian)
- BigInteger variable_name = new BigInteger (1, nuevo byte [] {- 85, 84, -87, -116, -21, 31, 10, -46}); // representación de complemento de dos sin signo de un entero positivo (big endian)

Observaciones

`BigInteger` es inmutable. Por lo tanto no puedes cambiar su estado. Por ejemplo, lo siguiente no funcionará ya que la `sum` no se actualizará debido a la inmutabilidad.

```
BigInteger sum = BigInteger.ZERO;
for(int i = 1; i < 5000; i++) {
   sum.add(BigInteger.valueOf(i));
}
```

Asigna el resultado a la variable `sum` para que funcione.

```
sum = sum.add(BigInteger.valueOf(i));
```

Java SE 8

La documentación oficial de `BigInteger` indica que las implementaciones de `BigInteger` deben admitir todos los enteros entre $-2^{2147483647}$ y $2^{2147483647}$ (exclusivo). ¡Esto significa que

`BigInteger` puede tener más de 2 *mil millones de* bits!

Examples

Inicialización

La clase `java.math.BigInteger` proporciona operaciones análogas a todos los operadores de enteros primitivos de Java y para todos los métodos relevantes de `java.lang.Math` . Como el paquete `java.math` no está disponible automáticamente, es posible que tenga que importar `java.math.BigInteger` antes de poder usar el nombre de clase simple.

Para convertir valores `long` o `int` en `BigInteger` use:

```
long longValue = Long.MAX_VALUE;
BigInteger valueFromLong = BigInteger.valueOf(longValue);
```

o, para enteros:

```
int intValue = Integer.MIN_VALUE; // negative
BigInteger valueFromInt = BigInteger.valueOf(intValue);
```

que *ampliarán* el `intValue` número entero de largo, utilizando la extensión bit de signo para los valores negativos, por lo que los valores negativos permanecerán negativo.

Para convertir una `String` numérica a `BigInteger` use:

```
String decimalString = "-1";
BigInteger valueFromDecimalString = new BigInteger(decimalString);
```

El siguiente constructor se utiliza para traducir la representación de cadena de un `BigInteger` en el radix especificado en un `BigInteger` .

```
String binaryString = "10";
int binaryRadix = 2;
BigInteger valueFromBinaryString = new BigInteger(binaryString , binaryRadix);
```

Java también admite la conversión directa de bytes a una instancia de `BigInteger` . Actualmente solo se puede usar la codificación big endian firmada y sin firmar:

```
byte[] bytes = new byte[] { (byte) 0x80 };
BigInteger valueFromBytes = new BigInteger(bytes);
```

Esto generará una instancia de `BigInteger` con valor -128 ya que el primer bit se interpreta como el bit de signo.

```
byte[] unsignedBytes = new byte[] { (byte) 0x80 };
int sign = 1; // positive
```

```
BigInteger valueFromUnsignedBytes = new BigInteger(sign, unsignedBytes);
```

Esto generará una instancia de `BigInteger` con valor 128, ya que los bytes se interpretan como un número sin firma y el signo se establece explícitamente en 1, un número positivo.

Hay constantes predefinidas para valores comunes:

- `BigInteger.ZERO` - valor de "0".
- `BigInteger.ONE` - valor de "1".
- `BigInteger.TEN` - valor de "10".

También hay `BigInteger.TWO` (valor de "2"), pero no puedes usarlo en tu código porque es `private` .

Comparando BigIntegers

Puede comparar `BigIntegers` igual que compara `String` u otros objetos en Java.

Por ejemplo:

```
BigInteger one = BigInteger.valueOf(1);
BigInteger two = BigInteger.valueOf(2);

if(one.equals(two)){
    System.out.println("Equal");
}
else{
    System.out.println("Not Equal");
}
```

Salida:

```
Not Equal
```

Nota:

En general, **no** use el operador `==` para comparar BigIntegers

- operador `==` : compara referencias; es decir, si dos valores se refieren al mismo objeto
- Método `equals()` : compara el contenido de dos BigIntegers.

Por ejemplo, BigIntegers **no** debe compararse de la siguiente manera:

```
if (firstBigInteger == secondBigInteger) {
  // Only checks for reference equality, not content equality!
}
```

Si lo hace, puede provocar un comportamiento inesperado, ya que el operador `==` solo comprueba la igualdad de referencia. Si ambos BigIntegers contienen el mismo contenido, pero no se refieren al mismo objeto, **esto fallará.** En su lugar, compare BigIntegers utilizando los métodos `equals` , como se explicó anteriormente.

También puede comparar su `BigInteger` con valores constantes como 0,1,10.

por ejemplo:

```
BigInteger reallyBig = BigInteger.valueOf(1);
if(BigInteger.ONE.equals(reallyBig)){
    //code when they are equal.
}
```

También puede comparar dos BigIntegers usando el `compareTo()` , como sigue: `compareTo()` devuelve 3 valores.

- **0:** Cuando ambos son **iguales** .
- **1:** Cuando el primero es **mayor que el** segundo (el que está entre paréntesis).
- **-1:** Cuando el primero es **menor que el** segundo.

```
BigInteger reallyBig = BigInteger.valueOf(10);
BigInteger reallyBig1 = BigInteger.valueOf(100);

if(reallyBig.compareTo(reallyBig1) == 0){
    //code when both are equal.
}
else if(reallyBig.compareTo(reallyBig1) == 1){
    //code when reallyBig is greater than reallyBig1.
}
else if(reallyBig.compareTo(reallyBig1) == -1){
    //code when reallyBig is less than reallyBig1.
}
```

Ejemplos de operaciones matemáticas de BigInteger

BigInteger se encuentra en un objeto inmutable, por lo que debe asignar los resultados de cualquier operación matemática a una nueva instancia de BigInteger.

Adición: *10 + 10 = 20*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("10");

BigInteger sum = value1.add(value2);
System.out.println(sum);
```

salida: 20

Sustracción: *10 - 9 = 1*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("9");

BigInteger sub = value1.subtract(value2);
System.out.println(sub);
```

salida: 1

División: *10/5 = 2*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

salida: 2

División: *17/4 = 4*

```
BigInteger value1 = new BigInteger("17");
BigInteger value2 = new BigInteger("4");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

salida: 4

Multiplicación: *10 * 5 = 50*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger mul = value1.multiply(value2);
System.out.println(mul);
```

salida: 50

Potencia: *10 ^ 3 = 1000*

```
BigInteger value1 = new BigInteger("10");
BigInteger power = value1.pow(3);
System.out.println(power);
```

salida: 1000

Resto: *10% 6 = 4*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("6");

BigInteger power = value1.remainder(value2);
System.out.println(power);
```

salida: 4

GCD: el divisor común más grande (GCD) para `12` y `18` es `6` .

```
BigInteger value1 = new BigInteger("12");
BigInteger value2 = new BigInteger("18");
```

```
System.out.println(value1.gcd(value2));
```

Salida: 6

Máximo de dos BigIntegers:

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.max(value2));
```

Salida: 11

Mínimo de dos BigIntegers:

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.min(value2));
```

Salida: 10

Operaciones de lógica binaria en BigInteger

BigInteger admite las operaciones lógicas binarias que también están disponibles para los tipos de `Number` . Al igual que con todas las operaciones, se implementan llamando a un método.

Binario o

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.or(val2);
```

Salida: 11 (que es equivalente a `10 | 9`)

Binario y

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.and(val2);
```

Salida: 8 (que es equivalente a `10 & 9`)

Xor binario:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");
```

```
val1.xor(val2);
```

Salida: 3 (que es equivalente a `10 ^ 9`)

Giro a la derecha:

```
BigInteger val1 = new BigInteger("10");

val1.shiftRight(1);   // the argument be an Integer
```

Salida: 5 (equivalente a `10 >> 1`)

Shift izquierdo:

```
BigInteger val1 = new BigInteger("10");

val1.shiftLeft(1);   // here parameter should be Integer
```

Salida: 20 (equivalente a `10 << 1`)

Inversión binaria (no):

```
BigInteger val1 = new BigInteger("10");

val1.not();
```

Salida: 5

*NAND (y no): **

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.andNot(val2);
```

Salida: 7

Generando BigIntegers aleatorios

La clase `BigInteger` tiene un constructor dedicado a generar `BigIntegers` aleatorios, dada una instancia de `java.util.Random` y un `int` que especifica cuántos bits tendrá el `BigInteger` . Su uso es bastante simple: cuando se llama al constructor `BigInteger(int, Random)` siguiente manera:

```
BigInteger randomBigInt = new BigInteger(bitCount, sourceOfRandomness);
```

luego terminarás con un `BigInteger` cuyo valor está entre 0 (inclusive) y 2 `bitCount` (exclusivo).

Esto también significa que el `new BigInteger(2147483647, sourceOfRandomness)` puede devolver todo `BigInteger` positivo dado suficiente tiempo.

¿Cuál será la `sourceOfRandomness` La `sourceOfRandomness` depende de usted. Por ejemplo, un `new Random()` es suficientemente bueno en la mayoría de los casos:

```
new BigInteger(32, new Random());
```

Si está dispuesto a renunciar a la velocidad para obtener números aleatorios de mayor calidad, puede usar un `new SecureRandom ()` lugar:

```
import java.security.SecureRandom;

// somewhere in the code...
new BigInteger(32, new SecureRandom());
```

¡Incluso puede implementar un algoritmo sobre la marcha con una clase anónima! Tenga en cuenta que el **despliegue de su propio algoritmo RNG *lo* terminará con una aleatoriedad de baja calidad** , por lo que siempre asegúrese de usar un algoritmo que se demuestre que sea decente a menos que desee que los `BigInteger` resultantes sean predecibles.

```
new BigInteger(32, new Random() {
    int seed = 0;

    @Override
    protected int next(int bits) {
        seed = ((22695477 * seed) + 1) & 2147483647; // Values shamelessly stolen from
Wikipedia
        return seed;
    }
});
```

Lea BigInteger en línea: https://riptutorial.com/es/java/topic/1514/biginteger

Capítulo 21: ByteBuffer

Introducción

La clase `ByteBuffer` se introdujo en Java 1.4 para facilitar el trabajo en datos binarios. Es especialmente adecuado para usar con datos de tipo primitivo. Permite la creación, pero también la manipulación posterior de un `byte[]` s en un nivel de abstracción superior

Sintaxis

- byte [] arr = new byte [1000];
- ByteBuffer buffer = ByteBuffer.wrap (arr);
- ByteBuffer buffer = ByteBuffer.allocate (1024);
- ByteBuffer buffer = ByteBuffer.allocateDirect (1024);
- byte b = buffer.get ();
- byte b = buffer.get (10);
- corto s = buffer.getShort (10);
- buffer.put ((byte) 120);
- buffer.putChar ('a');

Examples

Uso básico - Creación de un ByteBuffer

Hay dos formas de crear un `ByteBuffer` , donde uno puede subdividirse de nuevo.

Si ya tiene un `byte[]` existente, puede *"envolverlo"* en un `ByteBuffer` para simplificar el procesamiento:

```
byte[] reqBuffer = new byte[BUFFER_SIZE];
int readBytes = socketInputStream.read(reqBuffer);
final ByteBuffer reqBufferWrapper = ByteBuffer.wrap(reqBuffer);
```

Esta sería una posibilidad para el código que maneja las interacciones de redes de bajo nivel.

Si no tiene un `byte[]` ya existente `byte[]` , puede crear un `ByteBuffer` sobre una matriz que está asignada específicamente para el búfer de esta manera:

```
final ByteBuffer respBuffer = ByteBuffer.allocate(RESPONSE_BUFFER_SIZE);
putResponseData(respBuffer);
socketOutputStream.write(respBuffer.array());
```

Si la ruta del código es extremadamente crítica para el rendimiento y necesita **acceso directo a la memoria del sistema** , `ByteBuffer` puede incluso asignar búferes *directos* utilizando `#allocateDirect()`

Uso básico - Escribir datos en el búfer

Dado un `ByteBuffer` ejemplo uno puede escribir datos de tipo primitivo a él utilizando *relativa* y *absoluta* `put` . La sorprendente diferencia es que al poner los datos utilizando el método *relativo* , se realiza un seguimiento del índice en el que se insertan los datos, mientras que el método absoluto siempre requiere proporcionar un índice para `put` los datos.

Ambos métodos permiten *"encadenar"* llamadas. Dado un búfer suficientemente dimensionado, se puede hacer lo siguiente:

```
buffer.putInt(0xCAFEBABE).putChar('c').putFloat(0.25).putLong(0xDEADBEEFCAFEBABE);
```

que es equivalente a:

```
buffer.putInt(0xCAFEBABE);
buffer.putChar('c');
buffer.putFloat(0.25);
buffer.putLong(0xDEADBEEFCAFEBABE);
```

Tenga en cuenta que el método que opera en `byte` s no tiene un nombre especial. Además, tenga en cuenta que también es válido pasar tanto un `ByteBuffer` como un `byte[]` para `put` . Aparte de eso, todos los tipos primitivos tienen métodos de `put` especializados.

Una nota adicional: el índice dado cuando se usa `put*` absoluto `put*` siempre se cuenta en `byte` .

Uso básico - Uso de DirectByteBuffer

`DirectByteBuffer` es una implementación especial de `ByteBuffer` que no tiene `byte[]` debajo.

Podemos asignar dicho ByteBuffer llamando a:

```
ByteBuffer directBuffer = ByteBuffer.allocateDirect(16);
```

Esta operación asignará 16 bytes de memoria. El contenido de los buffers directos *puede* residir fuera del montón de recolección de basura normal.

Podemos verificar si ByteBuffer es directo llamando a:

```
directBuffer.isDirect(); // true
```

La principal característica de `DirectByteBuffer` es que JVM intentará trabajar de forma nativa en la memoria asignada sin ningún búfer adicional, por lo que las operaciones realizadas en él pueden ser más rápidas que las realizadas en ByteBuffers con arrays ubicados debajo.

Se recomienda usar `DirectByteBuffer` con operaciones de IO pesadas que dependen de la velocidad de ejecución, como la comunicación en tiempo real.

Debemos tener en cuenta que si intentamos utilizar el método `array()` obtendremos la

`UnsupportedOperationException` . Por lo tanto, es una buena práctica comprobar si nuestro ByteBuffer lo tiene (matriz de bytes) antes de intentar acceder a él:

```
byte[] arrayOfBytes;
if(buffer.hasArray()) {
    arrayOfBytes = buffer.array();
}
```

Otro uso del buffer de bytes directo es interoperar a través de JNI. Como un búfer de bytes directo no usa un `byte[]` , sino un bloque de memoria real, es posible acceder a esa memoria directamente a través de un puntero en el código nativo. Esto puede ahorrarle un poco de problemas y gastos generales al calcular entre Java y la representación nativa de los datos.

La interfaz JNI define varias funciones para manejar buffers de bytes directos: Soporte NIO .

Lea ByteBuffer en línea: https://riptutorial.com/es/java/topic/702/bytebuffer

Capítulo 22: Calendario y sus subclases

Observaciones

A partir de Java 8, `Calendar` y sus subclases han sido reemplazados por el paquete java.time y sus subpaquetes. Deberían ser preferibles, a menos que una API heredada requiera Calendario.

Examples

Creando objetos de calendario

`Calendar` objetos del `Calendar` se pueden crear usando `getInstance()` o usando el constructor `GregorianCalendar` .

Es importante tener en cuenta que los meses en el `Calendar` están basados en cero, lo que significa que ENERO se representa con un valor `int` 0. Para proporcionar un mejor código, siempre use las constantes del `Calendar` , como el `Calendar.JANUARY` . ENERO para evitar malentendidos.

```
Calendar calendar = Calendar.getInstance();
Calendar gregorianCalendar = new GregorianCalendar();
Calendar gregorianCalendarAtSpecificDay = new GregorianCalendar(2016, Calendar.JANUARY, 1);
Calendar gregorianCalendarAtSpecificDayAndTime = new GregorianCalendar(2016, Calendar.JANUARY,
1, 6, 55, 10);
```

Nota : utilice siempre las constantes del mes: la representación numérica es engañosa , por ejemplo, `Calendar.JANUARY` tiene el valor `0`

Aumentar / disminuir los campos del calendario

`add()` y `roll()` se pueden usar para aumentar / disminuir los campos del `Calendar` .

```
Calendar calendar = new GregorianCalendar(2016, Calendar.MARCH, 31); // 31 March 2016
```

El método `add()` afecta a todos los campos y se comporta de manera efectiva si se agregaran o restaran fechas reales del calendario

```
calendar.add(Calendar.MONTH, -6);
```

La operación anterior elimina seis meses del calendario y nos lleva al 30 de septiembre de 2015.

Para cambiar un campo en particular sin afectar a los otros campos, use `roll()` .

```
calendar.roll(Calendar.MONTH, -6);
```

La operación anterior elimina seis meses del *mes* actual, por lo que el mes se identifica como

septiembre. Ningún otro campo ha sido ajustado; El año no ha cambiado con esta operación.

Encontrando AM / PM

Con la clase Calendario es fácil encontrar AM o PM.

```
Calendar cal = Calendar.getInstance();
cal.setTime(new Date());
if (cal.get(Calendar.AM_PM) == Calendar.PM)
    System.out.println("It is PM");
```

Restando calendarios

Para obtener una diferencia entre dos `Calendar` , use el método `getTimeInMillis()` :

```
Calendar c1 = Calendar.getInstance();
Calendar c2 = Calendar.getInstance();
c2.set(Calendar.DATE, c2.get(Calendar.DATE) + 1);

System.out.println(c2.getTimeInMillis() - c1.getTimeInMillis()); //outputs 86400000 (24 * 60 *
60 * 1000)
```

Lea Calendario y sus subclases en línea: https://riptutorial.com/es/java/topic/165/calendario-y-sus-subclases

Capítulo 23: Características de Java SE 7

Introducción

En este tema, encontrará un resumen de las nuevas características agregadas al lenguaje de programación Java en Java SE 7. Hay muchas otras características nuevas en otros campos como JDBC y Java Virtual Machine (JVM) que no se cubrirán. en este tema.

Observaciones

Mejoras en Java SE 7

Examples

Nuevas características del lenguaje de programación Java SE 7.

- Literales binarios : los tipos integrales (byte, short, int y long) también se pueden expresar utilizando el sistema de números binarios. Para especificar un literal binario, agregue el prefijo 0b o 0B al número.
- Cadenas en declaraciones de switch : puede usar un objeto String en la expresión de una declaración de switch
- La declaración try-with-resources : La declaración try-with-resources es una declaración try que declara uno o más recursos. Un recurso es como un objeto que debe cerrarse después de que el programa haya terminado con él. La declaración try-with-resources asegura que cada recurso se cierre al final de la declaración. Cualquier objeto que implemente java.lang.AutoCloseable, que incluye todos los objetos que implementan java.io.Closeable, puede usarse como un recurso.
- Captura de múltiples tipos de excepciones y reingreso de excepciones con la verificación de tipos mejorada : un solo bloque de captura puede manejar más de un tipo de excepción. Esta característica puede reducir la duplicación de código y disminuir la tentación de atrapar una excepción demasiado amplia.
- Subrayados en literales numéricos : cualquier número de guiones bajos (_) puede aparecer en cualquier lugar entre los dígitos en un literal numérico. Esta función le permite, por ejemplo, separar grupos de dígitos en literales numéricos, lo que puede mejorar la legibilidad de su código.
- Inferencia de tipos para la creación de instancias genéricas : puede reemplazar los argumentos de tipo requeridos para invocar al constructor de una clase genérica con un conjunto vacío de parámetros de tipo (<>) siempre que el compilador pueda inferir los argumentos de tipo del contexto. Este par de soportes angulares se llama informalmente el diamante.
- Advertencias y errores mejorados del compilador cuando se usan parámetros formales no confiables con métodos de Varargs

Literales binarios

```
// An 8-bit 'byte' value:
byte aByte = (byte)0b00100001;

// A 16-bit 'short' value:
short aShort = (short)0b1010000101000101;

// Some 32-bit 'int' values:
int anInt1 = 0b10100001010001011010000101000101;
int anInt2 = 0b101;
int anInt3 = 0B101; // The B can be upper or lower case.

// A 64-bit 'long' value. Note the "L" suffix:
long aLong = 0b1010000101000101101000010100010110100001010001011010000101000101L;
```

La declaración de prueba con recursos

El ejemplo lee la primera línea de un archivo. Utiliza una instancia de `BufferedReader` para leer datos del archivo. `BufferedReader` es un recurso que debe cerrarse una vez que el programa haya terminado con él:

```
static String readFirstLineFromFile(String path) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

En este ejemplo, el recurso declarado en la declaración try-with-resources es un `BufferedReader` . La declaración de declaración aparece entre paréntesis inmediatamente después de la palabra clave try. La clase `BufferedReader` , en Java SE 7 y versiones posteriores, implementa la interfaz `java.lang.AutoCloseable` . Debido a que la instancia de `BufferedReader` se declara en una sentencia try-with-resource, se cerrará sin importar si la sentencia de prueba se completa normalmente o de manera abrupta (como resultado del método `BufferedReader.readLine` una `IOException`).

Subrayados en literales numéricos

El siguiente ejemplo muestra otras formas en que puede usar el guión bajo en literales numéricos:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

Puede colocar guiones bajos solo entre dígitos; No puedes colocar guiones bajos en los siguientes lugares:

- Al principio o al final de un número
- Adyacente a un punto decimal en un literal de punto flotante
- Antes de un sufijo F o L

- En posiciones donde se espera una cadena de dígitos

Inferencia de tipos para la creación de instancias genéricas

Puedes usar

```
Map<String, List<String>> myMap = new HashMap<>();
```

en lugar de

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

Sin embargo, no puedes usar

```
List<String> list = new ArrayList<>();
list.add("A");

  // The following statement should fail since addAll expects
  // Collection<? extends String>

list.addAll(new ArrayList<>());
```

porque no se puede compilar. Tenga en cuenta que el diamante a menudo funciona en las llamadas de método; sin embargo, se sugiere que use el diamante principalmente para declaraciones de variables.

Cadenas en las instrucciones del interruptor

```
public String getTypeOfDayWithSwitchStatement(String dayOfWeekArg) {
     String typeOfDay;
     switch (dayOfWeekArg) {
         case "Monday":
             typeOfDay = "Start of work week";
             break;
         case "Tuesday":
         case "Wednesday":
         case "Thursday":
             typeOfDay = "Midweek";
             break;
         case "Friday":
             typeOfDay = "End of work week";
             break;
         case "Saturday":
         case "Sunday":
             typeOfDay = "Weekend";
             break;
         default:
             throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeekArg);
     }
     return typeOfDay;
}
```

Lea Características de Java SE 7 en línea:

https://riptutorial.com/es/java/topic/8272/caracteristicas-de-java-se-7

Capítulo 24: Características de Java SE 8

Introducción

En este tema, encontrará un resumen de las nuevas características agregadas al lenguaje de programación Java en Java SE 8. Hay muchas otras características nuevas en otros campos, como JDBC y Java Virtual Machine (JVM) que no se cubrirán en este tema.

Observaciones

Referencia: Mejoras en Java SE 8.

Examples

Nuevas características del lenguaje de programación Java SE 8.

- Lambda Expressions , una nueva característica de lenguaje, se ha introducido en esta versión. Le permiten tratar la funcionalidad como un argumento de método, o un código como datos. Las expresiones Lambda le permiten expresar instancias de interfaces de un solo método (denominadas interfaces funcionales) de manera más compacta.
    - Las referencias de los métodos proporcionan expresiones lambda fáciles de leer para los métodos que ya tienen un nombre.
    - Los métodos predeterminados permiten agregar nuevas funciones a las interfaces de las bibliotecas y garantizar la compatibilidad binaria con el código escrito para las versiones anteriores de esas interfaces.
    - Las API nuevas y mejoradas que aprovechan las expresiones y secuencias de Lambda en Java SE 8 describen clases nuevas y mejoradas que aprovechan las expresiones y secuencias de lambda.
- Inferencia de tipos mejorada: el compilador de Java aprovecha la tipificación de destino para inferir los parámetros de tipo de una invocación de método genérico. El tipo de destino de una expresión es el tipo de datos que el compilador de Java espera dependiendo de dónde aparezca la expresión. Por ejemplo, puede usar el tipo de destino de una declaración de asignación para la inferencia de tipo en Java SE 7. Sin embargo, en Java SE 8, puede usar el tipo de destino para la inferencia de tipo en más contextos.
    - Mecanografía de objetivos en expresiones Lambda
    - Inferencia de tipos
- Las anotaciones repetidas permiten aplicar el mismo tipo de anotación más de una vez a la misma declaración o tipo de uso.
- Las anotaciones de tipo ofrecen la posibilidad de aplicar una anotación en cualquier lugar donde se use un tipo, no solo en una declaración. Usada con un sistema de tipo enchufable, esta característica permite una mejor verificación de tipos de su código.
- Reflexión de parámetros de método : puede obtener los nombres de los parámetros formales de cualquier método o constructor con el método `java.lang.reflect.Executable.getParameters` . (Las clases Method y Constructor extienden la

clase Executable y, por lo tanto, heredan el método `Executable.getParameters`) Sin embargo, los archivos `.class` no almacenan nombres de parámetros formales de forma predeterminada. Para almacenar nombres de parámetros formales en un archivo `.class` particular, y así habilitar la API de Reflection para recuperar nombres de parámetros formales, compile el archivo de origen con la opción `-parameters` del compilador javac.
- Date-time-api - Se agregó una nueva api de tiempo en `java.time` . Si se usa esto, no es necesario que designe zona horaria.

Lea Características de Java SE 8 en línea:
https://riptutorial.com/es/java/topic/8267/caracteristicas-de-java-se-8

Capítulo 25: Cargadores de clases

Observaciones

Un cargador de clases es una clase cuyo propósito principal es mediar la ubicación y carga de las clases utilizadas por una aplicación. Un cargador de clases también puede encontrar y cargar *recursos* .

Las clases estándar del cargador de clases pueden cargar clases y recursos de directorios en el sistema de archivos y de archivos JAR y ZIP. También pueden descargar y almacenar en caché los archivos JAR y ZIP desde un servidor remoto.

Los cargadores de clases normalmente están encadenados, de modo que la JVM intentará cargar las clases de las bibliotecas de clases estándar con preferencia a las fuentes proporcionadas por la aplicación. Los cargadores de clases personalizados permiten que el programador altere esto. También puede hacer cosas como descifrar archivos de bytecode y modificación de bytecode.

Examples

Instalar y usar un cargador de clases

Este ejemplo básico muestra cómo una aplicación puede instanciar un cargador de clases y usarlo para cargar dinámicamente una clase.

```
URL[] urls = new URL[] {new URL("file:/home/me/extras.jar")};
Classloader loader = new URLClassLoader(urls);
Class<?> myObjectClass = loader.findClass("com.example.MyObject");
```

El cargador de clases creado en este ejemplo tendrá el cargador de clases predeterminado como principal, y primero intentará encontrar cualquier clase en el cargador de clases principal antes de buscar en "extra.jar". Si la clase solicitada ya se ha cargado, la llamada `findClass` devolverá la referencia a la clase cargada anteriormente.

La llamada a `findClass` puede fallar de varias maneras. Los más comunes son:

- Si no se puede encontrar la clase nombrada, la llamada con la `ClassNotFoundException` .
- Si la clase nombrada depende de alguna otra clase que no se puede encontrar, la llamada lanzará `NoClassDefFoundError` .

Implementando un classLoader personalizado

Cada cargador personalizado debe extender directa o indirectamente la clase `java.lang.ClassLoader` . Los principales *puntos de extensión* son los siguientes métodos:

- `findClass(String)` : sobrecargue este método si su cargador de clases sigue el modelo de delegación estándar para la carga de clases.

- `loadClass(String, boolean)` : sobrecargue este método para implementar un modelo de delegación alternativo.
- `findResource` y `findResources` : sobrecargue estos métodos para personalizar la carga de recursos.

Los métodos `defineClass` que son responsables de cargar realmente la clase desde una matriz de bytes son `final` para evitar la sobrecarga. Cualquier comportamiento personalizado debe realizarse antes de llamar a `defineClass` .

Aquí hay un sencillo que carga una clase específica de una matriz de bytes:

```
public class ByteArrayClassLoader extends ClassLoader {
    private String classname;
    private byte[] classfile;

    public ByteArrayClassLoader(String classname, byte[] classfile) {
        this.classname = classname;
        this.classfile = classfile.clone();
    }

    @Override
    protected Class findClass(String classname) throws ClassNotFoundException {
        if (classname.equals(this.classname)) {
            return defineClass(classname, classfile, 0, classfile.length);
        } else {
            throw new ClassNotFoundException(classname);
        }
    }
}
```

Dado que solo hemos anulado el método `findClass` , este cargador de clases personalizado se comportará de la siguiente manera cuando se llame a `loadClass` .

1. El método `loadClass` del cargador de clases llama a `findLoadedClass` para ver si este cargador de clases ya ha cargado una clase con este nombre. Si eso tiene éxito, el objeto `Class` resultante se devuelve al solicitante.
2. El método `loadClass` luego delega al cargador de clases principal llamando a su llamada `loadClass` . Si el padre puede manejar la solicitud, devolverá un objeto de `Class` que luego se devolverá al solicitante.
3. Si el cargador de clases principal no puede cargar la clase, `findClass` luego llama a nuestro método `findClass` , pasando el nombre de la clase a cargar.
4. Si el nombre solicitado coincide con `this.classname` , llamamos a `defineClass` para cargar la clase real de la `this.classfile` bytes `this.classfile` . El objeto de `Class` resultante se devuelve.
5. Si el nombre no coincide, lanzamos la `ClassNotFoundException` .

Cargando un archivo .class externo

Para cargar una clase primero necesitamos definirla. La clase está definida por el `ClassLoader` . Solo hay un problema, Oracle no escribió el código del `ClassLoader` con esta característica disponible. Para definir la clase, necesitaremos acceder a un método llamado `defineClass()` que

es un método privado del `ClassLoader` .

Para acceder a él, lo que haremos es crear una nueva clase, `ByteClassLoader` , y extenderla a `ClassLoader` . Ahora que hemos extendido nuestra clase a `ClassLoader` , podemos acceder a los métodos privados de `ClassLoader` . Para hacer que `defineClass()` esté disponible, crearemos un nuevo método que actuará como un espejo para el método `defineClass()` privado. Para llamar al método privado necesitaremos el nombre de la clase, `name` , los bytes de clase, `classBytes` , offset del primer byte, que será `0` porque `classBytes` datos 'comienza en `classBytes[0]` , y offset del último byte, que será `classBytes.lenght` porque representa el tamaño de los datos, que será el último desplazamiento.

```
public class ByteClassLoader extends ClassLoader {

    public Class<?> defineClass(String name, byte[] classBytes) {
        return defineClass(name, classBytes, 0, classBytes.length);
    }

}
```

Ahora, tenemos un `defineClass()` público `defineClass()` . Puede invocarse pasando el nombre de la clase y los bytes de la clase como argumentos.

Digamos que tenemos una clase llamada `MyClass` en el paquete `stackoverflow` ...

Para llamar al método, necesitamos los bytes de la clase, así que creamos un objeto `Path` que representa la ruta de nuestra clase usando el método `Paths.get()` y pasando la ruta de la clase binaria como un argumento. Ahora, podemos obtener los bytes de la clase con `Files.readAllBytes(path)` . Así que creamos una instancia de `ByteClassLoader` y usamos el método que creamos, `defineClass()` . Ya tenemos los bytes de clase, pero para llamar a nuestro método también necesitamos el nombre de clase que viene dado por el nombre del paquete (punto) el nombre canónico de clase, en este caso `stackoverflow.MyClass` .

```
Path path = Paths.get("MyClass.class");

ByteClassLoader loader = new ByteClassLoader();
loader.defineClass("stackoverflow.MyClass", Files.readAllBytes(path);
```

Nota : el método `defineClass()` devuelve un objeto `Class<?>` . Puedes guardarlo si quieres.

Para cargar la clase, solo llamamos a `loadClass()` y pasamos el nombre de la clase. Este método puede lanzar una `ClassNotFoundException` por lo que necesitamos usar un bloque cat de prueba

```
try{
    loader.loadClass("stackoverflow.MyClass");
} catch(ClassNotFoundException e){
    e.printStackTrace();
}
```

Lea Cargadores de clases en línea: https://riptutorial.com/es/java/topic/5443/cargadores-de-clases

Capítulo 26: Cifrado RSA

Examples

Un ejemplo que utiliza un sistema criptográfico híbrido que consiste en OAEP y GCM

El siguiente ejemplo encripta los datos utilizando un sistema criptográfico híbrido que consiste en AES GCM y OAEP, utilizando sus tamaños de parámetros predeterminados y un tamaño de clave AES de 128 bits.

OAEP es menos vulnerable al relleno de los ataques de Oracle que el relleno PKCS # 1 v1.5. GCM también está protegido contra ataques oracle de relleno.

El descifrado se puede realizar recuperando primero la longitud de la clave encapsulada y luego recuperando la clave encapsulada. La clave encapsulada se puede descifrar utilizando la clave privada RSA que forma un par de claves con la clave pública. Después de eso, el texto cifrado cifrado AES / GCM se puede descifrar al texto plano original.

El protocolo consiste en:

1. un campo de longitud para la clave `RSAPrivateKey` (`RSAPrivateKey` pierde un método `getKeySize()`);
2. la clave encapsulada / encapsulada, del mismo tamaño que el tamaño de la clave RSA en bytes;
3. el texto cifrado GCM y la etiqueta de autenticación de 128 bits (agregada automáticamente por Java).

Notas:

- Para utilizar correctamente este código, debe proporcionar una clave RSA de al menos 2048 bits; cuanto más grande, mejor (pero más lento, especialmente durante el descifrado);
- Para utilizar AES-256, primero debe instalar los archivos de políticas de criptografía ilimitada ;
- En lugar de crear su propio protocolo, es posible que desee utilizar un formato de contenedor como la Sintaxis de mensajes criptográficos (CMS / PKCS # 7) o PGP en su lugar.

Así que aquí está el ejemplo:

```
/**
 * Encrypts the data using a hybrid crypto-system which uses GCM to encrypt the data and OAEP
to encrypt the AES key.
 * The key size of the AES encryption will be 128 bit.
 * All the default parameter choices are used for OAEP and GCM.
 *
 * @param publicKey the RSA public key used to wrap the AES key
 * @param plaintext the plaintext to be encrypted, not altered
```

```
 * @return the ciphertext
 * @throws InvalidKeyException if the key is not an RSA public key
 * @throws NullPointerException if the plaintext is null
 */
public static byte[] encryptData(PublicKey publicKey, byte[] plaintext)
        throws InvalidKeyException, NullPointerException {

    // --- create the RSA OAEP cipher ---

    Cipher oaep;
    try {
        // SHA-1 is the default and not vulnerable in this setting
        // use OAEPParameterSpec to configure more than just the hash
        oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for RSA cipher (mandatory algorithm for
runtimes)", e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for OAEP padding (present in the standard Java
runtime sinze XX)", e);
    }
    oaep.init(Cipher.WRAP_MODE, publicKey);

    // --- wrap the plaintext in a buffer

    // will throw NullPointerException if plaintext is null
    ByteBuffer plaintextBuffer = ByteBuffer.wrap(plaintext);

    // --- generate a new AES secret key ---

    KeyGenerator aesKeyGenerator;
    try {
        aesKeyGenerator = KeyGenerator.getInstance("AES");
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for AES key generator (mandatory algorithm for
runtimes)", e);
    }
    // for AES-192 and 256 make sure you've got the rights (install the
    // Unlimited Crypto Policy files)
    aesKeyGenerator.init(128);
    SecretKey aesKey = aesKeyGenerator.generateKey();

    // --- wrap the new AES secret key ---

    byte[] wrappedKey;
    try {
        wrappedKey = oaep.wrap(aesKey);
    } catch (IllegalBlockSizeException e) {
        throw new RuntimeException(
                "AES key should always fit OAEP with normal sized RSA key", e);
    }

    // --- setup the AES GCM cipher mode ---

    Cipher aesGCM;
    try {
        aesGCM = Cipher.getInstance("AES/GCM/Nopadding");
        // we can get away with a zero nonce since the key is randomly generated
```

```
        // 128 bits is the recommended (maximum) value for the tag size
        // 12 bytes (96 bits) is the default nonce size for GCM mode encryption
        GCMParameterSpec staticParameterSpec = new GCMParameterSpec(128, new byte[12]);
        aesGCM.init(Cipher.ENCRYPT_MODE, aesKey, staticParameterSpec);
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for AES cipher (mandatory algorithm for
runtimes)", e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for GCM (present in the standard Java runtime
sinze XX)", e);
    } catch (InvalidAlgorithmParameterException e) {
        throw new RuntimeException(
                "IvParameterSpec not accepted by this implementation of GCM", e);
    }

    // --- create a buffer of the right size for our own protocol ---

    ByteBuffer ciphertextBuffer = ByteBuffer.allocate(
            Short.BYTES
            + oaep.getOutputSize(128 / Byte.SIZE)
            + aesGCM.getOutputSize(plaintext.length));

    // - element 1: make sure that we know the size of the wrapped key
    ciphertextBuffer.putShort((short) wrappedKey.length);

    // - element 2: put in the wrapped key
    ciphertextBuffer.put(wrappedKey);

    // - element 3: GCM encrypt into buffer
    try {
        aesGCM.doFinal(plaintextBuffer, ciphertextBuffer);
    } catch (ShortBufferException | IllegalBlockSizeException | BadPaddingException e) {
        throw new RuntimeException("Cryptographic exception, AES/GCM encryption should not
fail here", e);
    }

    return ciphertextBuffer.array();
}
```

Por supuesto, el cifrado no es muy útil sin descifrado. Tenga en cuenta que esto devolverá información mínima si el descifrado falla.

```
/**
 * Decrypts the data using a hybrid crypto-system which uses GCM to encrypt
 * the data and OAEP to encrypt the AES key. All the default parameter
 * choices are used for OAEP and GCM.
 *
 * @param privateKey
 *            the RSA private key used to unwrap the AES key
 * @param ciphertext
 *            the ciphertext to be encrypted, not altered
 * @return the plaintext
 * @throws InvalidKeyException
 *             if the key is not an RSA private key
 * @throws NullPointerException
 *             if the ciphertext is null
 * @throws IllegalArgumentException
 *             with the message "Invalid ciphertext" if the ciphertext is invalid (minimize
```

```
information leakage)
 */
public static byte[] decryptData(PrivateKey privateKey, byte[] ciphertext)
        throws InvalidKeyException, NullPointerException {

    // --- create the RSA OAEP cipher ---

    Cipher oaep;
    try {
        // SHA-1 is the default and not vulnerable in this setting
        // use OAEPParameterSpec to configure more than just the hash
        oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for RSA cipher (mandatory algorithm for
runtimes)",
                e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for OAEP padding (present in the standard Java
runtime sinze XX)",
                e);
    }
    oaep.init(Cipher.UNWRAP_MODE, privateKey);

    // --- wrap the ciphertext in a buffer

    // will throw NullPointerException if ciphertext is null
    ByteBuffer ciphertextBuffer = ByteBuffer.wrap(ciphertext);

    // sanity check #1
    if (ciphertextBuffer.remaining() < 2) {
        throw new IllegalArgumentException("Invalid ciphertext");
    }
    // - element 1: the length of the encapsulated key
    int wrappedKeySize = ciphertextBuffer.getShort() & 0xFFFF;
    // sanity check #2
    if (ciphertextBuffer.remaining() < wrappedKeySize + 128 / Byte.SIZE) {
        throw new IllegalArgumentException("Invalid ciphertext");
    }

    // --- unwrap the AES secret key ---

    byte[] wrappedKey = new byte[wrappedKeySize];
    // - element 2: the encapsulated key
    ciphertextBuffer.get(wrappedKey);
    SecretKey aesKey;
    try {
        aesKey = (SecretKey) oaep.unwrap(wrappedKey, "AES",
                Cipher.SECRET_KEY);
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for AES cipher (mandatory algorithm for
runtimes)",
                e);
    } catch (InvalidKeyException e) {
        throw new RuntimeException(
                "Invalid ciphertext");
    }

    // --- setup the AES GCM cipher mode ---
```

```
    Cipher aesGCM;
    try {
        aesGCM = Cipher.getInstance("AES/GCM/Nopadding");
        // we can get away with a zero nonce since the key is randomly
        // generated
        // 128 bits is the recommended (maximum) value for the tag size
        // 12 bytes (96 bits) is the default nonce size for GCM mode
        // encryption
        GCMParameterSpec staticParameterSpec = new GCMParameterSpec(128,
                new byte[12]);
        aesGCM.init(Cipher.DECRYPT_MODE, aesKey, staticParameterSpec);
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for AES cipher (mandatory algorithm for
runtimes)",
                e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
                "Runtime doesn't have support for GCM (present in the standard Java runtime
sinze XX)",
                e);
    } catch (InvalidAlgorithmParameterException e) {
        throw new RuntimeException(
                "IvParameterSpec not accepted by this implementation of GCM",
                e);
    }

    // --- create a buffer of the right size for our own protocol ---

    ByteBuffer plaintextBuffer = ByteBuffer.allocate(aesGCM
            .getOutputSize(ciphertextBuffer.remaining()));

    // - element 3: GCM ciphertext
    try {
        aesGCM.doFinal(ciphertextBuffer, plaintextBuffer);
    } catch (ShortBufferException | IllegalBlockSizeException
            | BadPaddingException e) {
        throw new RuntimeException(
                "Invalid ciphertext");
    }

    return plaintextBuffer.array();
}
```

Lea Cifrado RSA en línea: https://riptutorial.com/es/java/topic/1889/cifrado-rsa

Capítulo 27: Clase - Reflexión de Java

Introducción

La clase java.lang.Class proporciona muchos métodos que pueden usarse para obtener metadatos, examinar y cambiar el comportamiento del tiempo de ejecución de una clase.

Los paquetes java.lang y java.lang.reflect proporcionan clases para la reflexión de java.

Donde se usa

La API de reflexión se utiliza principalmente en:

IDE (entorno de desarrollo integrado), por ejemplo, Eclipse, MyEclipse, NetBeans, etc.
Herramientas de prueba de depurador, etc.

Examples

Método getClass () de la clase Object.

```
class Simple { }

class Test {
     void printName(Object obj){
         Class c = obj.getClass();
         System.out.println(c.getName());
     }
     public static void main(String args[]){
         Simple s = new Simple();

         Test t = new Test();
         t.printName(s);
    }
}
```

Lea Clase - Reflexión de Java en línea: https://riptutorial.com/es/java/topic/10151/clase---reflexion-de-java

Capítulo 28: Clase de fecha

Sintaxis

- `Date object = new Date();`
- `Date object = new Date(long date);`

Parámetros

Parámetro	Explicación
Sin parametro	Crea un nuevo objeto Date utilizando el tiempo de asignación (al milisegundo más cercano)
fecha larga	Crea un nuevo objeto Fecha con la hora establecida en el número de milisegundos desde "la época" (1 de enero de 1970, 00:00:00 GMT)

Observaciones

Representación

Internamente, un objeto de fecha Java se representa como un largo; es el número de milisegundos desde un tiempo específico (conocido como la *época*). La clase de fecha de Java original tenía métodos para tratar con zonas horarias, etc., pero estos estaban en desuso en favor de la nueva clase de calendario.

Por lo tanto, si todo lo que desea hacer en su código es una hora específica, puede crear una clase de Fecha y almacenarla, etc. Si desea imprimir una versión legible por humanos de esa fecha, sin embargo, cree una clase de Calendario. y use su formato para producir horas, minutos, segundos, días, zonas horarias, etc. Recuerde que un milisegundo específico se muestra como horas diferentes en diferentes zonas horarias; Normalmente, usted desea mostrar uno en la zona horaria "local", pero los métodos de formato deben tener en cuenta que es posible que desee mostrarlo para otro.

También tenga en cuenta que los relojes utilizados por las JVM no suelen tener una precisión de milisegundos; el reloj solo puede "marcar" cada 10 milisegundos y, por lo tanto, si sincroniza las cosas, no puede confiar en medirlas con precisión a ese nivel.

Declaración de importación

```
import java.util.Date;
```

La clase de `Date` se puede importar del paquete `java.util` .

Precaución

`Date` instancias de `Date` son mutables, por lo que su uso puede dificultar la escritura de código seguro para subprocesos o puede proporcionar accidentalmente acceso de escritura al estado interno. Por ejemplo, en la siguiente clase, el método `getDate()` permite a la persona que llama modificar la fecha de la transacción:

```
public final class Transaction {
  private final Date date;

  public Date getTransactionDate() {
    return date;
  }
}
```

La solución es devolver una copia del campo de `date` o usar las nuevas API en `java.time` introducidas en Java 8.

La mayoría de los métodos de construcción en la clase `Date` han quedado en desuso y no deben usarse. En casi todos los casos, es recomendable utilizar la clase `Calendar` para las operaciones de fecha.

Java 8

Java 8 introduce una nueva API de fecha y hora en el paquete `java.time` , que incluye LocalDate y LocalTime . Las clases en el paquete `java.time` proporcionan una API revisada que es más fácil de usar. Si está escribiendo en Java 8, le recomendamos encarecidamente que utilice esta nueva API. Consulte Fechas y hora (java.time. *) .

Examples

Creando objetos de fecha

```
Date date = new Date();
System.out.println(date); // Thu Feb 25 05:03:59 IST 2016
```

Aquí, este objeto `Date` contiene la fecha y hora actuales en que se creó este objeto.

```
Calendar calendar = Calendar.getInstance();
calendar.set(90, Calendar.DECEMBER, 11);
Date myBirthDate = calendar.getTime();
System.out.println(myBirthDate); // Mon Dec 31 00:00:00 IST 1990
```

`Date` objetos de `Date` se crean mejor a través de una instancia de `Calendar` , ya que el uso de los constructores de datos está obsoleto y no se recomienda. Para hacerlo, necesitamos obtener una instancia de la clase `Calendar` del método de fábrica. Luego podemos establecer el año, mes y día del mes utilizando números o, en el caso de las constantes de los meses que se proporcionan en la clase Calendario, para mejorar la legibilidad y reducir los errores.

```
calendar.set(90, Calendar.DECEMBER, 11, 8, 32, 35);
Date myBirthDatenTime = calendar.getTime();
```

```
System.out.println(myBirthDatenTime); // Mon Dec 31 08:32:35 IST 1990
```

Junto con la fecha, también podemos pasar el tiempo en el orden de horas, minutos y segundos.

Comparando objetos de fecha

Calendario, fecha y fecha local

Java SE 8

Antes, después, compare y métodos iguales.

```
//Use of Calendar and Date objects
final Date today = new Date();
final Calendar calendar = Calendar.getInstance();
calendar.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date birthdate = calendar.getTime();

final Calendar calendar2 = Calendar.getInstance();
calendar2.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date samebirthdate = calendar2.getTime();

//Before example
System.out.printf("Is %1$tF before %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.before(birthdate)));
System.out.printf("Is %1$tF before %1$tF? %3$b%n", today, today,
Boolean.valueOf(today.before(today)));
System.out.printf("Is %2$tF before %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.before(today)));

//After example
System.out.printf("Is %1$tF after %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.after(birthdate)));
System.out.printf("Is %1$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.after(today)));
System.out.printf("Is %2$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.after(today)));

//Compare example
System.out.printf("Compare %1$tF to %2$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(birthdate)));
System.out.printf("Compare %1$tF to %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(today)));
System.out.printf("Compare %2$tF to %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(birthdate.compareTo(today)));

//Equal example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.equals(birthdate)));
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", birthdate, samebirthdate,
            Boolean.valueOf(birthdate.equals(samebirthdate)));
System.out.printf(
            "Because birthdate.getTime() -> %1$d is different from samebirthdate.getTime() ->
%2$d, there are millisecondes!%n",
```

```
            Long.valueOf(birthdate.getTime()), Long.valueOf(samebirthdate.getTime()));

//Clear ms from calendars
calendar.clear(Calendar.MILLISECOND);
calendar2.clear(Calendar.MILLISECOND);
birthdate = calendar.getTime();
samebirthdate = calendar2.getTime();

System.out.printf("Is %1$tF equal to %2$tF after clearing ms? %3$b%n", birthdate,
samebirthdate,
            Boolean.valueOf(birthdate.equals(samebirthdate)));
```

Java SE 8

isBefore, isAfter, compareTo y es igual a métodos

```
//Use of LocalDate
final LocalDate now = LocalDate.now();
final LocalDate birthdate2 = LocalDate.of(2012, 6, 30);
final LocalDate birthdate3 = LocalDate.of(2012, 6, 30);

//Hours, minutes, second and nanoOfsecond can also be configured with an other class
LocalDateTime
//LocalDateTime.of(year, month, dayOfMonth, hour, minute, second, nanoOfSecond);

//isBefore example
System.out.printf("Is %1$tF before %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(birthdate2)));
System.out.printf("Is %1$tF before %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(now)));
System.out.printf("Is %2$tF before %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isBefore(now)));

//isAfter example
System.out.printf("Is %1$tF after %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(birthdate2)));
System.out.printf("Is %1$tF after %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(now)));
System.out.printf("Is %2$tF after %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isAfter(now)));

//compareTo example
System.out.printf("Compare %1$tF to %2$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(birthdate2)));
System.out.printf("Compare %1$tF to %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(now)));
System.out.printf("Compare %2$tF to %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(birthdate2.compareTo(now)));

//equals example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.equals(birthdate2)));
System.out.printf("Is %1$tF to %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.equals(birthdate3)));
```

```
//isEqual example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isEqual(birthdate2)));
System.out.printf("Is %1$tF to %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.isEqual(birthdate3)));
```

Comparación de fechas antes de Java 8

Antes de Java 8, las fechas podían compararse utilizando las clases java.util.Calendar y java.util.Date . La clase de fecha ofrece 4 métodos para comparar fechas:

- después (fecha en que)
- antes (fecha en que)
- compareTo (fecha de otra fecha)
- es igual a (objeto obj)

`after` , `before` , `compareTo` y `equals` métodos comparan los valores devueltos por getTime () método para cada fecha.

`compareTo` método `compareTo` devuelve un entero positivo.

- Valor mayor que 0: cuando la fecha es posterior al argumento de fecha
- Valor mayor que 0: cuando la fecha es anterior al argumento de fecha
- El valor es igual a 0: cuando la fecha es igual al argumento de la fecha

`equals` resultados `equals` pueden ser sorprendentes como se muestra en el ejemplo porque los valores, como los milisegundos, no se inicializan con el mismo valor si no se dan explícitamente.

Desde Java 8

Con Java 8, está disponible un nuevo Objeto para trabajar con Fecha java.time.LocalDate . `LocalDate` implementa ChronoLocalDate , la representación abstracta de una fecha donde la cronología, o sistema de calendario, es conectable.

Para tener la precisión de fecha y hora, se debe usar el objeto java.time.LocalDateTime . `LocalDate` y `LocalDateTime` usan el mismo nombre de método para comparar.

La comparación de fechas con `LocalDate` es diferente de usar `ChronoLocalDate` porque la cronología o el sistema de calendario no se tienen en cuenta en la primera.

Debido a que la mayoría de las aplicaciones deben usar `LocalDate` , `ChronoLocalDate` no se incluye en los ejemplos. Lectura adicional aquí .

> La mayoría de las aplicaciones deben declarar firmas de métodos, campos y variables como LocalDate, no esta interfaz [ChronoLocalDate].

`LocalDate` tiene 5 métodos para comparar fechas:

- isAfter (ChronoLocalDate other)
- isBefore (ChronoLocalDate other)
- isEqual (ChronoLocalDate otro)
- compareTo (ChronoLocalDate otro)
- es igual a (objeto obj)

En el caso del parámetro `LocalDate` , `isAfter` , `isBefore` , `isEqual` , `equals` y `compareTo` ahora use este método:

```
int compareTo0(LocalDate otherDate) {
    int cmp = (year - otherDate.year);
    if (cmp == 0) {
        cmp = (month - otherDate.month);
        if (cmp == 0) {
            cmp = (day - otherDate.day);
        }
    }
    return cmp;
}
```

`equals` verificación de método si la referencia del parámetro es igual a la fecha primero, mientras que `isEqual` llama directamente a `compareTo0` .

En el caso de una instancia de otra clase de `ChronoLocalDate` las fechas se comparan utilizando el `Epoch Day` la `Epoch Day` . El recuento del día de la época es un recuento incremental simple de días donde el día 0 es 1970-01-01 (ISO).

Convertir la fecha a un determinado formato de cadena

`format()` de la clase `SimpleDateFormat` ayuda a convertir un objeto `Date` en cierto objeto `String` formato utilizando la *cadena de patrón* suministrada.

```
Date today = new Date();

SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MMM-yy"); //pattern is specified here
System.out.println(dateFormat.format(today)); //25-Feb-16
```

Los patrones se pueden aplicar de nuevo utilizando `applyPattern()`

```
dateFormat.applyPattern("dd-MM-yyyy");
System.out.println(dateFormat.format(today)); //25-02-2016

dateFormat.applyPattern("dd-MM-yyyy HH:mm:ss E");
System.out.println(dateFormat.format(today)); //25-02-2016 06:14:33 Thu
```

Nota: Aquí `mm` (letra minúscula m) denota minutos y `MM` (mayúscula M) denota mes. Preste mucha atención al formatear los años: el capital "Y" (`Y`) indica la "semana del año", mientras que la minúscula "y" (`y`) indica el año.

Convertir cadena en fecha

`parse()` de la clase `SimpleDateFormat` ayuda a convertir un patrón de `String` en un objeto `Date` .

```
DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.SHORT, Locale.US);
String dateStr = "02/25/2016"; // input String
Date date = dateFormat.parse(dateStr);
System.out.println(date.getYear()); // 116
```

Hay 4 estilos diferentes para el formato de texto, `SHORT` , `MEDIUM` (este es el valor predeterminado), `LONG` y `FULL` , todos los cuales dependen de la configuración regional. Si no se especifica una configuración regional, se utiliza la configuración regional predeterminada del sistema.

Estilo	Locale.US	Locale.France
CORTO	30/6/09	30/06/09
MEDIO	Jun 30, 2009	30 juin 2009
LARGO	30 de junio de 2009	30 juin 2009
COMPLETO	Martes 30 de junio de 2009	mardi 30 juin 2009

Una fecha básica de salida.

Usando el siguiente código con la cadena de formato `yyyy/MM/dd hh:mm.ss` , recibiremos la siguiente salida

> 2016/04/19 11: 45.36

```
// define the format to use
String formatString = "yyyy/MM/dd hh:mm.ss";

// get a current date object
Date date = Calendar.getInstance().getTime();

// create the formatter
SimpleDateFormat simpleDateFormat = new SimpleDateFormat(formatString);

// format the date
String formattedDate = simpleDateFormat.format(date);

// print it
System.out.println(formattedDate);


// single-line version of all above code
System.out.println(new SimpleDateFormat("yyyy/MM/dd
hh:mm.ss").format(Calendar.getInstance().getTime()));
```

Convertir la representación de cadena formateada de la fecha en el objeto Fecha

Este método se puede usar para convertir una representación de cadena formateada de una

fecha en un objeto `Date` .

```
    /**
     * Parses the date using the given format.
     *
     * @param formattedDate the formatted date string
     * @param dateFormat the date format which was used to create the string.
     * @return the date
     */
    public static Date parseDate(String formattedDate, String dateFormat) {
        Date date = null;
        SimpleDateFormat objDf = new SimpleDateFormat(dateFormat);
        try {
          date = objDf.parse(formattedDate);
        } catch (ParseException e) {
          // Do what ever needs to be done with exception.
        }
        return date;
    }
```

Creando una fecha específica

Si bien la clase de fecha de Java tiene varios constructores, notará que la mayoría está en desuso. La única forma aceptable de crear una instancia de Fecha directamente es usando el constructor vacío o pasando un largo (número de milisegundos desde el tiempo base estándar). Tampoco son útiles a menos que esté buscando la fecha actual o tenga otra instancia de Date ya disponible.

Para crear una nueva fecha, necesitará una instancia de Calendario. Desde allí puede establecer la instancia de Calendario en la fecha que necesite.

```
Calendar c = Calendar.getInstance();
```

Esto devuelve un nuevo conjunto de instancias de Calendario a la hora actual. El calendario tiene muchos métodos para mutar su fecha y hora o establecerlo de forma absoluta. En este caso, lo estableceremos en una fecha específica.

```
c.set(1974, 6, 2, 8, 0, 0);
Date d = c.getTime();
```

El método `getTime` devuelve la instancia de fecha que necesitamos. Tenga en cuenta que los métodos de configuración del calendario solo establecen uno o más campos, no los configuran todos. Es decir, si establece el año, los otros campos permanecerán sin cambios.

TRAMPA

En muchos casos, este fragmento de código cumple su propósito, pero tenga en cuenta que dos partes importantes de la fecha / hora no están definidas.

- los parámetros `(1974, 6, 2, 8, 0, 0)` se interpretan dentro de la zona horaria predeterminada, definida en otro lugar,

- los milisegundos no se establecen en cero, pero se llenan desde el reloj del sistema en el momento en que se crea la instancia de Calendario.

Objetos Java 8 LocalDate y LocalDateTime

Los objetos Date y LocalDate **no se** pueden convertir *exactamente* entre sí ya que un objeto Date representa un día y una hora específicos, mientras que un objeto LocalDate no contiene información de hora o zona horaria. Sin embargo, puede ser útil convertir entre los dos si solo te importa la información de la fecha real y no la información de la hora.

Crea una fecha local

```
// Create a default date
LocalDate lDate = LocalDate.now();

// Creates a date from values
lDate = LocalDate.of(2017, 12, 15);

// create a date from string
lDate = LocalDate.parse("2017-12-15");

// creates a date from zone
LocalDate.now(ZoneId.systemDefault());
```

Crea un LocalDateTime

```
// Create a default date time
LocalDateTime lDateTime = LocalDateTime.now();

// Creates a date time from values
lDateTime = LocalDateTime.of(2017, 12, 15, 11, 30);

// create a date time from string
lDateTime = LocalDateTime.parse("2017-12-05T11:30:30");

// create a date time from zone
LocalDateTime.now(ZoneId.systemDefault());
```

Fecha local hasta la fecha y viceversa

```
Date date = Date.from(Instant.now());
ZoneId defaultZoneId = ZoneId.systemDefault();

// Date to LocalDate
LocalDate localDate = date.toInstant().atZone(defaultZoneId).toLocalDate();

// LocalDate to Date
Date.from(localDate.atStartOfDay(defaultZoneId).toInstant());
```

LocalDateTime to Date y viceversa

```
Date date = Date.from(Instant.now());
ZoneId defaultZoneId = ZoneId.systemDefault();
```

```
// Date to LocalDateTime
LocalDateTime localDateTime =  date.toInstant().atZone(defaultZoneId).toLocalDateTime();

// LocalDateTime to Date
Date out = Date.from(localDateTime.atZone(defaultZoneId).toInstant());
```

Zonas horarias y java.util.Date

Un objeto `java.util.Date` *no* tiene un concepto de zona horaria.

- No hay manera de **establecer** una zona horaria para una fecha
- No hay forma de **cambiar** la zona horaria de un objeto Date
- Un objeto Date creado con el `new Date()` constructor predeterminado `new Date()` se inicializará con la hora actual en la zona horaria predeterminada del sistema

Sin embargo, es posible mostrar la fecha representada por el punto en el tiempo descrito por el objeto Date en una zona horaria diferente utilizando, por ejemplo, `java.text.SimpleDateFormat` :

```
Date date = new Date();
//print default time zone
System.out.println(TimeZone.getDefault().getDisplayName());
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"); //note: time zone not in
format!
//print date in the original time zone
System.out.println(sdf.format(date));
//current time in London
sdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));
System.out.println(sdf.format(date));
```

Salida:

```
Central European Time
2016-07-21 22:50:56
2016-07-21 21:50:56
```

Convertir java.util.Date a java.sql.Date

`java.util.Date` to `java.sql.Date` conversión suele ser necesaria cuando un objeto Date debe escribirse en una base de datos.

`java.sql.Date` es un envoltorio alrededor de un valor de milisegundos y `JDBC` utiliza para identificar un tipo de `SQL DATE`

En el siguiente ejemplo, usamos el constructor `java.util.Date()` , que crea un objeto Date y lo inicializa para representar el tiempo en el milisegundo más cercano. Esta fecha se utiliza en el método `convert(java.util.Date utilDate)` para devolver un objeto `java.sql.Date`

Ejemplo

```
public class UtilToSqlConversion {
```

```
    public static void main(String args[])
    {
        java.util.Date utilDate = new java.util.Date();
        System.out.println("java.util.Date is : " + utilDate);
        java.sql.Date sqlDate = convert(utilDate);
        System.out.println("java.sql.Date is : " + sqlDate);
        DateFormat df = new SimpleDateFormat("dd/MM/YYYY - hh:mm:ss");
        System.out.println("dateFormated date is : " + df.format(utilDate));
    }

    private static java.sql.Date convert(java.util.Date uDate) {
        java.sql.Date sDate = new java.sql.Date(uDate.getTime());
        return sDate;
    }

}
```

Salida

```
java.util.Date is : Fri Jul 22 14:40:35 IST 2016
java.sql.Date is : 2016-07-22
dateFormated date is : 22/07/2016 - 02:40:35
```

`java.util.Date` tiene información de fecha y hora, mientras que `java.sql.Date` solo tiene información de fecha

Hora local

Para usar solo la parte de tiempo de una fecha use LocalTime. Puede crear una instancia de un objeto LocalTime en un par de maneras

1. `LocalTime time = LocalTime.now();`
2. `time = LocalTime.MIDNIGHT;`
3. `time = LocalTime.NOON;`
4. `time = LocalTime.of(12, 12, 45);`

`LocalTime` también tiene un método toString incorporado que muestra el formato muy bien.

```
System.out.println(time);
```

También puede obtener, sumar y restar horas, minutos, segundos y nanosegundos del objeto LocalTime, es decir,

```
time.plusMinutes(1);
time.getMinutes();
time.minusMinutes(1);
```

Puedes convertirlo en un objeto Date con el siguiente código:

```
LocalTime lTime = LocalTime.now();
Instant instant = lTime.atDate(LocalDate.of(A_YEAR, A_MONTH, A_DAY)).
        atZone(ZoneId.systemDefault()).toInstant();
Date time = Date.from(instant);
```

esta clase funciona muy bien dentro de una clase de temporizador para simular un reloj de alarma.

Lea Clase de fecha en línea: https://riptutorial.com/es/java/topic/164/clase-de-fecha

Capítulo 29: Clase de objetos Métodos y constructor

Introducción

Esta página de documentación sirve para mostrar detalles con ejemplos sobre constructores de clases Java y sobre Métodos de clases de objetos que se heredan automáticamente de la Superclase `Object` de cualquier clase recién creada.

Sintaxis

- Clase nativa final pública <?> getClass ()
- pública final vacío nativo notificar ()
- vacío nativo final público notificar a todos ()
- la espera de final nativa pública pública (tiempo de espera largo) lanza InterruptedException
- pública final void wait () lanza InterruptedException
- la espera de final público (tiempo de espera largo, int nanos) lanza la excepción interrumpida
- código nativo int local ()
- booleanos públicos iguales (objeto obj)
- Cadena pública a la cadena ()
- El objeto nativo protegido clone () lanza la excepción CloneNotSupportedException
- vacío vacío finalizado () lanza Throwable

Examples

método toString ()

El método `toString()` se usa para crear una representación de `String` de un objeto utilizando el contenido del objeto. Este método debe ser anulado al escribir su clase. `toString()` se llama implícitamente cuando un objeto se concatena a una cadena como en `"hello " + anObject` .

Considera lo siguiente:

```
public class User {
    private String firstName;
    private String lastName;

    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return firstName + " " + lastName;
```

```
    }

    public static void main(String[] args) {
        User user = new User("John", "Doe");
        System.out.println(user.toString()); // Prints "John Doe"
    }
}
```

Aquí `toString()` de la clase `Object` se reemplaza en la clase `User` para proporcionar datos significativos sobre el objeto al imprimirlo.

Cuando se utiliza `println()` , el método `toString()` del objeto se llama implícitamente. Por lo tanto, estas declaraciones hacen lo mismo:

```
System.out.println(user); // toString() is implicitly called on `user`
System.out.println(user.toString());
```

Si el `toString()` no se reemplaza en la clase de `User` mencionada anteriormente, `System.out.println(user)` puede devolver `User@659e0bfd` o una cadena similar casi sin información útil, excepto el nombre de la clase. Esto se debe a que la llamada usará la implementación `toString()` de la clase de `Object` Java base que no sabe nada sobre la estructura de la clase de `User` o las reglas de negocios. Si desea cambiar esta funcionalidad en su clase, simplemente anule el método.

método igual ()

TL; DR

`==` pruebas de igualdad de referencia (si son el ***mismo objeto***)

`.equals()` comprueba la igualdad de valores (si son ***lógicamente "iguales"***)

`equals()` es un método usado para comparar dos objetos para la igualdad. La implementación predeterminada del método `equals()` en la clase `Object` devuelve `true` si y solo si ambas referencias apuntan a la misma instancia. Por lo tanto, se comporta igual que la comparación por `==` .

```
public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }

    public static void main(String[] args) {
        Foo foo1 = new Foo(0, 0, "bar");
        Foo foo2 = new Foo(0, 0, "bar");

        System.out.println(foo1.equals(foo2)); // prints false
```

```
    }
}
```

Aunque `foo1` y `foo2` se crean con los mismos campos, apuntan a dos objetos diferentes en la memoria. La implementación por defecto de `equals()` por lo tanto, se evalúa como `false` .

Para comparar el contenido de un objeto para la igualdad, se debe anular `equals()` .

```
public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }

        Foo f = (Foo) obj;
        return field1 == f.field1 &&
               field2 == f.field2 &&
               (field3 == null ? f.field3 == null : field3.equals(f.field3));
    }

    @Override
    public int hashCode() {
        int hash = 1;
        hash = 31 * hash + this.field1;
        hash = 31 * hash + this.field2;
        hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
        return hash;
    }

    public static void main(String[] args) {
        Foo foo1 = new Foo(0, 0, "bar");
        Foo foo2 = new Foo(0, 0, "bar");

        System.out.println(foo1.equals(foo2)); // prints true
    }
}
```

Aquí el método `equals()` anulado decide que los objetos son iguales si sus campos son iguales.

Observe que el método `hashCode()` también se sobrescribió. El contrato para ese método establece que cuando dos objetos son iguales, sus valores hash también deben ser iguales. Es por eso que uno casi siempre debe reemplazar `hashCode()` y `equals()` juntos.

Preste especial atención al tipo de argumento del método `equals` . Es el `Object obj` , no `Foo obj` . Si

coloca este último en su método, eso no es una anulación del método `equals` .

Al escribir su propia clase, tendrá que escribir una lógica similar cuando se reemplaza a `equals()` y `hashCode()` . La mayoría de los IDE pueden generar esto automáticamente por ti.

Se puede encontrar un ejemplo de una implementación `equals()` en la clase `String` , que forma parte de la API de Java central. En lugar de comparar punteros, la clase `String` compara el contenido de `String` .

Java SE 7

Java 1.7 introdujo la clase `java.util.Objects` que proporciona un método conveniente, `equals` , que compara dos referencias potencialmente `null` , por lo que puede usarse para simplificar implementaciones del método `equals` .

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    Foo f = (Foo) obj;
    return field1 == f.field1 && field2 == f.field2 && Objects.equals(field3, f.field3);
}
```

Comparación de clases

Dado que el método equals puede ejecutarse contra cualquier objeto, una de las primeras cosas que el método hace a menudo (después de verificar si es `null`) es verificar si la clase del objeto que se está comparando coincide con la clase actual.

```
@Override
public boolean equals(Object obj) {
    //...check for null
    if (getClass() != obj.getClass()) {
        return false;
    }
    //...compare fields
}
```

Esto se hace normalmente como se muestra arriba comparando los objetos de clase. Sin embargo, eso puede fallar en algunos casos especiales que pueden no ser obvios. Por ejemplo, algunos marcos generan proxies dinámicos de clases y estos proxies dinámicos son en realidad una clase diferente. Aquí hay un ejemplo usando JPA.

```
Foo detachedInstance = ...
Foo mergedInstance = entityManager.merge(detachedInstance);
if (mergedInstance.equals(detachedInstance)) {
    //Can never get here if equality is tested with getClass()
```

```
    //as mergedInstance is a proxy (subclass) of Foo
}
```

Un mecanismo para evitar esa limitación es comparar clases usando `instanceof`

```
@Override
public final boolean equals(Object obj) {
    if (!(obj instanceof Foo)) {
        return false;
    }
    //...compare fields
}
```

Sin embargo, hay algunos escollos que se deben evitar al usar `instanceof` . Como Foo podría tener otras subclases y esas subclases podrían reemplazar a `equals()` , podría entrar en un caso en el que un `Foo` es igual a una `FooSubclass` pero la `FooSubclass Foo` no es igual a `Foo` .

```
Foo foo = new Foo(7);
FooSubclass fooSubclass = new FooSubclass(7, false);
foo.equals(fooSubclass) //true
fooSubclass.equals(foo) //false
```

Esto viola las propiedades de simetría y transitividad y, por lo tanto, es una implementación no válida del método `equals()` . Como resultado, cuando se utiliza `instanceof` , una buena práctica es hacer que el método `equals() final` (como en el ejemplo anterior). Esto asegurará que ninguna anulación de subclase sea `equals()` y viole las suposiciones clave.

método hashCode ()

Cuando una clase de Java anula el método `equals` , también debería anular el método `hashCode` . Como se define en el contrato del método :

- Cada vez que se invoca en el mismo objeto más de una vez durante una ejecución de una aplicación Java, el método `hashCode` debe devolver constantemente el mismo número entero, siempre que no se modifique la información utilizada en comparaciones iguales en el objeto. No es necesario que este número entero permanezca consistente de una ejecución de una aplicación a otra ejecución de la misma aplicación.
- Si dos objetos son iguales según el método `equals(Object)` , entonces llamar al método `hashCode` en cada uno de los dos objetos debe producir el mismo resultado entero.
- No es necesario que si dos objetos son desiguales según el método `equals(Object)` , llamar al método `hashCode` en cada uno de los dos objetos debe producir resultados enteros distintos. Sin embargo, el programador debe tener en cuenta que producir resultados enteros distintos para objetos desiguales puede mejorar el rendimiento de las tablas hash.

Los códigos hash se utilizan en implementaciones de hash como `HashMap` , `HashTable` y `HashSet` . El resultado de la función `hashCode` determina el grupo en el que se colocará un objeto. Estas

implementaciones de hash son más eficientes si la implementación de `hashCode` proporcionada es buena. Una propiedad importante de una buena implementación de `hashCode` es que la distribución de los valores de `hashCode` es uniforme. En otras palabras, existe una pequeña probabilidad de que numerosas instancias se almacenen en el mismo cubo.

Un algoritmo para calcular un valor de código hash puede ser similar al siguiente:

```
public class Foo {
    private int field1, field2;
    private String field3;

    public Foo(int field1, int field2, String field3) {
        this.field1 = field1;
        this.field2 = field2;
        this.field3 = field3;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }

        Foo f = (Foo) obj;
        return field1 == f.field1 &&
               field2 == f.field2 &&
               (field3 == null ? f.field3 == null : field3.equals(f.field3);
    }

    @Override
    public int hashCode() {
        int hash = 1;
        hash = 31 * hash + field1;
        hash = 31 * hash + field2;
        hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
        return hash;
    }
}
```

Usando Arrays.hashCode () como un atajo

Java SE 1.2

En Java 1.2 y superior, en lugar de desarrollar un algoritmo para calcular un código hash, se puede generar uno utilizando `java.util.Arrays#hashCode` proporcionando un objeto o una matriz de primitivas que contiene los valores de campo:

```
@Override
public int hashCode() {
    return Arrays.hashCode(new Object[] {field1, field2, field3});
}
```

Java SE 7

Java 1.7 introdujo la clase `java.util.Objects` que proporciona un método conveniente, `hash(Object... objects)` , que calcula un código hash basado en los valores de los objetos que se le suministran. Este método funciona igual que `java.util.Arrays#hashCode` .

```
@Override
public int hashCode() {
    return Objects.hash(field1, field2, field3);
}
```

Nota: este enfoque es ineficiente y produce objetos de basura cada vez que se llama a su método `hashCode()` personalizado:

- Se crea un `Object[]` temporal `Object[]` . (En la versión `Objects.hash()` , la matriz se crea mediante el mecanismo "varargs").
- Si alguno de los campos son tipos primitivos, deben estar recuadrados y eso puede crear más objetos temporales.
- La matriz debe estar poblada.
- La matriz debe `Arrays.hashCode` `Objects.hash` método `Arrays.hashCode` o `Objects.hash` .
- Las llamadas a `Object.hashCode()` que `Arrays.hashCode` o `Objects.hash` tiene que hacer (probablemente) no se pueden insertar.

Caché interno de códigos hash

Dado que el cálculo del código hash de un objeto puede ser costoso, puede ser atractivo almacenar el valor del código hash en el objeto la primera vez que se calcula. Por ejemplo

```
public final class ImmutableArray {
    private int[] array;
    private volatile int hash = 0;

    public ImmutableArray(int[] initial) {
        array = initial.clone();
    }

    // Other methods

    @Override
    public boolean equals(Object obj) {
         // ...
    }

    @Override
    public int hashCode() {
        int h = hash;
        if (h == 0) {
            h = Arrays.hashCode(array);
            hash = h;
        }
        return h;
    }
}
```

Este enfoque intercambia el costo de (repetidamente) calcular el código hash contra la

sobrecarga de un campo adicional para almacenar en caché el código hash. Si esto se amortiza como una optimización del rendimiento dependerá de la frecuencia con la que se halle (busque) un objeto determinado y otros factores.

También notará que si el verdadero código hash de un `ImmutableArray` es cero (una posibilidad en 2^{32}), el caché no es efectivo.

Finalmente, este enfoque es mucho más difícil de implementar correctamente si el objeto que estamos haciendo hash es mutable. Sin embargo, hay mayores preocupaciones si los códigos hash cambian; Consulte el contrato anterior.

Métodos de esperar () y notificar ()

`wait()` y `notify()` trabajan en tándem: cuando un subproceso llama a `wait()` en un objeto, ese subproceso se bloqueará hasta que otro subproceso llame a `notify()` o `notifyAll()` en ese mismo objeto.

(Ver también: esperar () / notificar ())

```
package com.example.examples.object;

import java.util.concurrent.atomic.AtomicBoolean;

public class WaitAndNotify {

    public static void main(String[] args) throws InterruptedException {
        final Object obj = new Object();
        AtomicBoolean aHasFinishedWaiting = new AtomicBoolean(false);

        Thread threadA = new Thread("Thread A") {
            public void run() {
                System.out.println("A1: Could print before or after B1");
                System.out.println("A2: Thread A is about to start waiting...");
                try {
                    synchronized (obj) { // wait() must be in a synchronized block
                        // execution of thread A stops until obj.notify() is called
                        obj.wait();
                    }
                    System.out.println("A3: Thread A has finished waiting. "
                            + "Guaranteed to happen after B3");
                } catch (InterruptedException e) {
                    System.out.println("Thread A was interrupted while waiting");
                } finally {
                    aHasFinishedWaiting.set(true);
                }
            }
        };

        Thread threadB = new Thread("Thread B") {
            public void run() {
                System.out.println("B1: Could print before or after A1");

                System.out.println("B2: Thread B is about to wait for 10 seconds");
                for (int i = 0; i < 10; i++) {
                    try {
                        Thread.sleep(1000); // sleep for 1 second
```

```
                    } catch (InterruptedException e) {
                        System.err.println("Thread B was interrupted from waiting");
                    }
                }

                System.out.println("B3: Will ALWAYS print before A3 since "
                        + "A3 can only happen after obj.notify() is called.");

                while (!aHasFinishedWaiting.get()) {
                    synchronized (obj) {
                        // notify ONE thread which has called obj.wait()
                        obj.notify();
                    }
                }
            }
        };

        threadA.start();
        threadB.start();

        threadA.join();
        threadB.join();

        System.out.println("Finished!");
    }
}
```

Algunos ejemplos de salida:

```
A1: Could print before or after B1
B1: Could print before or after A1
A2: Thread A is about to start waiting...
B2: Thread B is about to wait for 10 seconds
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

B1: Could print before or after A1
B2: Thread B is about to wait for 10 seconds
A1: Could print before or after B1
A2: Thread A is about to start waiting...
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

A1: Could print before or after B1
A2: Thread A is about to start waiting...
B1: Could print before or after A1
B2: Thread B is about to wait for 10 seconds
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!
```

método getClass ()

El método `getClass()` se puede usar para encontrar el tipo de clase de tiempo de ejecución de un objeto. Vea el ejemplo a continuación:

```
public class User {

    private long userID;
    private String name;

    public User(long userID, String name) {
        this.userID = userID;
        this.name = name;
    }
}

public class SpecificUser extends User {
    private String specificUserID;

    public SpecificUser(String specificUserID, long userID, String name) {
        super(userID, name);
        this.specificUserID = specificUserID;
    }
}

public static void main(String[] args){
    User user = new User(879745, "John");
    SpecificUser specificUser = new SpecificUser("1AAAA", 877777, "Jim");
    User anotherSpecificUser = new SpecificUser("1BBBB", 812345, "Jenny");

    System.out.println(user.getClass()); //Prints "class User"
    System.out.println(specificUser.getClass()); //Prints "class SpecificUser"
    System.out.println(anotherSpecificUser.getClass()); //Prints "class SpecificUser"
}
```

El método `getClass()` devolverá el tipo de clase más específico, por lo que cuando se llama a `getClass()` en `anotherSpecificUser` , el valor devuelto es la `class SpecificUser` porque es más bajo en el árbol de herencia que el `User` .

Es de destacar que, mientras que el método `getClass` se declara como:

```
public final native Class<?> getClass();
```

El tipo estático real devuelto por una llamada a `getClass` es `Class<? extends T>` donde `T` es el tipo estático del objeto en el que se llama a `getClass` .

Es decir, lo siguiente compilará:

```
Class<? extends String> cls = "".getClass();
```

método clone ()

El método `clone()` se utiliza para crear y devolver una copia de un objeto. Este método discutible debe evitarse ya que es problemático y se debe utilizar un constructor de copia o algún otro método para copiar en favor de `clone()` .

Para que el método se use, todas las clases que llaman al método deben implementar la interfaz `Cloneable` .

La interfaz `Cloneable` en sí misma es solo una interfaz de etiqueta utilizada para cambiar el comportamiento del método `native clone()` que verifica si la clase de objetos llamantes implementa `Cloneable` . Si la persona que llama no implementa esta interfaz, se `CloneNotSupportedException` una `CloneNotSupportedException` .

La clase `Object` sí no implementa esta interfaz, por lo que se `CloneNotSupportedException` una `CloneNotSupportedException` si el objeto que llama es de la clase `Object` .

Para que un clon sea correcto, debe ser independiente del objeto desde el cual se está clonando, por lo tanto, puede ser necesario modificar el objeto antes de que se devuelva. Esto significa crear esencialmente una "copia profunda" copiando también cualquiera de los objetos *mutables* que conforman la estructura interna del objeto que se está clonando. Si esto no se implementa correctamente, el objeto clonado no será independiente y tendrá las mismas referencias a los objetos mutables que el objeto desde el que se clonó. Esto resultaría en un comportamiento inconsistente ya que cualquier cambio en aquellos en uno afectaría al otro.

```
class Foo implements Cloneable {
    int w;
    String x;
    float[] y;
    Date z;

    public Foo clone() {
        try {
            Foo result = new Foo();
            // copy primitives by value
            result.w = this.w;
            // immutable objects like String can be copied by reference
            result.x = this.x;

            // The fields y and z refer to a mutable objects; clone them recursively.
            if (this.y != null) {
              result.y = this.y.clone();
            }
            if (this.z != null) {
              result.z = this.z.clone();
            }

            // Done, return the new object
            return result;

        } catch (CloneNotSupportedException e) {
            // in case any of the cloned mutable fields do not implement Cloneable
            throw new AssertionError(e);
        }
    }
}
```

finalizar () método

Este es un método *protegido* y *no estático* de la clase `Object` . Este método se utiliza para realizar algunas operaciones finales o limpiar operaciones en un objeto antes de que se elimine de la memoria.

> Según el documento, este método es llamado por el recolector de basura en un objeto cuando la recolección de basura determina que no hay más referencias al objeto.

Pero no hay garantías de que se llame al método `finalize()` si el objeto aún es accesible o si no se ejecuta ningún recolector de basura cuando el objeto es elegible. Es por eso que es mejor **no confiar** en este método.

En las bibliotecas principales de Java, se pueden encontrar algunos ejemplos de uso, por ejemplo, en `FileInputStream.java` :

```
protected void finalize() throws IOException {
    if ((fd != null) &&  (fd != FileDescriptor.in)) {
        /* if fd is shared, the references in FileDescriptor
         * will ensure that finalizer is only called when
         * safe to do so. All references using the fd have
         * become unreachable. We can call close()
         */
        close();
    }
}
```

En este caso, es la última oportunidad de cerrar el recurso si ese recurso no se ha cerrado antes.

En general, se considera una mala práctica utilizar el método `finalize()` en aplicaciones de cualquier tipo y debe evitarse.

Los finalizadores *no* están destinados a liberar recursos (por ejemplo, cerrar archivos). Se llama al recolector de basura cuando (si!) El sistema se queda con poco espacio de almacenamiento. No se puede confiar en que se llame cuando el sistema se esté quedando sin los manejadores de archivos o, por cualquier otro motivo.

El caso de uso previsto para los finalizadores es para un objeto que está a punto de ser reclamado para notificar a otro objeto sobre su inminente destino. Ahora existe un mejor mecanismo para ese propósito: la clase `java.lang.ref.WeakReference<T>` . Si crees que necesitas escribir un método `finalize()` , entonces deberías investigar si puedes resolver el mismo problema usando `WeakReference` . Si eso no resuelve su problema, es posible que deba volver a pensar su diseño en un nivel más profundo.

Para leer más aquí, hay un artículo sobre el método `finalize()` del libro "Effective Java" de Joshua Bloch.

Constructor de objetos

Todos los constructores en Java deben hacer una llamada al constructor de `Object` . Esto se hace con la llamada `super()` . Esta tiene que ser la primera línea en un constructor. La razón de esto es para que el objeto se pueda crear realmente en el montón antes de que se realice una inicialización adicional.

Si no especifica la llamada a `super()` en un constructor, el compilador la colocará por usted.

Así que los tres de estos ejemplos son funcionalmente idénticos

con llamada explícita a `super()` constructor

```
public class MyClass {

    public MyClass() {
        super();
    }
}
```

con llamada implícita a `super()` constructor

```
public class MyClass {

    public MyClass() {
        // empty
    }
}
```

con constructor implícito

```
public class MyClass {

}
```

¿Qué pasa con el encadenamiento de constructores?

Es posible llamar a otros constructores como la primera instrucción de un constructor. Como tanto la llamada explícita a un súper constructor como la llamada a otro constructor tienen que ser las dos primeras instrucciones, se excluyen mutuamente.

```
public class MyClass {

    public MyClass(int size) {

        doSomethingWith(size);

    }

    public MyClass(Collection<?> initialValues) {

        this(initialValues.size());
        addInitialValues(initialValues);
    }
}
```

Llamar a la nueva `MyClass(Arrays.asList("a", "b", "c"))` llamará al segundo constructor con el argumento Lista, que a su vez delegará al primer constructor (que delegará implícitamente a `super()`) y luego llame a `addInitialValues(int size)` con el segundo tamaño de la lista. Esto se usa para reducir la duplicación de código donde varios constructores necesitan hacer el mismo trabajo.

¿Cómo llamo a un constructor específico?

Dado el ejemplo anterior, uno puede llamar a `new MyClass("argument")` o `new MyClass("argument", 0)` . En otras palabras, al igual que la sobrecarga de métodos , simplemente llame al constructor con los parámetros que son necesarios para su constructor elegido.

> ¿Qué pasará en el constructor de la clase Object?

Nada más de lo que sucedería en una subclase que tiene un constructor vacío predeterminado (menos la llamada a `super()`).

El constructor vacío predeterminado se puede definir explícitamente, pero si no, el compilador lo incluirá siempre que no haya otros constructores definidos.

> ¿Cómo se crea un Objeto a partir del constructor en Objeto?

La creación real de objetos se reduce a la JVM. Cada constructor en Java aparece como un método especial llamado `<init>` que es responsable de la inicialización de la instancia. El compilador suministra este método `<init>` y debido a que `<init>` no es un identificador válido en Java, no se puede usar directamente en el lenguaje.

> ¿Cómo invoca la JVM este método `<init>` ?

La JVM invocará el método `<init>` utilizando la instrucción especial `invokespecial` y solo puede invocarse en instancias de clase sin inicializar.

Para obtener más información, consulte la especificación JVM y la especificación del lenguaje Java:

- Métodos especiales (JVM) - JVMS - 2.9
- Constructores - JLS - 8.8

Lea Clase de objetos Métodos y constructor en línea:
https://riptutorial.com/es/java/topic/145/clase-de-objetos-metodos-y-constructor

Capítulo 30: Clase de propiedades

Introducción

El objeto de propiedades contiene un par de clave y valor tanto como una cadena. La clase java.util.Properties es la subclase de Hashtable.

Se puede utilizar para obtener el valor de la propiedad en función de la clave de propiedad. La clase de propiedades proporciona métodos para obtener datos del archivo de propiedades y almacenar datos en el archivo de propiedades. Por otra parte, se puede utilizar para obtener propiedades del sistema.

Ventaja del archivo de propiedades

No se requiere la compilación, si la información se cambia desde el archivo de propiedades: si se cambia alguna información de

Sintaxis

- En un archivo de propiedades:
- clave = valor
- #comentario

Observaciones

Un objeto de propiedades es un mapa cuyas claves y valores son cadenas por convención. Aunque los métodos de Map se pueden usar para acceder a los datos, los métodos getProperty , setProperty y stringPropertyNames más seguros para el uso de tipos se usan en su lugar.

Las propiedades se almacenan con frecuencia en archivos de propiedades Java, que son archivos de texto simples. Su formato está documentado a fondo en el método Properties.load . En resumen:

- Cada par clave / valor es una línea de texto con espacios en blanco, es igual a (`=`), o colon (`:`) entre la llave y el valor. Los iguales o dos puntos pueden tener cualquier cantidad de espacios en blanco antes y después, que se ignora.
- Los espacios en blanco iniciales siempre se ignoran, los espacios en blanco finales se incluyen siempre.
- Se puede usar una barra invertida para escapar de cualquier carácter (excepto en minúsculas `u`).
- Una barra invertida al final de la línea indica que la siguiente línea es una continuación de la línea actual. Sin embargo, al igual que con todas las líneas, los espacios en blanco iniciales en la línea de continuación se ignoran.
- Al igual que en el código fuente de Java, `\u` seguido de cuatro dígitos hexadecimales representa un carácter UTF-16.

La mayoría de los marcos, incluidas las instalaciones propias de Java SE como java.util.ResourceBundle, cargan archivos de propiedades como InputStreams. Al cargar un archivo de propiedades desde un InputStream, ese archivo solo puede contener caracteres ISO 8859-1 (es decir, caracteres en el rango de 0 a 255). Cualquier otro personaje debe ser representado como `\u` escapa. Sin embargo, puede escribir un archivo de texto en cualquier codificación y usar la herramienta native2ascii (que viene con cada JDK) para hacer ese escape por usted.

Si va a cargar un archivo de propiedades con su propio código, que puede ser en cualquier codificación, siempre y cuando se crea un lector (como un InputStreamReader) basado en el correspondiente conjunto de caracteres . Luego puede cargar el archivo usando load (Reader) en lugar del método de carga heredado (InputStream).

También puede almacenar propiedades en un archivo XML simple, lo que permite que el propio archivo defina la codificación. Dicho archivo se puede cargar con el método loadFromXML . La DTD que describe la estructura de dichos archivos XML se encuentra en http://java.sun.com/dtd/properties.dtd .

Examples

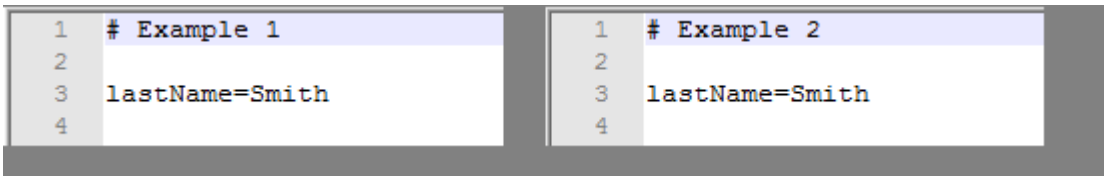
Cargando propiedades

Para cargar un archivo de propiedades empaquetado con su aplicación:

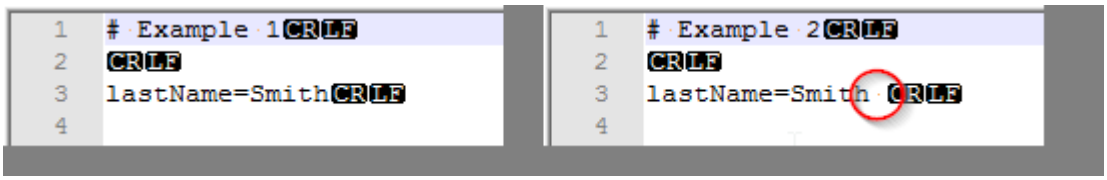
```
public class Defaults {

    public static Properties loadDefaults() {
        try (InputStream bundledResource =
            Defaults.class.getResourceAsStream("defaults.properties")) {

            Properties defaults = new Properties();
            defaults.load(bundledResource);
            return defaults;
        } catch (IOException e) {
            // Since the resource is bundled with the application,
            // we should never get here.
            throw new UncheckedIOException(
                "defaults.properties not properly packaged"
                + " with application", e);
        }
    }

}
```

Los archivos de propiedades advierten: espacios en blanco al final

Eche un vistazo de cerca a estos dos archivos de propiedades que parecen ser completamente idénticos:



excepto que en realidad no son idénticos:



(las capturas de pantalla son de Notepad ++)

Dado que el espacio en blanco se conserva, el valor de `lastName` sería `"Smith"` en el primer caso y `"Smith "` en el segundo.

Muy raramente, esto es lo que los usuarios esperan y uno y solo puede especular por qué este es el comportamiento predeterminado de la clase `Properties` . Sin embargo, es fácil crear una versión mejorada de `Properties` que solucione este problema. La siguiente clase, **TrimmedProperties** , hace justamente eso. Es un reemplazo directo para la clase de propiedades estándar.

```
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.Reader;
import java.util.Map.Entry;
import java.util.Properties;

/**
 * Properties class where values are trimmed for trailing whitespace if the
 * properties are loaded from a file.
 *
 * <p>
 * In the standard {@link java.util.Properties Properties} class trailing
 * whitespace is always preserved. When loading properties from a file such
 * trailing whitespace is almost always <i>unintentional</i>. This class fixes
 * this problem. The trimming of trailing whitespace only takes place if the
 * source of input is a file and only where the input is line oriented (meaning
 * that for example loading from XML file is <i>not</i> changed by this class).
 * For this reason this class is almost in all cases a safe drop-in replacement
 * for the standard <tt>Properties</tt>
 * class.
 *
 * <p>
 * Whitespace is defined here as any of space (U+0020) or tab (U+0009).
 * *
 */
public class TrimmedProperties extends Properties {

    /**
     * Reads a property list (key and element pairs) from the input byte stream.
     *
     * <p>Behaves exactly as {@link java.util.Properties#load(java.io.InputStream) }
     * with the exception that trailing whitespace is trimmed from property values
```

```
     * if <tt>inStream</tt> is an instance of <tt>FileInputStream</tt>.
     *
     * @see java.util.Properties#load(java.io.InputStream)
     * @param inStream the input stream.
     * @throws IOException if an error occurred when reading from the input stream.
     */
    @Override
    public void load(InputStream inStream) throws IOException {
        if (inStream instanceof FileInputStream) {
            // First read into temporary props using the standard way
            Properties tempProps = new Properties();
            tempProps.load(inStream);
            // Now trim and put into target
            trimAndLoad(tempProps);
        } else {
            super.load(inStream);
        }
    }

    /**
     * Reads a property list (key and element pairs) from the input character stream in a
simple line-oriented format.
     *
     * <p>Behaves exactly as {@link java.util.Properties#load(java.io.Reader)}
     * with the exception that trailing whitespace is trimmed on property values
     * if <tt>reader</tt> is an instance of <tt>FileReader</tt>.
     *
     * @see java.util.Properties#load(java.io.Reader) }
     * @param reader the input character stream.
     * @throws IOException if an error occurred when reading from the input stream.
     */
    @Override
    public void load(Reader reader) throws IOException {
        if (reader instanceof FileReader) {
            // First read into temporary props using the standard way
            Properties tempProps = new Properties();
            tempProps.load(reader);
            // Now trim and put into target
            trimAndLoad(tempProps);
        } else {
            super.load(reader);
        }
    }

    private void trimAndLoad(Properties p) {
        for (Entry<Object, Object> entry : p.entrySet()) {
            if (entry.getValue() instanceof String) {
                put(entry.getKey(), trimTrailing((String) entry.getValue()));
            } else {
                put(entry.getKey(), entry.getValue());
            }
        }
    }

    /**
     * Trims trailing space or tabs from a string.
     *
     * @param str
     * @return
     */
    public static String trimTrailing(String str) {
```

```
        if (str != null) {
            // read str from tail until char is no longer whitespace
            for (int i = str.length() - 1; i >= 0; i--) {
                if ((str.charAt(i) != ' ') && (str.charAt(i) != '\t')) {
                    return str.substring(0, i + 1);
                }
            }
        }
        return str;
    }
}
```

Guardar propiedades como XML

Almacenamiento de propiedades en un archivo XML

La forma en que almacena los archivos de propiedades como archivos XML es muy similar a la forma en que los almacenaría como archivos `.properties` . Solo en lugar de usar `store()` `storeToXML()` .

```
public void saveProperties(String location) throws IOException{
    // make new instance of properties
    Properties prop = new Properties();

    // set the property values
    prop.setProperty("name", "Steve");
    prop.setProperty("color", "green");
    prop.setProperty("age", "23");

    // check to see if the file already exists
    File file = new File(location);
    if (!file.exists()){
        file.createNewFile();
    }

    // save the properties
    prop.storeToXML(new FileOutputStream(file), "testing properties with xml");
}
```

Cuando abres el archivo se verá así.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
3  <properties>
4  <comment>testing properties with xml</comment>
5  <entry key="age">23</entry>
6  <entry key="color">green</entry>
7  <entry key="name">Steve</entry>
8  </properties>
9
```

Cargar propiedades desde un archivo XML

Ahora, para cargar este archivo como una `properties` , debe llamar a `loadFromXML()` lugar de a `load()` que usaría con los archivos `.propeties` normales.

```
public static void loadProperties(String location) throws FileNotFoundException, IOException{
    // make new properties instance to load the file into
    Properties prop = new Properties();

    // check to make sure the file exists
    File file = new File(location);
    if (file.exists()){
        // load the file
        prop.loadFromXML(new FileInputStream(file));

        // print out all the properties
        for (String name : prop.stringPropertyNames()){
            System.out.println(name + "=" + prop.getProperty(name));
        }
    } else {
        System.err.println("Error: No file found at: " + location);
    }
}
```

Cuando ejecute este código obtendrá lo siguiente en la consola:

```
age=23
color=green
name=Steve
```

Lea Clase de propiedades en línea: https://riptutorial.com/es/java/topic/576/clase-de-propiedades

Capítulo 31: Clase EnumSet

Introducción

La clase Java EnumSet es la implementación especializada de conjuntos para usar con tipos de enumeración. Hereda la clase AbstractSet e implementa la interfaz Set.

Examples

Ejemplo de conjunto de enumeración

```
import java.util.*;
enum days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
    // Traversing elements
    Iterator<days> iter = set.iterator();
    while (iter.hasNext())
      System.out.println(iter.next());
  }
}
```

Lea Clase EnumSet en línea: https://riptutorial.com/es/java/topic/10159/clase-enumset

Capítulo 32: Clase inmutable

Introducción

Los objetos inmutables son instancias cuyo estado no cambia después de que se haya inicializado. Por ejemplo, String es una clase inmutable y, una vez instanciada, su valor nunca cambia.

Observaciones

Algunas clases inmutables en Java:

1. java.lang.String
2. Las clases de envoltorio para los tipos primitivos: java.lang.Integer, java.lang.Byte, java.lang.Character, java.lang.Short, java.lang.Boolean, java.lang.Long, java.lang.Double, java.lang.Float
3. La mayoría de las clases de enumeración son inmutables, pero esto, de hecho, depende del caso concreto.
4. java.math.BigInteger y java.math.BigDecimal (al menos objetos de esas clases en sí)
5. java.io.File. Tenga en cuenta que esto representa un objeto externo a la VM (un archivo en el sistema local), que puede o no existir, y tiene algunos métodos para modificar y consultar el estado de este objeto externo. Pero el objeto File en sí permanece inmutable.

Examples

Reglas para definir clases inmutables.

Las siguientes reglas definen una estrategia simple para crear objetos inmutables.

1. No proporcione métodos de "establecimiento": métodos que modifican campos u objetos a los que se hace referencia por campos.
2. Hacer todos los campos finales y privados.
3. No permitir que las subclases anulen los métodos. La forma más sencilla de hacer esto es declarar la clase como final. Un enfoque más sofisticado es hacer que el constructor sea privado y construir instancias en métodos de fábrica.
4. Si los campos de la instancia incluyen referencias a objetos mutables, no permita que se cambien esos objetos:
5. No proporcione métodos que modifiquen los objetos mutables.
6. No comparta referencias a los objetos mutables. Nunca almacene referencias a objetos mutables externos pasados al constructor; Si es necesario, cree copias y almacene referencias a las copias. Del mismo modo, cree copias de sus objetos mutables internos cuando sea necesario para evitar devolver los originales en sus métodos.

Ejemplo sin referencias mutables.

```
public final class Color {
    final private int red;
    final private int green;
    final private int blue;

    private void check(int red, int green, int blue) {
        if (red < 0 || red > 255 || green < 0 || green > 255 || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public Color(int red, int green, int blue) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    public Color invert() {
        return new Color(255 - red, 255 - green, 255 - blue);
    }
}
```

Ejemplo con referencias mutables

En este caso, la clase Point es mutable y algunos usuarios pueden modificar el estado del objeto de esta clase.

```
class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }
}

//...

public final class ImmutableCircle {
    private final Point center;
    private final double radius;
```

```
    public ImmutableCircle(Point center, double radius) {
        // we create new object here because it shouldn't be changed
        this.center = new Point(center.getX(), center.getY());
        this.radius = radius;
    }
```

¿Cuál es la ventaja de la inmutabilidad?

La ventaja de la inmutabilidad viene con la concurrencia. Es difícil mantener la corrección en objetos mutables, ya que varios subprocesos podrían estar intentando cambiar el estado del mismo objeto, lo que lleva a que algunos subprocesos vean un estado diferente del mismo objeto, dependiendo de la sincronización de las lecturas y escrituras a dicho objeto.

Al tener un objeto inmutable, uno puede asegurarse de que todos los subprocesos que miran el objeto verán el mismo estado, ya que el estado de un objeto inmutable no cambiará.

Lea Clase inmutable en línea: https://riptutorial.com/es/java/topic/10561/clase-inmutable

Capítulo 33: Clase interna local

Introducción

Una clase, es decir, creada dentro de un método, se llama clase interna local en java. Si desea invocar los métodos de la clase interna local, debe crear una instancia de esta clase dentro del método.

Examples

Clase interna local

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

Lea Clase interna local en línea: https://riptutorial.com/es/java/topic/10160/clase-interna-local

Capítulo 34: Clases anidadas e internas

Introducción

Usando Java, los desarrolladores tienen la capacidad de definir una clase dentro de otra clase. Tal clase se llama una clase anidada . Las clases anidadas se denominan clases internas si se declararon como no estáticas; de lo contrario, simplemente se denominan clases anidadas estáticas. Esta página es para documentar y proporcionar detalles con ejemplos sobre cómo usar clases anidadas e internas de Java.

Sintaxis

- clase pública OuterClass {clase pública InnerClass {}} // las clases internas también pueden ser privadas
- clase pública OuterClass {clase estática pública StaticNestedClass {}} // Las clases anidadas estáticas también pueden ser privadas
- método de anulación público () {clase privada LocalClass {}} // Las clases locales son siempre privadas
- SomeClass anonymousClassInstance = new SomeClass () {}; // Las clases internas anónimas no pueden ser nombradas, por lo tanto el acceso es discutible. Si 'SomeClass ()' es abstracto, el cuerpo debe implementar todos los métodos abstractos.
- SomeInterface anonymousClassInstance = new SomeInterface () {}; // El cuerpo debe implementar todos los métodos de interfaz.

Observaciones

Terminologia y clasificacion

La especificación de lenguaje Java (JLS) clasifica los diferentes tipos de clases de Java de la siguiente manera:

> Una *clase de nivel superior* es una clase que no es una clase anidada.
>
> Una *clase anidada* es cualquier clase cuya declaración ocurre dentro del cuerpo de otra clase o interfaz.
>
> Una *clase interna* es una clase anidada que no se declara explícita o implícitamente como estática.
>
> Una clase interna puede ser una *clase miembro no estática* , una *clase local* o una *clase anónima* . Una clase miembro de una interfaz es implícitamente estática, por lo que nunca se considera una clase interna.

En la práctica, los programadores se refieren a una clase de nivel superior que contiene una clase interna como la "clase externa". Además, hay una tendencia a usar "clase anidada" para referirse

solo a clases anidadas estáticas (explícita o implícitamente).

Tenga en cuenta que existe una relación estrecha entre las clases internas anónimas y las lambdas, pero las lambdas son clases.

Diferencias semanticas

- Las clases de nivel superior son el "caso base". Son visibles a otras partes de un programa sujeto a reglas de visibilidad normales basadas en la semántica del modificador de acceso. Si no son abstractos, se pueden crear instancias de cualquier código en el que los constructores relevantes sean visibles según los modificadores de acceso.

- Las clases anidadas estáticas siguen las mismas reglas de acceso e instanciación que las clases de nivel superior, con dos excepciones:

  - Una clase anidada se puede declarar como `private` , lo que la hace inaccesible fuera de su clase de nivel superior.
  - Una clase anidada tiene acceso a los miembros `private` de la clase adjunta de nivel superior y toda su clase probada.

  Esto hace que las clases anidadas estáticas sean útiles cuando necesita representar múltiples "tipos de entidades" dentro de un límite de abstracción estricto; por ejemplo, cuando las clases anidadas se utilizan para ocultar "detalles de implementación".

- Las clases internas agregan la capacidad de acceder a variables no estáticas declaradas en los ámbitos adjuntos:

  - Una clase miembro no estática puede referirse a variables de instancia.
  - Una clase local (declarada dentro de un método) también puede referirse a las variables locales del método, siempre que sean `final` . (Para Java 8 y versiones posteriores, pueden ser *efectivamente definitivas*).
  - Una clase interna anónima se puede declarar dentro de una clase o un método, y puede acceder a las variables de acuerdo con las mismas reglas.

  El hecho de que una instancia de clase interna pueda referirse a variables en una instancia de clase adjunta tiene implicaciones para la creación de instancias. Específicamente, se debe proporcionar una instancia adjunta, ya sea implícita o explícitamente, cuando se crea una instancia de una clase interna.

Examples

Una pila simple usando una clase anidada

```
public class IntStack {

    private IntStackNode head;

    // IntStackNode is the inner class of the class IntStack
```

```
    // Each instance of this inner class functions as one link in the
    // Overall stack that it helps to represent
    private static class IntStackNode {

        private int val;
        private IntStackNode next;

        private IntStackNode(int v, IntStackNode n) {
            val = v;
            next = n;
        }
    }

    public IntStack push(int v) {
        head = new IntStackNode(v, head);
        return this;
    }

    public int pop() {
        int x = head.val;
        head = head.next;
        return x;
    }
}
```

Y su uso, que (en particular) no reconoce en absoluto la existencia de la clase anidada.

```
public class Main {
    public static void main(String[] args) {

        IntStack s = new IntStack();
        s.push(4).push(3).push(2).push(1).push(0);

        //prints: 0, 1, 2, 3, 4,
        for(int i = 0; i < 5; i++) {
            System.out.print(s.pop() + ", ");
        }
    }
}
```

Clases anidadas estáticas y no estáticas

Al crear una clase anidada, tiene la opción de tener esa clase estática anidada:

```
public class OuterClass1 {

    private static class StaticNestedClass {

    }

}
```

O no estático:

```
public class OuterClass2 {
```

```
    private class NestedClass {

    }

}
```

En su núcleo, las clases anidadas estáticas *no tienen una* ***instancia*** *circundante* de la clase externa, mientras que las clases anidadas no estáticas sí las tienen. Esto afecta a dónde y cuándo se permite a una instancia crear una clase anidada, y a qué instancias de esas clases anidadas se les permite acceder. Añadiendo al ejemplo anterior:

```
public class OuterClass1 {

    private int aField;
    public void aMethod(){}

    private static class StaticNestedClass {
        private int innerField;

        private StaticNestedClass() {
             innerField = aField; //Illegal, can't access aField from static context
             aMethod();           //Illegal, can't call aMethod from static context
        }

        private StaticNestedClass(OuterClass1 instance) {
             innerField = instance.aField; //Legal
        }

    }

    public static void aStaticMethod() {
        StaticNestedClass s = new StaticNestedClass(); //Legal, able to construct in static
context
        //Do stuff involving s...
    }

}

public class OuterClass2 {

    private int aField;

    public void aMethod() {}

    private class NestedClass {
        private int innerField;

        private NestedClass() {
             innerField = aField; //Legal
             aMethod(); //Legal
        }
    }

    public void aNonStaticMethod() {
        NestedClass s = new NestedClass(); //Legal
    }

    public static void aStaticMethod() {
        NestedClass s = new NestedClass(); //Illegal. Can't construct without surrounding
```

```
OuterClass2 instance.
                                        //As this is a static context, there is no
surrounding OuterClass2 instance
    }
}
```

Por lo tanto, su decisión de estática frente a no estática depende principalmente de si necesita poder acceder directamente a los campos y métodos de la clase externa, aunque también tiene consecuencias sobre cuándo y dónde puede construir la clase anidada.

Como regla general, haga que sus clases anidadas sean estáticas a menos que necesite acceder a los campos y métodos de la clase externa. Al igual que hacer que sus campos sean privados, a menos que los necesite, esto disminuye la visibilidad disponible para la clase anidada (al no permitir el acceso a una instancia externa), lo que reduce la posibilidad de error.

Modificadores de Acceso para Clases Internas

Una explicación completa de los modificadores de acceso en Java se puede encontrar aquí . ¿Pero cómo interactúan con las clases internas?

`public` , como de costumbre, da acceso sin restricciones a cualquier ámbito capaz de acceder al tipo.

```
public class OuterClass {

    public class InnerClass {

        public int x = 5;

    }

    public InnerClass createInner() {
        return new InnerClass();
    }
}

public class SomeOtherClass {

    public static void main(String[] args) {
        int x = new OuterClass().createInner().x; //Direct field access is legal
    }
}
```

tanto `protected` como el modificador predeterminado (de nada) se comportan como se espera también, de la misma manera que lo hacen para las clases no anidadas.

`private` , curiosamente, no se limita a la clase a la que pertenece. Más bien, se restringe a la unidad de compilación: el archivo .java. Esto significa que las clases externas tienen acceso total a los campos y métodos de la clase interna, incluso si están marcados como `private` .

```
public class OuterClass {

    public class InnerClass {
```

```
        private int x;
        private void anInnerMethod() {}
    }

    public InnerClass aMethod() {
        InnerClass a = new InnerClass();
        a.x = 5; //Legal
        a.anInnerMethod(); //Legal
        return a;
    }
}
```

La Clase Interna en sí misma puede tener una visibilidad que no sea `public` . Al marcarlo como `private` u otro modificador de acceso restringido, otras clases (externas) no podrán importar y asignar el tipo. Sin embargo, aún pueden obtener referencias a objetos de ese tipo.

```
public class OuterClass {

    private class InnerClass{}

    public InnerClass makeInnerClass() {
        return new InnerClass();
    }
}

public class AnotherClass {

    public static void main(String[] args) {
        OuterClass o = new OuterClass();

        InnerClass x = o.makeInnerClass(); //Illegal, can't find type
        OuterClass.InnerClass x = o.makeInnerClass(); //Illegal, InnerClass has visibility
private
        Object x = o.makeInnerClass(); //Legal
    }
}
```

Clases internas anónimas

Una clase interna anónima es una forma de clase interna que se declara y crea una instancia con una sola declaración. Como consecuencia, no hay un nombre para la clase que pueda usarse en otra parte del programa; Es decir, es anónimo.

Las clases anónimas se utilizan normalmente en situaciones en las que es necesario poder crear una clase de peso ligero que se pase como parámetro. Esto normalmente se hace con una interfaz. Por ejemplo:

```
public static Comparator<String> CASE_INSENSITIVE =
        new Comparator<String>() {
            @Override
            public int compare(String string1, String string2) {
                return string1.toUpperCase().compareTo(string2.toUpperCase());
            }
        };
```

Esta clase anónima define un objeto `Comparator<String>` (`CASE_INSENSITIVE`) que compara dos cadenas ignorando las diferencias en el caso.

Otras interfaces que se implementan e `Runnable` frecuencia usando clases anónimas son `Runnable` y `Callable` . Por ejemplo:

```
// An anonymous Runnable class is used to provide an instance that the Thread
// will run when started.
Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
              System.out.println("Hello world");
        }
    });
t.start();  // Prints "Hello world"
```

Las clases internas anónimas también pueden basarse en clases. En este caso, la clase anónima `extends` implícitamente la clase existente. Si la clase que se está extendiendo es abstracta, entonces la clase anónima debe implementar todos los métodos abstractos. También puede anular métodos no abstractos.

Constructores

Una clase anónima no puede tener un constructor explícito. En su lugar, se define un constructor implícito que usa `super(...)` para pasar cualquier parámetro a un constructor en la clase que se está extendiendo. Por ejemplo:

```
SomeClass anon = new SomeClass(1, "happiness") {
            @Override
            public int someMethod(int arg) {
                // do something
            }
        };
```

El constructor implícito para nuestra subclase anónima de `SomeClass` llamará a un constructor de `SomeClass` que coincida con la firma de llamada `SomeClass(int, String)` . Si no hay ningún constructor disponible, obtendrá un error de compilación. Cualquier excepción lanzada por el constructor emparejado también es lanzada por el constructor implícito.

Naturalmente, esto no funciona cuando se extiende una interfaz. Cuando creas una clase anónima desde una interfaz, la clase superclase es `java.lang.Object` que solo tiene un constructor sin argumentos.

Método de clases internas locales

Una clase escrita dentro de un método llamado **método clase interna local** . En ese caso, el alcance de la clase interna está restringido dentro del método.

Una clase interna de método local puede instanciarse solo dentro del método donde se define la clase interna.

El ejemplo de usar el método local de clase interna:

```
public class OuterClass {
    private void outerMethod() {
       final int outerInt = 1;
        // Method Local Inner Class
        class MethodLocalInnerClass {
            private void print() {
                System.out.println("Method local inner class " + outerInt);
            }
        }
        // Accessing the inner class
        MethodLocalInnerClass inner = new MethodLocalInnerClass();
        inner.print();
    }

    public static void main(String args[]) {
        OuterClass outer = new OuterClass();
        outer.outerMethod();
    }
}
```

La ejecución dará una salida: `Method local inner class 1` .

Acceso a la clase externa desde una clase interna no estática

La referencia a la clase externa usa el nombre de la clase y `this`

```
public class OuterClass {
    public class InnerClass {
        public void method() {
            System.out.println("I can access my enclosing class: " + OuterClass.this);
        }
    }
}
```

Puede acceder a los campos y métodos de la clase externa directamente.

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        public void method() {
            System.out.println("I can access " + counter);
        }
    }
}
```

Pero en caso de colisión de nombres, puede utilizar la referencia de clase externa.

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        private int counter;
```

```
        public void method() {
            System.out.println("My counter: " + counter);
            System.out.println("Outer counter: " + OuterClass.this.counter);

            // updating my counter
            counter = OuterClass.this.counter;
        }
    }
}
```

Crear instancia de clase interna no estática desde el exterior.

También se puede crear una clase interna que sea visible para cualquier clase externa a partir de esta clase.

La clase interna depende de la clase externa y requiere una referencia a una instancia de ella. Para crear una instancia de la clase interna, el `new` operador solo necesita ser llamado en una instancia de la clase externa.

```
class OuterClass {

    class InnerClass {
    }
}

class OutsideClass {

    OuterClass outer = new OuterClass();

    OuterClass.InnerClass createInner() {
        return outer.new InnerClass();
    }
}
```

Tenga en cuenta el uso como `outer.new` .

Lea Clases anidadas e internas en línea: https://riptutorial.com/es/java/topic/3317/clases-anidadas-e-internas

Capítulo 35: Clases y objetos

Introducción

Los objetos tienen estados y comportamientos. Ejemplo: Un perro tiene estados (color, nombre, raza y comportamientos): menear la cola, ladrar, comer. Un objeto es una instancia de una clase.

Clase: una clase se puede definir como una plantilla / modelo que describe el comportamiento / estado que admite el objeto de su tipo.

Sintaxis

- clase Ejemplo {} // clase palabra clave, nombre, cuerpo

Examples

Clase posible más simple

```
class TrivialClass {}
```

Una clase consta de un mínimo de la palabra clave de `class` , un nombre y un cuerpo, que pueden estar vacíos.

Se crea una instancia de una clase con el `new` operador.

```
TrivialClass tc = new TrivialClass();
```

Miembro Objeto vs Miembro Estático

Con esta clase:

```
class ObjectMemberVsStaticMember {

    static int staticCounter = 0;
    int memberCounter = 0;

    void increment() {
        staticCounter ++;
        memberCounter++;
    }
}
```

el siguiente fragmento de código:

```
final ObjectMemberVsStaticMember o1 = new ObjectMemberVsStaticMember();
final ObjectMemberVsStaticMember o2 = new ObjectMemberVsStaticMember();
```

```
o1.increment();

o2.increment();
o2.increment();

System.out.println("o1 static counter " + o1.staticCounter);
System.out.println("o1 member counter " + o1.memberCounter);
System.out.println();

System.out.println("o2 static counter " + o2.staticCounter);
System.out.println("o2 member counter " + o2.memberCounter);
System.out.println();

System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
ObjectMemberVsStaticMember.staticCounter);

// the following line does not compile. You need an object
// to access its members
//System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
ObjectMemberVsStaticMember.memberCounter);
```

produce esta salida:

```
o1 static counter 3
o1 member counter 1

o2 static counter 3
o2 member counter 2

ObjectMemberVsStaticMember.staticCounter = 3
```

Nota: No debe llamar a miembros `static` en objetos, sino en clases. Si bien no hace una diferencia para la JVM, los lectores humanos lo apreciarán.

`static` miembros `static` son parte de la clase y existen solo una vez por clase. Los miembros no `static` existen en las instancias, hay una copia independiente para cada instancia. Esto también significa que necesita acceder a un objeto de esa clase para acceder a sus miembros.

Métodos de sobrecarga

A veces, se debe escribir la misma funcionalidad para diferentes tipos de entradas. En ese momento, uno puede usar el mismo nombre de método con un conjunto diferente de parámetros. Cada conjunto diferente de parámetros se conoce como una firma de método. Como se ve en el ejemplo, un solo método puede tener varias firmas.

```
public class Displayer {

    public void displayName(String firstName) {
        System.out.println("Name is: " + firstName);
    }

    public void displayName(String firstName, String lastName) {
        System.out.println("Name is: " + firstName + " " + lastName);
    }
```

```
    public static void main(String[] args) {
        Displayer displayer = new Displayer();
        displayer.displayName("Ram");          //prints "Name is: Ram"
        displayer.displayName("Jon", "Skeet"); //prints "Name is: Jon Skeet"
    }
}
```

La ventaja es que se llama a la misma funcionalidad con dos números diferentes de entradas. Al invocar el método de acuerdo con la entrada que estamos pasando, (en este caso, un valor de cadena o dos valores de cadena) se ejecuta el método correspondiente.

Los métodos pueden ser sobrecargados:

1. Basado en el **número de parámetros** pasados.

   Ejemplo: `method(String s)` y `method(String s1, String s2)` .

2. Basado en el **orden de los parámetros** .

   Ejemplo: `method(int i, float f)` y `method(float f, int i))` .

***Nota: los** métodos no pueden sobrecargarse cambiando solo el tipo de retorno (`int method()` se considera el mismo que `String method()` y lanzará una `RuntimeException` si se intenta). Si cambia el tipo de retorno, también debe cambiar los parámetros para sobrecargar.*

Construcción y uso de objetos básicos

Los objetos vienen en su propia clase, por lo que un ejemplo simple sería un automóvil (explicaciones detalladas a continuación):

```
public class Car {

    //Variables describing the characteristics of an individual car, varies per  object
   private int milesPerGallon;
   private String name;
   private String color;
   public int numGallonsInTank;

    public Car(){
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }

    //this is where an individual object is created
    public Car(int mpg, int, gallonsInTank, String carName, String carColor){
        milesPerGallon = mpg;
        name = carName;
        color = carColor;
        numGallonsInTank = gallonsInTank;
    }

    //methods to make the object more usable
```

```
    //Cars need to drive
    public void drive(int distanceInMiles){
        //get miles left in car
        int miles = numGallonsInTank * milesPerGallon;

        //check that car has enough gas to drive distanceInMiles
        if (miles <= distanceInMiles){
            numGallonsInTank = numGallonsInTank - (distanceInMiles / milesPerGallon)
            System.out.println("Drove " + numGallonsInTank + " miles!");
        } else {
            System.out.println("Could not drive!");
        }
    }

    public void paintCar(String newColor){
        color = newColor;
    }
        //set new Miles Per Gallon
    public void setMPG(int newMPG){
        milesPerGallon = newMPG;
    }

       //set new number of Gallon In Tank
    public void setGallonsInTank(int numGallons){
        numGallonsInTank = numGallons;
    }

    public void nameCar(String newName){
        name = newName;
    }

    //Get the Car color
    public String getColor(){
        return color;
    }

    //Get the Car name
    public String getName(){
        return name;
    }

    //Get the number of Gallons
    public String getGallons(){
        return numGallonsInTank;
    }

}
```

Los objetos son **instancias de** su clase. Entonces, la forma en que **crearía un objeto** sería llamando a la clase Car de **una de las dos formas** en su clase principal (método principal en Java u onCreate en Android).

Opción 1

```
`Car newCar = new Car(30, 10, "Ferrari", "Red");
```

La opción 1 es donde esencialmente le dice al programa todo sobre el Coche al crear el objeto.

Cambiar cualquier propiedad del automóvil requeriría llamar a uno de los métodos, como el método `repaintCar` . Ejemplo:

```
 newCar.repaintCar("Blue");
```

Nota: asegúrese de pasar el tipo de datos correcto al método. En el ejemplo anterior, también puede pasar una variable al método `repaintCar` **siempre que el tipo de datos sea correcto** .

Ese fue un ejemplo de cambio de propiedades de un objeto, recibir propiedades de un objeto requeriría el uso de un método de la clase Car que tenga un valor de retorno (es decir, un método que no sea `void`). Ejemplo:

```
String myCarName = newCar.getName();  //returns string "Ferrari"
```

La opción 1 es la **mejor** opción cuando tiene **todos los datos del objeto** en el momento de la creación.

opcion 2

```
`Car newCar = new Car();
```

La opción 2 obtiene el mismo efecto pero requiere más trabajo para crear un objeto correctamente. Quiero recordar a este Constructor en la clase de Automóviles:

```
public void Car(){
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }
```

Tenga en cuenta que no tiene que pasar ningún parámetro al objeto para crearlo. Esto es muy útil para cuando no tiene todos los aspectos del objeto pero necesita usar las partes que tiene. Esto establece datos genéricos en cada una de las variables de instancia del objeto, de modo que, si solicita un dato que no existe, no se generan errores.

Nota: No olvide que debe configurar las partes del objeto más adelante con las que no lo haya inicializado. Por ejemplo,

```
Car myCar = new Car();
String color = Car.getColor(); //returns empty string
```

Este es un error común entre los objetos que no están inicializados con todos sus datos. Se evitaron los errores porque hay un Constructor que permite que se cree un objeto de Coche vacío con **variables de apoyo** (`public Car(){}`), pero ninguna parte de myCar fue realmente personalizada. **Ejemplo correcto de crear un objeto de coche:**

```
Car myCar = new Car();
myCar.nameCar("Ferrari");
```

```
myCar.paintCar("Purple");
myCar.setGallonsInTank(10);
myCar.setMPG(30);
```

Y, como recordatorio, obtenga las propiedades de un objeto llamando a un método en su clase principal. Ejemplo:

```
String myCarName = myCar.getName(); //returns string "Ferrari"
```

Constructores

Los constructores son métodos especiales nombrados después de la clase y sin un tipo de retorno, y se utilizan para construir objetos. Los constructores, como los métodos, pueden tomar parámetros de entrada. Los constructores se utilizan para inicializar objetos. Las clases abstractas también pueden tener constructores.

```
public class Hello{
    // constructor
    public Hello(String wordToPrint){
        printHello(wordToPrint);
    }
    public void printHello(String word){
        System.out.println(word);
    }
}
// instantiates the object during creating and prints out the content
// of wordToPrint
```

Es importante entender que los constructores son diferentes de los métodos de varias maneras:

1. Los constructores solo pueden tomar los modificadores `public` , `private` y `protected` , y no pueden ser declarados `abstract` , `final` , `static` o `synchronized` .
2. Los constructores no tienen un tipo de retorno.
3. Los constructores DEBEN tener el mismo nombre que el nombre de la clase. En el ejemplo de `Hello` , el nombre del constructor del objeto `Hello` es el mismo que el nombre de la clase.
4. La palabra clave `this` tiene un uso adicional dentro de los constructores. `this.method(...)` llama a un método en la instancia actual, mientras que `this(...)` refiere a otro constructor en la clase actual con diferentes firmas.

A los constructores también se les puede llamar por herencia usando la palabra clave `super` .

```
public class SuperManClass{

    public SuperManClass(){
        // some implementation
    }

    // ... methods
}
```

```
public class BatmanClass extends SupermanClass{
    public BatmanClass(){
        super();
    }
    //... methods...
}
```

Ver la especificación del lenguaje Java # 8.8 y # 15.9

Inicializando campos finales estáticos usando un inicializador estático

Para inicializar campos `static final` que requieren el uso de más de una expresión, se puede usar un inicializador `static` para asignar el valor. El siguiente ejemplo inicializa un conjunto no modificable de `String` s:

```
public class MyClass {

    public static final Set<String> WORDS;

    static {
        Set<String> set = new HashSet<>();
        set.add("Hello");
        set.add("World");
        set.add("foo");
        set.add("bar");
        set.add("42");
        WORDS = Collections.unmodifiableSet(set);
    }
}
```

Explicando qué es el método de sobrecarga y anulación.

El método de anulación y sobrecarga son dos formas de polimorfismo soportado por Java.

Método de sobrecarga

La sobrecarga de métodos (también conocida como Polimorfismo estático) es una forma en que puede tener dos (o más) métodos (funciones) con el mismo nombre en una sola clase. Sí, es tan simple como eso.

```
public class Shape{
    //It could be a circle or rectangle or square
    private String type;

    //To calculate area of rectangle
    public Double area(Long length, Long breadth){
        return (Double) length * breadth;
    }

     //To calculate area of a circle
     public Double area(Long radius){
        return (Double) 3.14 * r * r;
```

```
    }
}
```

De esta manera, el usuario puede llamar al mismo método para el área dependiendo del tipo de forma que tenga.

Pero la pregunta real ahora es, ¿cómo distinguirá el compilador Java qué cuerpo del método se ejecutará?

Bueno, Java ha dejado claro que aunque los **nombres de** los **métodos** (`area()` en nuestro caso) **pueden ser iguales, el método de los argumentos debe ser diferente.**

> Los métodos sobrecargados deben tener diferentes listas de argumentos (cantidad y tipos).

Dicho esto, no podemos agregar otro método para calcular el área de un cuadrado como este: `public Double area(Long side)` porque, en este caso, entrará en conflicto con el método del círculo del área y causará **ambigüedad** para el compilador Java.

Gracias a Dios, hay algunas relajaciones al escribir métodos sobrecargados como

> Puede tener diferentes tipos de retorno.
>
> Puede tener diferentes modificadores de acceso.
>
> Puede lanzar diferentes excepciones.

¿Por qué se llama esto polimorfismo estático?

Bueno, eso se debe a que los métodos sobrecargados que se van a invocar se deciden en el momento de la compilación, según el número real de argumentos y los tipos de tiempo de compilación de los argumentos.

> Una de las razones más comunes para usar la sobrecarga de métodos es la simplicidad del código que proporciona. Por ejemplo, recuerde `String.valueOf()` que toma casi cualquier tipo de argumento? Lo que está escrito detrás de la escena es probablemente algo como esto:

```
static String valueOf(boolean b)
static String valueOf(char c)
static String valueOf(char[] data)
static String valueOf(char[] data, int offset, int count)
static String valueOf(double d)
static String valueOf(float f)
static String valueOf(int i)
static String valueOf(long l)
static String valueOf(Object obj)
```

Método Anulando

Bueno, el método de anulación (sí, supongo que es correcto, también se conoce como

polimorfismo dinámico) es un tema algo más interesante y complejo.

En la sustitución de métodos, sobrescribimos el cuerpo del método proporcionado por la clase padre. ¿Lo tengo? ¿No? Veamos un ejemplo.

```
public abstract class Shape{

    public abstract Double area(){
        return 0.0;
    }
}
```

Así que tenemos una clase llamada Forma y tiene un método llamado área que probablemente devolverá el área de la forma.

Digamos que ahora tenemos dos clases llamadas Círculo y Rectángulo.

```
public class Circle extends Shape {
    private Double radius = 5.0;

    // See this annotation @Override, it is telling that this method is from parent
    // class Shape and is overridden here
    @Override
    public Double area(){
        return 3.14 * radius * radius;
    }
}
```

De forma similar clase de rectángulo:

```
 public class Rectangle extends Shape {
    private Double length = 5.0;
    private Double breadth= 10.0;


    // See this annotation @Override, it is telling that this method is from parent
    // class Shape and is overridden here
    @Override
    public Double area(){
        return length * breadth;
    }
}
```

Entonces, ahora las dos clases de sus hijos tienen el cuerpo del método actualizado proporcionado por la clase principal (`Shape`). Ahora la pregunta es ¿cómo ver el resultado? Bueno, `psvm` la vieja `psvm` .

```
public class AreaFinder{

    public static void main(String[] args){

        //This will create an object of circle class
        Shape circle = new Circle();
        //This will create an object of Rectangle class
        Shape rectangle = new Rectangle();
```

```
        // Drumbeats ......
        //This should print 78.5
        System.out.println("Shape of circle : "+circle.area());

        //This should print 50.0
        System.out.println("Shape of rectangle: "+rectangle.area());

    }
}
```

¡Guauu! no es genial? Dos objetos del mismo tipo que llaman a los mismos métodos y devuelven valores diferentes. Mi amigo, ese es el poder del polimorfismo dinámico.

Aquí hay una tabla para comparar mejor las diferencias entre estos dos:

Método de sobrecarga	Método Anulando
El método de sobrecarga se utiliza para aumentar la legibilidad del programa.	La anulación del método se utiliza para proporcionar la implementación específica del método que ya está provisto por su superclase.
La sobrecarga de métodos se realiza dentro de la clase.	La anulación del método se produce en dos clases que tienen una relación IS-A (herencia).
En caso de sobrecarga del método, el parámetro debe ser diferente.	En caso de anulación del método, el parámetro debe ser el mismo.
La sobrecarga de métodos es el ejemplo del polimorfismo de tiempo de compilación.	La anulación del método es el ejemplo del polimorfismo de tiempo de ejecución.
En java, la sobrecarga de métodos no se puede realizar cambiando solo el tipo de retorno del método. El tipo de retorno puede ser igual o diferente en la sobrecarga de métodos. Pero debes tener que cambiar el parámetro.	El tipo de devolución debe ser igual o covariante en la invalidación del método.

Lea Clases y objetos en línea: https://riptutorial.com/es/java/topic/114/clases-y-objetos

Capítulo 36: Clonación de objetos

Observaciones

La clonación puede ser complicada, especialmente cuando los campos del objeto contienen otros objetos. Hay situaciones en las que desea realizar una copia profunda , en lugar de copiar solo los valores de campo (es decir, referencias a los otros objetos).

La conclusión es que la copia está rota , y debe pensarlo dos veces antes de implementar la interfaz `Cloneable` y anular el método de `clone` . El método de `clone` se declara en la clase `Object` y no en la interfaz `Cloneable` , por lo que `Cloneable` no funciona como interfaz porque carece de un método de `clone` público. El resultado es que el contrato para usar el `clone` está escasamente documentado y se aplica de manera débil. Por ejemplo, una clase que anula la `clone` veces depende de todas sus clases primarias y también anula la `clone` . No están obligados a hacerlo, y si no lo hacen, su código puede generar excepciones.

Una solución mucho mejor para proporcionar funcionalidad de clonación es proporcionar un *constructor de copias* o una *fábrica de copias* . Consulte el ítem 11 de Java efectivo de Joshua Bloch : Sobrescriba el clon de forma juiciosa.

Examples

Clonación utilizando un constructor de copia.

Una forma fácil de clonar un objeto es implementando un constructor de copia.

```
public class Sheep {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    // copy constructor
    // copies the fields of other into the new object
    public Sheep(Sheep other) {
        this.name = other.name;
        this.weight = other.weight;
    }

}

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);
// clone the sheep
Sheep dolly = new Sheep(sheep); // dolly.name is "Dolly" and dolly.weight is 20
```

Clonación implementando la interfaz clonable

Clonando un objeto implementando la interfaz Cloneable .

```
public class Sheep implements Cloneable {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

}

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);
// clone the sheep
Sheep dolly =  (Sheep) sheep.clone(); // dolly.name is "Dolly" and dolly.weight is 20
```

Clonación realizando una copia superficial.

El comportamiento predeterminado al clonar un objeto es realizar una copia superficial de los campos del objeto. En ese caso, tanto el objeto original como el objeto clonado mantienen referencias a los mismos objetos.

Este ejemplo muestra ese comportamiento.

```
import java.util.List;

public class Sheep implements Cloneable {

    private String name;

    private int weight;

    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public List<Sheep> getChildren() {
        return children;
```

```
    }

    public void setChildren(List<Sheep> children) {
        this.children = children;
    }

}

import java.util.Arrays;
import java.util.List;

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);

// create children
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// clone the sheep
Sheep dolly =  (Sheep) sheep.clone();
List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // prints true, both arrays contain the same objects
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}
```

Clonación realizando una copia profunda.

Para copiar objetos anidados, se debe realizar una copia profunda , como se muestra en este ejemplo.

```
import java.util.ArrayList;
import java.util.List;

public class Sheep implements Cloneable {

    private String name;

    private int weight;

    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        Sheep clone = (Sheep) super.clone();
        if (children != null) {
            // make a deep copy of the children
            List<Sheep> cloneChildren = new ArrayList<>(children.size());
            for (Sheep child : children) {
                cloneChildren.add((Sheep) child.clone());
            }
```

```
            clone.setChildren(cloneChildren);
        }
        return clone;
    }

    public List<Sheep> getChildren() {
        return children;
    }

    public void setChildren(List<Sheep> children) {
        this.children = children;
    }

}

import java.util.Arrays;
import java.util.List;

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);

// create children
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// clone the sheep
Sheep dolly =  (Sheep) sheep.clone();
List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // prints false, both arrays contain copies of the objects inside
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}
```

Clonación utilizando una fábrica de copias.

```
public class Sheep {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public static Sheep newInstance(Sheep other);
        return new Sheep(other.name, other.weight)
    }

}
```

Lea Clonación de objetos en línea: https://riptutorial.com/es/java/topic/2830/clonacion-de-objetos

Capítulo 37: Codificación de caracteres

Examples

Leyendo texto de un archivo codificado en UTF-8

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ReadingUTF8TextFile {

    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {
            wr.write("Strange cyrillic symbol Ы");
        }
        /* First Way. For big files */
        try (BufferedReader reader = Files.newBufferedReader(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {

            String line;
            while ((line = reader.readLine()) != null) {
                System.out.print(line);
            }
        }

        System.out.println(); //just separating output

        /* Second way. For small files */
        String s = new String(Files.readAllBytes(Paths.get("test.txt")),
StandardCharsets.UTF_8);
        System.out.print(s);
    }
}
```

Escribiendo texto a un archivo en UTF-8

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class WritingUTF8TextFile {
    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test2.txt"),
StandardCharsets.UTF_8)) {
```

```
            wr.write("Cyrillic symbol Ы");
        }
    }
}
```

Obtención de la representación en byte de una cadena en UTF-8

```
import java.nio.charset.StandardCharsets;
import java.util.Arrays;

public class GetUtf8BytesFromString {

    public static void main(String[] args) {
        String str = "Cyrillic symbol Ы";
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        byte[] textInUtf8 = str.getBytes(StandardCharsets.UTF_8);

        System.out.println(Arrays.toString(textInUtf8));
    }
}
```

Lea Codificación de caracteres en línea: https://riptutorial.com/es/java/topic/2735/codificacion-de-caracteres

Capítulo 38: Código oficial de Oracle estándar

Introducción

La guía de estilo oficial de Oracle para el lenguaje de programación Java es un estándar seguido por los desarrolladores de Oracle y se recomienda que sea seguido por cualquier otro desarrollador de Java. Cubre nombres de archivos, organización de archivos, sangría, comentarios, declaraciones, declaraciones, espacios en blanco, convenciones de nomenclatura, prácticas de programación e incluye un ejemplo de código.

Observaciones

- Los ejemplos anteriores siguen estrictamente la nueva guía de estilo oficial de Oracle. En otras palabras, *no están* constituidos subjetivamente por los autores de esta página.
- La guía de estilo oficial se ha escrito cuidadosamente para que sea compatible con la guía de estilo original y con la mayoría de los códigos en libertad.
- La guía oficial de estilo ha sido pares revisado por, entre otros, Brian Goetz (lenguaje Java Architect) y Mark Reinhold (Arquitecto Jefe de la Plataforma Java).
- Los ejemplos no son normativos; Si bien pretenden ilustrar la forma correcta de formatear el código, puede haber otras formas de formatear el código correctamente. Este es un principio general: puede haber varias formas de formatear el código, todas de acuerdo con las pautas oficiales.

Examples

Convenciones de nombres

Nombres de paquetes

- Los nombres de los paquetes deben estar en minúsculas sin guiones bajos u otros caracteres especiales.
- Los nombres de los paquetes comienzan con la parte de autoridad anulada de la dirección web de la empresa del desarrollador. Esta parte puede ser seguida por una subestructura de paquete dependiente de la estructura del proyecto / programa.
- No utilice la forma plural. Siga la convención de la API estándar que utiliza, por ejemplo, `java.lang.annotation` y no `java.lang.annotations` .
- **Ejemplos:** `com.yourcompany.widget.button` , `com.yourcompany.core.api`

Clase, Interfaz y Nombres Enum

- Los nombres de clase y enumeración deben ser nombres.
- Los nombres de las interfaces normalmente deben ser sustantivos o adjetivos que terminen con ... capaces.
- Use mayúsculas y minúsculas con la primera letra de cada palabra en mayúsculas (es decir, CamelCase).
- Empareja la expresión regular `^[AZ][a-zA-Z0-9]*$` .
- Use palabras completas y evite usar abreviaturas a menos que la abreviatura se use más ampliamente que la forma larga.
- Formatee una abreviatura como una palabra si es parte de un nombre de clase más largo.
- **Ejemplos:** `ArrayList` , `BigInteger` , `ArrayIndexOutOfBoundsException` , `Iterable` .

Nombres de los métodos

Los nombres de los métodos suelen ser verbos u otras descripciones de acciones

- Deben coincidir con la expresión regular `^[az][a-zA-Z0-9]*$` .
- Use mayúsculas y minúsculas con la primera letra en minúsculas.
- **Ejemplos:** `toString` , `hashCode`

Variables

Los nombres de las variables deben estar en mayúsculas y minúsculas con la primera letra en minúscula

- Empareja la expresión regular `^[az][a-zA-Z0-9]*$`
- Recomendación adicional: Variables
- **Ejemplos:** `elements` , `currentIndex`

Variables de tipo

Para casos simples donde hay pocas variables de tipo involucradas, use una sola letra mayúscula.

- Coincidir con la expresión regular `^[AZ][0-9]?$`
- Si una letra es más descriptiva que otra (como `K` y `V` para claves y valores en mapas o `R` para un tipo de retorno de función) use eso, de lo contrario use `T`
- Para los casos complejos en los que las variables de una sola letra se vuelven confusas, use nombres más largos escritos con mayúsculas y use el guión bajo (_) para separar las palabras.
- **Ejemplos:** `T` , `V` , `SRC_VERTEX`

Constantes

Las constantes (campos `static final` cuyo contenido es inmutable, por reglas de idioma o por convención) deben nombrarse con todas las letras mayúsculas y el subrayado (_) para separar las palabras.

- Coincidir con la expresión regular `^[AZ][A-Z0-9]*(_[A-Z0-9]+)*$`
- **Ejemplos:** `BUFFER_SIZE` , `MAX_LEVEL`

Otras directrices sobre nombramiento

- Evite ocultar / sombrear métodos, variables y variables de tipo en ámbitos externos.
- Deje que la verbosidad del nombre se correlacione con el tamaño del alcance. (Por ejemplo, use nombres descriptivos para campos de clases grandes y nombres breves para variables locales de corta duración).
- Al nombrar miembros públicos estáticos, deje que el identificador sea auto-descriptivo si cree que serán importados estáticamente.
- Más información: Sección de nombres (en la guía de estilo de Java oficial)

Fuente: Directrices de estilo de Java de Oracle

Archivos fuente de Java

- Todas las líneas deben terminarse con un carácter de avance de línea (LF, valor ASCII 10) y no, por ejemplo, CR o CR + LF.

- Puede que no haya ningún espacio en blanco al final de una línea.

- El nombre de un archivo de origen debe ser igual al nombre de la clase que contiene, seguido de la extensión `.java` , incluso para los archivos que solo contienen una clase privada de paquete. Esto no se aplica a los archivos que no contienen ninguna declaración de clase, como `package-info.java` .

Caracteres especiales

- Aparte de LF, el único carácter de espacio en blanco permitido es el espacio (valor ASCII 32). Tenga en cuenta que esto implica que otros caracteres de espacios en blanco (en, por ejemplo, literales de cadenas y caracteres) deben escribirse en forma de escape.

- `\'` , `\"` , `\\` , `\t` , `\b` , `\r` , `\f` , y `\n` deben preferirse a los octales correspondientes (por ejemplo, `\047`) o Unicode (por ejemplo, `\u0027`) caracteres escapados.

- En caso de que sea necesario ir en contra de las reglas anteriores por el bien de la prueba, la prueba debe *generar* la entrada requerida programáticamente.

Declaración del paquete

```
package com.example.my.package;
```

La declaración del paquete no debe incluirse en línea, independientemente de si excede la longitud máxima recomendada de una línea.

Declaraciones de importación

```
// First java/javax packages
import java.util.ArrayList;
import javax.tools.JavaCompiler;

// Then third party libraries
import com.fasterxml.jackson.annotation.JsonProperty;

// Then project imports
import com.example.my.package.ClassA;
import com.example.my.package.ClassB;

// Then static imports (in the same order as above)
import static java.util.stream.Collectors.toList;
```

- Las declaraciones de importación deben ser ordenadas ...
    - ... principalmente por importaciones no estáticas / estáticas con importaciones no estáticas primero.
    - ... secundariamente por origen de paquete según el siguiente orden
        - paquetes de java
        - paquetes javax
        - paquetes externos (ej. org.xml)
        - paquetes internos (ej. com.sun)
    - … Terciario por paquete e identificador de clase en orden lexicográfico
- Las declaraciones de importación no deben incluirse en línea, independientemente de si exceden la longitud máxima recomendada de una línea.
- No deben estar presentes las importaciones no utilizadas.

Importaciones de comodines

- En general, las importaciones de comodines no deben utilizarse.
- Cuando se importa una gran cantidad de clases estrechamente relacionadas (como la implementación de un visitante en un árbol con docenas de clases distintas de "nodos"), se puede usar una importación de comodines.
- En cualquier caso, no se debe utilizar más de una importación de comodines por archivo.

Estructura de clase

Orden de los miembros de la clase

Los miembros de la clase deben ordenarse de la siguiente manera:

1. Campos (en orden público, protegido y privado).
2. Constructores
3. Métodos de fábrica
4. Otros Métodos (en orden público, protegido y privado).

No es necesario ordenar campos y métodos principalmente por sus modificadores de acceso o identificador.

Aquí hay un ejemplo de este orden:

```
class Example {

    private int i;

    Example(int i) {
        this.i = i;
    }

    static Example getExample(int i) {
        return new Example(i);
    }

    @Override
    public String toString() {
        return "An example [" + i + "]";
    }

}
```

Agrupación de miembros de la clase.

- Los campos relacionados deben agruparse.
- Un tipo anidado puede ser declarado justo antes de su primer uso; De lo contrario se debe declarar antes de los campos.
- Los constructores y los métodos sobrecargados deben agruparse por funcionalidad y ordenarse con una aridad creciente. Esto implica que la delegación entre estas construcciones fluye hacia abajo en el código.
- Los constructores deben agruparse sin otros miembros entre ellos.
- Las variantes sobrecargadas de un método deben agruparse juntas sin otros miembros entre ellas.

Modificadores

```
class ExampleClass {
    // Access modifiers first (don't do for instance "static public")
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

```
interface ExampleInterface {
    // Avoid 'public' and 'abstract' since they are implicit
    void sayHello();
}
```

- Los modificadores deben ir en el siguiente orden
    - Modificador de acceso (`public` / `private` / `protected`)
    - `abstract`
    - `static`
    - `final`
    - `transient`
    - `volatile`
    - `default`
    - `synchronized`
    - `native`
    - `strictfp`
- Los modificadores no deben escribirse cuando están implícitos. Por ejemplo, los métodos de interfaz no deben declararse `public` ni `abstract` , y las enums e interfaces anidadas no deben declararse estáticas.
- Los parámetros del método y las variables locales no deben declararse `final` menos que mejore la legibilidad o documente una decisión de diseño real.
- Los campos deben declararse `final` menos que haya una razón convincente para hacerlos mutables.

Sangría

- El nivel de sangrado es de **cuatro espacios** .
- Solo se pueden usar caracteres de espacio para la sangría. **No hay pestañas.**
- Las líneas vacías no deben ser sangradas. (Esto está implícito en la regla de espacio en blanco final).
- `case` líneas del `case` deben estar sangradas con cuatro espacios, y las declaraciones dentro del caso deben estar sangradas con otros cuatro espacios.

```
switch (var) {
    case TWO:
        setChoice("two");
        break;
    case THREE:
        setChoice("three");
        break;
    default:
        throw new IllegalArgumentException();
}
```

Consulte las instrucciones de ajuste para obtener instrucciones sobre cómo sangrar líneas de continuación.

Envolver declaraciones

- El código fuente y los comentarios generalmente no deben exceder los 80 caracteres por línea y rara vez, si alguna vez, exceden los 100 caracteres por línea, incluida la sangría.

  El límite de caracteres se debe juzgar caso por caso. Lo que realmente importa es la "densidad" semántica y la legibilidad de la línea. Hacer líneas gratuitamente largas las hace difíciles de leer; de manera similar, hacer "intentos heroicos" para encajarlos en 80 columnas también puede hacer que sean difíciles de leer. La flexibilidad descrita aquí apunta a permitir a los desarrolladores evitar estos extremos, no maximizar el uso del monitor de bienes raíces.

- Las URL o los comandos de ejemplo no se deben envolver.

```
// Ok even though it might exceed max line width when indented.
Error e = isTypeParam
        ? Errors.InvalidRepeatableAnnotationNotApplicable(targetContainerType, on)
        : Errors.InvalidRepeatableAnnotationNotApplicableInContext(targetContainerType));

// Wrapping preferable
String pretty = Stream.of(args)
                      .map(Argument::prettyPrint)
                      .collectors(joining(", "));

// Too strict interpretation of max line width. Readability suffers.
Error e = isTypeParam
        ? Errors.InvalidRepeatableAnnotationNotApplicable(
                targetContainerType, on)
        : Errors.InvalidRepeatableAnnotationNotApplicableInContext(
                targetContainerType);

// Should be wrapped even though it fits within the character limit
String pretty = Stream.of(args).map(Argument::prettyPrint).collectors(joining(", "));
```

- Se prefiere envolver en un nivel sintáctico más alto que envolver en un nivel sintáctico más bajo.

- Debe haber como máximo 1 declaración por línea.

- Una línea de continuación debe estar sangrada de una de las siguientes cuatro formas

  - **Variante 1** : Con 8 espacios adicionales relativos a la sangría de la línea anterior.
  - **Variante 2** : con 8 espacios adicionales en relación con la columna inicial de la expresión envuelta.
  - **Variante 3** : alineada con la expresión de hermano anterior (siempre que quede claro que es una línea de continuación)
  - **Variante 4** : Alineado con la llamada al método anterior en una expresión encadenada.

Declaraciones del método de envoltura

```
int someMethod(String aString,
               List<Integer> aList,
               Map<String, String> aMap,
               int anInt,
               long aLong,
```

```
                Set<Number> aSet,
                double aDouble) {
    …
}

int someMethod(String aString, List<Integer> aList,
        Map<String, String> aMap, int anInt, long aLong,
        double aDouble, long aLong) {
    …
}

int someMethod(String aString,
                List<Map<Integer, StringBuffer>> aListOfMaps,
                Map<String, String> aMap)
        throws IllegalArgumentException {
    …
}

int someMethod(String aString, List<Integer> aList,
        Map<String, String> aMap, int anInt)
                throws IllegalArgumentException {
    …
}
```

- Las declaraciones de métodos se pueden formatear enumerando los argumentos verticalmente, o por una nueva línea y +8 espacios adicionales
- Si es necesario ajustar una cláusula de lanzamientos, coloque el salto de línea delante de la cláusula de lanzamientos y asegúrese de que se destaque de la lista de argumentos, ya sea sangrando +8 con relación a la declaración de la función, o +8 con respecto a la línea anterior.

Expresiones de envoltura

- Si una línea se acerca al límite máximo de caracteres, siempre considere dividirlo en múltiples declaraciones / expresiones en lugar de envolver la línea.
- Romper ante los operadores.
- Romper antes de la. En las llamadas de método encadenado.

```
popupMsg("Inbox notification: You have "
        + newMsgs + " new messages");

// Don't! Looks like two arguments
popupMsg("Inbox notification: You have " +
         newMsgs + " new messages");
```

Espacio en blanco

Espacio en blanco vertical

- Se debe usar una sola línea en blanco para separar ...
    - Declaración del paquete

    - Declaraciones de clase
    - Constructores
    - Métodos
    - Inicializadores estáticos
    - Inicializadores de instancia

- ... y puede ser usado para separar grupos lógicos de

    - declaraciones de importación
    - campos
    - declaraciones

- Múltiples líneas en blanco consecutivas solo deben usarse para separar grupos de miembros relacionados y no como el espaciado entre miembros estándar.

Espacio en blanco horizontal

- Se debe usar un solo espacio ...

    - Para separar palabras clave de corchetes y llaves de apertura o cierre vecinos
    - Antes y después de todos los operadores binarios y el operador, como símbolos como las flechas en las expresiones lambda y los dos puntos mejorados para los bucles (pero no antes de los dos puntos de una etiqueta)
    - Después de `//` eso comienza un comentario.
    - Después de comas, separa los argumentos y los puntos y comas separan las partes de un bucle for.
    - Tras el paréntesis de cierre de un reparto.

- En declaraciones de variables no se recomienda alinear tipos y variables.

Declaraciones Variables

- Una variable por declaración (y como máximo una declaración por línea)
- Los corchetes de las matrices deben estar en el tipo (`String[] args`) y no en la variable (`String args[]`).
- Declare una variable local justo antes de usarla por primera vez e inicialícela lo más cerca posible de la declaración.

Anotaciones

Las anotaciones de la declaración deben colocarse en una línea separada de la declaración que se está anotando.

```
@SuppressWarnings("unchecked")
public T[] toArray(T[] typeHolder) {
    ...
}
```

Sin embargo, pocas anotaciones o anotaciones cortas que anotan un método de una sola línea se pueden colocar en la misma línea que el método si mejora la legibilidad. Por ejemplo, uno puede escribir:

```
@Nullable String getName() { return name; }
```

Por cuestiones de coherencia y legibilidad, todas las anotaciones deben colocarse en la misma línea o cada anotación debe colocarse en una línea separada.

```
// Bad.
@Deprecated @SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}

// Even worse.
@Deprecated @SafeVarargs
@CustomAnnotation public final Tuple<T> extend(T... elements) {
    ...
}

// Good.
@Deprecated
@SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}

// Good.
@Deprecated @SafeVarargs @CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}
```

Expresiones lambda

```
Runnable r = () -> System.out.println("Hello World");

Supplier<String> c = () -> "Hello World";

// Collection::contains is a simple unary method and its behavior is
// clear from the context. A method reference is preferred here.
appendFilter(goodStrings::contains);

// A lambda expression is easier to understand than just tempMap::put in this case
trackTemperature((time, temp) -> tempMap.put(time, temp));
```

- Las lambdas de expresión se prefieren a las lambdas de bloque de una sola línea.
- Las referencias a los métodos generalmente deben preferirse a las expresiones lambda.
- Para referencias de métodos de instancia unida, o métodos con una aridad mayor que uno, una expresión lambda puede ser más fácil de entender y, por lo tanto, preferible. Especialmente si el comportamiento del método no está claro en el contexto.

- Los tipos de parámetros deben omitirse a menos que mejoren la legibilidad.
- Si una expresión lambda se extiende sobre más de unas pocas líneas, considere crear un método.

Paréntesis redundantes

```
return flag ? "yes" : "no";

String cmp = (flag1 != flag2) ? "not equal" : "equal";

// Don't do this
return (flag ? "yes" : "no");
```

- Se pueden usar paréntesis de agrupación redundantes (es decir, paréntesis que no afectan la evaluación) si mejoran la legibilidad.
- Los paréntesis de agrupación redundantes normalmente se deben omitir en expresiones más cortas que involucren operadores comunes, pero se deben incluir en expresiones más largas o que involucren operadores cuya prioridad y asociatividad no estén claras sin paréntesis. Las expresiones ternarias con condiciones no triviales pertenecen a esta última.
- La expresión completa que sigue a una palabra clave de `return` no debe estar entre paréntesis.

Literales

```
long l = 5432L;
int i = 0x123 + 0xABC;
byte b = 0b1010;
float f1 = 1 / 5432f;
float f2 = 0.123e4f;
double d1 = 1 / 5432d;  // or 1 / 5432.0
double d2 = 0x1.3p2;
```

- `long` literales `long` deben usar el sufijo de la letra `L` mayúscula.
- Los literales hexadecimales deben usar letras mayúsculas `A` - `F`
- Todos los demás prefijos numéricos, infijos y sufijos deben usar letras minúsculas.

Tirantes

```
class Example {
    void method(boolean error) {
        if (error) {
            Log.error("Error occurred!");
            System.out.println("Error!");
        } else { // Use braces since the other block uses braces.
            System.out.println("No error");
        }
    }
}
```

- Las llaves de apertura deben colocarse en el final de la línea actual en lugar de en una línea por sí sola.

- Debería haber una nueva línea delante de una abrazadera de cierre, a menos que el bloque esté vacío (vea las formas cortas a continuación)
- Los frenos se recomiendan incluso cuando el lenguaje los hace opcionales, como los cuerpos de bucle y de una sola línea.
    - Si un bloque abarca más de una línea (incluidos los comentarios) debe tener llaves.
    - Si uno de los bloques en una sentencia `if` / `else` tiene llaves, el otro bloque también debe hacerlo.
    - Si el bloque es el último en un bloque adjunto, debe tener llaves.
- La palabra clave `else` , `catch` y `while` en `do…while` bucles van en la misma línea que la llave de cierre del bloque anterior.

Formas cortas

```
enum Response { YES, NO, MAYBE }
public boolean isReference() { return true; }
```

Las recomendaciones anteriores están destinadas a mejorar la uniformidad (y por lo tanto aumentar la familiaridad / legibilidad). En algunos casos, las "formas cortas" que se desvían de las pautas anteriores son tan legibles y pueden usarse en su lugar. Estos casos incluyen, por ejemplo, declaraciones de enumeración simples y métodos triviales y expresiones lambda.

Lea Código oficial de Oracle estándar en línea: https://riptutorial.com/es/java/topic/2697/codigo-oficial-de-oracle-estandar

Capítulo 39: Colas y deques

Examples

El uso de la PriorityQueue

`PriorityQueue` es una estructura de datos. Al igual que `SortedSet` , `PriorityQueue` también clasifica sus elementos según sus prioridades. Los elementos, que tienen una prioridad más alta, vienen primero. El tipo de `PriorityQueue` debe implementar `comparable` interfaz `comparable` o de `comparator` , cuyos métodos deciden las prioridades de los elementos de la estructura de datos.

```
//The type of the PriorityQueue is Integer.
PriorityQueue<Integer> queue = new PriorityQueue<Integer>();

//The elements are added to the PriorityQueue
queue.addAll( Arrays.asList( 9, 2, 3, 1, 3, 8 ) );

//The PriorityQueue sorts the elements by using compareTo method of the Integer Class
//The head of this queue is the least element with respect to the specified ordering
System.out.println( queue );  //The Output: [1, 2, 3, 9, 3, 8]
queue.remove();
System.out.println( queue );  //The Output: [2, 3, 3, 9, 8]
queue.remove();
System.out.println( queue );  //The Output: [3, 8, 3, 9]
queue.remove();
System.out.println( queue );  //The Output: [3, 8, 9]
queue.remove();
System.out.println( queue );  //The Output: [8, 9]
queue.remove();
System.out.println( queue );  //The Output: [9]
queue.remove();
System.out.println( queue );  //The Output: []
```

LinkedList como una cola FIFO

La clase `java.util.LinkedList` , mientras implementa `java.util.List` es una implementación de propósito general de la interfaz `java.util.Queue` que también funciona con un principio FIFO (First In, First Out) .

En el siguiente ejemplo, con el método `offer()` , los elementos se insertan en `LinkedList` . Esta operación de inserción se llama `enqueue` . En el `while` bucle a continuación, los elementos se eliminan de la `Queue` sobre la base de FIFO. Esta operación se llama `dequeue` .

```
Queue<String> queue = new LinkedList<String>();

queue.offer( "first element" );
queue.offer( "second element" );
queue.offer( "third element" );
queue.offer( "fourth. element" );
queue.offer( "fifth. element" );
```

```
while ( !queue.isEmpty() ) {
  System.out.println( queue.poll() );
}
```

La salida de este código es

```
first element
second element
third element
fourth element
fifth element
```

Como se ve en la salida, el primer elemento insertado "primer elemento" se elimina en primer lugar, el "segundo elemento" se elimina en el segundo lugar, etc.

Pilas

¿Qué es una pila?

En Java, las pilas son una estructura de datos LIFO (Last In, First Out) para objetos.

API de pila

Java contiene una API de pila con los siguientes métodos

```
Stack()             //Creates an empty Stack
isEmpty()           //Is the Stack Empty?             Return Type: Boolean
push(Item item)     //push an item onto the stack
pop()               //removes item from top of stack  Return Type: Item
size()              //returns # of items in stack     Return Type: Int
```

Ejemplo

```
import java.util.*;

public class StackExample {

   public static void main(String args[]) {
      Stack st = new Stack();
      System.out.println("stack: " + st);

      st.push(10);
      System.out.println("10 was pushed to the stack");
      System.out.println("stack: " + st);

      st.push(15);
      System.out.println("15 was pushed to the stack");
      System.out.println("stack: " + st);
```

```
      st.push(80);
      System.out.println("80 was pushed to the stack");
      System.out.println("stack: " + st);

      st.pop();
      System.out.println("80 was popped from the stack");
      System.out.println("stack: " + st);

      st.pop();
      System.out.println("15 was popped from the stack");
      System.out.println("stack: " + st);

      st.pop();
      System.out.println("10 was popped from the stack");
      System.out.println("stack: " + st);

     if(st.isEmpty())
      {
         System.out.println("empty stack");
      }
   }
}
```

Esto devuelve:

```
stack: []
10 was pushed to the stack
stack: [10]
15 was pushed to the stack
stack: [10, 15]
80 was pushed to the stack
stack: [10, 15, 80]
80 was popped from the stack
stack: [10, 15]
15 was popped from the stack
stack: [10]
10 was popped from the stack
stack: []
empty stack
```

BlockingQueue

Un BlockingQueue es una interfaz, que es una cola que se bloquea cuando intenta salir de la cola y la cola está vacía, o si intenta poner en cola elementos en ella y la cola ya está llena. Se bloquea un subproceso que intenta salir de una cola vacía hasta que otro subproceso inserta un elemento en la cola. Un subproceso que intenta poner en cola un elemento en una cola completa se bloquea hasta que algún otro subproceso haga espacio en la cola, ya sea retirando uno o más elementos o eliminando la cola por completo.

Los métodos de BlockingQueue vienen en cuatro formas, con diferentes formas de manejar las operaciones que no pueden satisfacerse inmediatamente, pero que pueden satisfacerse en algún momento en el futuro: uno lanza una excepción, el segundo devuelve un valor especial (ya sea nulo o falso, dependiendo de la operación), el tercero bloquea el subproceso actual indefinidamente hasta que la operación pueda tener éxito, y el cuarto bloquea solo un límite de tiempo máximo dado antes de rendirse.

Operación	Lanza la excepción	Valor especial	Bloques	Se acabó el tiempo
Insertar	añadir()	oferta (e)	poner (e)	oferta (e, tiempo, unidad)
retirar	retirar()	encuesta()	tomar()	encuesta (tiempo, unidad)
Examinar	elemento()	ojeada()	N / A	N / A

Un `BlockingQueue` puede ser **limitado** o **ilimitado** . Un BlockingQueue limitado es aquel que se inicializa con la capacidad inicial.

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);
```

Cualquier llamada a un método put () se bloqueará si el tamaño de la cola es igual a la capacidad inicial definida.

Una cola ilimitada es aquella que se inicializa sin capacidad, de hecho, de forma predeterminada, se inicializa con Integer.MAX_VALUE.

Algunas implementaciones comunes de `BlockingQueue` son:

1. ArrayBlockingQueue
2. LinkedBlockingQueue
3. PriorityBlockingQueue

Ahora veamos un ejemplo de `ArrayBlockingQueue` :

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<>(2);
bQueue.put("This is entry 1");
System.out.println("Entry one done");
bQueue.put("This is entry 2");
System.out.println("Entry two done");
bQueue.put("This is entry 3");
System.out.println("Entry three done");
```

Esto imprimirá:

```
Entry one done
Entry two done
```

Y el hilo será bloqueado después de la segunda salida.

Interfaz de cola

Lo esencial

Una `Queue` es una colección para contener elementos antes del procesamiento. Las colas normalmente, pero no necesariamente, ordenan los elementos de una manera FIFO (primero en

entrar, primero en salir).

El encabezado de la cola es el elemento que se eliminaría con una llamada para eliminar o sondear. En una cola FIFO, todos los elementos nuevos se insertan en la cola de la cola.

La interfaz de cola

```
public interface Queue<E> extends Collection<E> {
    boolean add(E e);

    boolean offer(E e);

    E remove();

    E poll();

    E element();

    E peek();
}
```

Cada método de `Queue` existe en dos formas:

- uno lanza una excepción si la operación falla;
- otro devuelve un valor especial si la operación falla (ya sea `null` o `false` según la operación).

Tipo de operación	Lanza excepción	Devuelve valor especial
Insertar	`add(e)`	`offer(e)`
retirar	`remove()`	`poll()`
Examinar	`element()`	`peek()`

Deque

Un `Deque` es una "cola de doble finalización", lo que significa que se pueden agregar elementos en la parte delantera o en la cola de la cola. La cola solo puede agregar elementos a la cola de una cola.

`Deque` hereda la interfaz de la `Queue` , lo que significa que los métodos regulares permanecen, sin embargo, la interfaz de Deque ofrece métodos adicionales para ser más flexible con una cola. Los métodos adicionales realmente hablan por sí mismos si sabes cómo funciona una cola, ya que esos métodos están destinados a agregar más flexibilidad:

Método	Breve descripción
`getFirst()`	Obtiene el primer elemento de la **cabecera** de la cola sin eliminarlo.
`getLast()`	Obtiene el primer elemento de la **cola** de la cola sin eliminarlo.

Método	Breve descripción
`addFirst(E e)`	Agrega un elemento a la **cabecera** de la cola.
`addLast(E e)`	Agrega un elemento a la **cola** de la cola.
`removeFirst()`	Elimina el primer elemento en la **cabeza** de la cola
`removeLast()`	Elimina el primer elemento en la **cola** de la cola

Por supuesto las mismas opciones para `offer` , `poll` y `peek` están disponibles, sin embargo, no funcionan con excepciones, sino más bien con los valores especiales. No tiene sentido mostrar lo que hacen aquí.

Agregar y acceder a los elementos

Para agregar elementos a la cola de un Deque, se llama a su método `add()` . También puede usar los `addFirst()` y `addLast()` , que agregan elementos a la cabeza y la cola del deque.

```
Deque<String> dequeA = new LinkedList<>();

dequeA.add("element 1");       //add element at tail
dequeA.addFirst("element 2"); //add element at head
dequeA.addLast("element 3");  //add element at tail
```

Puede echar un vistazo al elemento que se encuentra al principio de la cola sin sacar el elemento de la cola. Esto se hace a través del método `element()` . También puede usar los `getFirst()` y `getLast()` , que devuelven el primer y último elemento en el `Deque` . Aquí es cómo se ve:

```
String firstElement0 = dequeA.element();
String firstElement1 = dequeA.getFirst();
String lastElement = dequeA.getLast();
```

Removiendo elementos

Para eliminar elementos de un deque, debe llamar a los métodos `remove()` , `removeFirst()` y `removeLast()` . Aquí están algunos ejemplos:

```
String firstElement = dequeA.remove();
String firstElement = dequeA.removeFirst();
String lastElement  = dequeA.removeLast();
```

Lea Colas y deques en línea: https://riptutorial.com/es/java/topic/7196/colas-y-deques

Capítulo 40: Colecciones

Introducción

El marco de colecciones en `java.util` proporciona una serie de clases genéricas para conjuntos de datos con una funcionalidad que no pueden ser proporcionadas por matrices regulares.

El marco de colecciones contiene interfaces para la `Collection<O>` , con la lista de subinterfaces principales `List<O>` y `Set<O>` , y el `Map<K,V>` colección de mapeo `Map<K,V>` . Las colecciones son la interfaz raíz y están siendo implementadas por muchos otros marcos de colección.

Observaciones

Las colecciones son objetos que pueden almacenar colecciones de otros objetos dentro de ellos. Puede especificar el tipo de datos almacenados en una colección usando Genéricos .

Las colecciones generalmente usan los espacios de nombres `java.util` o `java.util.concurrent` .

Java SE 1.4

Java 1.4.2 y versiones anteriores no son compatibles con los genéricos. Como tal, no puede especificar los parámetros de tipo que contiene una colección. Además de no tener seguridad de tipo, también debe usar moldes para recuperar el tipo correcto de una colección.

Además de la `Collection<E>` , hay varios tipos principales de colecciones, algunas de las cuales tienen subtipos.

- `List<E>` es una colección ordenada de objetos. Es similar a una matriz, pero no define un límite de tamaño. Las implementaciones generalmente aumentarán de tamaño internamente para acomodar nuevos elementos.
- `Set<E>` es una colección de objetos que no permite duplicados.
  - `SortedSet<E>` es un `Set<E>` que también especifica el orden de los elementos.
- `Map<K,V>` es una colección de pares clave / valor.
  - `SortedMap<K,V>` es un `Map<K,V>` que también especifica el orden de los elementos.

Java SE 5

Java 5 agrega un nuevo tipo de colección:

- `Queue<E>` es una colección de elementos destinados a ser procesados en un orden específico. La implementación especifica si esto es FIFO o LIFO. Esto obsoleta la clase `Stack` .

Java SE 6

Java 6 agrega algunos nuevos subtipos de colecciones.

- `NavigableSet<E>` es un `Set<E>` con métodos de navegación especiales integrados.
- `NavigableMap<K,V>` es un `Map<K,V>` con métodos de navegación especiales integrados.
- `Deque<E>` es una `Queue<E>` que puede leerse desde cualquier extremo.

Tenga en cuenta que los elementos anteriores son todas las interfaces. Para utilizarlos, debe encontrar las clases de implementación adecuadas, como `ArrayList` , `HashSet` , `HashMap` o `PriorityQueue` .

Cada tipo de colección tiene implementaciones múltiples que tienen diferentes métricas de rendimiento y casos de uso.

Tenga en cuenta que el principio de sustitución de Liskov se aplica a los subtipos de colección. Es decir, se puede pasar un `SortedSet<E>` a una función que espera un `Set<E>` . También es útil leer sobre Parámetros delimitados en la sección Genéricos para obtener más información sobre cómo usar colecciones con herencia de clase.

Si desea crear sus propias colecciones, puede ser más fácil heredar una de las clases abstractas (como `AbstractList`) en lugar de implementar la interfaz.

Java SE 1.2

Antes de la versión 1.2, tenías que usar las siguientes clases / interfaces en su lugar:

- `Vector` lugar de `ArrayList`
- `Dictionary` lugar de `Map` . Tenga en cuenta que el diccionario también es una clase abstracta en lugar de una interfaz.
- `Hashtable` lugar de `HashMap`

Estas clases son obsoletas y no deben usarse en el código moderno.

Examples

Declarar una ArrayList y agregar objetos

Podemos crear un `ArrayList` (siguiendo la interfaz de la `List`):

```
List aListOfFruits = new ArrayList();
```

Java SE 5

```
List<String> aListOfFruits = new ArrayList<String>();
```

Java SE 7

```
List<String> aListOfFruits = new ArrayList<>();
```

Ahora, usa el método `add` para agregar una `String` :

```
aListOfFruits.add("Melon");
```

```
aListOfFruits.add("Strawberry");
```

En el ejemplo anterior, `ArrayList` contendrá la `String` "Melon" en el índice 0 y la `String` "Strawberry" en el índice 1.

También podemos agregar varios elementos con el método `addAll(Collection<? extends E> c)`

```
List<String> aListOfFruitsAndVeggies = new ArrayList<String>();
aListOfFruitsAndVeggies.add("Onion");
aListOfFruitsAndVeggies.addAll(aListOfFruits);
```

Ahora "Cebolla" se coloca en el índice 0 en `aListOfFruitsAndVeggies` , "Melón" está en el índice 1 y "Fresa" está en el índice 2.

Construyendo colecciones a partir de datos existentes

Colecciones estandar

Marco de Colecciones Java

Una forma sencilla de construir una `List` partir de valores de datos individuales es usar el método de `Arrays.asList` `java.utils.Arrays` :

```
List<String> data = Arrays.asList("ab", "bc", "cd", "ab", "bc", "cd");
```

Todas las implementaciones de colección estándar proporcionan constructores que toman otra colección como un argumento que agrega todos los elementos a la nueva colección en el momento de la construcción:

```
List<String> list = new ArrayList<>(data); // will add data as is
Set<String> set1 = new HashSet<>(data); // will add data keeping only unique values
SortedSet<String> set2 = new TreeSet<>(data); // will add data keeping unique values and
sorting
Set<String> set3 = new LinkedHashSet<>(data); // will add data keeping only unique values and
preserving the original order
```

Marco de colecciones de guayaba de Google

Otra gran marco es `Google Guava` que está clase de utilidad increíble (proporcionando métodos estáticos de conveniencia) para la construcción de diferentes tipos de colecciones estándar `Lists` y `Sets` :

```
import com.google.common.collect.Lists;
import com.google.common.collect.Sets;
...
List<String> list1 = Lists.newArrayList("ab", "bc", "cd");
List<String> list2 = Lists.newArrayList(data);
```

```
Set<String> set4 = Sets.newHashSet(data);
SortedSet<String> set5 = Sets.newTreeSet("bc", "cd", "ab", "bc", "cd");
```

Mapeo de Colecciones

Marco de Colecciones Java

De manera similar, para los mapas, dado un `Map<String, Object> map` se puede construir un nuevo mapa con todos los elementos de la siguiente manera:

```
Map<String, Object> map1 = new HashMap<>(map);
SortedMap<String, Object> map2 = new TreeMap<>(map);
```

Marco de Apache Commons Collections

Usando `Apache Commons` puede crear un mapa usando una matriz en `ArrayUtils.toMap` así como `MapUtils.toMap` :

```
import org.apache.commons.lang3.ArrayUtils;
...
// Taken from org.apache.commons.lang.ArrayUtils#toMap JavaDoc

// Create a Map mapping colors.
Map colorMap = MapUtils.toMap(new String[][] {{
    {"RED", "#FF0000"},
    {"GREEN", "#00FF00"},
    {"BLUE", "#0000FF"}});
```

Cada elemento de la matriz debe ser Map.Entry o Array, que contenga al menos dos elementos, donde el primer elemento se utiliza como clave y el segundo como valor.

Marco de colecciones de guayaba de Google

La clase de utilidad del marco de `Google Guava` se llama `Maps` :

```
import com.google.common.collect.Maps;
...
void howToCreateMapsMethod(Function<? super K,V> valueFunction,
          Iterable<K> keys1,
          Set<K> keys2,
          SortedSet<K> keys3) {
    ImmutableMap<K, V> map1 = toMap(keys1, valueFunction); // Immutable copy
    Map<K, V> map2 = asMap(keys2, valueFunction); // Live Map view
    SortedMap<K, V> map3 = toMap(keys3, valueFunction); // Live Map view
}
```

Java SE 8

Utilizando `Stream` ,

```
Stream.of("xyz", "abc").collect(Collectors.toList());
```

o

```
Arrays.stream("xyz", "abc").collect(Collectors.toList());
```

Unir listas

Se pueden utilizar las siguientes formas para unir listas sin modificar la (s) lista (s) de origen.

Primer enfoque. Tiene más líneas pero es fácil de entender.

```
List<String> newList = new ArrayList<String>();
newList.addAll(listOne);
newList.addAll(listTwo);
```

Segundo enfoque. Tiene una línea menos pero menos legible.

```
List<String> newList = new ArrayList<String>(listOne);
newList.addAll(listTwo);
```

Tercer enfoque. Requiere de la biblioteca de colecciones comunes de Apache de terceros.

```
ListUtils.union(listOne,listTwo);
```

Java SE 8

Usando Streams se puede lograr lo mismo por

```
List<String> newList = Stream.concat(listOne.stream(),
listTwo.stream()).collect(Collectors.toList());
```

Referencias. Lista de interfaces

Eliminar elementos de una lista dentro de un bucle

Es difícil eliminar elementos de una lista mientras se encuentra dentro de un bucle, esto se debe al hecho de que el índice y la longitud de la lista se modifican.

Dada la siguiente lista, aquí hay algunos ejemplos que darán un resultado inesperado y otros que darán el resultado correcto.

```
List<String> fruits = new ArrayList<String>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Strawberry");
```

INCORRECTO

Extracción de iteración de `for` declaración *Omite "Banana":*

El ejemplo de código solo imprimirá `Apple` y `Strawberry` . `Banana` se omite porque se mueve al índice `0` vez `Apple` se elimina, pero, al mismo tiempo `i` se incrementa a `1` .

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

Eliminando en la *excepción* mejorada `for` *tiros de* instrucción *:*

Debido a iterar sobre la colección y modificarla al mismo tiempo.

> Emite: java.util.ConcurrentModificationException

```
for (String fruit : fruits) {
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruits.remove(fruit);
    }
}
```

CORRECTO

Eliminando en bucle while usando un `Iterator`

```
Iterator<String> fruitIterator = fruits.iterator();
while(fruitIterator.hasNext()) {
    String fruit = fruitIterator.next();
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruitIterator.remove();
    }
}
```

La interfaz `Iterator` tiene un método `remove()` incorporado solo para este caso. Sin embargo, este método está marcado como "opcional" en la documentación y podría generar una `UnsupportedOperationException` .

> Emite: UnsupportedOperationException - si la operación de eliminación no es

compatible con este iterador

Por lo tanto, es recomendable consultar la documentación para asegurarse de que esta operación sea compatible (en la práctica, a menos que la recopilación sea una inmutable obtenida a través de una biblioteca de terceros o el uso de uno de los métodos `Collections.unmodifiable...()` , la operación es casi siempre soportada).

Al utilizar un `Iterator` una `Iterator` `ConcurrentModificationException` cuando se `modCount` el `modCount` de la `List` desde que se creó el `Iterator` . Esto podría haber ocurrido en el mismo hilo o en una aplicación multihilo compartiendo la misma lista.

Un `modCount` es una variable `int` que cuenta el número de veces que esta lista ha sido modificada estructuralmente. Un cambio estructural significa esencialmente una operación `add()` o `remove()` que se invoca en el objeto `Collection` (los cambios realizados por `Iterator` no se cuentan). Cuando se crea el `Iterator` , almacena este `modCount` y en cada iteración de la `List` comprueba si el `modCount` actual es el mismo que cuando se creó el `Iterator` . Si hay un cambio en el valor de `modCount` , lanza una `ConcurrentModificationException` .

Por lo tanto, para la lista declarada anteriormente, una operación como la siguiente no arrojará ninguna excepción:

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.set(0, "Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());
}
```

Pero agregar un nuevo elemento a la `List` después de inicializar un `Iterator` lanzará una `ConcurrentModificationException` :

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.add("Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());    //ConcurrentModificationException here
}
```

Iterando hacia atrás

```
for (int i = (fruits.size() - 1); i >=0; i--) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

Esto no se salta nada. La desventaja de este enfoque es que la salida es inversa. Sin embargo, en la mayoría de los casos, elimina elementos que no importan. Nunca debes hacer esto con `LinkedList` .

Iterando hacia adelante, ajustando el índice de bucle.

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
         i--;
    }
}
```

Esto no se salta nada. Cuando el elemento `i` th se elimina de la `List` , el elemento originalmente posicionado en el índice `i+1` convierte en el nuevo elemento `i` th. Por lo tanto, el bucle puede decrementar `i` para que la siguiente iteración procese el siguiente elemento, sin saltarse.

Usando una lista de "debería ser eliminado"

```
ArrayList shouldBeRemoved = new ArrayList();
for (String str : currentArrayList) {
    if (condition) {
        shouldBeRemoved.add(str);
    }
}
currentArrayList.removeAll(shouldBeRemoved);
```

Esta solución permite al desarrollador verificar si los elementos correctos se eliminan de forma más limpia.

Java SE 8

En Java 8 son posibles las siguientes alternativas. Estos son más limpios y más directos si la eliminación no tiene que suceder en un bucle.

Filtrando una corriente

Una `List` puede ser transmitida y filtrada. Se puede utilizar un filtro adecuado para eliminar todos los elementos no deseados.

```
List<String> filteredList =
    fruits.stream().filter(p -> !"Apple".equals(p)).collect(Collectors.toList());
```

Tenga en cuenta que a diferencia de todos los otros ejemplos aquí, este ejemplo produce una nueva instancia de `List` y mantiene la `List` original sin cambios.

Utilizando `removeIf`

Ahorra la sobrecarga de construir un flujo si todo lo que se necesita es eliminar un conjunto de elementos.

```
fruits.removeIf(p -> "Apple".equals(p));
```

Colección no modificable

A veces no es una buena práctica exponer una colección interna, ya que puede provocar una vulnerabilidad de código malicioso debido a su característica mutable. Para proporcionar colecciones de "solo lectura", java proporciona sus versiones no modificables.

Una colección no modificable es a menudo una copia de una colección modificable que garantiza que la colección en sí no puede ser alterada. Los intentos de modificarlo darán como resultado una excepción UnsupportedOperationException.

Es importante notar que los objetos que están presentes dentro de la colección todavía pueden ser alterados.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MyPojoClass {
    private List<Integer> intList = new ArrayList<>();

    public void addValueToIntList(Integer value){
        intList.add(value);
    }

    public List<Integer> getIntList() {
        return Collections.unmodifiableList(intList);
    }
}
```

El siguiente intento de modificar una colección no modificable generará una excepción:

```
import java.util.List;

public class App {

    public static void main(String[] args) {
        MyPojoClass pojo = new MyPojoClass();
        pojo.addValueToIntList(42);

        List<Integer> list = pojo.getIntList();
        list.add(69);
    }
}
```

salida:

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
    at App.main(App.java:12)
```

Iterando sobre colecciones

Iterando sobre la lista

```
List<String> names  = new ArrayList<>(Arrays.asList("Clementine", "Duran", "Mike"));
```

Java SE 8

```
names.forEach(System.out::println);
```

Si necesitamos paralelismo de uso.

```
names.parallelStream().forEach(System.out::println);
```

Java SE 5

```
for (String name : names) {
    System.out.println(name);
}
```

Java SE 5

```
for (int i = 0; i < names.size(); i++) {
    System.out.println(names.get(i));
}
```

Java SE 1.2

```
//Creates ListIterator which supports both forward as well as backward traversel
ListIterator<String> listIterator = names.listIterator();

//Iterates list in forward direction
while(listIterator.hasNext()){
    System.out.println(listIterator.next());
}

//Iterates list in backward direction once reaches the last element from above iterator in
forward direction
while(listIterator.hasPrevious()){
    System.out.println(listIterator.previous());
}
```

Iterando sobre Set

```
Set<String> names = new HashSet<>(Arrays.asList("Clementine", "Duran", "Mike"));
```

Java SE 8

```
names.forEach(System.out::println);
```

Java SE 5

```
for (Iterator<String> iterator = names.iterator(); iterator.hasNext(); ) {
    System.out.println(iterator.next());
}

for (String name : names) {
    System.out.println(name);
}
```

Java SE 5

```
Iterator iterator = names.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

Iterando sobre el mapa

```
Map<Integer, String> names = new HashMap<>();
names.put(1, "Clementine");
names.put(2, "Duran");
names.put(3, "Mike");
```

Java SE 8

```
names.forEach((key, value) -> System.out.println("Key: " + key + " Value: " + value));
```

Java SE 5

```
for (Map.Entry<Integer, String> entry : names.entrySet()) {
    System.out.println(entry.getKey());
    System.out.println(entry.getValue());
}

// Iterating over only keys
for (Integer key : names.keySet()) {
    System.out.println(key);
}
// Iterating over only values
for (String value : names.values()) {
    System.out.println(value);
}
```

Java SE 5

```
Iterator entries = names.entrySet().iterator();
while (entries.hasNext()) {
    Map.Entry entry = (Map.Entry) entries.next();
    System.out.println(entry.getKey());
    System.out.println(entry.getValue());
}
```

Colecciones vacias inmutables

A veces es apropiado utilizar una colección vacía inmutable. La clase `Collections` proporciona métodos para obtener dichas colecciones de una manera eficiente:

```
List<String> anEmptyList = Collections.emptyList();
Map<Integer, Date> anEmptyMap = Collections.emptyMap();
Set<Number> anEmptySet = Collections.emptySet();
```

Estos métodos son genéricos y convertirán automáticamente la colección devuelta al tipo al que está asignado. Es decir, se puede asignar una invocación de, por ejemplo, `emptyList()` a cualquier tipo de `List` y de la misma manera para `emptySet()` y `emptyMap()` .

Las colecciones devueltas por estos métodos son inmutables, ya que lanzarán la `UnsupportedOperationException` si intenta llamar a métodos que cambiarían su contenido (`add` , `put` , etc.). Estas colecciones son útiles principalmente como sustitutos de resultados de métodos vacíos u otros valores predeterminados, en lugar de usar `null` o crear objetos con `new` .

Colecciones y valores primitivos

Las colecciones en Java solo funcionan para objetos. Es decir, no hay `Map<int, int>` en Java. En su lugar, los valores primitivos deben *encuadrarse* en objetos, como en el `Map<Integer, Integer>` . Java auto-boxing permitirá el uso transparente de estas colecciones:

```
Map<Integer, Integer> map = new HashMap<>();
map.put(1, 17); // Automatic boxing of int to Integer objects
int a = map.get(1); // Automatic unboxing.
```

Desafortunadamente, la sobrecarga de esto es *sustancial* . Un `HashMap<Integer, Integer>` requerirá alrededor de 72 bytes por entrada (por ejemplo, en JVM de 64 bits con punteros comprimidos, y suponiendo que los enteros son mayores que 256, y suponiendo una carga del 50% del mapa). Debido a que los datos reales son solo 8 bytes, esto produce una sobrecarga masiva. Además, requiere dos niveles de direccionamiento indirecto (Mapa -> Entrada -> Valor) es innecesariamente lento.

Existen varias bibliotecas con colecciones optimizadas para tipos de datos primitivos (que requieren solo ~ 16 bytes por entrada con una carga del 50%, es decir, 4 veces menos memoria, y un nivel de direccionamiento indirecto menos), que pueden proporcionar beneficios sustanciales de rendimiento cuando se utilizan grandes colecciones de primitivos Valores en Java.

Eliminar elementos coincidentes de las listas utilizando Iterator.

Más arriba noté un ejemplo para eliminar elementos de una lista dentro de un bucle y pensé en otro ejemplo que podría ser útil esta vez usando la interfaz del `Iterator` .
Esta es una demostración de un truco que puede ser útil cuando se trata de elementos duplicados en listas de las que desea deshacerse.

Nota: Esto solo se agrega a la **eliminación de elementos de una lista dentro de un** ejemplo de **bucle** :

Así que definamos nuestras listas como de costumbre.

```
    String[] names = {"James","Smith","Sonny","Huckle","Berry","Finn","Allan"};
    List<String> nameList = new ArrayList<>();

    //Create a List from an Array
    nameList.addAll(Arrays.asList(names));

    String[] removeNames = {"Sonny","Huckle","Berry"};
    List<String> removeNameList = new ArrayList<>();

    //Create a List from an Array
    removeNameList.addAll(Arrays.asList(removeNames));
```

El siguiente método toma dos objetos de la Colección y realiza la magia de eliminar los elementos de nuestra `removeNameList` que coinciden con los elementos de `nameList` .

```
private static void removeNames(Collection<String> collection1, Collection<String>
collection2) {
      //get Iterator.
    Iterator<String> iterator = collection1.iterator();

    //Loop while collection has items
    while(iterator.hasNext()){
        if (collection2.contains(iterator.next()))
            iterator.remove(); //remove the current Name or Item
    }
}
```

Al llamar al método y que pasa en la `nameList` y la `removeNameList` como sigue
`removeNames(nameList,removeNameList);`
Producirá la siguiente salida:

> Lista de matrices antes de eliminar nombres: **James Smith Sonny Huckle Berry Finn Allan**
> Lista de matrices después de eliminar nombres: **James Smith Finn Allan**

Un uso simple y simple para colecciones que puede ser útil para eliminar elementos que se repiten en las listas.

Creando su propia estructura de Iterable para usar con Iterator o para cada bucle.

Para asegurarnos de que nuestra colección pueda iterarse utilizando iterador o para cada bucle, debemos seguir los siguientes pasos:

1. Lo que queremos iterar debe ser `Iterable` y exponer `iterator()` .
2. Diseñe un `java.util.Iterator` anulando `hasNext()` , `next()` y `remove()` .

A continuación, agregué una implementación de lista vinculada genérica simple que utiliza las entidades anteriores para hacer que la lista vinculada sea iterable.

```
package org.algorithms.linkedlist;

import java.util.Iterator;
import java.util.NoSuchElementException;


public class LinkedList<T> implements Iterable<T> {

    Node<T> head, current;

    private static class Node<T> {
        T data;
        Node<T> next;

        Node(T data) {
            this.data = data;
        }
    }

    public LinkedList(T data) {
        head = new Node<>(data);
    }

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements Iterator<T> {

        Node<T> node = head;

        @Override
        public boolean hasNext() {
            return node != null;
        }

        @Override
        public T next() {
            if (!hasNext())
                throw new NoSuchElementException();
            Node<T> prevNode = node;
            node = node.next;
            return prevNode.data;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException("Removal logic not implemented.");
        }
    }

    public void add(T data) {
        Node current = head;
        while (current.next != null)
            current = current.next;
        current.next = new Node<>(data);
    }

}

class App {
```

```
    public static void main(String[] args) {

        LinkedList<Integer> list = new LinkedList<>(1);
        list.add(2);
        list.add(4);
        list.add(3);

        //Test #1
        System.out.println("using Iterator:");
        Iterator<Integer> itr = list.iterator();
        while (itr.hasNext()) {
            Integer i = itr.next();
            System.out.print(i + " ");
        }

        //Test #2
        System.out.println("\n\nusing for-each:");
        for (Integer data : list) {
            System.out.print(data + " ");
        }
    }
}
```

Salida

```
using Iterator:
1 2 4 3
using for-each:
1 2 4 3
```

Esto se ejecutará en Java 7+. Puede hacer que se ejecute en Java 5 y Java 6 también sustituyendo:

```
LinkedList<Integer> list = new LinkedList<>(1);
```

con

```
LinkedList<Integer> list = new LinkedList<Integer>(1);
```

o cualquier otra versión incorporando los cambios compatibles.

Pitfall: excepciones de modificaciones concurrentes

Esta excepción se produce cuando se modifica una colección al iterar sobre ella utilizando métodos distintos a los proporcionados por el objeto iterador. Por ejemplo, tenemos una lista de sombreros y queremos eliminar todos aquellos que tienen orejeras:

```
List<IHat> hats = new ArrayList<>();
hats.add(new Ushanka()); // that one has ear flaps
hats.add(new Fedora());
hats.add(new Sombrero());
for (IHat hat : hats) {
    if (hat.hasEarFlaps()) {
        hats.remove(hat);
```

```
    }
}
```

Si ejecutamos este código, ***ConcurrentModificationException se*** generará ya que el código modifica la colección mientras lo iteramos. La misma excepción puede ocurrir si uno de los múltiples subprocesos que trabajan con la misma lista está intentando modificar la colección mientras que otros iteran sobre ella. La modificación simultánea de colecciones en varios subprocesos es algo natural, pero debe tratarse con las herramientas habituales de la caja de herramientas de programación concurrente, como los bloqueos de sincronización, las colecciones especiales adoptadas para la modificación concurrente, la modificación de la colección clonada desde el inicio, etc.

Sub Colecciones

Lista de sublistas (int fromIndex, int toIndex)

Aquí fromIndex es inclusivo y toIndex es exclusivo.

```
List list = new ArrayList();
List list1 = list.subList(fromIndex,toIndex);
```

1. Si la lista no existe en el rango de dar, lanza la excepción IndexOutofBoundException.
2. Los cambios realizados en la lista1 afectarán a los mismos cambios en la lista. Esto se denomina colecciones respaldadas.
3. Si el fromnIndex es mayor que el toIndex (fromIndex> toIndex) lanza la excepción IllegalArgumentException.

Ejemplo:

```
List<String> list = new ArrayList<String>();
List<String> list = new ArrayList<String>();
list.add("Hello1");
list.add("Hello2");
System.out.println("Before Sublist "+list);
List<String> list2 = list.subList(0, 1);
list2.add("Hello3");
System.out.println("After sublist changes "+list);
```

Salida:
Antes de Sublist [Hello1, Hello2]
Después de cambios de lista [Hello1, Hello3, Hello2]

Establecer subSet (fromIndex, toIndex)

Aquí fromIndex es inclusivo y toIndex es exclusivo.

```
Set set = new TreeSet();
Set set1 = set.subSet(fromIndex,toIndex);
```

El conjunto devuelto lanzará una IllegalArgumentException en un intento de insertar un elemento fuera de su rango.

Mapa de mapa secundario (fromKey, toKey)

fromKey es inclusivo y toKey es exclusivo

```
Map map = new TreeMap();
Map map1 = map.get(fromKey,toKey);
```

Si fromKey es mayor que toKey o si este mapa en sí tiene un rango restringido, y fromKey o toKey se encuentra fuera de los límites del rango, arroja IllegalArgumentException.

Todas las colecciones admiten colecciones respaldadas significa que los cambios realizados en la sub colección tendrán el mismo cambio en la colección principal.

Lea Colecciones en línea: https://riptutorial.com/es/java/topic/90/colecciones

Capítulo 41: Colecciones alternativas

Observaciones

Este tema sobre las colecciones de Java de guava, apache, eclipse: Multiset, Bag, Multimap, utils funciona desde esta biblioteca y así sucesivamente.

Examples

Apache HashBag, Guava HashMultiset y Eclipse HashBag

Una Bolsa / ultiset almacena cada objeto en la colección junto con un conteo de ocurrencias. Los métodos adicionales en la interfaz permiten agregar o eliminar varias copias de un objeto a la vez. JDK analógico es HashMap <T, Integer>, cuando los valores son el número de copias de esta clave.

Tipo	Guayaba	Colecciones Apache Commons	Colecciones GS	JDK
Orden no definida	HashMultiset	HashBag	HashBag	HashMa
Ordenados	TreeMultiset	TreeBag	TreeBag	TreeMa
Orden de insercion	LinkedHashMultiset	-	-	LinkedH
Variante concurrente	ConcurrentHashMultiset	Bolsa sincronizada	Bolsa sincronizada	`Collecti`
Concurrentes y ordenados	-	SynchronizedSortedBag	SynchronizedSortedBag	`Collecti`
Colección inmutable	ImmutableMultiset	UnmodifiableBag	UnmodifiableBag	`Collecti`
Inmutable y ordenado.	InmutableSortedMultiset	UnmodifiableSortedBag	UnmodifiableSortedBag	`Collecti`)

Ejemplos :

1. Usando SynchronizedSortedBag de Apache :

```
    // Parse text to separate words
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
```

```
    // Create Multiset
    Bag bag = SynchronizedSortedBag.synchronizedBag(new
TreeBag(Arrays.asList(INPUT_TEXT.split(" "))));

    // Print count words
    System.out.println(bag); // print [1:All!,2:Hello,1:Hi,2:World!]- in natural (alphabet)
order
    // Print all unique words
    System.out.println(bag.uniqueSet());    // print [All!, Hello, Hi, World!]- in natural
(alphabet) order


    // Print count occurrences of words
    System.out.println("Hello = " + bag.getCount("Hello"));    // print 2
    System.out.println("World = " + bag.getCount("World!"));    // print 2
    System.out.println("All = " + bag.getCount("All!"));    // print 1
    System.out.println("Hi = " + bag.getCount("Hi"));    // print 1
    System.out.println("Empty = " + bag.getCount("Empty"));    // print 0

    // Print count all words
    System.out.println(bag.size());    //print 6

    // Print count unique words
    System.out.println(bag.uniqueSet().size());    //print 4
```

2. Usando TreeBag de Eclipse (GC) :

```
    // Parse text to separate words
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
    // Create Multiset
    MutableSortedBag<String> bag =  TreeBag.newBag(Arrays.asList(INPUT_TEXT.split(" ")));

    // Print count words
    System.out.println(bag); // print [All!, Hello, Hello, Hi, World!, World!]- in natural
order
    // Print all unique words
    System.out.println(bag.toSortedSet());    // print [All!, Hello, Hi, World!]- in natural
order

    // Print count occurrences of words
    System.out.println("Hello = " + bag.occurrencesOf("Hello"));    // print 2
    System.out.println("World = " + bag.occurrencesOf("World!"));    // print 2
    System.out.println("All = " + bag.occurrencesOf("All!"));    // print 1
    System.out.println("Hi = " + bag.occurrencesOf("Hi"));    // print 1
    System.out.println("Empty = " + bag.occurrencesOf("Empty"));    // print 0

    // Print count all words
    System.out.println(bag.size());    //print 6

    // Print count unique words
    System.out.println(bag.toSet().size());    //print 4
```

3. Usando LinkedHashMultiset desde Guava :

```
    // Parse text to separate words
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
    // Create Multiset
```

```
    Multiset<String> multiset = LinkedHashMultiset.create(Arrays.asList(INPUT_TEXT.split(" ")));

    // Print count words
    System.out.println(multiset); // print [Hello x 2, World! x 2, All!, Hi]- in predictable iteration order
    // Print all unique words
    System.out.println(multiset.elementSet());    // print [Hello, World!, All!, Hi] - in predictable iteration order

    // Print count occurrences of words
    System.out.println("Hello = " + multiset.count("Hello"));    // print 2
    System.out.println("World = " + multiset.count("World!"));    // print 2
    System.out.println("All = " + multiset.count("All!"));    // print 1
    System.out.println("Hi = " + multiset.count("Hi"));    // print 1
    System.out.println("Empty = " + multiset.count("Empty"));    // print 0

    // Print count all words
    System.out.println(multiset.size());    //print 6

    // Print count unique words
    System.out.println(multiset.elementSet().size());    //print 4
```

Más ejemplos:

I. Colección Apache:

1. HashBag - orden no definida
2. SynchronizedBag - concurrente y orden no definido
3. SynchronizedSortedBag - - orden concurrente y ordenado
4. TreeBag - ordenado

II. Colección GS / Eclipse

5. MutableBag - orden no definida
6. MutableSortedBag - ordenado

III. Guayaba

7. HashMultiset - orden no definida
8. TreeMultiset - ordenado
9. LinkedHashMultiset - orden de inserción
10. ConcurrentHashMultiset - concurrente y orden no definido

Multimap en colecciones de guayaba, apache y eclipse

Este multimapa permite duplicar pares clave-valor. Los análogos de JDK son HashMap <K, List>, HashMap <K, Set> y así sucesivamente.

Orden de llaves	Orden del valor	Duplicar	Tecla analógica	Valor analógico	Guayaba	apache
no definida	Orden de insercion	sí	HashMap	Lista de arreglo	ArrayListMultimap	`MultiValu`
no definida	no definida	no	HashMap	HashSet	HashMultimap	`MultiValu multiValu HashMap<K HashSet.c`
no definida	ordenado	no	HashMap	TreeSet	`Multimaps. newMultimap( HashMap, Supplier <TreeSet>)`	`MultiValu new HashM TreeSet.c`
Orden de insercion	Orden de insercion	sí	LinkedHashMap	Lista de arreglo	LinkedListMultimap	MultiValu multiValu LinkedHa (), ArrayL
Orden de insercion	Orden de insercion	no	LinkedHashMap	LinkedHashSet	LinkedHashMultimap	`MultiValu multiValu LinkedHas LinkedHas`
ordenado	ordenado	no	TreeMap	TreeSet	TreeMultimap	`MultiValu multiValu TreeMap<K Set>(),Tr`

Ejemplos usando Multimap

Tarea : Analizar "¡Hola mundo! ¡Hola a todos! ¡Hola mundo!" cadena para separar palabras e imprimir todos los índices de cada palabra usando MultiMap (por ejemplo, Hello = [0, 2], World! = [1, 5] y así sucesivamente)

1. MultiValueMap de Apache

```
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
    // Parse text to words and index
    List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
    // Create Multimap
    MultiMap<String, Integer> multiMap = new MultiValueMap<String, Integer>();


    // Fill Multimap
    int i = 0;
    for(String word: words) {
        multiMap.put(word, i);
        i++;
    }

    // Print all words
    System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} -
in random orders
    // Print all unique words
    System.out.println(multiMap.keySet());    // print [Hi, Hello, World!, All!] - in random
```

```
orders

    // Print all indexes
    System.out.println("Hello = " + multiMap.get("Hello"));    // print [0, 2]
    System.out.println("World = " + multiMap.get("World!"));    // print [1, 5]
    System.out.println("All = " + multiMap.get("All!"));    // print [3]
    System.out.println("Hi = " + multiMap.get("Hi"));    // print [4]
    System.out.println("Empty = " + multiMap.get("Empty"));    // print null

    // Print count unique words
    System.out.println(multiMap.keySet().size());    //print 4
```

2. HashBiMap de GS / Eclipse Collection

```
    String[] englishWords = {"one", "two", "three","ball","snow"};
    String[] russianWords = {"jeden", "dwa", "trzy", "kula", "snieg"};

    // Create Multiset
    MutableBiMap<String, String> biMap = new HashBiMap(englishWords.length);
    // Create English-Polish dictionary
    int i = 0;
    for(String englishWord: englishWords) {
        biMap.put(englishWord, russianWords[i]);
        i++;
    }

    // Print count words
    System.out.println(biMap); // print {two=dwa, ball=kula, one=jeden, snow=snieg,
three=trzy} - in random orders
    // Print all unique words
    System.out.println(biMap.keySet());    // print [snow, two, one, three, ball] - in random
orders
    System.out.println(biMap.values());    // print [dwa, kula, jeden, snieg, trzy] - in
random orders

    // Print translate by words
    System.out.println("one = " + biMap.get("one"));    // print one = jeden
    System.out.println("two = " + biMap.get("two"));    // print two = dwa
    System.out.println("kula = " + biMap.inverse().get("kula"));    // print kula = ball
    System.out.println("snieg = " + biMap.inverse().get("snieg"));    // print snieg = snow
    System.out.println("empty = " + biMap.get("empty"));    // print empty = null

    // Print count word's pair
    System.out.println(biMap.size());    //print 5
```

3. HashMultiMap de guayaba

```
String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// Parse text to words and index
List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
// Create Multimap
Multimap<String, Integer> multiMap = HashMultimap.create();

// Fill Multimap
int i = 0;
for(String word: words) {
    multiMap.put(word, i);
    i++;
```

```
 }

 // Print all words
 System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} -
keys and values in random orders
 // Print all unique words
 System.out.println(multiMap.keySet());    // print [Hi, Hello, World!, All!] - in random
orders

 // Print all indexes
 System.out.println("Hello = " + multiMap.get("Hello"));    // print [0, 2]
 System.out.println("World = " + multiMap.get("World!"));    // print [1, 5]
 System.out.println("All = " + multiMap.get("All!"));    // print [3]
 System.out.println("Hi = " + multiMap.get("Hi"));    // print [4]
 System.out.println("Empty = " + multiMap.get("Empty"));    // print []

 // Print count all words
 System.out.println(multiMap.size());    //print 6

 // Print count unique words
 System.out.println(multiMap.keySet().size());    //print 4
```

Más ejemplos:

I. Colección Apache:

1. MultiValueMap
2. MultiValueMapLinked
3. MultiValueMapTree

II. Colección GS / Eclipse

1. FastListMultimap
2. HashBagMultimap
3. TreeSortedSetMultimap
4. UnifiedSetMultimap

III. Guayaba

1. HashMultiMap
2. LinkedHashMultimap
3. LinkedListMultimap
4. TreeMultimap
5. ArrayListMultimap

Comparar operación con colecciones - Crear colecciones

Comparar operación con colecciones - Crear colecciones

1. Crear lista

Descripción	JDK	guayaba	gs-colecciones
Crear lista vacía	`new ArrayList<>()`	`Lists.newArrayList()`	`FastList.newList()`
Crear lista a partir de valores.	`Arrays.asList("1", "2", "3")`	`Lists.newArrayList("1", "2", "3")`	`FastList.newListWith("1", "2", "3")`
Crear lista con capacidad = 100	`new ArrayList<>(100)`	`Lists.newArrayListWithCapacity(100)`	`FastList.newList(100)`
Crear una lista de cualquier colección	`new ArrayList<>(collection)`	`Lists.newArrayList(collection)`	`FastList.newList(collecti`
Crear lista desde cualquier iterable.	-	`Lists.newArrayList(iterable)`	`FastList.newList(iterable`
Crear lista de iterador	-	`Lists.newArrayList(iterator)`	-
Crear lista de la matriz	`Arrays.asList(array)`	`Lists.newArrayList(array)`	`FastList.newListWith(arra`
Crear lista usando la fábrica	-	-	`FastList.newWithNValues(1 () -> "1")`

Ejemplos:

```
System.out.println("createArrayList start");
// Create empty list
List<String> emptyGuava = Lists.newArrayList(); // using guava
List<String> emptyJDK = new ArrayList<>(); // using JDK
MutableList<String>  emptyGS = FastList.newList(); // using gs

// Create list with 100 element
List < String > exactly100 = Lists.newArrayListWithCapacity(100); // using guava
List<String> exactly100JDK = new ArrayList<>(100); // using JDK
MutableList<String>  empty100GS = FastList.newList(100); // using gs

// Create list with about 100 element
List<String> approx100 = Lists.newArrayListWithExpectedSize(100); // using guava
List<String> approx100JDK = new ArrayList<>(115); // using JDK
MutableList<String>  approx100GS = FastList.newList(115); // using gs
```

```
    // Create list with some elements
    List<String> withElements = Lists.newArrayList("alpha", "beta", "gamma"); // using guava
    List<String> withElementsJDK = Arrays.asList("alpha", "beta", "gamma"); // using JDK
    MutableList<String>  withElementsGS = FastList.newListWith("alpha", "beta", "gamma"); //
using gs

    System.out.println(withElements);
    System.out.println(withElementsJDK);
    System.out.println(withElementsGS);

    // Create list from any Iterable interface (any collection)
    Collection<String> collection = new HashSet<>(3);
    collection.add("1");
    collection.add("2");
    collection.add("3");

    List<String> fromIterable = Lists.newArrayList(collection); // using guava
    List<String> fromIterableJDK = new ArrayList<>(collection); // using JDK
    MutableList<String>  fromIterableGS = FastList.newList(collection); // using gs

    System.out.println(fromIterable);
    System.out.println(fromIterableJDK);
    System.out.println(fromIterableGS);
    /* Attention: JDK create list only from Collection, but guava and gs can create list from
Iterable and Collection */

    // Create list from any Iterator
    Iterator<String> iterator = collection.iterator();
    List<String> fromIterator = Lists.newArrayList(iterator); // using guava
    System.out.println(fromIterator);

    // Create list from any array
    String[] array = {"4", "5", "6"};
    List<String> fromArray = Lists.newArrayList(array); // using guava
    List<String> fromArrayJDK = Arrays.asList(array); // using JDK
    MutableList<String>  fromArrayGS = FastList.newListWith(array); // using gs
    System.out.println(fromArray);
    System.out.println(fromArrayJDK);
    System.out.println(fromArrayGS);

    // Create list using fabric
    MutableList<String>  fromFabricGS = FastList.newWithNValues(10, () ->
String.valueOf(Math.random())); // using gs
    System.out.println(fromFabricGS);

    System.out.println("createArrayList end");
```

2 Crear Set

Descripción	JDK	guayaba	gs-colecciones
Crear conjunto vacío	`new HashSet<>()`	`Sets.newHashSet()`	`UnifiedSet.newSet()`
Conjunto de valores a partir de.	`new HashSet<>(Arrays.asList("alpha", "beta", "gamma")`)	`Sets.newHashSet("alpha", "beta", "gamma")`	`UnifiedSet.newSetWith("a` `"beta", "gamma")`

Descripción	JDK	guayaba	gs-colecciones
Crear conjunto a partir de cualquier colección.	`new HashSet<>(collection)`	`Sets.newHashSet(collection)`	`UnifiedSet.newSet(colle`
Crear set desde cualquier iterable.	-	`Sets.newHashSet(iterable)`	`UnifiedSet.newSet(itera`
Crear conjunto desde cualquier iterador	-	`Sets.newHashSet(iterator)`	-
Crear conjunto desde Array	`new HashSet<>(Arrays.asList(array))`	`Sets.newHashSet(array)`	`UnifiedSet.newSetWith(a`

Ejemplos:

```
    System.out.println("createHashSet start");
    // Create empty set
    Set<String> emptyGuava = Sets.newHashSet(); // using guava
    Set<String> emptyJDK = new HashSet<>(); // using JDK
    Set<String> emptyGS = UnifiedSet.newSet(); // using gs

    // Create set with 100 element
    Set<String> approx100 = Sets.newHashSetWithExpectedSize(100); // using guava
    Set<String> approx100JDK = new HashSet<>(130); // using JDK
    Set<String> approx100GS = UnifiedSet.newSet(130); // using gs

    // Create set from some elements
    Set<String> withElements =  Sets.newHashSet("alpha", "beta", "gamma"); // using guava
    Set<String> withElementsJDK = new HashSet<>(Arrays.asList("alpha", "beta", "gamma")); //
using JDK
    Set<String> withElementsGS  = UnifiedSet.newSetWith("alpha", "beta", "gamma"); // using gs

    System.out.println(withElements);
    System.out.println(withElementsJDK);
    System.out.println(withElementsGS);

    // Create set from any Iterable interface (any collection)
    Collection<String> collection = new ArrayList<>(3);
    collection.add("1");
    collection.add("2");
    collection.add("3");

    Set<String> fromIterable = Sets.newHashSet(collection); // using guava
    Set<String> fromIterableJDK = new HashSet<>(collection); // using JDK
```

```
    Set<String> fromIterableGS  = UnifiedSet.newSet(collection); // using gs

    System.out.println(fromIterable);
    System.out.println(fromIterableJDK);
    System.out.println(fromIterableGS);
    /* Attention: JDK create set only from Collection, but guava and gs can create set from
Iterable and Collection */

    // Create set from any Iterator
    Iterator<String> iterator = collection.iterator();
    Set<String> fromIterator = Sets.newHashSet(iterator); // using guava
    System.out.println(fromIterator);

    // Create set from any array
    String[] array = {"4", "5", "6"};
    Set<String> fromArray = Sets.newHashSet(array); // using guava
    Set<String> fromArrayJDK = new HashSet<>(Arrays.asList(array)); // using JDK
    Set<String> fromArrayGS  = UnifiedSet.newSetWith(array); // using gs
    System.out.println(fromArray);
    System.out.println(fromArrayJDK);
    System.out.println(fromArrayGS);

    System.out.println("createHashSet end");
```

3 Crear Mapa

Descripción	JDK	guayaba	gs-colecciones
Crear mapa vacío	`new HashMap<>()`	`Maps.newHashMap()`	`UnifiedMap.newMap()`
Crear mapa con capacidad = 130	`new HashMap<>(130)`	`Maps.newHashMapWithExpectedSize(100)`	`UnifiedMap.newMap(130)`
Crear mapa desde otro mapa	`new HashMap<>(map)`	`Maps.newHashMap(map)`	`UnifiedMap.newMap(map)`
Crear mapa desde las teclas	-	-	`UnifiedMap.newWithKeysValues("1", "a", "2", "b")`

Ejemplos:

```
    System.out.println("createHashMap start");
    // Create empty map
    Map<String, String> emptyGuava = Maps.newHashMap(); // using guava
    Map<String, String> emptyJDK = new HashMap<>(); // using JDK
    Map<String, String> emptyGS = UnifiedMap.newMap(); // using gs

    // Create map with about 100 element
    Map<String, String> approx100 = Maps.newHashMapWithExpectedSize(100); // using guava
    Map<String, String> approx100JDK = new HashMap<>(130); // using JDK
```

```
    Map<String, String> approx100GS = UnifiedMap.newMap(130); // using gs

    // Create map from another map
    Map<String, String> map = new HashMap<>(3);
    map.put("k1","v1");
    map.put("k2","v2");
    Map<String, String> withMap =  Maps.newHashMap(map); // using guava
    Map<String, String> withMapJDK = new HashMap<>(map); // using JDK
    Map<String, String> withMapGS = UnifiedMap.newMap(map); // using gs

    System.out.println(withMap);
    System.out.println(withMapJDK);
    System.out.println(withMapGS);

    // Create map from keys
    Map<String, String> withKeys =  UnifiedMap.newWithKeysValues("1", "a", "2", "b");
    System.out.println(withKeys);

    System.out.println("createHashMap end");
```

Más ejemplos: CreateCollectionTest

1. ColecciónComparar
2. ColecciónBúsqueda
3. JavaTransform

Lea Colecciones alternativas en línea: https://riptutorial.com/es/java/topic/2958/colecciones-alternativas

Capítulo 42: Colecciones concurrentes

Introducción

Una *colección concurrente* es una [colección] [1] que permite el acceso por más de un hilo al mismo tiempo. Normalmente, diferentes hilos pueden recorrer el contenido de la colección y agregar o eliminar elementos. La colección es responsable de garantizar que la colección no se corrompa. [1]: http://stackoverflow.com/documentation/java/90/collections#t=201612221936497298484

Examples

Colecciones a prueba de hilos

De forma predeterminada, los distintos tipos de Colección no son seguros para subprocesos.

Sin embargo, es bastante fácil hacer una colección segura para subprocesos.

```
List<String> threadSafeList = Collections.synchronizedList(new ArrayList<String>());
Set<String> threadSafeSet = Collections.synchronizedSet(new HashSet<String>());
Map<String, String> threadSafeMap = Collections.synchronizedMap(new HashMap<String,
String>());
```

Cuando realice una colección segura para subprocesos, nunca debe acceder a ella a través de la colección original, solo a través de la envoltura segura para hilos.

Java SE 5

A partir de Java 5, `java.util.collections` tiene varias colecciones nuevas seguras para subprocesos que no necesitan los diversos métodos de `Collections.synchronized` .

```
List<String> threadSafeList = new CopyOnWriteArrayList<String>();
Set<String> threadSafeSet = new ConcurrentHashSet<String>();
Map<String, String> threadSafeMap = new ConcurrentHashMap<String, String>();
```

Colecciones concurrentes

Las colecciones concurrentes son una generalización de las colecciones seguras para subprocesos, que permiten un uso más amplio en un entorno concurrente.

Si bien las colecciones seguras para subprocesos tienen la adición o eliminación segura de elementos de varios subprocesos, no necesariamente tienen una iteración segura en el mismo contexto (es posible que uno no pueda recorrer la colección en un subproceso en una secuencia, mientras que otro lo modifica agregando quitando elementos).

Aquí es donde se utilizan las colecciones concurrentes.

Como la iteración suele ser la implementación básica de varios métodos masivos en colecciones, como `addAll` , `removeAll` , o también la copia de colecciones (a través de un constructor u otros medios), clasificación, ... el caso de uso de colecciones concurrentes es en realidad bastante grande.

Por ejemplo, Java SE 5 `java.util.concurrent.CopyOnWriteArrayList` es una implementación de `Lis` t segura para subprocesos, su javadoc afirma:

> El método del iterador de estilo "instantánea" utiliza una referencia al estado de la matriz en el punto en que se creó el iterador. Esta matriz nunca cambia durante la vida útil del iterador, por lo que la interferencia es imposible y se garantiza que el iterador no lanzará ConcurrentModificationException.

Por lo tanto, el siguiente código es seguro:

```
public class ThreadSafeAndConcurrent {

public static final List<Integer> LIST = new CopyOnWriteArrayList<>();

public static void main(String[] args) throws InterruptedException {
    Thread modifier = new Thread(new ModifierRunnable());
    Thread iterator = new Thread(new IteratorRunnable());
    modifier.start();
    iterator.start();
    modifier.join();
    iterator.join();
}

public static final class ModifierRunnable implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 0; i < 50000; i++) {
                LIST.add(i);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public static final class IteratorRunnable implements Runnable {
    @Override
    public void run() {
        try {
            for (int i = 0; i < 10000; i++) {
                long total = 0;
                for(Integer inList : LIST) {
                    total += inList;
                }
                System.out.println(total);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
}
```

Otra colección concurrente con respecto a la iteración es `ConcurrentLinkedQueue` , que establece:

> Los iteradores son débilmente consistentes, devolviendo elementos que reflejan el estado de la cola en algún momento en o desde la creación del iterador. No lanzan java.util.ConcurrentModificationException, y pueden proceder simultáneamente con otras operaciones. Los elementos contenidos en la cola desde la creación del iterador se devolverán exactamente una vez.

Uno debe revisar los javadocs para ver si una colección es concurrente o no. Los atributos del iterador devueltos por el método `iterator()` ("falla rápidamente", "débilmente consistente", ...) es el atributo más importante que debe buscarse.

Hilo seguro pero ejemplos no concurrentes

En el código anterior, cambiando la declaración `LIST` a

```
public static final List<Integer> LIST = Collections.synchronizedList(new ArrayList<>());
```

Podría (y estadísticamente lo hará en la mayoría de las arquitecturas de CPU / núcleo con múltiples CPU) dar lugar a excepciones.

Las colecciones sincronizadas de los métodos de utilidad de `Collections` son seguras para subprocesos para la adición / eliminación de elementos, pero no la iteración (a menos que la colección subyacente que se le pasa ya lo sea).

Inserción en el mapa de hash concurrente

```
public class InsertIntoConcurrentHashMap
{

    public static void main(String[] args)
    {
        ConcurrentHashMap<Integer, SomeObject> concurrentHashMap = new ConcurrentHashMap<>();

        SomeObject value = new SomeObject();
        Integer key = 1;

        SomeObject previousValue = concurrentHashMap.putIfAbsent(1, value);
        if (previousValue != null)
        {
            //Then some other value was mapped to key = 1. 'value' that was passed to
            //putIfAbsent method is NOT inserted, hence, any other thread which calls
            //concurrentHashMap.get(1) would NOT receive a reference to the 'value'
            //that your thread attempted to insert. Decide how you wish to handle
            //this situation.
        }

       else
       {
```

```
            //'value' reference is mapped to key = 1.
        }
    }
}
```

Lea Colecciones concurrentes en línea: https://riptutorial.com/es/java/topic/8363/colecciones-concurrentes

Capítulo 43: Comandos de tiempo de ejecución

Examples

Añadiendo ganchos de apagado

A veces, se necesita un fragmento de código para ejecutarse cuando el programa se detiene, como liberar los recursos del sistema que abre. Puede hacer que un hilo se ejecute cuando el programa se detiene con el método `addShutdownHook` :

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    ImportantStuff.someImportantIOStream.close();
}));
```

Lea Comandos de tiempo de ejecución en línea:
https://riptutorial.com/es/java/topic/7304/comandos-de-tiempo-de-ejecucion

Capítulo 44: Comparable y comparador

Sintaxis

- clase pública implementa MyClass Comparable `<MyClass` >
- la clase pública MyComparator implementa Comparator `<SomeOtherClass` >
- public int compareTo (MyClass other)
- public int compare (SomeOtherClass o1, SomeOtherClass o2)

Observaciones

Al implementar un `compareTo(..)` que depende de un `double` , **no** haga lo siguiente:

```
public int comareTo(MyClass other) {
    return (int)(doubleField - other.doubleField); //THIS IS BAD
}
```

El truncamiento causado por `(int)` cast causará que el método a veces devuelva incorrectamente `0` lugar de un número positivo o negativo, y por lo tanto puede llevar a errores de comparación y clasificación.

En su lugar, la implementación correcta más simple es usar Double.compare , como tal:

```
public int comareTo(MyClass other) {
    return Double.compare(doubleField,other.doubleField); //THIS IS GOOD
}
```

Una versión no genérica de `Comparable<T>` , simplemente `Comparable` , ha existido desde Java 1.2 . Además de interactuar con el código heredado, siempre es mejor implementar la versión genérica `Comparable<T>` , ya que no requiere conversión en la comparación.

Es muy estándar que una clase sea comparable a sí misma, como en:

```
public class A implements Comparable<A>
```

Si bien es posible romper con este paradigma, tenga cuidado al hacerlo.

Un `Comparator<T>` todavía se puede usar en instancias de una clase si esa clase implementa `Comparable<T>` . En este caso, se utilizará la lógica del `Comparator` ; El orden natural especificado por la implementación `Comparable` será ignorado.

Examples

Ordenar una lista usando Comparable o un comparador

Digamos que estamos trabajando en una clase que representa a una persona por su nombre y apellido. Hemos creado una clase básica para hacer esto e implementado los métodos `equals` y `hashCode` adecuados.

```
public class Person {

    private final String lastName; //invariant - nonnull
    private final String firstName; //invariant - nonnull

    public Person(String firstName, String lastName){
        this.firstName = firstName != null ? firstName : "";
        this.lastName = lastName != null ? lastName : "";
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String toString() {
        return lastName + ", " + firstName;
    }

    @Override
    public boolean equals(Object o) {
        if (! (o instanceof Person)) return false;
        Person p = (Person)o;
        return firstName.equals(p.firstName) && lastName.equals(p.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }
}
```

Ahora nos gustaría ordenar una lista de objetos `Person` por su nombre, como en el siguiente escenario:

```
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));
    Collections.sort(people); //This currently won't work.
}
```

Desafortunadamente, como está marcado, lo anterior actualmente no compilará. `Collections.sort(..)` solo sabe cómo ordenar una lista si los elementos de esa lista son comparables o si se proporciona un método personalizado de comparación.

Si le pidieran que clasificara la siguiente lista: `1,3,5,4,2` , no tendría ningún problema en decir que la respuesta es `1,2,3,4,5` . Esto se debe a que los enteros (tanto en Java como matemáticamente) tienen un *orden natural* , un orden de comparación estándar predeterminado. Para dar a nuestra clase Person un orden natural, implementamos la `Comparable<Person>` , que requiere la implementación del método `compareTo(Person p)`:

```
public class Person implements Comparable<Person> {

    private final String lastName; //invariant - nonnull
    private final String firstName; //invariant - nonnull

    public Person(String firstName, String lastName) {
        this.firstName = firstName != null ? firstName : "";
        this.lastName = lastName != null ? lastName : "";
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String toString() {
        return lastName + ", " + firstName;
    }

    @Override
    public boolean equals(Object o) {
        if (! (o instanceof Person)) return false;
        Person p = (Person)o;
        return firstName.equals(p.firstName) && lastName.equals(p.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }

    @Override
    public int compareTo(Person other) {
        // If this' lastName and other's lastName are not comparably equivalent,
        // Compare this to other by comparing their last names.
        // Otherwise, compare this to other by comparing their first names
        int lastNameCompare = lastName.compareTo(other.lastName);
        if (lastNameCompare != 0) {
            return lastNameCompare;
        } else {
            return firstName.compareTo(other.firstName);
        }
    }
}
```

Ahora, el método principal dado funcionará correctamente.

```
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
```

```
                                    new Person("Bob", "Dole"),
                                    new Person("Ronald", "McDonald"),
                                    new Person("Alice", "McDonald"),
                                    new Person("Jill", "Doe"));
    Collections.sort(people); //Now functions correctly

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

Sin embargo, si no desea o no puede modificar la clase `Person` , puede proporcionar un `Comparator<T>` personalizado `Comparator<T>` que maneje la comparación de cualquiera de los dos objetos `Person` . Si le pidieran que clasificara la siguiente lista: `circle, square, rectangle, triangle, hexagon` no podría, pero si le pidieran que clasificara esa lista *según el número de esquinas* , podría hacerlo. De este modo, al proporcionar un comparador se le indica a Java cómo comparar dos objetos que normalmente no son comparables.

```
public class PersonComparator implements Comparator<Person> {

    public int compare(Person p1, Person p2) {
        // If p1's lastName and p2's lastName are not comparably equivalent,
        // Compare p1 to p2 by comparing their last names.
        // Otherwise, compare p1 to p2 by comparing their first names
        if (p1.getLastName().compareTo(p2.getLastName()) != 0) {
            return p1.getLastName().compareTo(p2.getLastName());
        } else {
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
}

//Assume the first version of Person (that does not implement Comparable) is used here
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                    new Person("Bob", "Dole"),
                                    new Person("Ronald", "McDonald"),
                                    new Person("Alice", "McDonald"),
                                    new Person("Jill", "Doe"));
    Collections.sort(people); //Illegal, Person doesn't implement Comparable.
    Collections.sort(people, new PersonComparator()); //Legal

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

Los comparadores también se pueden crear / usar como una clase interna anónima

```
//Assume the first version of Person (that does not implement Comparable) is used here
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                    new Person("Bob", "Dole"),
                                    new Person("Ronald", "McDonald"),
                                    new Person("Alice", "McDonald"),
                                    new Person("Jill", "Doe"));
    Collections.sort(people); //Illegal, Person doesn't implement Comparable.

    Collections.sort(people, new PersonComparator()); //Legal
```

```
    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald

    //Anonymous Class
    Collections.sort(people, new Comparator<Person>() { //Legal
        public int compare(Person p1, Person p2) {
            //Method code...
        }
    });
}
```

Java SE 8

Comparadores basados en expresiones Lambda

A partir de Java 8, los comparadores también se pueden expresar como expresiones lambda

```
    //Lambda
    Collections.sort(people, (p1, p2) -> { //Legal
        //Method code....
    });
```

Métodos predeterminados del comparador

Por otra parte, existen métodos predeterminados interesantes en la interfaz de comparación para la construcción de comparadores: la siguiente construye un comparador compara por `lastName` y luego `firstName` .

```
Collections.sort(people, Comparator.comparing(Person::getLastName)
                                .thenComparing(Person::getFirstName));
```

Invertir el orden de un comparador

Cualquier comparador también puede revertirse fácilmente usando el método `reversedMethod` que cambiará el orden ascendente a descendente.

Los métodos de comparar y comparar

La interfaz `Comparable<T>` requiere un método:

```
public interface Comparable<T> {

    public int compareTo(T other);
```

```
}
```

Y la interfaz del `Comparator<T>` requiere un método:

```
public interface Comparator<T> {

    public int compare(T t1, T t2);

}
```

Estos dos métodos hacen esencialmente lo mismo, con una pequeña diferencia: `compareTo` compara `this` con `other` , mientras que `compare` compara `t1` con `t2` , sin preocuparse en absoluto por `this` .

Aparte de esa diferencia, los dos métodos tienen requisitos similares. Específicamente (para compareTo), compara este objeto con el objeto especificado para orden. Devuelve un entero negativo, cero o un entero positivo, ya que este objeto es menor, igual o mayor que el objeto especificado. Así, para la comparación de `a` y `b` :

- Si `a < b` , `a.compareTo(b)` y `compare(a,b)` debe devolver un entero negativo, y `b.compareTo(a)` y `compare(b,a)` deben devolver un entero positivo
- Si `a > b` , `a.compareTo(b)` y `compare(a,b)` debe devolver un entero positivo, y `b.compareTo(a)` y `compare(b,a)` deben devolver un entero negativo
- Si `a` es igual a `b` para la comparación, todas las comparaciones deben devolver `0` .

Clasificación natural (comparable) vs explícita (comparativa)

Hay dos métodos `Collections.sort()` :

- Uno que toma una `List<T>` como parámetro donde `T` debe implementar Comparable y anular el `compareTo()` que determina el orden de clasificación.
- Uno que toma una Lista y un Comparador como los argumentos, donde el Comparador determina el orden de clasificación.

Primero, aquí hay una clase de persona que implementa Comparable:

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
```

```
    @Override
    public int compareTo(Person o) {
        return this.getAge() - o.getAge();
    }
    @Override
    public String toString() {
        return this.getAge()+"-"+this.getName();
    }

}
```

Aquí es cómo usaría la clase anterior para ordenar una Lista en el orden natural de sus elementos, definido por la anulación del método `compareTo()` :

```
//-- usage
List<Person> pList = new ArrayList<Person>();
            Person p = new Person();
            p.setName("A");
            p.setAge(10);
            pList.add(p);
            p = new Person();
            p.setName("Z");
            p.setAge(20);
            pList.add(p);
            p = new Person();
            p.setName("D");
            p.setAge(30);
            pList.add(p);

            //-- natural sorting i.e comes with object implementation, by age
            Collections.sort(pList);

            System.out.println(pList);
```

Aquí es cómo usaría un Comparador en línea anónimo para ordenar una Lista que no implementa Comparable, o en este caso, para ordenar una Lista en un orden diferente al orden natural:

```
            //-- explicit sorting, define sort on another property here goes with name
            Collections.sort(pList, new Comparator<Person>() {

                @Override
                public int compare(Person o1, Person o2) {
                    return o1.getName().compareTo(o2.getName());
                }
            });
            System.out.println(pList);
```

Ordenar entradas de mapa

A partir de Java 8, hay métodos predeterminados en la interfaz `Map.Entry` para permitir la clasificación de iteraciones de mapas.

Java SE 8

```
Map<String, Integer> numberOfEmployees = new HashMap<>();
numberOfEmployees.put("executives", 10);
numberOfEmployees.put("human ressources", 32);
numberOfEmployees.put("accounting", 12);
numberOfEmployees.put("IT", 100);

// Output the smallest departement in terms of number of employees
numberOfEmployees.entrySet().stream()
    .sorted(Map.Entry.comparingByValue())
    .limit(1)
    .forEach(System.out::println);   // outputs : executives=10
```

Por supuesto, estos también se pueden utilizar fuera de la api de flujo:

Java SE 8

```
List<Map.Entry<String, Integer>> entries = new ArrayList<>(numberOfEmployees.entrySet());
Collections.sort(entries, Map.Entry.comparingByValue());
```

Creando un comparador usando el método de comparación

```
Comparator.comparing(Person::getName)
```

Esto crea un comparador para la clase `Person` que usa este nombre de persona como fuente de comparación. También es posible usar la versión del método para comparar long, int y double. Por ejemplo:

```
Comparator.comparingInt(Person::getAge)
```

Orden invertida

Para crear un comparador que imponga el orden inverso, use el método `reversed()` :

```
Comparator.comparing(Person::getName).reversed()
```

Cadena de comparadores

```
Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstName)
```

Esto creará un comparador que se compara con el apellido y luego con el nombre. Puedes encadenar tantos comparadores como quieras.

Lea Comparable y comparador en línea: https://riptutorial.com/es/java/topic/3137/comparable-y-comparador

Capítulo 45: Comparación de C ++

Introducción

Java y C ++ son lenguajes similares. Este tema sirve como una guía de referencia rápida para los ingenieros de Java y C ++.

Observaciones

Clases definidas dentro de otras construcciones

Definido dentro de otra clase

C ++

Clase anidada [ref] (necesita una referencia para encerrar clase)

```
class Outer {
   class Inner {
      public:
         Inner(Outer* o) :outer(o) {}

      private:
         Outer*  outer;
   };
};
```

Java

Clase anidada [no estática] (clase interna o clase miembro)

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Definido estáticamente dentro de otra clase

C ++

Clase anidada estática

```
class Outer {
   class Inner {
      ...
   };
};
```

Java

Clase anidada estática (también conocida como clase de miembro estática) [ref]

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

Definido dentro de un método

(por ejemplo, manejo de eventos)

C ++

Clase local [ref]

```
void fun() {
   class Test {
      /* members of Test class */
   };
}
```

Ver también expresiones Lambda.

Java

Clase local [ref]

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
            }
        }).start();
```

```
    }
}
```

Anulando vs Sobrecarga

Los siguientes puntos de Anulación frente a Sobrecarga se aplican tanto a C ++ como a Java:

- Un método anulado tiene el mismo nombre y argumentos que su método base.
- Un método sobrecargado tiene el mismo nombre pero diferentes argumentos y no se basa en la herencia.
- Dos métodos con el mismo nombre y argumentos pero diferente tipo de devolución son ilegales. Consulte las preguntas relacionadas sobre Stackoverflow sobre "sobrecarga con diferentes tipos de retorno en Java" - Pregunta 1 ; Pregunta 2

Polimorfismo

El polimorfismo es la capacidad de los objetos de diferentes clases relacionadas por herencia para responder de manera diferente a la misma llamada de método. Aquí hay un ejemplo:

- Clase base Forma con área como método abstracto
- Dos clases derivadas, Cuadrado y Círculo, implementan métodos de área.
- Forma los puntos de referencia a Plaza y área se invoca.

En C ++, el polimorfismo es habilitado por métodos virtuales. En Java, los métodos son virtuales por defecto.

Orden de Construcción / Destrucción

Limpieza de objetos

En C ++, es una buena idea declarar un destructor como virtual para garantizar que se llamará al destructor de la subclase si se elimina el puntero de la clase base.

En Java, un método de finalización es similar a un destructor en C ++; sin embargo, los finalizadores son impredecibles (dependen de GC). Práctica recomendada: utilice un método "cerrado" para la limpieza explícita.

```
protected void close() {
    try {
       // do subclass cleanup
    }
    finally {
       isClosed = true;
       super.close();
```

```
    }
}

protected void finalize() {
    try {
       if(!isClosed) close();
    }
    finally {
       super.finalize();
    }
}
```

Métodos y clases abstractas

Concepto	C ++	Java
Método abstracto declarado sin implementación	método virtual puro `virtual void eat(void) = 0;`	método abstracto `abstract void draw();`
Clase abstracta no puede ser instanciado	no puede ser instanciado; tiene al menos 1 método virtual puro `class AB {public: virtual void f() = 0;};`	no puede ser instanciado; puede tener métodos no abstractos `abstract class GraphicObject {}`
Interfaz campos sin instancia	ninguna palabra clave de "interfaz", pero puede imitar una interfaz de Java con instalaciones de una clase abstracta	muy similar a la clase abstracta, pero 1) admite herencia múltiple; 2) campos sin instancia `interface TestInterface {}`

Modificadores de accesibilidad

Modificador	C ++	Java
Público - accesible por todos	*sin notas especiales*	*sin notas especiales*
Protegido - accesible por subclases	también accesible por amigos	También accesible dentro del mismo paquete
Privado - accesible por los miembros	también accesible por amigos	*sin notas especiales*
defecto	la clase por defecto es privada; La estructura por defecto es pública	Accesible por todas las clases dentro del mismo

Modificador	C ++	Java
		paquete.
otro	Amigo: una forma de otorgar acceso a miembros privados y protegidos sin herencia (ver más abajo)	

Ejemplo de C ++ Friend

```
class Node {
  private:
    int key;  Node *next;
    // LinkedList::search() can access "key" & "next"
    friend int LinkedList::search();
};
```

El temido problema del diamante

> El problema del diamante es una ambigüedad que surge cuando dos clases B y C heredan de A, y la clase D hereda de B y C. Si hay un método en A que B y C se han invalidado, y D no lo reemplaza, entonces ¿Qué versión del método hereda D: la de B o la de C? (de Wikipedia)

Si bien C ++ siempre ha sido susceptible al problema diamante, Java fue susceptible hasta Java 8. Originalmente, Java no admitía herencia múltiple, pero con la llegada de los métodos de interfaz predeterminados, las clases Java no pueden heredar la "implementación" de más de una clase. .

clase java.lang.Object

En Java, todas las clases heredan, ya sea implícita o explícitamente, de la clase Object. Cualquier referencia de Java se puede convertir al tipo de objeto.

C ++ no tiene una clase de "Objeto" comparable.

Colecciones Java y Contenedores C ++

Las colecciones de Java son sinónimas con los contenedores de C ++.

Diagrama de flujo de colecciones Java

Diagrama de flujo de contenedores C ++

Tipos enteros

Bits	Min	Max	Tipo C ++ (en LLP64 o LP64)	Tipo de Java
8	-2 (8-1) = -128	2 (8-1) -1 = 127	carbonizarse	byte
8	0	2 (8) -1 = 255	sin firmar	-
dieciséis	-2 (16-1) = -32,768	2 (16-1) -1 = 32,767	corto	corto
dieciséis	0 (\ u0000)	2 (16) -1 = 65,535 (\ uFFFF)	corto sin firmar	char (sin firmar)
32	-2 (32-1) = -2.147 mil millones	2 (32-1) -1 = 2.147 millones	En t	En t
32	0	2 (32) -1 = 4.295 mil millones	sin firmar int	-
64	-2 (64-1)	2 (16-1) -1	largo*	largo largo
64	0	2 (16) -1	sin firmar largo * sin firmar mucho tiempo	-

* Win64 API es solo de 32 bits

Muchos más tipos de C ++

Examples

Miembros de la clase estática

Los miembros estáticos tienen un alcance de clase en lugar de un objeto.

Ejemplo de C ++

```
// define in header
class Singleton {
   public:
      static Singleton *getInstance();

   private:
      Singleton() {}
      static Singleton *instance;
};
```

```
// initialize in .cpp
Singleton* Singleton::instance = 0;
```

Ejemplo de Java

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Clases definidas dentro de otras construcciones

Definido dentro de otra clase

C ++

Clase anidada [ref] (necesita una referencia para encerrar clase)

```
class Outer {
   class Inner {
      public:
         Inner(Outer* o) :outer(o) {}

      private:
         Outer*  outer;
   };
};
```

Java

Clase anidada [no estática] (clase interna o clase miembro)

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Definido estáticamente dentro de otra clase

C ++

Clase anidada estática

```
class Outer {
   class Inner {
      ...
   };
};
```

Java

Clase anidada estática (también conocida como clase de miembro estática) [ref]

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

Definido dentro de un método

(por ejemplo, manejo de eventos)

C ++

Clase local [ref]

```
void fun() {
   class Test {
      /* members of Test class */
   };
}
```

Java

Clase local [ref]

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
```

```
            }
        }).start();
    }
}
```

Pasar por valor y Pasar por referencia

Muchos argumentan que Java SÓLO es paso por valor, pero tiene más matices que eso. Compare los siguientes ejemplos de C ++ y Java para ver los muchos sabores de paso por valor (también conocido como copia) y paso por referencia (también conocido como alias).

Ejemplo de C ++ (código completo)

```
  // passes a COPY of the object
  static void passByCopy(PassIt obj) {
     obj.i = 22;  // only a "local" change
  }

  // passes a pointer
  static void passByPointer(PassIt* ptr) {
     ptr->i = 33;
     ptr = 0; // better to use nullptr instead if '0'
  }

  // passes an alias (aka reference)
  static void passByAlias(PassIt& ref) {
     ref.i = 44;
  }

  // This is an old-school way of doing it.
  // Check out std::swap for the best way to do this
  static void swap(PassIt** pptr1, PassIt** pptr2) {
     PassIt* tmp = *pptr1;
     *pptr1 = *pptr2;
     *pptr2 = tmp;
  }
```

Ejemplo de Java (código completo)

```
   // passes a copy of the variable
   // NOTE: in java only primitives are pass-by-copy
   public static void passByCopy(int copy) {
      copy = 33;  // only a "local" change
   }

   // No such thing as pointers in Java
   /*
   public static void passByPointer(PassIt *ptr) {
      ptr->i = 33;
      ptr = 0; // better to use nullptr instead if '0'
   }
   */
```

```
   // passes an alias (aka reference)
   public static void passByAlias(PassIt ref) {
      ref.i = 44;
   }

   // passes aliases (aka references),
   // but need to do "manual", potentially expensive copies
   public static void swap(PassIt ref1, PassIt ref2) {
      PassIt tmp = new PassIt(ref1);
      ref1.copy(ref2);
      ref2.copy(tmp);
   }
```

Herencia vs Composición

C ++ y Java son lenguajes orientados a objetos, por lo que el siguiente diagrama se aplica a ambos.

Despreciación por desvalorización

Tenga cuidado con el uso de "downcasting" - Downcasting es derribar la jerarquía de herencia de una clase base a una subclase (es decir, opuesta al polimorfismo). En general, use polimorfismo y reemplazo en lugar de instanceof y downcasting.

Ejemplo de C ++

```
// explicit type case required
Child *pChild =  (Child *) &parent;
```

Ejemplo de Java

```
if(mySubClass instanceof SubClass) {
   SubClass mySubClass = (SubClass)someBaseClass;
   mySubClass.nonInheritedMethod();
}
```

Métodos y clases abstractas

Método abstracto

declarado sin implementación

C ++

método virtual puro

```
virtual void eat(void) = 0;
```

Java

método abstracto

```
abstract void draw();
```

Clase abstracta

no puede ser instanciado

C ++

no puede ser instanciado; tiene al menos 1 método virtual puro

```
class AB {public: virtual void f() = 0;};
```

Java

no puede ser instanciado; puede tener métodos no abstractos

```
abstract class GraphicObject {}
```

Interfaz

campos sin instancia

C ++

nada comparable a Java

Java

muy similar a la clase abstracta, pero 1) admite herencia múltiple; 2) campos sin instancia

```
interface TestInterface {}
```

Lea Comparación de C ++ en línea: https://riptutorial.com/es/java/topic/10849/comparacion-de-c-plusplus

Capítulo 46: Compilador de Java - 'javac'

Observaciones

El comando `javac` se utiliza para compilar archivos fuente de Java a archivos de código de bytes. Los archivos de bytecode son independientes de la plataforma. Esto significa que puede compilar su código en un tipo de hardware y sistema operativo, y luego ejecutar el código en cualquier otra plataforma que admita Java.

El comando `javac` se incluye en las distribuciones del Kit de desarrollo de Java (JDK).

El compilador de Java y el resto de la cadena de herramientas estándar de Java coloca las siguientes restricciones en el código:

- El código fuente se guarda en archivos con el sufijo ".java"
- Los códigos de bytes se guardan en archivos con el sufijo ".class"
- Para los archivos de código fuente y bytecode en el sistema de archivos, las rutas de los archivos deben reflejar el nombre del paquete y la clase.

Nota: `javac` compilador `javac` no debe confundirse con el compilador Just in Time (JIT) que compila los códigos de bytes en código nativo.

Examples

El comando 'javac' - empezando

Ejemplo simple

Suponiendo que el "HelloWorld.java" contiene la siguiente fuente de Java:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

(Para obtener una explicación del código anterior, consulte Cómo comenzar con el lenguaje Java).

Podemos compilar el archivo anterior usando este comando:

```
$ javac HelloWorld.java
```

Esto produce un archivo llamado "HelloWorld.class", que luego podemos ejecutar de la siguiente manera:

```
$ java HelloWorld
Hello world!
```

Los puntos clave a tener en cuenta en este ejemplo son:

1. El nombre de archivo de origen "HelloWorld.java" debe coincidir con el nombre de la clase en el archivo de origen ... que es `HelloWorld` . Si no coinciden, obtendrá un error de compilación.
2. El nombre de archivo de bytecode "HelloWorld.class" corresponde al nombre de clase. Si tuviera que cambiar el nombre de "HelloWorld.class", obtendría un error cuando intentara ejecutarlo.
3. Al ejecutar una aplicación Java utilizando `java` , debe proporcionar el nombre de clase NO el nombre de archivo de bytecode.

Ejemplo con paquetes

El código Java más práctico utiliza paquetes para organizar el espacio de nombres para las clases y reducir el riesgo de colisión accidental de nombres de clases.

Si quisiéramos declarar la clase `HelloWorld` en un paquete, llame a `com.example` , el "HelloWorld.java" contendría la siguiente fuente Java:

```
package com.example;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Este archivo de código fuente necesita almacenarse en un árbol de directorios cuya estructura corresponde a la denominación del paquete.

```
.    # the current directory (for this example)
|
 ----com
     |
      ----example
          |
           ----HelloWorld.java
```

Podemos compilar el archivo anterior usando este comando:

```
$ javac com/example/HelloWorld.java
```

Esto produce un archivo llamado "com / example / HelloWorld.class"; es decir, después de la compilación, la estructura del archivo debería verse así:

```
.    # the current directory (for this example)
|
```

```
 ----com
     |
      ----example
          |
           ----HelloWorld.java
           ----HelloWorld.class
```

Luego podemos ejecutar la aplicación de la siguiente manera:

```
$ java com.example.HelloWorld
Hello world!
```

Los puntos adicionales a tener en cuenta en este ejemplo son:

1. La estructura del directorio debe coincidir con la estructura del nombre del paquete.
2. Cuando ejecuta la clase, se debe proporcionar el nombre completo de la clase; es decir, "com.example.HelloWorld" no "HelloWorld".
3. No es necesario compilar y ejecutar código Java desde el directorio actual. Solo lo estamos haciendo aquí para ilustrar.

Compilando múltiples archivos a la vez con 'javac'.

Si su aplicación consta de varios archivos de código fuente (y la mayoría lo hace), puede compilarlos uno por uno. Alternativamente, puede compilar varios archivos al mismo tiempo enumerando las rutas de acceso:

```
$ javac Foo.java Bar.java
```

o usando la funcionalidad de comodín de nombre de archivo de su shell de comandos ....

```
$ javac *.java
$ javac com/example/*.java
$ javac */**/*.java #Only works on Zsh or with globstar enabled on your shell
```

Esto compilará todos los archivos fuente de Java en el directorio actual, en el directorio "com / example", y recursivamente en directorios secundarios respectivamente. Una tercera alternativa es proporcionar una lista de nombres de archivos de origen (y opciones del compilador) como un archivo. Por ejemplo:

```
$ javac @sourcefiles
```

donde el archivo `sourcefiles` contiene:

```
Foo.java
Bar.java
com/example/HelloWorld.java
```

Nota: compilar código como este es apropiado para proyectos pequeños de una sola persona y para programas únicos. Más allá de eso, es recomendable seleccionar y utilizar una herramienta

de compilación Java. Alternativamente, la mayoría de los programadores usan un IDE de Java (por ejemplo, NetBeans , eclipse , IntelliJ IDEA) que ofrece un compilador integrado y la construcción incremental de "proyectos".

Opciones 'javac' usadas comúnmente

Aquí hay algunas opciones para el comando `javac` que probablemente sean útiles para usted.

- La opción `-d` establece un directorio de destino para escribir los archivos ".class".
- La opción `-sourcepath` establece una ruta de búsqueda de código fuente.
- La opción `-cp` o `-classpath` establece la ruta de búsqueda para encontrar clases externas compiladas previamente. Para obtener más información sobre el classpath y cómo especificarlo, consulte el tema de Classpath .
- La opción `-version` imprime la información de la versión del compilador.

Una lista más completa de las opciones del compilador se describirá en un ejemplo separado.

Referencias

La referencia definitiva para el comando `javac` es la página de manual de Oracle para `javac` .

Compilando para una versión diferente de Java

El lenguaje de programación Java (y su tiempo de ejecución) ha sufrido numerosos cambios desde su lanzamiento desde su lanzamiento público inicial. Estos cambios incluyen:

- Cambios en la sintaxis y semántica del lenguaje de programación Java.
- Cambios en las API proporcionadas por las bibliotecas de clases estándar de Java.
- Cambios en el conjunto de instrucciones Java (bytecode) y el formato de archivo de clase.

Con muy pocas excepciones (por ejemplo, la palabra clave `enum` , los cambios en algunas clases "internas", etc.), estos cambios son compatibles con versiones anteriores.

- Un programa Java que se compiló usando una versión anterior de la cadena de herramientas de Java se ejecutará en una versión más reciente de la plataforma Java sin recompilación.
- Un programa Java que se escribió en una versión anterior de Java se compilará con éxito con un nuevo compilador de Java.

Compilando Java antiguo con un compilador más nuevo

Si necesita (volver a compilar) el código Java más antiguo en una plataforma Java más nueva para ejecutarlo en la plataforma más nueva, por lo general, no necesita dar ninguna marca de compilación especial. En algunos casos (por ejemplo, si ha usado `enum` como identificador), puede usar la opción `-source` para deshabilitar la nueva sintaxis. Por ejemplo, dada la siguiente clase:

```
public class OldSyntax {
    private static int enum;  // invalid in Java 5 or later
}
```

se requiere lo siguiente para compilar la clase utilizando un compilador Java 5 (o posterior):

```
$ javac -source 1.4 OldSyntax.java
```

Compilando para una plataforma de ejecución anterior

Si necesita compilar Java para que se ejecute en plataformas Java más antiguas, el método más simple es instalar un JDK para la versión más antigua que necesita admitir y usar el compilador de JDK en sus compilaciones.

También puede compilar con un compilador de Java más nuevo, pero los hay complicados. En primer lugar, hay algunas condiciones previas importantes que deben cumplirse:

- El código que está compilando no debe usar construcciones del lenguaje Java que no estaban disponibles en la versión de Java que está seleccionando.
- El código no debe depender de clases Java estándar, campos, métodos, etc. que no estuvieran disponibles en las plataformas anteriores.
- Las bibliotecas de terceros de las que depende el código también deben construirse para la plataforma más antigua y estar disponibles en tiempo de compilación y tiempo de ejecución.

Dado que se cumplen las condiciones previas, puede volver a compilar el código para una plataforma anterior utilizando la opción `-target` . Por ejemplo,

```
$ javac -target 1.4 SomeClass.java
```

compilará la clase anterior para producir códigos de bytes que sean compatibles con Java 1.4 o posterior JVM. (De hecho, el `-source` opción implica una compatibles `-target` , por lo `javac -source 1.4 ...` tendría el mismo efecto. La relación entre `-source` y `-target` se describe en la documentación de Oracle.)

Una vez dicho esto, si usted utiliza simplemente `-target` o `-source` , usted seguirá siendo compilando contra las bibliotecas de clases estándar proporcionados por JDK del compilador. Si no tiene cuidado, puede terminar con clases con la versión correcta de bytecode, pero con dependencias en las API que no están disponibles. La solución es usar la opción `-bootclasspath` . Por ejemplo:

```
$ javac -target 1.4 --bootclasspath path/to/java1.4/rt.jar SomeClass.java
```

Se compilará contra un conjunto alternativo de bibliotecas de tiempo de ejecución. Si la clase que se está compilando tiene dependencias (accidentales) en las bibliotecas más nuevas, esto le dará errores de compilación.

Lea Compilador de Java - 'javac' en línea: https://riptutorial.com/es/java/topic/4478/compilador-de-

java----javac-

Capítulo 47: CompletableFuture

Introducción

CompletableFuture es una clase agregada a Java SE 8 que implementa la interfaz Future de Java SE 5. Además de admitir la interfaz Future, agrega muchos métodos que permiten la devolución de llamadas asíncronas cuando se completa el futuro.

Examples

Convertir el método de bloqueo a asíncrono.

El siguiente método tomará uno o dos segundos, dependiendo de su conexión para recuperar una página web y contar la longitud del texto. Cualquiera que sea el hilo que llame, se bloqueará durante ese período de tiempo. También vuelve a lanzar una excepción que es útil más adelante.

```
public static long blockingGetWebPageLength(String urlString) {
    try (BufferedReader br = new BufferedReader(new InputStreamReader(new
URL(urlString).openConnection().getInputStream()))) {
        StringBuilder sb = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            sb.append(line);
        }
        return sb.toString().length();
    } catch (IOException ex) {
        throw new RuntimeException(ex);
    }
}
```

Esto lo convierte en un método que regresará inmediatamente moviendo la llamada al método de bloqueo a otro hilo. De forma predeterminada, el método supplyAsync ejecutará el proveedor en el grupo común. Para un método de bloqueo, probablemente esta no sea una buena opción ya que uno podría agotar los subprocesos en ese grupo, por lo que agregué el parámetro de servicio opcional.

```
static private ExecutorService service = Executors.newCachedThreadPool();

static public CompletableFuture<Long> asyncGetWebPageLength(String url) {
    return CompletableFuture.supplyAsync(() -> blockingGetWebPageLength(url), service);
}
```

Para usar la función de forma asíncrona, se debe usar uno de los métodos que aceptan una lamda para que se llame con el resultado del proveedor cuando se complete, como por ejemplo, Aceptar. También es importante usar excepcionalmente o manejar un método para registrar cualquier excepción que pueda haber ocurrido.

```
public static void main(String[] args) {
```

```
    asyncGetWebPageLength("https://stackoverflow.com/")
            .thenAccept(l -> {
                System.out.println("Stack Overflow returned " + l);
            })
            .exceptionally((Throwable throwable) -> {
                Logger.getLogger("myclass").log(Level.SEVERE, "", throwable);
                return null;
            });

}
```

Ejemplo simple de CompletableFuture

En el ejemplo siguiente, el `calculateShippingPrice` método calcula el coste de envío, lo que lleva algún tiempo de procesamiento. En un ejemplo del mundo real, esto sería, por ejemplo, ponerse en contacto con otro servidor que devuelve el precio según el peso del producto y el método de envío.

Al modelar esto de forma asíncrona a través de `CompletableFuture` , podemos continuar con diferentes trabajos en el método (es decir, calcular los costos de empaque).

```
public static void main(String[] args) {
    int price = 15; // Let's keep it simple and work with whole number prices here
    int weightInGrams = 900;

    calculateShippingPrice(weightInGrams) // Here, we get the future
        .thenAccept(shippingPrice -> { // And then immediately work on it!
            // This fluent style is very useful for keeping it concise
            System.out.println("Your total price is: " + (price + shippingPrice));
        });
    System.out.println("Please stand by. We are calculating your total price.");
}


public static CompletableFuture<Integer> calculateShippingPrice(int weightInGrams) {
    return CompletableFuture.supplyAsync(() -> {
        // supplyAsync is a factory method that turns a given
        // Supplier<U> into a CompletableFuture<U>

        // Let's just say each 200 grams is a new dollar on your shipping costs
        int shippingCosts = weightInGrams / 200;

        try {
            Thread.sleep(2000L); // Now let's simulate some waiting time...
        } catch(InterruptedException e) { /* We can safely ignore that */ }

        return shippingCosts; // And send the costs back!
    });
}
```

Lea CompletableFuture en línea: https://riptutorial.com/es/java/topic/10935/completablefuture

Capítulo 48: Conjuntos

Examples

Declarando un HashSet con valores

Puedes crear una nueva clase que hereda de HashSet:

```
Set<String> h = new HashSet<String>() {{
    add("a");
    add("b");
}};
```

Una solución de línea:

```
Set<String> h = new HashSet<String>(Arrays.asList("a", "b"));
```

Utilizando guayaba:

```
Sets.newHashSet("a", "b", "c")
```

Utilizando Streams:

```
Set<String> set3 = Stream.of("a", "b", "c").collect(toSet());
```

Tipos y uso de conjuntos

Generalmente, los conjuntos son un tipo de colección que almacena valores únicos. La unicidad está determinada por los métodos `equals()` y `hashCode()` .

La clasificación está determinada por el tipo de conjunto.

`HashSet` - Clasificación aleatoria

Java SE 7

```
Set<String> set = new HashSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Strawberry", "Banana", "Apple"]
```

`LinkedHashSet` - Orden de inserción

Java SE 7

```
Set<String> set = new LinkedHashSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Banana", "Apple", "Strawberry"]
```

`TreeSet` - por `compareTo()` o `Comparator`

Java SE 7

```
Set<String> set = new TreeSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Apple", "Banana", "Strawberry"]
```

Java SE 7

```
Set<String> set = new TreeSet<> ((string1, string2) -> string2.compareTo(string1));
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// Set Elements: ["Strawberry", "Banana", "Apple"]
```

Inicialización

Un conjunto es una colección que no puede contener elementos duplicados. Modela la abstracción del conjunto matemático.

`Set` tiene su implementación en varias clases como `HashSet` , `TreeSet` , `LinkedHashSet` .

Por ejemplo:

HashSet:

```
Set<T> set = new HashSet<T>();
```

Aquí `T` puede ser `String` , `Integer` o cualquier otro **objeto** . **HashSet** permite una búsqueda rápida de O (1) pero no ordena los datos agregados y pierde el orden de inserción de los elementos.

TreeSet:

Almacena los datos de forma ordenada, sacrificando cierta velocidad para las operaciones básicas que toman O (lg (n)). No mantiene el orden de inserción de los artículos.

```
TreeSet<T> sortedSet = new TreeSet<T>();
```

LinkedHashSet:

Es una implementación de lista enlazada de `HashSet` Una vez puede iterar sobre los elementos en el orden en que se agregaron. La clasificación no está prevista para su contenido. O (1) se proporcionan operaciones básicas, sin embargo, hay un costo más alto que `HashSet` para mantener la lista de vínculos de respaldo.

```
LinkedHashSet<T> linkedhashset = new LinkedHashSet<T>();
```

Fundamentos de set

¿Qué es un conjunto?

Un conjunto es una estructura de datos que contiene un conjunto de elementos con una propiedad importante que no hay dos elementos iguales en el conjunto.

Tipos de juego:

1. **HashSet:** un conjunto respaldado por una tabla hash (en realidad una instancia de HashMap)
2. **HashSet vinculado:** un conjunto respaldado por una tabla de hash y una lista enlazada, con un orden de iteración predecible
3. **TreeSet:** Una implementación de NavigableSet basada en un TreeMap.

Creando un conjunto

```
Set<Integer> set = new HashSet<Integer>(); // Creates an empty Set of Integers

Set<Integer> linkedHashSet = new LinkedHashSet<Integer>(); //Creates a empty Set of Integers,
with predictable iteration order
```

Añadiendo elementos a un conjunto

Los elementos se pueden agregar a un conjunto usando el método `add()`

```
 set.add(12); //  - Adds element 12 to the set
 set.add(13); //  - Adds element 13 to the set
```

Nuestro set después de ejecutar este método:

```
set = [12,13]
```

Eliminar todos los elementos de un conjunto.

```
set.clear();  //Removes all objects from the collection.
```

Después de este conjunto será:

```
set = []
```

Compruebe si un elemento es parte del conjunto

La existencia de un elemento en el conjunto se puede verificar usando el método de `contains()`

```
set.contains(0);  //Returns true if a specified object is an element within the set.
```

Salida: `False`

Compruebe si un conjunto está vacío

`isEmpty()` método `isEmpty()` se puede usar para verificar si un conjunto está vacío.

```
set.isEmpty();  //Returns true if the set has no elements
```

Salida: Verdadero

Eliminar un elemento del conjunto.

```
 set.remove(0); // Removes first occurrence of a specified object from the collection
```

Compruebe el tamaño del conjunto

```
set.size(); //Returns the number of elements in the collection
```

Salida: 0

Crear una lista de un conjunto existente

Usando una nueva lista

```
List<String> list = new ArrayList<String>(listOfElements);
```

Usando el método List.addAll ()

```
    Set<String> set = new HashSet<String>();
    set.add("foo");
    set.add("boo");

    List<String> list = new ArrayList<String>();
    list.addAll(set);
```

Usando Java 8 Steam API

```
List<String> list = set.stream().collect(Collectors.toList());
```

Eliminando duplicados utilizando Set

Supongamos que tiene `elements` una colección y desea crear otra colección que contenga los mismos elementos pero con todos los **duplicados eliminados** :

```
Collection<Type> noDuplicates = new HashSet<Type>(elements);
```

Ejemplo :

```
List<String> names = new ArrayList<>(
        Arrays.asList("John", "Marco", "Jenny", "Emily", "Jenny", "Emily", "John"));
Set<String> noDuplicates = new HashSet<>(names);
System.out.println("noDuplicates = " + noDuplicates);
```

Salida :

```
noDuplicates = [Marco, Emily, John, Jenny]
```

Lea Conjuntos en línea: https://riptutorial.com/es/java/topic/3102/conjuntos

Capítulo 49: Constructores

Introducción

Aunque no es obligatorio, los constructores en Java son métodos reconocidos por el compilador para instanciar valores específicos para la clase que pueden ser esenciales para el rol del objeto. Este tema demuestra el uso adecuado de los constructores de clases Java.

Observaciones

La especificación del lenguaje Java habla en detalle sobre la naturaleza exacta de la semántica del constructor. Se pueden encontrar en JLS §8.8

Examples

Constructor predeterminado

El "predeterminado" para los constructores es que no tienen ningún argumento. En caso de que no especifique **ningún** constructor, el compilador generará un constructor predeterminado por usted.
Esto significa que los siguientes dos fragmentos son semánticamente equivalentes:

```
public class TestClass {
    private String test;
}
```

```
public class TestClass {
    private String test;
    public TestClass() {

    }
}
```

La visibilidad del constructor predeterminado es la misma que la visibilidad de la clase. Por lo tanto, una clase definida de paquete privado tiene un constructor predeterminado privado de paquete

Sin embargo, si tiene un constructor no predeterminado, el compilador no generará un constructor predeterminado para usted. Así que estos no son equivalentes:

```
public class TestClass {
    private String test;
    public TestClass(String arg) {
    }
}
```

```
public class TestClass {
    private String test;
    public TestClass() {
    }
    public TestClass(String arg) {
    }
}
```

Tenga en cuenta que el constructor generado no realiza una inicialización no estándar. Esto significa que todos los campos de su clase tendrán su valor predeterminado, a menos que tengan un inicializador.

```
public class TestClass {

    private String testData;

    public TestClass() {
        testData = "Test"
    }
}
```

Los constructores se llaman así:

```
TestClass testClass = new TestClass();
```

Constructor con Argumentos

Los constructores se pueden crear con cualquier tipo de argumentos.

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
        this.testData = testData;
    }
}
```

Llamado así:

```
TestClass testClass = new TestClass("Test Data");
```

Una clase puede tener múltiples constructores con diferentes firmas. Para encadenar llamadas de constructor (llamar a un constructor diferente de la misma clase al crear instancias) use `this()` .

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
        this.testData = testData;
    }
```

```
    public TestClass() {
        this("Test"); // testData defaults to "Test"
    }
}
```

Llamado así:

```
TestClass testClass1 = new TestClass("Test Data");
TestClass testClass2 = new TestClass();
```

Llamar al constructor padre

Digamos que tienes una clase para padres y una clase para niños. Para construir una instancia de Child siempre se requiere que se ejecute algún constructor Parent desde el principio del constructor de Child. Podemos seleccionar el constructor principal que queremos llamando explícitamente a `super(...)` con los argumentos apropiados como nuestra primera declaración del constructor secundario. Hacer esto nos ahorra tiempo al reutilizar el constructor de las clases principales en lugar de volver a escribir el mismo código en el constructor de las clases secundarias.

Sin `super(...)` método:

(implícitamente, la versión no-args `super()` se llama invisiblemente)

```
class Parent {
    private String name;
    private int age;

    public Parent() {} // necessary because we call super() without arguments

    public Parent(String tName, int tAge) {
        name = tName;
        age = tAge;
    }
}

// This does not even compile, because name and age are private,
// making them invisible even to the child class.
class Child extends Parent {
    public Child() {
        // compiler implicitly calls super() here
        name = "John";
        age = 42;
    }
}
```

Con el método `super()` :

```
class Parent {
    private String name;
    private int age;
    public Parent(String tName, int tAge) {
        name = tName;
        age = tAge;
```

```
    }
}

class Child extends Parent {
    public Child() {
        super("John", 42);   // explicit super-call
    }
}
```

Nota: las llamadas a otro constructor (encadenamiento) o el súper constructor **DEBEN** ser la primera declaración dentro del constructor.

Si llama al constructor `super(...)` explícitamente, debe existir un constructor padre coincidente (eso es sencillo, ¿no es así?).

Si no llama explícitamente a ningún constructor `super(...)` , su clase principal debe tener un constructor sin argumentos, y esto puede ser escrito explícitamente o creado por defecto por el compilador si la clase principal no proporciona cualquier constructor

```
class Parent{
    public Parent(String tName, int tAge) {}
}

class Child extends Parent{
    public Child(){}
}
```

La clase Parent no tiene un constructor predeterminado, por lo tanto, el compilador no puede agregar `super` en el constructor Child. Este código no se compilará. Debes cambiar los constructores para que encajen en ambos lados, o escribir tu propia `super` llamada, así:

```
class Child extends Parent{
    public Child(){
          super("",0);
    }
}
```

Lea Constructores en línea: https://riptutorial.com/es/java/topic/682/constructores

Capítulo 50: Convertir hacia y desde cuerdas

Examples

Convertir otros tipos de datos a String

- Puede obtener el valor de otros tipos de datos primitivos como una cadena utilizando uno de los métodos `valueOf` la clase String.

  Por ejemplo:

  ```
  int i = 42;
  String string = String.valueOf(i);
  //string now equals "42”.
  ```

  Este método también está sobrecargado para otros tipos de datos, como `float` , `double` , `boolean` e incluso `Object` .

- También puede obtener cualquier otro Objeto (cualquier instancia de cualquier clase) como una Cadena llamando a `.toString` en él. Para que esto dé un resultado útil, la clase debe anular `toString()` . La mayoría de las clases estándar de la biblioteca de Java lo hacen, como `Date` y otras.

  Por ejemplo:

  ```
  Foo foo = new Foo(); //Any class.
  String stringifiedFoo = foo.toString().
  ```

  Aquí `stringifiedFoo` contiene una representación de `foo` como una cadena.

También puede convertir cualquier tipo de número a Cadena con una notación corta como la que se muestra a continuación.

```
int i = 10;
String str = i + "";
```

O simplemente una forma simple es

```
String str = 10 + "";
```

Conversión a / desde bytes

Para codificar una cadena en una matriz de bytes, simplemente puede usar el método `String#getBytes()` , con uno de los juegos de caracteres estándar disponibles en cualquier tiempo de ejecución de Java:

```
byte[] bytes = "test".getBytes(StandardCharsets.UTF_8);
```

y para decodificar:

```
String testString = new String(bytes, StandardCharsets.UTF_8);
```

Puede simplificar aún más la llamada utilizando una importación estática:

```
import static java.nio.charset.StandardCharsets.UTF_8;
...
byte[] bytes = "test".getBytes(UTF_8);
```

Para conjuntos de caracteres menos comunes, puede indicar el conjunto de caracteres con una cadena:

```
byte[] bytes = "test".getBytes("UTF-8");
```

y al revés:

```
String testString = new String (bytes, "UTF-8");
```

sin embargo, esto significa que debe manejar la excepción `UnsupportedCharsetException` .

La siguiente llamada utilizará el conjunto de caracteres predeterminado. El conjunto de caracteres predeterminado es específico de la plataforma y generalmente difiere entre las plataformas Windows, Mac y Linux.

```
byte[] bytes = "test".getBytes();
```

y al revés:

```
String testString = new String(bytes);
```

Tenga en cuenta que los caracteres y los bytes no válidos pueden ser reemplazados o omitidos por estos métodos. Para obtener más control, por ejemplo para validar la entrada, se recomienda utilizar las clases `CharsetEncoder` y `CharsetDecoder` .

Codificación / decodificación de Base64

Ocasionalmente encontrará la necesidad de codificar datos binarios como una cadena codificada en *base64* .

Para esto podemos usar la clase `DatatypeConverter` del paquete `javax.xml.bind` :

```
import javax.xml.bind.DatatypeConverter;
```

```
import java.util.Arrays;

// arbitrary binary data specified as a byte array
byte[] binaryData = "some arbitrary data".getBytes("UTF-8");

// convert the binary data to the base64-encoded string
String encodedData = DatatypeConverter.printBase64Binary(binaryData);
// encodedData is now "c29tZSBhcmJpdHJhcnkgZGF0YQ=="

// convert the base64-encoded string back to a byte array
byte[] decodedData = DatatypeConverter.parseBase64Binary(encodedData);

// assert that the original data and the decoded data are equal
assert Arrays.equals(binaryData, decodedData);
```

Apache commons-codec

Alternativamente, podemos usar `Base64` de Apache commons-codec .

```
import org.apache.commons.codec.binary.Base64;

// your blob of binary as a byte array
byte[] blob = "someBinaryData".getBytes();

// use the Base64 class to encode
String binaryAsAString = Base64.encodeBase64String(blob);

// use the Base64 class to decode
byte[] blob2 = Base64.decodeBase64(binaryAsAString);

// assert that the two blobs are equal
System.out.println("Equal : " + Boolean.toString(Arrays.equals(blob, blob2)));
```

Si inspecciona este programa mientras se ejecuta, verá que `someBinaryData` codifica para `c29tZUJpbmFyeURhdGE=` , un objeto de cadena *UTF-8* muy `c29tZUJpbmFyeURhdGE=` .

Java SE 8

Detalles para el mismo se pueden encontrar en Base64

```
// encode with padding
String encoded = Base64.getEncoder().encodeToString(someByteArray);

// encode without padding
String encoded = Base64.getEncoder().withoutPadding().encodeToString(someByteArray);

// decode a String
byte [] barr = Base64.getDecoder().decode(encoded);
```

Referencia

Analizar cadenas a un valor numérico

Cadena a un tipo numérico primitivo o un tipo de contenedor numérico:

Cada clase envoltura numérica proporciona un método `parseXxx` que convierte una `String` al tipo primitivo correspondiente. El siguiente código convierte una `String` en un `int` usando el método `Integer.parseInt` :

```
String string = "59";
int primitive = Integer.parseInteger(string);
```

Para convertir a una `String` a una instancia de una clase de envoltura numérica, puede utilizar una sobrecarga del método `valueOf` clases de envoltorio:

```
String string = "59";
Integer wrapper = Integer.valueOf(string);
```

o confiar en el boxeo automático (Java 5 y posterior):

```
String string = "59";
Integer wrapper = Integer.parseInteger(string);  // 'int' result is autoboxed
```

El patrón anterior funciona para `byte` , `short` , `int` , `long` , `float` y `double` y las clases de envoltorio correspondientes (`Byte` , `Short` , `Integer` , `Long` , `Float` y `Double`).

Cadena a entero usando radix:

```
String integerAsString = "0101"; // binary representation
int parseInt = Integer.parseInt(integerAsString,2);
Integer valueOfInteger = Integer.valueOf(integerAsString,2);
System.out.println(valueOfInteger); // prints 5
System.out.println(parseInt); // prints 5
```

Excepciones

La excepción NumberFormatException no marcada se lanzará si se llama a un método numérico `valueOf(String)` o `parseXxx(...)` para una cadena que no es una representación numérica aceptable, o que representa un valor fuera de rango.

Obteniendo un `String` de un` InputStream`

Una `String` se puede leer desde un `InputStream` usando el constructor de matriz de bytes.

```
import java.io.*;

public String readString(InputStream input) throws IOException {
    byte[] bytes = new byte[50]; // supply the length of the string in bytes here
    input.read(bytes);
    return new String(bytes);
}
```

Esto utiliza el conjunto de caracteres predeterminado del sistema, aunque se puede especificar un conjunto de caracteres alternativo:

```
return new String(bytes, Charset.forName("UTF-8"));
```

Convertir cadena a otros tipos de datos.

Puede convertir una cadena **numérica** a varios tipos numéricos de Java de la siguiente manera:

Cadena a int:

```
String number = "12";
int num = Integer.parseInt(number);
```

Cadena para flotar:

```
String number = "12.0";
float num = Float.parseFloat(number);
```

Cadena para doblar:

```
String double = "1.47";
double num = Double.parseDouble(double);
```

Cadena a booleano:

```
String falseString = "False";
boolean falseBool = Boolean.parseBoolean(falseString);   // falseBool = false

String trueString = "True";
boolean trueBool = Boolean.parseBoolean(trueString);     // trueBool = true
```

Cadena a largo:

```
String number = "47";
long num = Long.parseLong(number);
```

Cadena a BigInteger:

```
String bigNumber = "21";
BigInteger reallyBig = new BigInteger(bigNumber);
```

Cadena a BigDecimal:

```
String bigFraction = "17.21455";
BigDecimal reallyBig = new BigDecimal(bigFraction);
```

Excepciones de conversión:

Las conversiones numéricas anteriores arrojarán una `NumberFormatException` (sin `NumberFormatException`) si intenta analizar una cadena que no es un número con formato adecuado, o está fuera del rango para el tipo de destino. El tema de Excepciones discute cómo

lidiar con tales excepciones.

Si desea probar que puede analizar una cadena, podría implementar un método `tryParse...` como este:

```
boolean tryParseInt (String value) {
    try {
        String somechar = Integer.parseInt(value);
        return true;
     } catch (NumberFormatException e) {
        return false;
     }
}
```

Sin embargo, llamar a este método `tryParse...` inmediatamente antes de analizar es (posiblemente) una mala práctica. Sería mejor simplemente llamar al método `parse...` y lidiar con la excepción.

Lea Convertir hacia y desde cuerdas en línea: https://riptutorial.com/es/java/topic/6678/convertir-hacia-y-desde-cuerdas

Capítulo 51: Corrientes

Introducción

Un `Stream` representa una secuencia de elementos y admite diferentes tipos de operaciones para realizar cálculos sobre esos elementos. Con Java 8, la interfaz de la `Collection` tiene dos métodos para generar un `Stream` : `stream()` y `parallelStream()` . `Stream` operaciones de `Stream` son intermedias o terminales. Las operaciones intermedias devuelven un `Stream` por lo que se pueden encadenar múltiples operaciones intermedias antes de que se cierre el `Stream` . Las operaciones de la terminal son nulas o devuelven un resultado no continuo.

Sintaxis

- collection.stream ()
- Arrays.stream (array)
- Stream.iterate (firstValue, currentValue -> nextValue)
- Stream.generate (() -> valor)
- Stream.of (elementOfT [, elementOfT, ...])
- Stream.empty ()
- StreamSupport.stream (iterable.spliterator (), false)

Examples

Usando Streams

Un `Stream` es una secuencia de elementos sobre los cuales se pueden realizar operaciones agregadas secuenciales y paralelas. Cualquier `Stream` dado puede potencialmente tener una cantidad ilimitada de datos que fluyen a través de él. Como resultado, los datos recibidos de un `Stream` se procesan individualmente a medida que llegan, en lugar de realizar el procesamiento por lotes de los datos por completo. Cuando se combinan con expresiones lambda , proporcionan una forma concisa de realizar operaciones en secuencias de datos utilizando un enfoque funcional.

Ejemplo: (verlo funciona en Ideone)

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

fruitStream.filter(s -> s.contains("a"))
           .map(String::toUpperCase)
           .sorted()
           .forEach(System.out::println);
```

Salida:

MANZANA

PLÁTANO
NARANJA
PERA

Las operaciones realizadas por el código anterior se pueden resumir de la siguiente manera:

1. Crear una `Stream<String>` contiene un secuenciado clasificadas `Stream` de fruta `String` elementos usando el método de fábrica estática `Stream.of(values)` .

2. La operación de `filter()` retiene solo los elementos que coinciden con un predicado dado (los elementos que cuando son probados por el predicado devuelven verdadero). En este caso, retiene los elementos que contienen una `"a"` . El predicado se da como una expresión lambda .

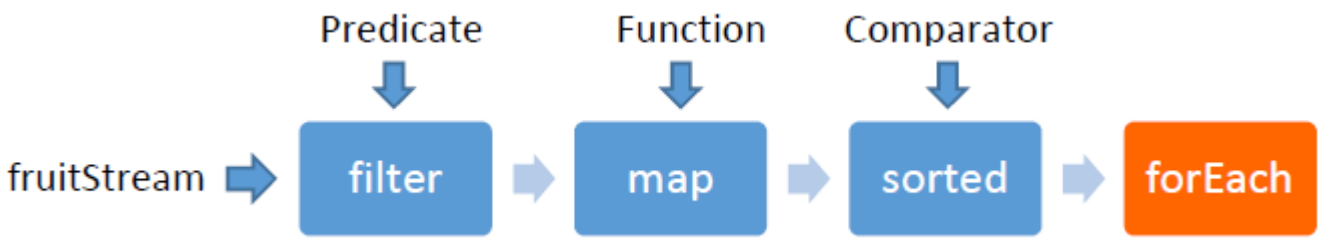
3. La operación `map()` transforma cada elemento usando una función dada, llamada mapeador. En este caso, cada `String` fruta se asigna a su versión de `String` mayúsculas utilizando el método `String::toUppercase` .

   > **Tenga en cuenta** que la operación `map()` devolverá un flujo con un tipo genérico diferente si la función de mapeo devuelve un tipo diferente a su parámetro de entrada. Por ejemplo, en una `Stream<String>` llamada a `.map(String::isEmpty)` devuelve un `Stream<Boolean>`

4. La operación `sorted()` ordena los elementos de la `Stream` acuerdo con su ordenamiento natural (lexicográficamente, en el caso de `String`).

5. Finalmente, la `forEach(action)` realiza una acción que actúa en cada elemento de la `Stream` , pasándola a un Consumidor . En el ejemplo, cada elemento simplemente se imprime en la consola. Esta operación es una operación de terminal, por lo que es imposible volver a operar en ella.

   > **Tenga en cuenta** que las operaciones definidas en el `Stream` se realizan *debido* a la operación del terminal. Sin una operación de terminal, el flujo no se procesa. Las transmisiones no pueden ser reutilizadas. Una vez que se llama a una operación de terminal, el objeto `Stream` vuelve inutilizable.



Las operaciones (como se ve arriba) se encadenan para formar lo que se puede ver como una consulta en los datos.

Cierre de arroyos

> **Tenga en cuenta que un `Stream` generalmente no tiene que estar cerrado.** Solo se requiere cerrar secuencias que operan en canales IO. La mayoría de `Stream` tipos no operan sobre los recursos y por lo tanto no requieren de cierre.

La interfaz de `Stream` extiende a `AutoCloseable` . Las secuencias se pueden cerrar llamando al método de `close` o usando declaraciones try-with-resource.

Un ejemplo de caso de uso en el que se debería cerrar una `Stream` es cuando crea una `Stream` de líneas desde un archivo:

```
try (Stream<String> lines = Files.lines(Paths.get("somePath"))) {
    lines.forEach(System.out::println);
}
```

La interfaz de `Stream` también declara el método `Stream.onClose()` que le permite registrar los controladores de `Runnable` que se `Runnable` cuando se cierre el stream. Un ejemplo de caso de uso es cuando el código que produce una transmisión necesita saber cuándo se consume para realizar una limpieza.

```
public Stream<String>streamAndDelete(Path path) throws IOException {
    return Files.lines(path).onClose(() -> someClass.deletePath(path));
}
```

El controlador de ejecución solo se ejecutará si se llama al método `close()` , ya sea explícita o implícitamente mediante una declaración try-with-resources.

Orden de procesamiento

El procesamiento de un objeto `Stream` puede ser secuencial o paralelo .

En un modo **secuencial** , los elementos se procesan en el orden de la fuente de la `Stream` . Si se ordena el `Stream` (como una implementación de `SortedMap` o una `List`), se garantiza que el procesamiento coincidirá con el orden de la fuente. En otros casos, sin embargo, se debe tener cuidado de no depender de la ordenación (ver: ¿ es consistente el orden de iteración de Java `HashMap keySet()` ?).

Ejemplo:

```
List<Integer> integerList = Arrays.asList(0, 1, 2, 3, 42);

// sequential
long howManyOddNumbers = integerList.stream()
                                    .filter(e -> (e % 2) == 1)
                                    .count();

System.out.println(howManyOddNumbers); // Output: 2
```

Vivir en Ideone

El modo **paralelo** permite el uso de múltiples hilos en múltiples núcleos, pero no hay garantía del orden en que se procesan los elementos.

Si se invocan varios métodos en una secuencia `Stream` , no es necesario invocar todos los métodos. Por ejemplo, si un `Stream` se filtra y el número de elementos se reduce a uno, no se producirá una llamada posterior a un método como la `sort` . Esto puede aumentar el rendimiento de un `Stream` secuencial, una optimización que no es posible con un `Stream` paralelo.

Ejemplo:

```
// parallel
long howManyOddNumbersParallel = integerList.parallelStream()
                                            .filter(e -> (e % 2) == 1)
                                            .count();

System.out.println(howManyOddNumbersParallel); // Output: 2
```

Vivir en Ideone

Diferencias de Contenedores (o Colecciones)

Si bien algunas acciones se pueden realizar tanto en Contenedores como en Flujos, en última instancia tienen propósitos diferentes y admiten diferentes operaciones. Los contenedores están más enfocados en cómo se almacenan los elementos y cómo se puede acceder a esos elementos de manera eficiente. Un `Stream` , por otro lado, no proporciona acceso directo y manipulación a sus elementos; está más dedicado al grupo de objetos como una entidad colectiva y realiza operaciones en esa entidad en su conjunto. `Stream` y `Collection` son abstracciones de alto nivel separadas para estos propósitos diferentes.

Recoge los elementos de una corriente en una colección

Recopilar con `toList()` y `toSet()`

Los elementos de un `Stream` se pueden recopilar fácilmente en un contenedor utilizando la operación `Stream.collect` :

```
System.out.println(Arrays
    .asList("apple", "banana", "pear", "kiwi", "orange")
    .stream()
    .filter(s -> s.contains("a"))
    .collect(Collectors.toList())
);
// prints: [apple, banana, pear, orange]
```

Se pueden crear otras instancias de recopilación, como un `Set` , utilizando otros métodos integrados de `Collectors` . Por ejemplo, `Collectors.toSet()` recopila los elementos de un `Stream` en un `Set` .

Control explícito sobre la implementación de `List` o `Set`

De acuerdo con la documentación de los `Collectors#toList()` y los `Collectors#toSet()` , no hay garantías sobre el tipo, mutabilidad, serialización o seguridad de subprocesos de la `List` o el `Set` devuelto.

Para que el control explícito sobre la implementación sea devuelto, los `Collectors#toCollection(Supplier)` pueden usarse en su lugar, donde el proveedor dado devuelve una colección nueva y vacía.

```
// syntax with method reference
System.out.println(strings
        .stream()
        .filter(s -> s != null && s.length() <= 3)
        .collect(Collectors.toCollection(ArrayList::new))
);

// syntax with lambda
System.out.println(strings
        .stream()
        .filter(s -> s != null && s.length() <= 3)
        .collect(Collectors.toCollection(() -> new LinkedHashSet<>()))
);
```

Recopilando elementos usando toMap

El recopilador acumula elementos en un mapa, donde la clave es la identificación del estudiante y el valor es el valor del estudiante.

```
  List<Student> students = new ArrayList<Student>();
    students.add(new Student(1,"test1"));
    students.add(new Student(2,"test2"));
    students.add(new Student(3,"test3"));

    Map<Integer, String> IdToName = students.stream()
        .collect(Collectors.toMap(Student::getId, Student::getName));
    System.out.println(IdToName);
```

Salida:

```
{1=test1, 2=test2, 3=test3}
```

Collectors.toMap tiene otra implementación `Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper, BinaryOperator<U> mergeFunction)` . El mergeFunction se usa principalmente para seleccionar un nuevo valor o retener un valor antiguo si la clave se repite al agregar un nuevo miembro en el Mapa de una lista.

El mergeFunction a menudo se ve así: `(s1, s2) -> s1` para retener el valor correspondiente a la tecla repetida, o `(s1, s2) -> s2` para poner un nuevo valor para la tecla repetida.

Recopilación de elementos al mapa de colecciones

Ejemplo: de ArrayList a Map <String, List <>>

A menudo se requiere hacer un mapa de la lista de una lista primaria. Ejemplo: de un estudiante de la lista, necesitamos hacer un mapa de la lista de temas para cada estudiante.

```
    List<Student> list = new ArrayList<>();
    list.add(new Student("Davis", SUBJECT.MATH, 35.0));
    list.add(new Student("Davis", SUBJECT.SCIENCE, 12.9));
    list.add(new Student("Davis", SUBJECT.GEOGRAPHY, 37.0));

    list.add(new Student("Sascha", SUBJECT.ENGLISH, 85.0));
    list.add(new Student("Sascha", SUBJECT.MATH, 80.0));
    list.add(new Student("Sascha", SUBJECT.SCIENCE, 12.0));
    list.add(new Student("Sascha", SUBJECT.LITERATURE, 50.0));

    list.add(new Student("Robert", SUBJECT.LITERATURE, 12.0));

    Map<String, List<SUBJECT>> map = new HashMap<>();
    list.stream().forEach(s -> {
                map.computeIfAbsent(s.getName(), x -> new ArrayList<>()).add(s.getSubject());
            });
    System.out.println(map);
```

Salida:

```
{ Robert=[LITERATURE],
Sascha=[ENGLISH, MATH, SCIENCE, LITERATURE],
Davis=[MATH, SCIENCE, GEOGRAPHY] }
```

Ejemplo: de ArrayList a Map <String, Map <>>

```
    List<Student> list = new ArrayList<>();
    list.add(new Student("Davis", SUBJECT.MATH, 1, 35.0));
    list.add(new Student("Davis", SUBJECT.SCIENCE, 2, 12.9));
    list.add(new Student("Davis", SUBJECT.MATH, 3, 37.0));
    list.add(new Student("Davis", SUBJECT.SCIENCE, 4, 37.0));

    list.add(new Student("Sascha", SUBJECT.ENGLISH, 5, 85.0));
    list.add(new Student("Sascha", SUBJECT.MATH, 1, 80.0));
    list.add(new Student("Sascha", SUBJECT.ENGLISH, 6, 12.0));
    list.add(new Student("Sascha", SUBJECT.MATH, 3, 50.0));

    list.add(new Student("Robert", SUBJECT.ENGLISH, 5, 12.0));

    Map<String, Map<SUBJECT, List<Double>>> map = new HashMap<>();

    list.stream().forEach(student -> {
        map.computeIfAbsent(student.getName(), s -> new HashMap<>())
                .computeIfAbsent(student.getSubject(), s -> new ArrayList<>())
                .add(student.getMarks());
    });

    System.out.println(map);
```

Salida:

```
{ Robert={ENGLISH=[12.0]},
Sascha={MATH=[80.0, 50.0], ENGLISH=[85.0, 12.0]},
Davis={MATH=[35.0, 37.0], SCIENCE=[12.9, 37.0]} }
```

Hoja de trucos

Gol	Código
Recoger en una `List`	`Collectors.toList()`
Recoger en un `ArrayList` con tamaño asignado previamente	`Collectors.toCollection(() -> new ArrayList<>(size))`
Recoger a un `Set`	`Collectors.toSet()`
Recoger a un `Set` con un mejor rendimiento de iteración	`Collectors.toCollection(() -> new LinkedHashSet<>())`
Recoger en un `Set<String>` distinga mayúsculas y minúsculas `Set<String>`	`Collectors.toCollection(() -> new TreeSet<>(String.CASE_INSENSITIVE_ORDER))`
Recopilar en un `EnumSet<AnEnum>` (mejor rendimiento para enums)	`Collectors.toCollection(() -> EnumSet.noneOf(AnEnum.class))`
Recoger en un `Map<K,V>` con claves únicas	`Collectors.toMap(keyFunc,valFunc)`
Asignar MyObject.getter () a MyObject único	`Collectors.toMap(MyObject::getter, Function.identity())`
Mapear MyObject.getter () a múltiples MyObjects	`Collectors.groupingBy(MyObject::getter)`

Corrientes infinitas

Es posible generar un `Stream` que no termine. Al llamar a un método de terminal en un `Stream` infinito `Stream` el `Stream` entra en un bucle infinito. El método de `limit` de un `Stream` se puede usar para limitar la cantidad de términos del `Stream` que Java procesa.

Este ejemplo genera una `Stream` de todos los números naturales, comenzando con el número 1. Cada término sucesivo de la `Stream` es uno más alto que el anterior. Al llamar al método de límite de este `Stream` , solo se consideran e imprimen los primeros cinco términos del `Stream` .

```
// Generate infinite stream - 1, 2, 3, 4, 5, 6, 7, ...
IntStream naturalNumbers = IntStream.iterate(1, x -> x + 1);

// Print out only the first 5 terms
naturalNumbers.limit(5).forEach(System.out::println);
```

Salida:

1
2
3
4
5

Otra forma de generar un flujo infinito es usando el método Stream.generate . Este método lleva una lambda de tipo Proveedor .

```
// Generate an infinite stream of random numbers
Stream<Double> infiniteRandomNumbers = Stream.generate(Math::random);

// Print out only the first 10 random numbers
infiniteRandomNumbers.limit(10).forEach(System.out::println);
```

Corrientes de consumo

Un `Stream` solo se atravesará cuando haya una *operación de terminal* , como `count()` , `collect()` o `forEach()` . De lo contrario, no se realizará ninguna operación en el `Stream` .

En el siguiente ejemplo, no se agrega ninguna operación de terminal a la `Stream` , por lo que la operación de `filter()` no se invocará y no se producirá ninguna salida porque `peek()` NO es una *operación de terminal* .

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).peek(System.out::println);
```

Vivir en Ideone

Esta es una secuencia de `Stream` con una *operación de terminal* válida, por lo que se produce una salida.

También `forEach` usar `forEach` lugar de `peek` :

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).forEach(System.out::println);
```

Vivir en Ideone

Salida:

2
4
6
8

Una vez realizada la operación del terminal, el `Stream` se consume y no se puede reutilizar.

Aunque un objeto de flujo dado no puede ser reutilizado, es fácil crear un `Iterable` reutilizable que delega a un flujo de flujo. Esto puede ser útil para devolver una vista modificada de un conjunto de datos en vivo sin tener que recopilar resultados en una estructura temporal.

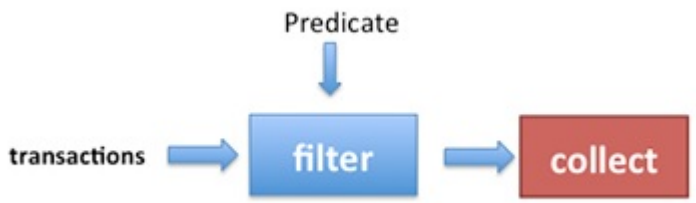
```
List<String> list = Arrays.asList("FOO", "BAR");
Iterable<String> iterable = () -> list.stream().map(String::toLowerCase).iterator();

for (String str : iterable) {
    System.out.println(str);
}
for (String str : iterable) {
    System.out.println(str);
}
```

Salida:

> foo
> bar
> foo
> bar

Esto funciona porque `Iterable` declara un solo método abstracto `Iterator<T> iterator()` . Eso hace que sea efectivamente una interfaz funcional, implementada por un lambda que crea una nueva transmisión en cada llamada.

En general, un `Stream` funciona como se muestra en la siguiente imagen:



NOTA : Las comprobaciones de argumentos siempre se realizan, incluso sin una *operación de terminal* :

```
try {
    IntStream.range(1, 10).filter(null);
} catch (NullPointerException e) {
    System.out.println("We got a NullPointerException as null was passed as an argument to
filter()");
}
```

Vivir en Ideone

Salida:

> Obtuvimos una NullPointerException ya que null se pasó como un argumento para filtrar ()

Creando un Mapa de Frecuencia

El recopilador `groupingBy(classifier, downstream)` permite la recopilación de elementos `Stream` en un `Map` al clasificar cada elemento en un grupo y realizar una operación descendente en los elementos clasificados en el mismo grupo.

Un ejemplo clásico de este principio es usar un `Map` para contar las ocurrencias de elementos en una `Stream` . En este ejemplo, el clasificador es simplemente la función de identidad, que devuelve el elemento como está. La operación descendente cuenta el número de elementos iguales, utilizando `counting()` .

```
Stream.of("apple", "orange", "banana", "apple")
      .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
      .entrySet()
      .forEach(System.out::println);
```

La operación descendente es en sí misma un colector (`Collectors.counting()`) que opera en elementos de tipo String y produce un resultado de tipo `Long` . El resultado de la llamada al método de `collect` es un `Map<String, Long>` .

Esto produciría el siguiente resultado:

> plátano = 1
> naranja = 1
> manzana = 2

Corriente paralela

Nota: antes de decidir qué `Stream` usar, consulte ParallelStream vs Sequential Stream .

Cuando desee realizar operaciones de `Stream` simultánea, puede utilizar cualquiera de estas formas.

```
List<String> data = Arrays.asList("One", "Two", "Three", "Four", "Five");
Stream<String> aParallelStream = data.stream().parallel();
```

O:

```
Stream<String> aParallelStream = data.parallelStream();
```

Para ejecutar las operaciones definidas para el flujo paralelo, llame a un operador de terminal:

```
aParallelStream.forEach(System.out::println);
```

(Una posible) salida de la `Stream` paralela:

> Tres
> Cuatro

Uno
Dos
Cinco

El orden puede cambiar ya que todos los elementos se procesan en paralelo (lo que *puede* hacer que sea más rápido). Use `parallelStream` cuando ordene no importa.

Impacto en el rendimiento

En el caso de que se trate de redes, los `Stream` paralelo pueden degradar el rendimiento general de una aplicación porque todos los `Stream` paralelo utilizan un conjunto común de subprocesos para la red.

Por otro lado, los `Stream` paralelo pueden mejorar significativamente el rendimiento en muchos otros casos, dependiendo del número de núcleos disponibles en la CPU en ejecución en este momento.

Convertir un flujo de opcionales a un flujo de valores

Es posible que necesite convertir un `Stream` emite un `Optional` en un `Stream` de valores, emitiendo solo valores del `Optional` existente. (es decir, sin valor `null` y sin tratar con `Optional.empty()`).

```
Optional<String> op1 = Optional.empty();
Optional<String> op2 = Optional.of("Hello World");

List<String> result = Stream.of(op1, op2)
                            .filter(Optional::isPresent)
                            .map(Optional::get)
                            .collect(Collectors.toList());

System.out.println(result); //[Hello World]
```

Creando un Stream

Todos los java `Collection<E>` tienen métodos `stream()` y `parallelStream()` partir de los cuales se puede construir un `Stream<E>` :

```
Collection<String> stringList = new ArrayList<>();
Stream<String> stringStream = stringList.parallelStream();
```

Se puede crear un `Stream<E>` partir de una matriz usando uno de dos métodos:

```
String[] values = { "aaa", "bbbb", "ddd", "cccc" };
Stream<String> stringStream = Arrays.stream(values);
Stream<String> stringStreamAlternative = Stream.of(values);
```

La diferencia entre `Arrays.stream()` y `Stream.of()` es que `Stream.of()` tiene un parámetro varargs, por lo que se puede usar como:

```
Stream<Integer> integerStream = Stream.of(1, 2, 3);
```

También hay `Stream` primitivas que puedes usar. Por ejemplo:

```
IntStream intStream = IntStream.of(1, 2, 3);
DoubleStream doubleStream = DoubleStream.of(1.0, 2.0, 3.0);
```

Estas corrientes primitivas también se pueden construir utilizando el método `Arrays.stream()` :

```
IntStream intStream = Arrays.stream(new int[]{ 1, 2, 3 });
```

Es posible crear un `Stream` desde una matriz con un rango específico.

```
int[] values= new int[]{1, 2, 3, 4, 5};
IntStream intStram = Arrays.stream(values, 1, 3);
```

Tenga en cuenta que cualquier flujo primitivo se puede convertir a flujo de tipo en caja utilizando el método en `boxed` :

```
Stream<Integer> integerStream = intStream.boxed();
```

Esto puede ser útil en algunos casos si desea recopilar datos, ya que la secuencia primitiva no tiene ningún método de `collect` que tome un `Collector` como argumento.

Reutilización de operaciones intermedias de una cadena de flujo.

La secuencia se cierra cuando se llama a la operación del terminal. Reutilizando el flujo de operaciones intermedias, cuando solo la operación del terminal solo varía. podríamos crear un proveedor de flujo para construir un nuevo flujo con todas las operaciones intermedias ya configuradas.

```
Supplier<Stream<String>> streamSupplier = () -> Stream.of("apple", "banana","orange",
"grapes", "melon","blueberry","blackberry")
.map(String::toUpperCase).sorted();

  streamSupplier.get().filter(s ->   s.startsWith("A")).forEach(System.out::println);

// APPLE

  streamSupplier.get().filter(s -> s.startsWith("B")).forEach(System.out::println);

  // BANANA
  // BLACKBERRY
  // BLUEBERRY
```

`int[]` matrices `int[]` se pueden convertir a la `List<Integer>` usando flujos

```
int[] ints = {1,2,3};
List<Integer> list = IntStream.of(ints).boxed().collect(Collectors.toList());
```

Encontrar estadísticas sobre flujos numéricos

Java 8 proporciona clases denominadas `IntSummaryStatistics` , `DoubleSummaryStatistics` y `LongSummaryStatistics` que proporcionan un objeto de estado para recopilar estadísticas como `count` , `min` , `max` , `sum` y `average` .

Java SE 8

```
List<Integer> naturalNumbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
IntSummaryStatistics stats = naturalNumbers.stream()
                                          .mapToInt((x) -> x)
                                          .summaryStatistics();
System.out.println(stats);
```

Lo que resultará en:

Java SE 8

```
IntSummaryStatistics{count=10, sum=55, min=1, max=10, average=5.500000}
```

Obtener un trozo de un arroyo

Ejemplo: Obtenga un `Stream` de 30 elementos, que contiene del elemento 21 al 50 (inclusive) de una colección.

```
final long n = 20L; // the number of elements to skip
final long maxSize = 30L; // the number of elements the stream should be limited to
final Stream<T> slice = collection.stream().skip(n).limit(maxSize);
```

Notas:

- `IllegalArgumentException` se lanza si `n` es negativo o `maxSize` es negativo
- tanto el `skip(long)` como el `limit(long)` son operaciones intermedias
- si un flujo contiene menos de `n` elementos, entonces `skip(n)` devuelve un flujo vacío
- tanto `skip(long)` como `limit(long)` son operaciones baratas en tuberías de flujo secuencial, pero pueden ser bastante caras en tuberías ordenadas en paralelo

Corrientes de concatenación

Declaración de variables para ejemplos:

```
Collection<String> abc = Arrays.asList("a", "b", "c");
Collection<String> digits = Arrays.asList("1", "2", "3");
Collection<String> greekAbc = Arrays.asList("alpha", "beta", "gamma");
```

Ejemplo 1 - Concatenar dos `Stream`

```
final Stream<String> concat1 = Stream.concat(abc.stream(), digits.stream());

concat1.forEach(System.out::print);
```

```
// prints: abc123
```

Ejemplo 2 - Concatenar más de dos `Stream` s

```
final Stream<String> concat2 = Stream.concat(
    Stream.concat(abc.stream(), digits.stream()),
    greekAbc.stream());

System.out.println(concat2.collect(Collectors.joining(", ")));
// prints: a, b, c, 1, 2, 3, alpha, beta, gamma
```

Alternativamente, para simplificar la sintaxis `concat()` anidada, los `Stream` también se pueden concatenar con `flatMap()` :

```
final Stream<String> concat3 = Stream.of(
    abc.stream(), digits.stream(), greekAbc.stream())
    .flatMap(s -> s);
    // or `.flatMap(Function.identity());` (java.util.function.Function)

System.out.println(concat3.collect(Collectors.joining(", ")));
// prints: a, b, c, 1, 2, 3, alpha, beta, gamma
```

Tenga cuidado al construir `Stream` a partir de una concatenación repetida, ya que el acceso a un elemento de un `Stream` profundamente concatenado puede dar lugar a cadenas de llamadas profundas o incluso a una `StackOverflowException` .

IntStream to String

Java no tiene un *flujo de caracteres* , por lo que al trabajar con `String` y construir un `Stream` de `Character` , una opción es obtener un `IntStream` de puntos de código utilizando el método `String.codePoints()` . Por `IntStream` tanto, `IntStream` se puede obtener de la siguiente manera:

```
public IntStream stringToIntStream(String in) {
  return in.codePoints();
}
```

Es un poco más complicado hacer la conversión de otra manera, por ejemplo, IntStreamToString. Eso se puede hacer de la siguiente manera:

```
public String intStreamToString(IntStream intStream) {
  return intStream.collect(StringBuilder::new, StringBuilder::appendCodePoint,
StringBuilder::append).toString();
}
```

Ordenar usando Stream

```
List<String> data = new ArrayList<>();
data.add("Sydney");
data.add("London");
data.add("New York");
data.add("Amsterdam");
```

```
data.add("Mumbai");
data.add("California");

System.out.println(data);

List<String> sortedData = data.stream().sorted().collect(Collectors.toList());

System.out.println(sortedData);
```

Salida:

```
[Sydney, London, New York, Amsterdam, Mumbai, California]
[Amsterdam, California, London, Mumbai, New York, Sydney]
```

También es posible usar un mecanismo de comparación diferente, ya que hay una versión `sorted` sobrecargada que toma un comparador como argumento.

Además, puedes usar una expresión lambda para ordenar:

```
List<String> sortedData2 = data.stream().sorted((s1,s2) ->
s2.compareTo(s1)).collect(Collectors.toList());
```

Esto generaría `[Sydney, New York, Mumbai, London, California, Amsterdam]`

Puede usar `Comparator.reverseOrder()` para tener un comparador que impone el orden `reverse` al `reverse` .

```
List<String> reverseSortedData =
data.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
```

Arroyos de primitivos

Java proporciona `Stream` especializadas para tres tipos de primitivas `IntStream` (para `int` s), `LongStream` (para s `long`) y `DoubleStream` (para s `double`). Además de ser implementaciones optimizadas para sus primitivas respectivas, también proporcionan varios métodos terminales específicos, típicamente para operaciones matemáticas. P.ej:

```
IntStream is = IntStream.of(10, 20, 30);
double average = is.average().getAsDouble(); // average is 20.0
```

Recopilar los resultados de una secuencia en una matriz

Analógico para obtener una colección para un `Stream` por `collect()` se puede obtener una matriz mediante el método `Stream.toArray()` :

```
List<String> fruits = Arrays.asList("apple", "banana", "pear", "kiwi", "orange");

String[] filteredFruits = fruits.stream()
    .filter(s -> s.contains("a"))
    .toArray(String[]::new);
```

```
// prints: [apple, banana, pear, orange]
System.out.println(Arrays.toString(filteredFruits));
```

`String[]::new` es un tipo especial de referencia de método: una referencia de constructor.

Encontrar el primer elemento que coincide con un predicado

Es posible encontrar el primer elemento de un `Stream` que coincida con una condición.

Para este ejemplo, encontraremos el primer `Integer` cuyo cuadrado es más de `50000` .

```
IntStream.iterate(1, i -> i + 1) // Generate an infinite stream 1,2,3,4...
    .filter(i -> (i*i) > 50000) // Filter to find elements where the square is >50000
    .findFirst(); // Find the first filtered element
```

Esta expresión devolverá un `OptionalInt` con el resultado.

Tenga en cuenta que con un `Stream` infinito, Java seguirá revisando cada elemento hasta que encuentre un resultado. Con un `Stream` finito, si Java se queda sin elementos pero aún no puede encontrar un resultado, devuelve un `OptionalInt` vacío.

Usando IntStream para iterar sobre los índices

`Stream` de elementos generalmente no permiten el acceso al valor de índice del elemento actual. Para iterar sobre una matriz o `ArrayList` mientras tiene acceso a los índices, use `IntStream.range(start, endExclusive)` .

```
String[] names = { "Jon", "Darin", "Bauke", "Hans", "Marc" };

IntStream.range(0, names.length)
    .mapToObj(i -> String.format("#%d %s", i + 1, names[i]))
    .forEach(System.out::println);
```

El método `range(start, endExclusive)` devuelve otro `IntStream` y `mapToObj(mapper)` devuelve una secuencia de `String` .

Salida:

```
# 1 Jon
# 2 Darin
# 3 Bauke
# 4 Hans
# 5 Marc
```

Esto es muy similar a usar un bucle normal `for` con un contador, pero con el beneficio de la canalización y la paralelización:

```
for (int i = 0; i < names.length; i++) {
    String newName = String.format("#%d %s", i + 1, names[i]);
```

```
    System.out.println(newName);
}
```

Aplanar arroyos con mapa plano ()

Un `Stream` de elementos que a su vez se pueden transmitir puede aplanarse en un solo `Stream` continuo:

La matriz de Lista de elementos se puede convertir en una sola Lista.

```
List<String> list1 =  Arrays.asList("one", "two");
      List<String> list2 =  Arrays.asList("three","four","five");
      List<String> list3 =  Arrays.asList("six");
          List<String> finalList = Stream.of(list1, list2,
list3).flatMap(Collection::stream).collect(Collectors.toList());
System.out.println(finalList);

// [one, two, three, four, five, six]
```

El mapa que contiene la Lista de elementos como valores se puede acoplar a una lista combinada

```
Map<String, List<Integer>> map = new LinkedHashMap<>();
map.put("a", Arrays.asList(1, 2, 3));
map.put("b", Arrays.asList(4, 5, 6));

List<Integer> allValues = map.values() // Collection<List<Integer>>
        .stream()                      // Stream<List<Integer>>
        .flatMap(List::stream)         // Stream<Integer>
        .collect(Collectors.toList());

System.out.println(allValues);
// [1, 2, 3, 4, 5, 6]
```

`List` del `Map` se puede aplanar en una sola `Stream` continua

```
List<Map<String, String>> list = new ArrayList<>();
Map<String,String> map1 = new HashMap();
map1.put("1", "one");
map1.put("2", "two");

Map<String,String> map2 = new HashMap();
map2.put("3", "three");
map2.put("4", "four");
list.add(map1);
list.add(map2);


Set<String> output= list.stream()  //  Stream<Map<String, String>>
    .map(Map::values)              // Stream<List<String>>
    .flatMap(Collection::stream)   // Stream<String>
    .collect(Collectors.toSet());  //Set<String>
// [one, two, three,four]
```

Crear un mapa basado en una corriente

Caso simple sin llaves duplicadas

```
Stream<String> characters = Stream.of("A", "B", "C");

Map<Integer, String> map = characters
            .collect(Collectors.toMap(element -> element.hashCode(), element -> element));
// map = {65=A, 66=B, 67=C}
```

Para hacer las cosas más declarativa, podemos utilizar el método estático en `Function` la interfaz - `Function.identity()` . Podemos reemplazar este `element -> element` lambda `element -> element` con `Function.identity()` .

Caso donde puede haber duplicados de llaves.

El javadoc para `Collectors.toMap` establece:

> Si las claves asignadas contienen duplicados (de acuerdo con `Object.equals(Object)`), se lanza una `IllegalStateException` cuando se realiza la operación de recolección. Si las claves asignadas pueden tener duplicados, use `toMap(Function, Function, BinaryOperator)` lugar.

```
Stream<String> characters = Stream.of("A", "B", "B", "C");

Map<Integer, String> map = characters
            .collect(Collectors.toMap(
                element -> element.hashCode(),
                element -> element,
                (existingVal, newVal) -> (existingVal + newVal)));

// map = {65=A, 66=BB, 67=C}
```

El `BinaryOperator` pasado a `Collectors.toMap(...)` genera el valor que se almacenará en el caso de una colisión. Puede:

- devuelve el valor anterior, de modo que el primer valor de la secuencia tenga prioridad,
- devuelve el nuevo valor, de modo que el último valor de la secuencia tenga prioridad, o
- Combina los valores antiguos y nuevos.

Agrupación por valor

Puede usar `Collectors.groupingBy` cuando necesite realizar el equivalente de una base de datos en cascada "agrupar por" operación. Para ilustrar, lo siguiente crea un mapa en el que los nombres de las personas se asignan a los apellidos:

```
List<Person> people = Arrays.asList(
    new Person("Sam", "Rossi"),
    new Person("Sam", "Verdi"),
    new Person("John", "Bianchi"),
    new Person("John", "Rossi"),
    new Person("John", "Verdi")
);

Map<String, List<String>> map = people.stream()
```

```
        .collect(
                // function mapping input elements to keys
                Collectors.groupingBy(Person::getName,
                // function mapping input elements to values,
                // how to store values
                Collectors.mapping(Person::getSurname, Collectors.toList()))
        );

// map = {John=[Bianchi, Rossi, Verdi], Sam=[Rossi, Verdi]}
```

Vivir en Ideone

Generando cadenas aleatorias utilizando Streams

A veces es útil crear `Strings` aleatorias, tal vez como Session-ID para un servicio web o una contraseña inicial después del registro para una aplicación. Esto se puede lograr fácilmente usando `Stream` s.

Primero necesitamos inicializar un generador de números aleatorios. Para mejorar la seguridad de las `String` generadas, es una buena idea usar `SecureRandom` .

Nota : crear un `SecureRandom` es bastante costoso, por lo que es una buena práctica hacer esto solo una vez y llamar a uno de sus métodos `setSeed()` de vez en cuando para reiniciarlo.

```
private static final SecureRandom rng = new SecureRandom(SecureRandom.generateSeed(20));
//20 Bytes as a seed is rather arbitrary, it is the number used in the JavaDoc example
```

Al crear `String` aleatorias, generalmente queremos que utilicen solo ciertos caracteres (por ejemplo, solo letras y dígitos). Por lo tanto, podemos crear un método que devuelva un valor `boolean` que luego se puede usar para filtrar el `Stream` .

```
//returns true for all chars in 0-9, a-z and A-Z
boolean useThisCharacter(char c){
    //check for range to avoid using all unicode Letter (e.g. some chinese symbols)
    return c >= '0' && c <= 'z' && Character.isLetterOrDigit(c);
}
```

A continuación, podemos utilizar el RNG para generar una cadena aleatoria de longitud específica que contiene el conjunto de caracteres que pasa nuestra verificación de uso Este `useThisCharacter` .

```
public String generateRandomString(long length){
    //Since there is no native CharStream, we use an IntStream instead
    //and convert it to a Stream<Character> using mapToObj.
    //We need to specify the boundaries for the int values to ensure they can safely be cast
to char
    Stream<Character> randomCharStream = rng.ints(Character.MIN_CODE_POINT,
Character.MAX_CODE_POINT).mapToObj(i -> (char)i).filter(c ->
this::useThisCharacter).limit(length);

    //now we can use this Stream to build a String utilizing the collect method.
    String randomString = randomCharStream.collect(StringBuilder::new, StringBuilder::append,
StringBuilder::append).toString();
```

```
    return randomString;
}
```

Usando Streams para Implementar Funciones Matemáticas

`Stream` , y especialmente los `IntStream` , son una forma elegante de implementar los términos de resumen (∑). Los rangos de la `Stream` se pueden usar como los límites de la suma.

Por ejemplo, la aproximación de Madhava de Pi viene dada por la fórmula (Fuente: wikipedia):

$$\pi = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-3)^{-k}}{2k+1} = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-\frac{1}{3})^{k}}{2k+1} = \sqrt{12}\left(\frac{1}{1\cdot 3^0} - \frac{1}{3\cdot 3^1} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots\right)$$

Esto se puede calcular con una precisión arbitraria. Por ejemplo, para 101 términos:

```
double pi = Math.sqrt(12) *
            IntStream.rangeClosed(0, 100)
                     .mapToDouble(k -> Math.pow(-3, -1 * k) / (2 * k + 1))
                     .sum();
```

Nota: Con precisión de `double` , seleccionar un límite superior de 29 es suficiente para obtener un resultado que no se puede distinguir de `Math.Pi`

Uso de flujos y referencias de métodos para escribir procesos de autodocumentación

Las referencias de métodos son un excelente código de auto-documentación, y el uso de referencias de métodos con `Stream` hace que los procesos complicados sean fáciles de leer y comprender. Considere el siguiente código:

```
public interface Ordered {
    default int getOrder(){
        return 0;
    }
}

public interface Valued<V extends Ordered> {
    boolean hasPropertyTwo();
    V getValue();
}

public interface Thing<V extends Ordered> {
    boolean hasPropertyOne();
    Valued<V> getValuedProperty();
}

public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    List<V> results = new ArrayList<V>();
    for (Thing<V> thing : things) {
        if (thing.hasPropertyOne()) {
            Valued<V> valued = thing.getValuedProperty();
            if (valued != null && valued.hasPropertyTwo()){
                V value = valued.getValue();
                if (value != null){
```

```
                    results.add(value);
                }
            }
        }
    }
    results.sort((a, b)->{
        return Integer.compare(a.getOrder(), b.getOrder());
    });
    return results;
}
```

Este último método reescrito utilizando `Stream` s y las referencias de métodos es mucho más legible y cada paso del proceso se comprende rápida y fácilmente. No solo es más corto, también muestra de un vistazo qué interfaces y clases son responsables del código en cada paso:

```
public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    return things.stream()
        .filter(Thing::hasPropertyOne)
        .map(Thing::getValuedProperty)
        .filter(Objects::nonNull)
        .filter(Valued::hasPropertyTwo)
        .map(Valued::getValue)
        .filter(Objects::nonNull)
        .sorted(Comparator.comparing(Ordered::getOrder))
        .collect(Collectors.toList());
}
```

Uso de flujos de Map.Entry para conservar los valores iniciales después del mapeo

Cuando tiene un `Stream` , debe `Map.Entry<K,V>` pero también desea conservar los valores iniciales, puede mapear el `Stream` a un `Map.Entry<K,V>` usando un método de utilidad como el siguiente:

```
public static <K, V> Function<K, Map.Entry<K, V>> entryMapper(Function<K, V> mapper){
    return (k)->new AbstractMap.SimpleEntry<>(k, mapper.apply(k));
}
```

Luego, puede usar su convertidor para procesar los `Stream` tienen acceso a los valores originales y asignados:

```
Set<K> mySet;
Function<K, V> transformer = SomeClass::transformerMethod;
Stream<Map.Entry<K, V>> entryStream = mySet.stream()
    .map(entryMapper(transformer));
```

A continuación, puede continuar procesando esa `Stream` forma normal. Esto evita la sobrecarga de crear una colección intermedia.

Categorías de operaciones de flujo

Las operaciones de flujo se dividen en dos categorías principales, operaciones intermedias y terminales, y dos subcategorías, sin estado y con estado.

Operaciones intermedias:

Una operación intermedia siempre es *perezosa* , como un simple `Stream.map` . No se invoca hasta que la secuencia se consume realmente. Esto se puede verificar fácilmente:

```
Arrays.asList(1, 2 ,3).stream().map(i -> {
    throw new RuntimeException("not gonna happen");
    return i;
});
```

Las operaciones intermedias son los bloques de construcción comunes de una secuencia, encadenados después de la fuente y generalmente son seguidos por una operación de terminal que dispara la cadena de la corriente.

Terminal de Operaciones

Las operaciones de terminal son las que activan el consumo de un flujo. Algunos de los más comunes son `Stream.forEach` o `Stream.collect` . Por lo general, se colocan después de una cadena de operaciones intermedias y casi siempre están *ansiosos* .

Operaciones sin Estado

La falta de estado significa que cada artículo se procesa sin el contexto de otros artículos. Las operaciones sin estado permiten el procesamiento eficiente de la memoria de flujos. Las operaciones como `Stream.map` y `Stream.filter` que no requieren información sobre otros elementos de la transmisión se consideran sin estado.

Operaciones de estado

La notificación de estado significa que la operación en cada elemento depende de (algunos) otros elementos de la transmisión. Esto requiere un estado para ser preservado. Las operaciones de estado de estado pueden romperse con flujos largos o infinitos. Las operaciones como `Stream.sorted` requieren que la totalidad de la secuencia se procese antes de que se emita cualquier elemento que se rompa en una secuencia lo suficientemente larga. Esto se puede demostrar mediante un flujo largo (**ejecutado bajo su propio riesgo**):

```
// works - stateless stream
long BIG_ENOUGH_NUMBER = 999999999;
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).forEach(System.out::println);
```

Esto causará una falta de memoria debido a la condición de estado de `Stream.sorted` :

```
// Out of memory - stateful stream
IntStream.iterate(0, i -> i +
1).limit(BIG_ENOUGH_NUMBER).sorted().forEach(System.out::println);
```

Convertir un iterador a un flujo

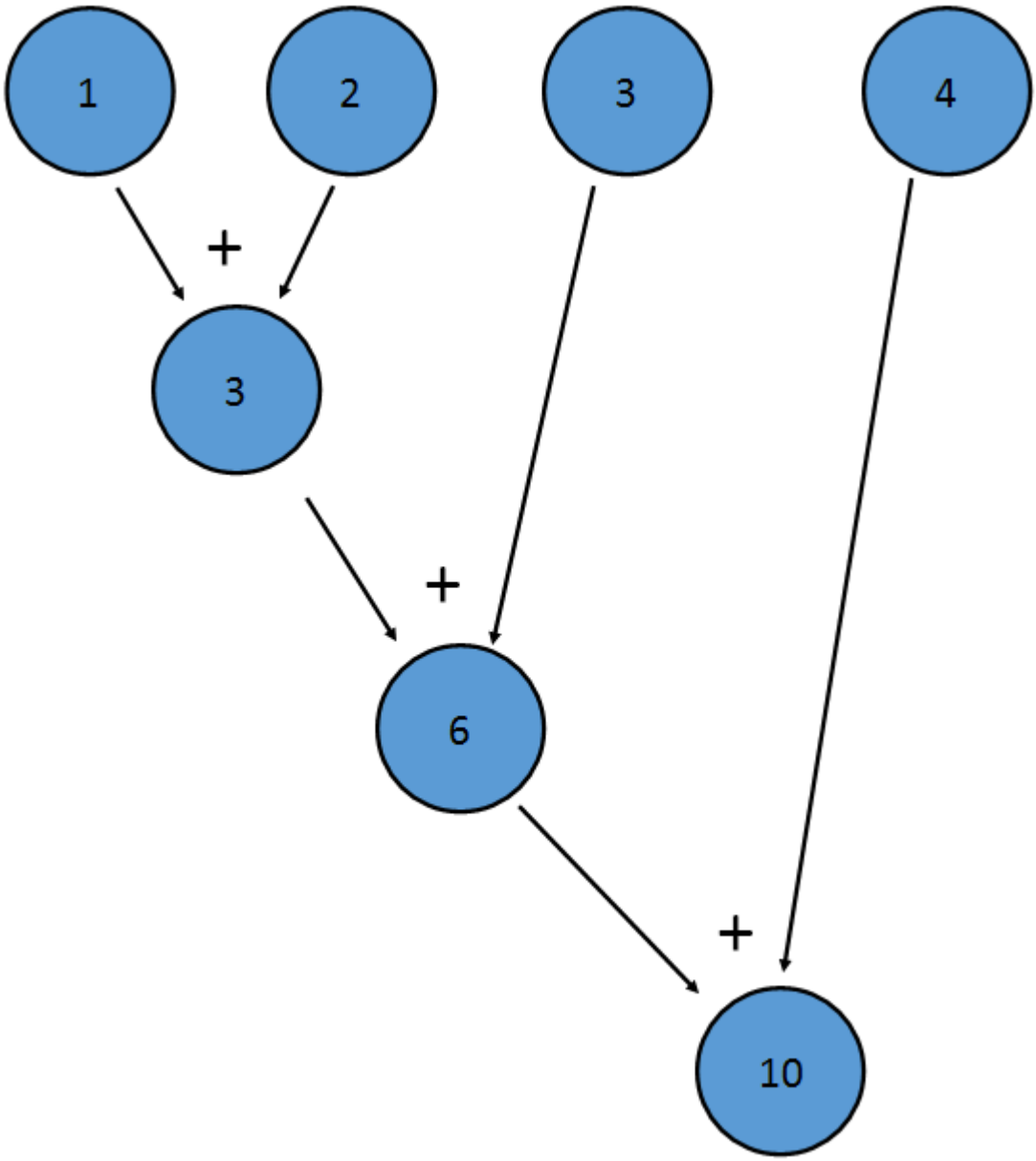
Use `Spliterators.spliterator()` o `Spliterators.spliteratorUnknownSize()` para convertir un iterador en una secuencia:

```
Iterator<String> iterator = Arrays.asList("A", "B", "C").iterator();
Spliterator<String> spliterator = Spliterators.spliteratorUnknownSize(iterator, 0);
Stream<String> stream = StreamSupport.stream(spliterator, false);
```

Reducción con arroyos

La reducción es el proceso de aplicar un operador binario a cada elemento de un flujo para obtener un valor.

El método `sum()` de un `IntStream` es un ejemplo de una reducción; se aplica la adición a cada término de la corriente, lo que resulta en un valor final:



Esto es equivalente a `(((1+2)+3)+4)`

El método de `reduce` de un flujo permite crear una reducción personalizada. Es posible utilizar el método de `reduce` para implementar el método `sum()` :

```
IntStream istr;

//Initialize istr

OptionalInt istr.reduce((a,b)->a+b);
```

La versión `Optional` se devuelve para que las secuencias vacías se puedan manejar adecuadamente.

Otro ejemplo de reducción es la combinación de un `Stream<LinkedList<T>>` en un solo `LinkedList<T>`

:

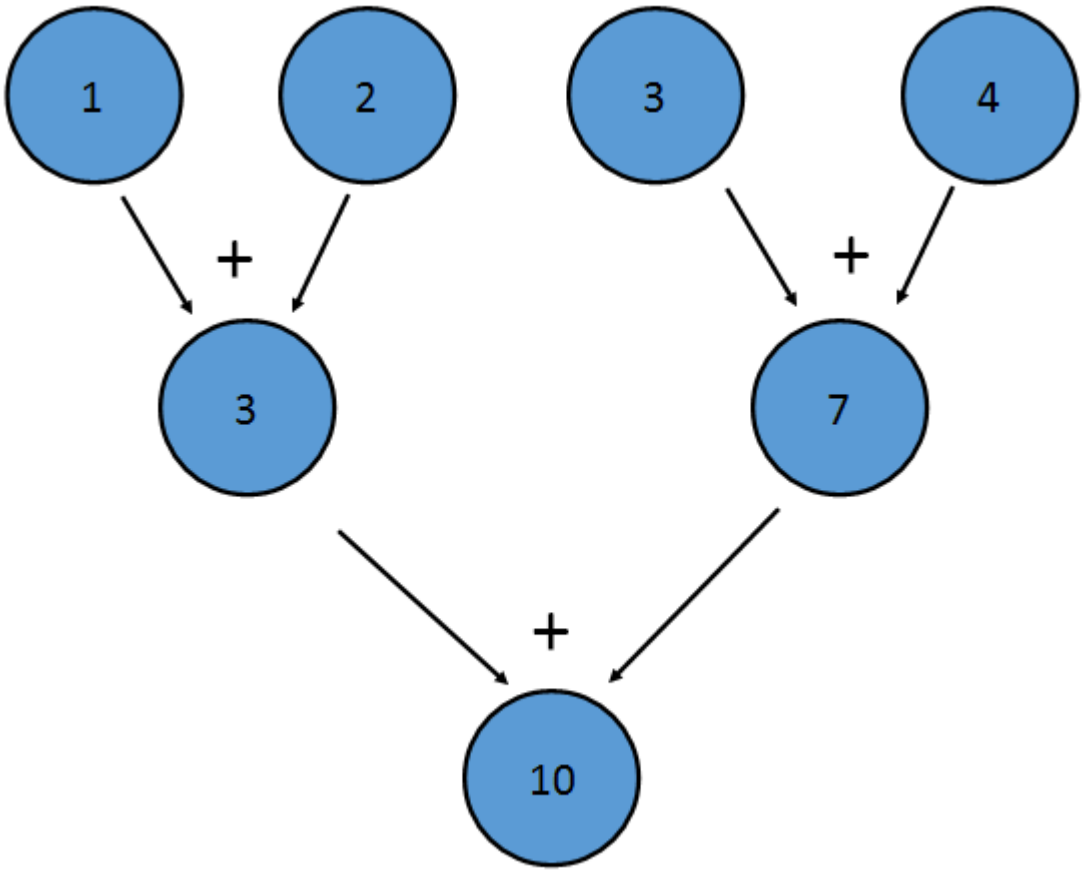
```
Stream<LinkedList<T>> listStream;

//Create a Stream<LinkedList<T>>

Optional<LinkedList<T>> bigList = listStream.reduce((LinkedList<T> list1, LinkedList<T>
list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

También puede proporcionar un *elemento de identidad* . Por ejemplo, el elemento de identidad para la adición es 0, como `x+0==x` . Para la multiplicación, el elemento de identidad es 1, como `x*1==x` . En el caso anterior, el elemento de identidad es una lista `LinkedList<T>` vacía `LinkedList<T>` , porque si agrega una lista vacía a otra lista, la lista a la que está "agregando" no cambia:

```
Stream<LinkedList<T>> listStream;

//Create a Stream<LinkedList<T>>

LinkedList<T> bigList = listStream.reduce(new LinkedList<T>(), (LinkedList<T> list1,
LinkedList<T> list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

Tenga en cuenta que cuando se proporciona un elemento de identidad, el valor de retorno no se ajusta en un `Optional` si se llama en una secuencia vacía, `reduce()` devolverá el elemento de identidad.

El operador binario también debe ser *asociativo* , lo que significa que `(a+b)+c==a+(b+c)` . Esto se debe a que los elementos pueden reducirse en cualquier orden. Por ejemplo, la reducción de adición anterior se podría realizar de la siguiente manera:



Esta reducción es equivalente a la escritura `((1+2)+(3+4))` . La propiedad de asociatividad también permite que Java reduzca el `Stream` en paralelo; cada procesador puede reducir una parte del flujo, con una reducción que combina el resultado de cada procesador al final.

Unir un flujo a una sola cadena

Un caso de uso que aparece con frecuencia es crear una `String` partir de una secuencia, donde los elementos de la secuencia están separados por un determinado carácter. El método `Collectors.joining()` se puede usar para esto, como en el siguiente ejemplo:

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

String result = fruitStream.filter(s -> s.contains("a"))
           .map(String::toUpperCase)
           .sorted()
           .collect(Collectors.joining(", "));

System.out.println(result);
```

Salida:

> Manzana, plátano, naranja, pera.

El método `Collectors.joining()` también puede atender pre y postfixes:

```
String result = fruitStream.filter(s -> s.contains("e"))
           .map(String::toUpperCase)
           .sorted()
           .collect(Collectors.joining(", ", "Fruits: ", "."));

System.out.println(result);
```

Salida:

> Frutos: MANZANA, NARANJA, PERA.

Vivir en Ideone

Lea Corrientes en línea: https://riptutorial.com/es/java/topic/88/corrientes

Capítulo 52: Creando imágenes programáticamente

Observaciones

`BufferedImage.getGraphics()` siempre devuelve `Graphics2D` .

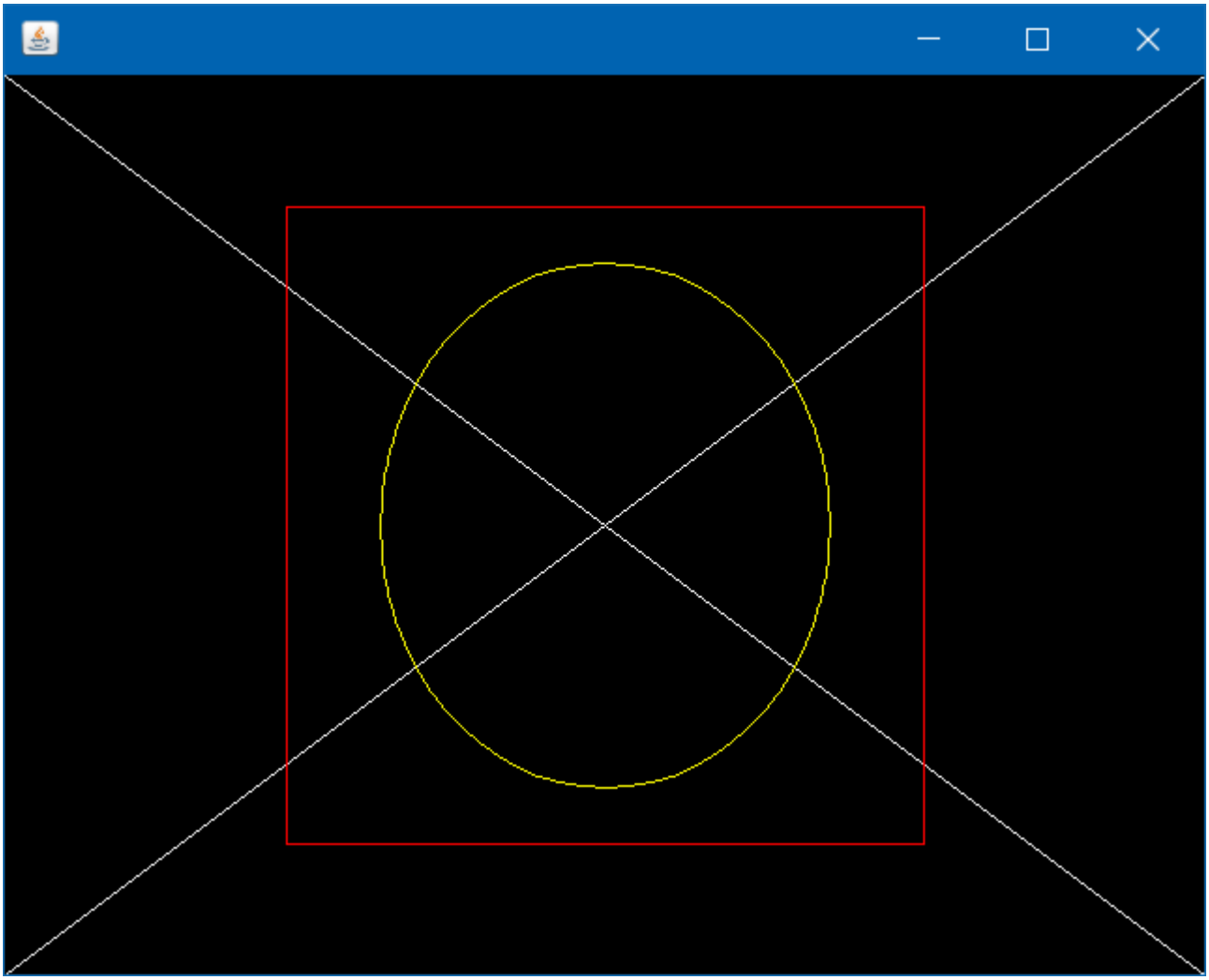
El uso de `VolatileImage` puede mejorar significativamente la velocidad de las operaciones de dibujo, pero también tiene sus inconvenientes: su contenido puede perderse en cualquier momento y es posible que deba volver a dibujarse desde cero.

Examples

Creando una imagen simple programáticamente y mostrándola.

```
class ImageCreationExample {

    static Image createSampleImage() {
        // instantiate a new BufferedImage (subclass of Image) instance
        BufferedImage img = new BufferedImage(640, 480, BufferedImage.TYPE_INT_ARGB);

        //draw something on the image
        paintOnImage(img);

        return img;
    }

    static void paintOnImage(BufferedImage img) {
        // get a drawable Graphics2D (subclass of Graphics) object
        Graphics2D g2d = (Graphics2D) img.getGraphics();

        // some sample drawing
        g2d.setColor(Color.BLACK);
        g2d.fillRect(0, 0, 640, 480);
        g2d.setColor(Color.WHITE);
        g2d.drawLine(0, 0, 640, 480);
        g2d.drawLine(0, 480, 640, 0);
        g2d.setColor(Color.YELLOW);
        g2d.drawOval(200, 100, 240, 280);
        g2d.setColor(Color.RED);
        g2d.drawRect(150, 70, 340, 340);

        // drawing on images can be very memory-consuming
        // so it's better to free resources early
        // it's not necessary, though
        g2d.dispose();
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Image img = createSampleImage();
```

```
        ImageIcon icon = new ImageIcon(img);
        frame.add(new JLabel(icon));
        frame.pack();
        frame.setVisible(true);
    }
}
```



Guardar una imagen en el disco

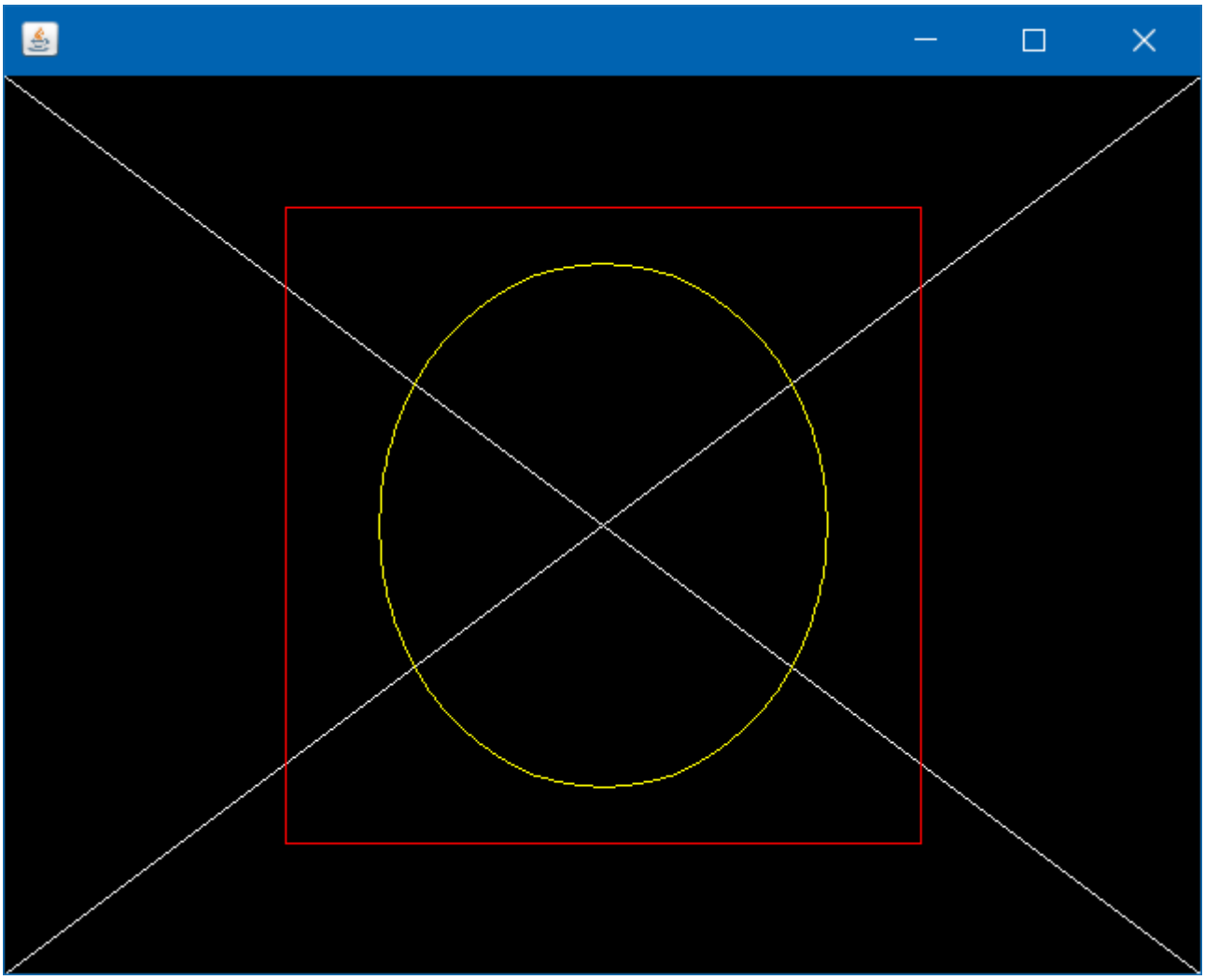
```
public static void saveImage(String destination) throws IOException {
    // method implemented in "Creating a simple image Programmatically and displaying it"
example
    BufferedImage img = createSampleImage();

    // ImageIO provides several write methods with different outputs
    ImageIO.write(img, "png", new File(destination));
}
```

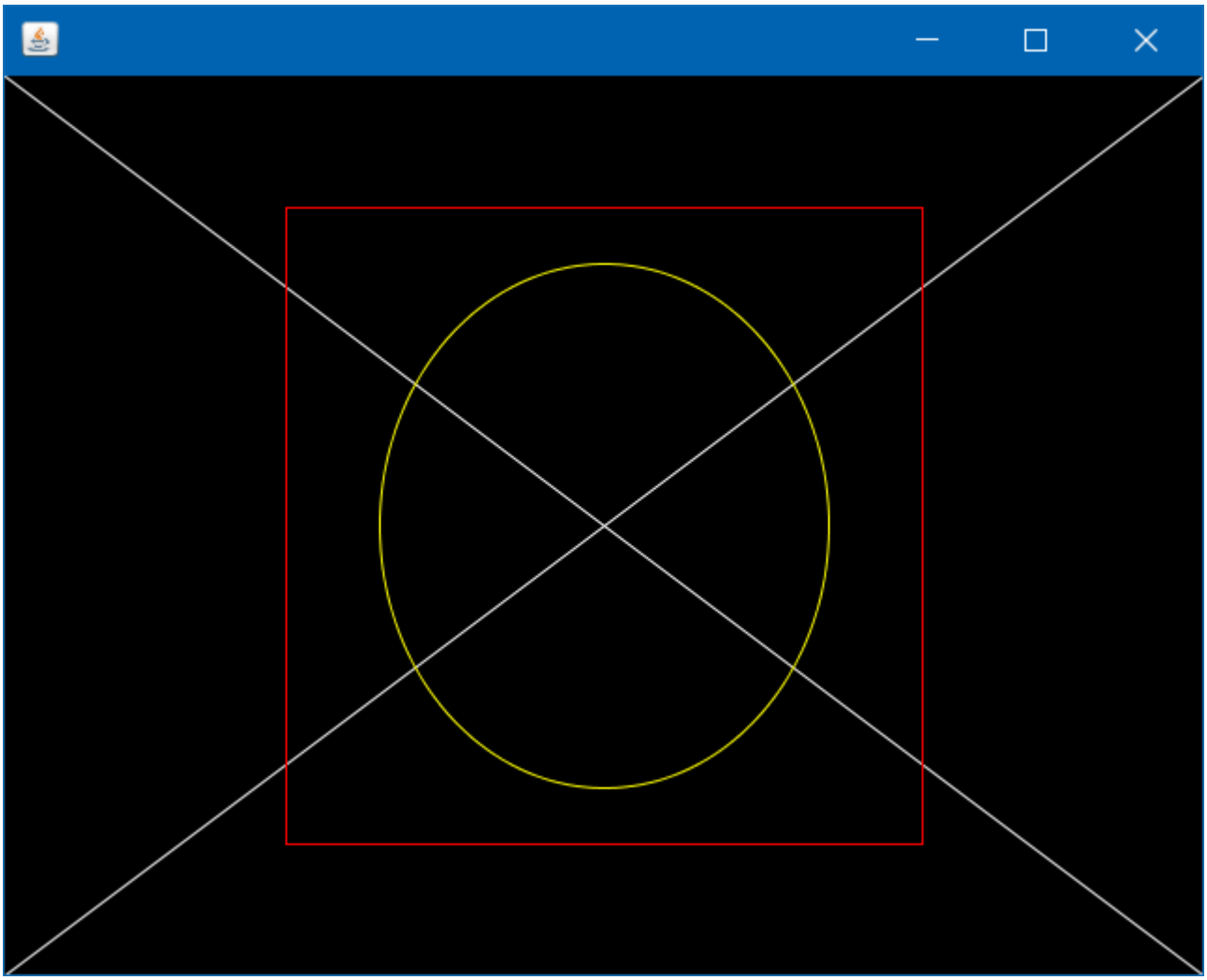
Especificando la calidad de renderizado de la imagen.

```
static void setupQualityHigh(Graphics2D g2d) {
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_QUALITY);
    // many other RenderingHints KEY/VALUE pairs to specify
```

```
}


static void setupQualityLow(Graphics2D g2d) {
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_OFF);
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_SPEED);
}
```

Una comparación de la representación de CALIDAD y VELOCIDAD de la imagen de muestra:

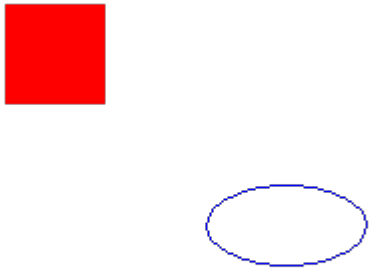




Creando una imagen con la clase BufferedImage

```
int width = 256; //in pixels
int height = 256; //in pixels
BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_4BYTE_ABGR);
//BufferedImage.TYPE_4BYTE_ABGR - store RGB color and visibility (alpha), see javadoc for more
info

Graphics g = image.createGraphics();

//draw whatever you like, like you would in a drawComponent(Graphics g) method in an UI
application
g.setColor(Color.RED);
g.fillRect(20, 30, 50, 50);

g.setColor(Color.BLUE);
g.drawOval(120, 120, 80, 40);

g.dispose(); //dispose graphics objects when they are no longer needed

//now image has programmatically generated content, you can use it in graphics.drawImage() to
draw it somewhere else
//or just simply save it to a file
ImageIO.write(image, "png", new File("myimage.png"));
```

Salida:



Edición y reutilización de imágenes con BufferedImage.

```
BufferedImage cat = ImageIO.read(new File("cat.jpg")); //read existing file

//modify it
Graphics g = cat.createGraphics();
g.setColor(Color.RED);
g.drawString("Cat", 10, 10);
g.dispose();

//now create a new image
BufferedImage cats = new BufferedImage(256, 256, BufferedImage.TYPE_4BYTE_ABGR);

//and draw the old one on it, 16 times
g = cats.createGraphics();
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        g.drawImage(cat, i * 64, j * 64, null);
    }
}

g.setColor(Color.BLUE);
g.drawRect(0, 0, 255, 255); //add some nice border
g.dispose(); //and done

ImageIO.write(cats, "png", new File("cats.png"));
```

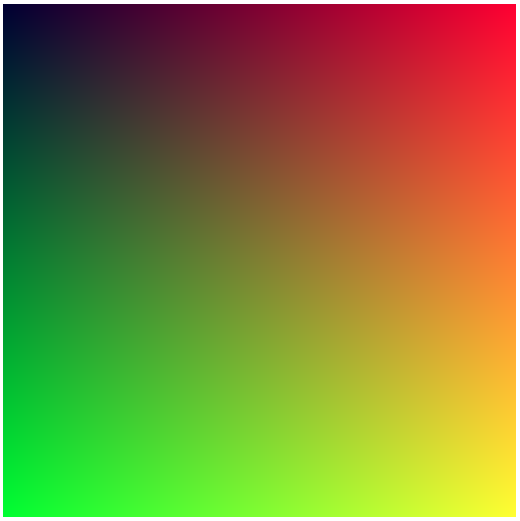
Archivo cat original:

Archivo producido:

Configuración del color de píxel individual en BufferedImage

```
BufferedImage image = new BufferedImage(256, 256, BufferedImage.TYPE_INT_ARGB);

//you don't have to use the Graphics object, you can read and set pixel color individually
for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 256; j++) {
        int alpha = 255; //don't forget this, or use BufferedImage.TYPE_INT_RGB instead
        int red = i; //or any formula you like
        int green = j; //or any formula you like
        int blue = 50; //or any formula you like
        int color = (alpha << 24) | (red << 16) | (green << 8) | blue;
        image.setRGB(i, j, color);
    }
}

ImageIO.write(image, "png", new File("computed.png"));
```

Salida:



Cómo escalar una imagen almacenada

```
/**
* Resizes an image using a Graphics2D object backed by a BufferedImage.
```

```
 * @param srcImg - source image to scale
 * @param w - desired width
 * @param h - desired height
 * @return - the new resized image
 */
private BufferedImage getScaledImage(Image srcImg, int w, int h){

    //Create a new image with good size that contains or might contain arbitrary alpha values
between and including 0.0 and 1.0.
    BufferedImage resizedImg = new BufferedImage(w, h, BufferedImage.TRANSLUCENT);

    //Create a device-independant object to draw the resized image
    Graphics2D g2 = resizedImg.createGraphics();

    //This could be changed, Cf. http://stackoverflow.com/documentation/java/5482/creating-
images-programmatically/19498/specifying-image-rendering-quality
    g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
RenderingHints.VALUE_INTERPOLATION_BILINEAR);

    //Finally draw the source image in the Graphics2D with the desired size.
    g2.drawImage(srcImg, 0, 0, w, h, null);

    //Disposes of this graphics context and releases any system resources that it is using
    g2.dispose();

    //Return the image used to create the Graphics2D
    return resizedImg;
}
```

Lea Creando imágenes programáticamente en línea:
https://riptutorial.com/es/java/topic/5482/creando-imagenes-programaticamente

Capítulo 53: Desmontaje y Descompilación.

Sintaxis

- javap [opciones] <clases>

Parámetros

Nombre	Descripción
`<classes>`	Lista de clases para desmontar. Puede estar en el formato `package1.package2.Classname` , *O* `package1/package2/Classname` . **No** incluya el `.class` extensión.
`-help` , `--help` , `-?`	Imprimir este mensaje de uso
`-version`	Información de versión
`-v` , `-verbose`	Imprimir información adicional
`-l`	Imprimir número de línea y tablas de variables locales
`-public`	Mostrar solo clases públicas y miembros
`-protected`	Mostrar clases protegidas / públicas y miembros
`-package`	Mostrar paquete / clases protegidas / públicas y miembros (predeterminado)
`-p` , `-private`	Mostrar todas las clases y miembros
`-c`	Desmontar el codigo
`-s`	Imprimir firmas de tipo interno
`-sysinfo`	Mostrar información del sistema (ruta, tamaño, fecha, hash MD5) de la clase en proceso
`-constants`	Mostrar constantes finales
`-classpath <path>`	Especifique dónde encontrar los archivos de clase de usuario
`-cp <path>`	Especifique dónde encontrar los archivos de clase de usuario
`-bootclasspath <path>`	Anular la ubicación de los archivos de clase de arranque

Examples

Viendo bytecode con javap

Si desea ver el código de bytes generado para un programa Java, puede usar el comando `javap` proporcionado para verlo.

Suponiendo que tenemos el siguiente archivo fuente de Java:

```
package com.stackoverflow.documentation;

import org.springframework.stereotype.Service;

import java.io.IOException;
import java.io.InputStream;
import java.util.List;

@Service
public class HelloWorldService {

    public void sayHello() {
        System.out.println("Hello, World!");
    }

    private Object[] pvtMethod(List<String> strings) {
        return new Object[]{strings};
    }

    protected String tryCatchResources(String filename) throws IOException {
        try (InputStream inputStream = getClass().getResourceAsStream(filename)) {
            byte[] bytes = new byte[8192];
            int read = inputStream.read(bytes);
            return new String(bytes, 0, read);
        } catch (IOException | RuntimeException e) {
            e.printStackTrace();
            throw e;
        }
    }

    void stuff() {
        System.out.println("stuff");
    }
}
```

Después de compilar el archivo fuente, el uso más simple es:

```
cd <directory containing classes> (e.g. target/classes)
javap com/stackoverflow/documentation/SpringExample
```

Lo que produce la salida.

```
Compiled from "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService {
  public com.stackoverflow.documentation.HelloWorldService();
  public void sayHello();
```

```
  protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
  void stuff();
}
```

Esto enumera todos los métodos no privados en la clase, pero eso no es particularmente útil para la mayoría de los propósitos. El siguiente comando es mucho más útil:

```
javap -p -c -s -constants -l -v com/stackoverflow/documentation/HelloWorldService
```

Lo que produce la salida:

```
Classfile /Users/pivotal/IdeaProjects/stackoverflow-spring-
docs/target/classes/com/stackoverflow/documentation/HelloWorldService.class
  Last modified Jul 22, 2016; size 2167 bytes
  MD5 checksum 6e33b5c292ead21701906353b7f06330
  Compiled from "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService
  minor version: 0
  major version: 51
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
    #1 = Methodref          #5.#60        // java/lang/Object."<init>":()V
    #2 = Fieldref           #61.#62       // java/lang/System.out:Ljava/io/PrintStream;
    #3 = String             #63           // Hello, World!
    #4 = Methodref          #64.#65       // java/io/PrintStream.println:(Ljava/lang/String;)V
    #5 = Class              #66           // java/lang/Object
    #6 = Methodref          #5.#67        // java/lang/Object.getClass:()Ljava/lang/Class;
    #7 = Methodref          #68.#69       //
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
    #8 = Methodref          #70.#71       // java/io/InputStream.read:([B)I
    #9 = Class              #72           // java/lang/String
   #10 = Methodref          #9.#73        // java/lang/String."<init>":([BII)V
   #11 = Methodref          #70.#74       // java/io/InputStream.close:()V
   #12 = Class              #75           // java/lang/Throwable
   #13 = Methodref          #12.#76       //
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
   #14 = Class              #77           // java/io/IOException
   #15 = Class              #78           // java/lang/RuntimeException
   #16 = Methodref          #79.#80       // java/lang/Exception.printStackTrace:()V
   #17 = String             #55           // stuff
   #18 = Class              #81           // com/stackoverflow/documentation/HelloWorldService
   #19 = Utf8               <init>
   #20 = Utf8               ()V
   #21 = Utf8               Code
   #22 = Utf8               LineNumberTable
   #23 = Utf8               LocalVariableTable
   #24 = Utf8               this
   #25 = Utf8               Lcom/stackoverflow/documentation/HelloWorldService;
   #26 = Utf8               sayHello
   #27 = Utf8               pvtMethod
   #28 = Utf8               (Ljava/util/List;)[Ljava/lang/Object;
   #29 = Utf8               strings
   #30 = Utf8               Ljava/util/List;
   #31 = Utf8               LocalVariableTypeTable
   #32 = Utf8               Ljava/util/List<Ljava/lang/String;>;
   #33 = Utf8               Signature
   #34 = Utf8               (Ljava/util/List<Ljava/lang/String;>;)[Ljava/lang/Object;
   #35 = Utf8               tryCatchResources
```

```
   #36 = Utf8               (Ljava/lang/String;)Ljava/lang/String;
   #37 = Utf8               bytes
   #38 = Utf8               [B
   #39 = Utf8               read
   #40 = Utf8               I
   #41 = Utf8               inputStream
   #42 = Utf8               Ljava/io/InputStream;
   #43 = Utf8               e
   #44 = Utf8               Ljava/lang/Exception;
   #45 = Utf8               filename
   #46 = Utf8               Ljava/lang/String;
   #47 = Utf8               StackMapTable
   #48 = Class              #81           // com/stackoverflow/documentation/HelloWorldService
   #49 = Class              #72           // java/lang/String
   #50 = Class              #82           // java/io/InputStream
   #51 = Class              #75           // java/lang/Throwable
   #52 = Class              #38           // "[B"
   #53 = Class              #83           // java/lang/Exception
   #54 = Utf8               Exceptions
   #55 = Utf8               stuff
   #56 = Utf8               SourceFile
   #57 = Utf8               HelloWorldService.java
   #58 = Utf8               RuntimeVisibleAnnotations
   #59 = Utf8               Lorg/springframework/stereotype/Service;
   #60 = NameAndType        #19:#20       // "<init>":()V
   #61 = Class              #84           // java/lang/System
   #62 = NameAndType        #85:#86       // out:Ljava/io/PrintStream;
   #63 = Utf8               Hello, World!
   #64 = Class              #87           // java/io/PrintStream
   #65 = NameAndType        #88:#89       // println:(Ljava/lang/String;)V
   #66 = Utf8               java/lang/Object
   #67 = NameAndType        #90:#91       // getClass:()Ljava/lang/Class;
   #68 = Class              #92           // java/lang/Class
   #69 = NameAndType        #93:#94       //
getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
   #70 = Class              #82           // java/io/InputStream
   #71 = NameAndType        #39:#95       // read:([B)I
   #72 = Utf8               java/lang/String
   #73 = NameAndType        #19:#96       // "<init>":([BII)V
   #74 = NameAndType        #97:#20       // close:()V
   #75 = Utf8               java/lang/Throwable
   #76 = NameAndType        #98:#99       // addSuppressed:(Ljava/lang/Throwable;)V
   #77 = Utf8               java/io/IOException
   #78 = Utf8               java/lang/RuntimeException
   #79 = Class              #83           // java/lang/Exception
   #80 = NameAndType        #100:#20      // printStackTrace:()V
   #81 = Utf8               com/stackoverflow/documentation/HelloWorldService
   #82 = Utf8               java/io/InputStream
   #83 = Utf8               java/lang/Exception
   #84 = Utf8               java/lang/System
   #85 = Utf8               out
   #86 = Utf8               Ljava/io/PrintStream;
   #87 = Utf8               java/io/PrintStream
   #88 = Utf8               println
   #89 = Utf8               (Ljava/lang/String;)V
   #90 = Utf8               getClass
   #91 = Utf8               ()Ljava/lang/Class;
   #92 = Utf8               java/lang/Class
   #93 = Utf8               getResourceAsStream
   #94 = Utf8               (Ljava/lang/String;)Ljava/io/InputStream;
   #95 = Utf8               ([B)I
```

```
   #96 = Utf8               ([BII)V
   #97 = Utf8               close
   #98 = Utf8               addSuppressed
   #99 = Utf8               (Ljava/lang/Throwable;)V
  #100 = Utf8               printStackTrace
{
  public com.stackoverflow.documentation.HelloWorldService();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 10: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/stackoverflow/documentation/HelloWorldService;

  public void sayHello();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3                  // String Hello, World!
         5: invokevirtual #4                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 13: 0
        line 14: 8
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  this   Lcom/stackoverflow/documentation/HelloWorldService;

  private java.lang.Object[] pvtMethod(java.util.List<java.lang.String>);
    descriptor: (Ljava/util/List;)[Ljava/lang/Object;
    flags: ACC_PRIVATE
    Code:
      stack=4, locals=2, args_size=2
         0: iconst_1
         1: anewarray     #5                  // class java/lang/Object
         4: dup
         5: iconst_0
         6: aload_1
         7: aastore
         8: areturn
      LineNumberTable:
        line 17: 0
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  this   Lcom/stackoverflow/documentation/HelloWorldService;
            0       9     1 strings   Ljava/util/List;
      LocalVariableTypeTable:
        Start  Length  Slot  Name   Signature
            0       9     1 strings   Ljava/util/List<Ljava/lang/String;>;
    Signature: #34                          //
(Ljava/util/List<Ljava/lang/String;>;)[Ljava/lang/Object;
```

```
  protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
    descriptor: (Ljava/lang/String;)Ljava/lang/String;
    flags: ACC_PROTECTED
    Code:
      stack=5, locals=10, args_size=2
         0: aload_0
         1: invokevirtual #6                  // Method
java/lang/Object.getClass:()Ljava/lang/Class;
         4: aload_1
         5: invokevirtual #7                  // Method
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
         8: astore_2
         9: aconst_null
        10: astore_3
        11: sipush        8192
        14: newarray       byte
        16: astore        4
        18: aload_2
        19: aload         4
        21: invokevirtual #8                  // Method java/io/InputStream.read:([B)I
        24: istore        5
        26: new           #9                  // class java/lang/String
        29: dup
        30: aload         4
        32: iconst_0
        33: iload         5
        35: invokespecial #10                 // Method java/lang/String."<init>":([BII)V
        38: astore        6
        40: aload_2
        41: ifnull        70
        44: aload_3
        45: ifnull        66
        48: aload_2
        49: invokevirtual #11                 // Method java/io/InputStream.close:()V
        52: goto          70
        55: astore        7
        57: aload_3
        58: aload         7
        60: invokevirtual #13                 // Method
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
        63: goto          70
        66: aload_2
        67: invokevirtual #11                 // Method java/io/InputStream.close:()V
        70: aload         6
        72: areturn
        73: astore        4
        75: aload         4
        77: astore_3
        78: aload         4
        80: athrow
        81: astore        8
        83: aload_2
        84: ifnull        113
        87: aload_3
        88: ifnull        109
        91: aload_2
        92: invokevirtual #11                 // Method java/io/InputStream.close:()V
        95: goto          113
        98: astore        9
       100: aload_3
```

```
       101: aload         9
       103: invokevirtual #13                 // Method
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
       106: goto          113
       109: aload_2
       110: invokevirtual #11                 // Method java/io/InputStream.close:()V
       113: aload         8
       115: athrow
       116: astore_2
       117: aload_2
       118: invokevirtual #16                 // Method
java/lang/Exception.printStackTrace:()V
       121: aload_2
       122: athrow
      Exception table:
         from    to  target type
            48    52    55   Class java/lang/Throwable
            11    40    73   Class java/lang/Throwable
            11    40    81   any
            91    95    98   Class java/lang/Throwable
            73    83    81   any
             0    70   116   Class java/io/IOException
             0    70   116   Class java/lang/RuntimeException
            73   116   116   Class java/io/IOException
            73   116   116   Class java/lang/RuntimeException
      LineNumberTable:
        line 21: 0
        line 22: 11
        line 23: 18
        line 24: 26
        line 25: 40
        line 21: 73
        line 25: 81
        line 26: 117
        line 27: 121
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
           18      55     4 bytes   [B
           26      47     5  read   I
            9     107     2 inputStream   Ljava/io/InputStream;
          117       6     2     e   Ljava/lang/Exception;
            0     123     0  this   Lcom/stackoverflow/documentation/HelloWorldService;
            0     123     1 filename   Ljava/lang/String;
      StackMapTable: number_of_entries = 9
        frame_type = 255 /* full_frame */
          offset_delta = 55
          locals = [ class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String, class java/io/InputStream, class java/lang/Throwable, class "[B", int, class
java/lang/String ]
          stack = [ class java/lang/Throwable ]
        frame_type = 10 /* same */
        frame_type = 3 /* same */
        frame_type = 255 /* full_frame */
          offset_delta = 2
          locals = [ class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String, class java/io/InputStream, class java/lang/Throwable ]
          stack = [ class java/lang/Throwable ]
        frame_type = 71 /* same_locals_1_stack_item */
          stack = [ class java/lang/Throwable ]
        frame_type = 255 /* full_frame */
          offset_delta = 16
```

```
          locals = [ class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String, class java/io/InputStream, class java/lang/Throwable, top, top, top, top,
class java/lang/Throwable ]
          stack = [ class java/lang/Throwable ]
        frame_type = 10 /* same */
        frame_type = 3 /* same */
        frame_type = 255 /* full_frame */
          offset_delta = 2
          locals = [ class com/stackoverflow/documentation/HelloWorldService, class
java/lang/String ]
          stack = [ class java/lang/Exception ]
    Exceptions:
      throws java.io.IOException

  void stuff();
    descriptor: ()V
    flags:
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #17                 // String stuff
         5: invokevirtual #4                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 32: 0
        line 33: 8
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  this   Lcom/stackoverflow/documentation/HelloWorldService;
}
SourceFile: "HelloWorldService.java"
RuntimeVisibleAnnotations:
  0: #59()
```

Lea Desmontaje y Descompilación. en línea: https://riptutorial.com/es/java/topic/2318/desmontaje-y-descompilacion-

Capítulo 54: Despliegue de Java

Introducción

Existe una variedad de tecnologías para "empaquetar" aplicaciones Java, aplicaciones web, etc., para su implementación en la plataforma en la que se ejecutarán. Abarcan desde bibliotecas simples o archivos `JAR` ejecutables, archivos `WAR` y `EAR` , hasta instaladores y ejecutables independientes.

Observaciones

En el nivel más fundamental, un programa Java puede implementarse copiando una clase compilada (es decir, un archivo ".class") o un árbol de directorios que contiene clases compiladas. Sin embargo, Java normalmente se implementa de una de las siguientes maneras:

- Al copiar un archivo JAR o una colección de archivos JAR en el sistema donde se ejecutarán; por ejemplo, utilizando `javac` .
- Al copiar o cargar un archivo WAR, EAR o similar en un "contenedor de servlets" o "servidor de aplicaciones".
- Al ejecutar algún tipo de instalador de aplicaciones que automatiza lo anterior. El instalador también puede instalar un JRE incorporado.
- Al colocar los archivos JAR para la aplicación en un servidor web para permitir que se inicien utilizando Java WebStart.

El ejemplo de Creación de archivos JAR, WAR y EAR resume las diferentes formas de crear estos archivos.

Existen numerosas herramientas de código abierto y comercial "generador de instalador" y "generador EXE" para Java. De manera similar, existen herramientas para ofuscar archivos de clase Java (para dificultar la ingeniería inversa) y para agregar la verificación de licencias en tiempo de ejecución. Todo esto está fuera del alcance de la documentación del "Lenguaje de programación Java".

Examples

Haciendo un JAR ejecutable desde la línea de comando

Para hacer un jar, necesita uno o más archivos de clase. Esto debería tener un método principal si se va a ejecutar haciendo doble clic.

Para este ejemplo, utilizaremos:

```
import javax.swing.*;
import java.awt.Container;

public class HelloWorld {

    public static void main(String[] args) {
        JFrame f = new JFrame("Hello, World");
        JLabel label = new JLabel("Hello, World");
        Container cont = f.getContentPane();
        cont.add(label);
        f.setSize(400,100);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

}
```

Ha sido nombrado HelloWorld.java

A continuación, queremos compilar este programa.

Puede utilizar cualquier programa que desee hacer esto. Para ejecutar desde la línea de comandos, consulte la documentación sobre cómo compilar y ejecutar su primer programa java.

Una vez que tengas HelloWorld.class, crea una nueva carpeta y llámala como quieras.

Haz otro archivo llamado manifest.txt y pégalo en él.

```
Main-Class: HelloWorld
Class-Path: HelloWorld.jar
```

Ponlo en la misma carpeta con HelloWorld.class
Use la línea de comando para hacer de su directorio actual (`cd C:\Your\Folder\Path\Here` en windows) su carpeta.

Use Terminal y cambie el directorio al directorio (`cd /Users/user/Documents/Java/jarfolder` en Mac) su carpeta

Cuando haya terminado, escriba `jar -cvfm HelloWorld.jar manifest.txt HelloWorld.class` y presione Entrar. Esto crea un archivo jar (en la carpeta con su manifiesto y HelloWorld.class) utilizando los archivos .class especificados y llamados HelloWorld.jar. Consulte la sección Sintaxis para obtener información sobre las opciones (como -m y -v).
Después de estos pasos, vaya a su directorio con el archivo de manifiesto y debería encontrar HelloWorld.jar
Al hacer clic en él debería aparecer `Hello, World` en un cuadro de texto.

Creando archivos JAR, WAR y EAR

Los tipos de archivos JAR, WAR y EAR son fundamentalmente archivos ZIP con un archivo "manifiesto" y (para archivos WAR y EAR) un directorio / estructura interna particular.

La forma recomendada de crear estos archivos es utilizar una herramienta de compilación

específica de Java que "entienda" los requisitos para los tipos de archivo respectivos. Si no usa una herramienta de compilación, la opción siguiente es intentar "exportar" al IDE.

(*Nota editorial: las descripciones de cómo crear estos archivos se ubican mejor en la documentación de las herramientas respectivas. Póngalas allí. ¡Por favor, muestre un poco de autocontrol y NO las calce en este ejemplo!*)

Creando archivos JAR y WAR usando Maven

Crear un JAR o WAR utilizando Maven es simplemente una cuestión de colocar el elemento `<packaging>` correcto en el archivo POM; p.ej,

```
<packaging>jar</packaging>
```

o

```
<packaging>war</packaging>
```

Para más detalles. Maven se puede configurar para crear archivos JAR "ejecutables" agregando la información necesaria sobre la clase de punto de entrada y las dependencias externas como propiedades de plugin para el plugin de maven jar. Incluso hay un complemento para crear archivos "uberJAR" que combinan una aplicación y sus dependencias en un solo archivo JAR.

Consulte la documentación de Maven (http://www.riptutorial.com/topic/898) para obtener más información.

Creando archivos JAR, WAR y EAR usando Ant

La herramienta de compilación Ant tiene "tareas" separadas para construir JAR, WAR y EAR. Consulte la documentación de Ant (http://www.riptutorial.com/topic/4223) para obtener más información.

Creando archivos JAR, WAR y EAR usando un IDE

Los tres IDE de Java más populares tienen soporte incorporado para crear archivos de implementación. La funcionalidad a menudo se describe como "exportar".

- Eclipse - http://www.riptutorial.com/topic/1143
- NetBeans - http://www.riptutorial.com/topic/5438
- Intellij-IDEA - Exportando

Creando archivos JAR, WAR y EAR usando el comando `jar` .

También es posible crear estos archivos "a mano" usando el comando `jar` . Es simplemente una cuestión de ensamblar un árbol de archivos con los archivos del componente correcto en el lugar correcto, crear un archivo de manifiesto y ejecutar `jar` para crear el archivo JAR.

Consulte el tema del comando `jar` (Crear y modificar archivos JAR) para obtener más información.

Introducción a Java Web Start

Los tutoriales de Oracle Java resumen Web Start de la siguiente manera:

> El software Java Web Start brinda el poder de lanzar aplicaciones completas con un solo clic. Los usuarios pueden descargar e iniciar aplicaciones, como un programa completo de hoja de cálculo o un cliente de chat de Internet, sin pasar por largos procedimientos de instalación.

Otras ventajas de Java Web Start son la compatibilidad con el código firmado y la declaración explícita de las dependencias de la plataforma, y la compatibilidad con el almacenamiento en caché de códigos y la implementación de actualizaciones de aplicaciones.

Java Web Start también se conoce como JavaWS y JAWS. Las principales fuentes de información son:

- Los tutoriales de Java - Lección: Java Web Start
- Guía de inicio de Java Web
- Preguntas frecuentes sobre Java Web Start
- Especificación JNLP
- Documentación API `javax.jnlp`
- Sitio de desarrolladores de Java Web Start

Prerrequisitos

En un nivel alto, Web Start funciona distribuyendo aplicaciones Java empaquetadas como archivos JAR desde un servidor web remoto. Los requisitos previos son:

- Una instalación Java preexistente (JRE o JDK) en la máquina de destino donde se ejecutará la aplicación. Se requiere Java 1.2.2 o superior:
  - Desde Java 5.0 en adelante, el soporte de Web Start se incluye en el JRE / JDK.
  - Para versiones anteriores, el soporte de Web Start se instala por separado.
  - La infraestructura de Web Start incluye algunos Javascript que se pueden incluir en una página web para ayudar al usuario a instalar el software necesario.
- El servidor web que aloja el software debe estar accesible para la máquina de destino.
- Si el usuario va a iniciar una aplicación Web Start utilizando un enlace en una página web, entonces:
  - necesitan un navegador web compatible, y
  - para los navegadores modernos (seguros), se les debe decir cómo decirle al navegador que permita que Java se ejecute ... sin comprometer la seguridad del navegador web.

Un ejemplo de archivo JNLP

El siguiente ejemplo pretende ilustrar la funcionalidad básica de JNLP.

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="https://www.example.com/demo"
    href="demo_webstart.jnlp">
    <information>
        <title>Demo</title>
        <vendor>The Example.com Team</vendor>
    </information>
    <resources>
        <!-- Application Resources -->
        <j2se version="1.7+" href="http://java.sun.com/products/autodl/j2se"/>
        <jar href="Demo.jar" main="true"/>
    </resources>
    <application-desc
         name="Demo Application"
         main-class="com.example.jwsdemo.Main"
         width="300"
         height="300">
     </application-desc>
     <update check="background"/>
</jnlp>
```

Como puede ver, un archivo JNLP está basado en XML y toda la información está contenida en el elemento `<jnlp>` .

- El atributo `spec` proporciona la versión de la especificación JNPL que cumple este archivo.
- El atributo `codebase` proporciona la URL base para resolver las URL `href` relativas en el resto del archivo.
- El atributo `href` proporciona la URL definitiva para este archivo JNLP.
- El elemento `<information>` contiene metadatos de la aplicación, incluidos su título, autores, descripción y sitio web de ayuda.
- El elemento `<resources>` describe las dependencias para la aplicación, incluida la versión de Java, la plataforma del sistema operativo y los archivos JAR requeridos.
- El elemento `<application-desc>` (o `<applet-desc>`) proporciona la información necesaria para iniciar la aplicación.

Configurando el servidor web

El servidor web debe estar configurado para usar la `application/x-java-jnlp-file` como el tipo MIME para los archivos `.jnlp` .

El archivo JNLP y los archivos JAR de la aplicación deben instalarse en el servidor web para que estén disponibles utilizando las URL indicadas por el archivo JNLP.

Habilitar el lanzamiento a través de una página web.

Si la aplicación se inicia a través de un enlace web, la página que contiene el enlace debe crearse

en el servidor web.

- Si puede asumir que Java Web Start ya está instalado en la máquina del usuario, entonces la página web simplemente necesita contener un enlace para iniciar la aplicación. Por ejemplo.

```
<a href="https://www.example.com/demo_webstart.jnlp">Launch the application</a>
```

- De lo contrario, la página también debe incluir algunas secuencias de comandos para detectar el tipo de navegador que el usuario está utilizando y solicitar descargar e instalar la versión requerida de Java.

NOTA: es una mala idea animar a los usuarios a animar a instalar Java de esta manera, o incluso habilitar Java en sus navegadores web para que el lanzamiento de la página web JNLP funcione.

Lanzar aplicaciones Web Start desde la línea de comandos.

Las instrucciones para iniciar una aplicación Web Start desde la línea de comandos son simples. Suponiendo que el usuario tiene un JRE o JDK de Java 5.0 instalado, simplemente debe ejecutar esto:

```
$ javaws <url>
```

donde `<url>` es la URL del archivo JNLP en el servidor remoto.

Creando un UberJAR para una aplicación y sus dependencias

Un requisito común para una aplicación Java es que puede implementarse copiando un solo archivo. Para aplicaciones simples que dependen solo de las bibliotecas de clases estándar de Java SE, este requisito se cumple creando un archivo JAR que contiene todas las clases de aplicaciones (compiladas).

Las cosas no son tan sencillas si la aplicación depende de bibliotecas de terceros. Si simplemente coloca los archivos JAR de dependencia dentro de un JAR de aplicación, el cargador de clases Java estándar no podrá encontrar las clases de la biblioteca y su aplicación no se iniciará. En su lugar, debe crear un solo archivo JAR que contenga las clases de la aplicación y los recursos asociados junto con las clases de dependencia y los recursos. Estos deben organizarse como un único espacio de nombres para que el cargador de clases busque.

El archivo JAR de este tipo suele denominarse UberJAR.

Creando un UberJAR usando el comando "jar"

El procedimiento para crear un UberJAR es sencillo. (Usaré los comandos de Linux por

simplicidad. Los comandos deben ser idénticos para Mac OS y similares para Windows).

1. Cree un directorio temporal y cámbielo.

```
$ mkdir tempDir
$ cd tempDir
```

2. Para cada archivo JAR dependiente, *en el orden inverso al* que deben aparecer en la ruta de clase de la aplicación, usó el comando `jar` para descomprimir el JAR en el directorio temporal.

```
$ jar -xf <path/to/file.jar>
```

   Al hacer esto para múltiples archivos JAR se *superpondrán los* contenidos de los JAR.

3. Copie las clases de aplicación del árbol de compilación en el directorio temporal

```
$ cp -r path/to/classes .
```

4. Cree el UberJAR a partir del contenido del directorio temporal:

```
$ jar -cf ../myApplication.jar
```

   Si está creando un archivo JAR ejecutable, incluya un MANIFEST.MF apropiado como se describe aquí.

Creando un UberJAR utilizando Maven

Si su proyecto se construye utilizando Maven, puede crear un UberJAR utilizando los complementos "maven-assembly" o "maven-shade". Consulte el tema de la Asamblea de Maven (en la documentación de Maven) para obtener más información.

Las ventajas y desventajas de los UberJARs.

Algunas de las ventajas de los UberJAR son evidentes:

- Un UberJAR es fácil de distribuir.
- No puede romper las dependencias de la biblioteca para un UberJAR, ya que las bibliotecas son autónomas.

Además, si usa una herramienta adecuada para crear UberJAR, tendrá la opción de excluir las clases de biblioteca que no se usan del archivo JAR. Sin embargo, esto se hace típicamente mediante análisis estático de las clases. Si su aplicación utiliza la reflexión, el procesamiento de anotaciones y técnicas similares, debe tener cuidado de que las clases no se excluyan incorrectamente.

Los UberJAR también tienen algunas desventajas:

- Si tiene muchos UberJAR con las mismas dependencias, cada uno contendrá una copia de las dependencias.
- Algunas bibliotecas de código abierto tienen licencias que *puede* impedir su uso [1] en un UberJAR.

---

1 - Algunas licencias de biblioteca de código abierto le permiten usar la biblioteca, solo si el usuario final puede reemplazar una versión de la biblioteca por otra. UberJARs puede dificultar el reemplazo de dependencias de versión.

Lea Despliegue de Java en línea: https://riptutorial.com/es/java/topic/6840/despliegue-de-java

Capítulo 55: Dividir una cadena en partes de longitud fija

Observaciones

El objetivo aquí es no perder contenido, por lo que la expresión regular no debe consumir (igualar) ninguna entrada. En su lugar, debe coincidir *entre* el último carácter de la entrada de destino anterior y el primer carácter de la siguiente entrada de destino. por ejemplo, para las subcadenas de 8 caracteres, necesitamos dividir la entrada (es decir, hacer coincidir) en los lugares marcados a continuación:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
              ^               ^               ^
```

Ignore los espacios en la entrada que fueron necesarios para mostrar *entre las* posiciones de los caracteres.

Examples

Rompe una cuerda en subcadenas, todas de una longitud conocida

El truco es usar un look-behind con el regex `\G` , que significa "final de la coincidencia anterior":

```
String[] parts = str.split("(?<=\\G.{8})");
```

La expresión regular coincide con 8 caracteres después del final de la última coincidencia. Como en este caso la coincidencia es de ancho cero, podríamos decir más simplemente "8 caracteres después de la última coincidencia".

Convenientemente, `\G` se inicializa al inicio de la entrada, por lo que también funciona para la primera parte de la entrada.

Rompe una cadena en subcadenas todas de longitud variable

Igual que el ejemplo de longitud conocida, pero inserte la longitud en expresiones regulares:

```
int length = 5;
String[] parts = str.split("(?<=\\G.{" + length + "})");
```

Lea Dividir una cadena en partes de longitud fija en línea:
https://riptutorial.com/es/java/topic/5613/dividir-una-cadena-en-partes-de-longitud-fija

Capítulo 56: Documentando el código de Java

Introducción

La documentación para el código java a menudo se genera utilizando javadoc . Javadoc fue creado por Sun Microsystems con el propósito de generar documentación de API en formato HTML desde el código fuente de Java. El uso del formato HTML brinda la comodidad de poder vincular documentos relacionados entre sí.

Sintaxis

- / ** - inicio de JavaDoc en una clase, campo, método o paquete
- @author // Para nombrar al autor de la clase, interfaz o enumeración. Es requerido.
- @version // La versión de esa clase, interfaz o enumeración. Es requerido. Puede usar macros como% I% o% G% para que el software de control de origen complete en el proceso de pago.
- @param // Para mostrar los argumentos (parámetros) de un método o un constructor. Especifique una etiqueta @param para cada parámetro.
- @return // Para mostrar los tipos de retorno para métodos no nulos.
- @exception // Muestra qué excepciones se pueden lanzar desde el método o el constructor. Las excepciones que **DEBEN** ser capturadas deben enumerarse aquí. Si lo desea, también puede incluir aquellos que no necesitan ser capturados, como ArrayIndexOutOfBoundsException. Especifique una @excepción para cada excepción que se pueda lanzar.
- @throws // Igual que @exception.
- @see // Enlaces a un método, campo, clase o paquete. Utilizar en la forma de package.Class # algo.
- @since // Cuando este método, campo o clase fue agregado. Por ejemplo, JDK-8 para una clase como java.util.Optional <T> .
- @serial, @serialField, @serialData // Se usa para mostrar el serialVersionUID.
- @deprecated // Para marcar una clase, método o campo como obsoleto. Por ejemplo, uno sería java.io.StringBufferInputStream . Vea una lista completa de las clases en desuso existentes aquí .
- {@link} // Similar a @see, pero se puede usar con texto personalizado: {@link #setDefaultCloseOperation (int closeOperation) vea JFrame # setDefaultCloseOperation para más información}.
- {@linkplain} // Similar a {@link}, pero sin la fuente del código.
- {@code} // Para código literal, como etiquetas HTML. Por ejemplo: {@code <html> </html>}. Sin embargo, esto usará una fuente monoespaciada. Para obtener el mismo resultado sin la fuente monoespaciado, use {@literal}.
- {@literal} // Igual que {@code}, pero sin la fuente monoespaciada.
- {@value} // Muestra el valor de un campo estático: El valor de JFrame # EXIT_ON_CLOSE

es {@value}. O, podría vincular a un campo determinado: utiliza el nombre de la aplicación {@value AppConstants # APP_NAME}.
- {@docRoot} // La carpeta raíz del HTML de JavaDoc relativa al archivo actual. Ejemplo: Créditos .
- Se permite HTML: <code> "Hola cookies" .substring (3) </code>.
- * / - final de la declaración de JavaDoc

Observaciones

Javadoc es una herramienta incluida con el JDK que permite que los comentarios en código se conviertan a una documentación HTML. La especificación de la API de Java se generó utilizando Javadoc. Lo mismo es cierto para gran parte de la documentación de las bibliotecas de terceros.

Examples

Documentación de la clase

Todos los comentarios de Javadoc comienzan con un comentario de bloqueo seguido de un asterisco (`/**`) y finalizan cuando el comentario de bloqueo (`*/`). Opcionalmente, cada línea puede comenzar con espacios en blanco arbitrarios y un solo asterisco; estos se ignoran cuando se generan los archivos de documentación.

```
/**
 * Brief summary of this class, ending with a period.
 *
 * It is common to leave a blank line between the summary and further details.
 * The summary (everything before the first period) is used in the class or package
 * overview section.
 *
 * The following inline tags can be used (not an exhaustive list):
 * {@link some.other.class.Documentation} for linking to other docs or symbols
 * {@link some.other.class.Documentation Some Display Name} the link's appearance can be
 * customized by adding a display name after the doc or symbol locator
 * {@code code goes here} for formatting as code
 * {@literal <>[]()foo} for interpreting literal text without converting to HTML markup
 * or other tags.
 *
 * Optionally, the following tags may be used at the end of class documentation
 * (not an exhaustive list):
 *
 * @author John Doe
 * @version 1.0
 * @since 5/10/15
 * @see some.other.class.Documentation
 * @deprecated This class has been replaced by some.other.package.BetterFileReader
 *
 * You can also have custom tags for displaying additional information.
 * Using the @custom.<NAME> tag and the -tag custom.<NAME>:htmltag:"context"
 * command line option, you can create a custom tag.
 *
 * Example custom tag and generation:
 * @custom.updated 2.0
 * Javadoc flag: -tag custom.updated:a:"Updated in version:"
```

```
 * The above flag will display the value of @custom.updated under "Updated in version:"
 *
 */
public class FileReader {
}
```

Las mismas etiquetas y formato que se usan para las `Classes` se pueden usar para `Enums` e `Interfaces` .

Método de Documentación

Todos los comentarios de Javadoc comienzan con un comentario de bloqueo seguido de un asterisco (`/**`) y finalizan cuando el comentario de bloqueo (`*/`). Opcionalmente, cada línea puede comenzar con espacios en blanco arbitrarios y un solo asterisco; estos se ignoran cuando se generan los archivos de documentación.

```
/**
 * Brief summary of method, ending with a period.
 *
 * Further description of method and what it does, including as much detail as is
 * appropriate.  Inline tags such as
 * {@code code here}, {@link some.other.Docs}, and {@literal text here} can be used.
 *
 * If a method overrides a superclass method, {@inheritDoc} can be used to copy the
 * documentation
 * from the superclass method
 *
 * @param stream Describe this parameter.  Include as much detail as is appropriate
 *               Parameter docs are commonly aligned as here, but this is optional.
 *               As with other docs, the documentation before the first period is
 *               used as a summary.
 *
 * @return Describe the return values.  Include as much detail as is appropriate
 *         Return type docs are commonly aligned as here, but this is optional.
 *         As with other docs, the documentation before the first period is used as a
 *         summary.
 *
 * @throws IOException Describe when and why this exception can be thrown.
 *                     Exception docs are commonly aligned as here, but this is
 *                     optional.
 *                     As with other docs, the documentation before the first period
 *                     is used as a summary.
 *                     Instead of @throws, @exception can also be used.
 *
 * @since 2.1.0
 * @see some.other.class.Documentation
 * @deprecated  Describe why this method is outdated. A replacement can also be specified.
 */
public String[] read(InputStream stream) throws IOException {
    return null;
}
```

Documentacion de campo

Todos los comentarios de Javadoc comienzan con un comentario de bloqueo seguido de un asterisco (`/**`) y finalizan cuando el comentario de bloqueo (`*/`). Opcionalmente, cada línea

puede comenzar con espacios en blanco arbitrarios y un solo asterisco; estos se ignoran cuando se generan los archivos de documentación.

```
/**
 * Fields can be documented as well.
 *
 * As with other javadocs, the documentation before the first period is used as a
 * summary, and is usually separated from the rest of the documentation by a blank
 * line.
 *
 * Documentation for fields can use inline tags, such as:
 * {@code code here}
 * {@literal text here}
 * {@link other.docs.Here}
 *
 * Field documentation can also make use of the following tags:
 *
 * @since 2.1.0
 * @see some.other.class.Documentation
 * @deprecated Describe why this field is outdated
 */
public static final String CONSTANT_STRING = "foo";
```

Documentación del paquete

Java SE 5

Es posible crear documentación a nivel de paquete en Javadocs usando un archivo llamado `package-info.java` . Este archivo se debe formatear como se muestra a continuación. Los espacios en blanco iniciales y los asteriscos son opcionales, normalmente están presentes en cada línea por razones de formato

```
/**
 * Package documentation goes here; any documentation before the first period will
 * be used as a summary.
 *
 * It is common practice to leave a blank line between the summary and the rest
 * of the documentation; use this space to describe the package in as much detail
 * as is appropriate.
 *
 * Inline tags such as {@code code here}, {@link reference.to.other.Documentation},
 * and {@literal text here} can be used in this documentation.
 */
package com.example.foo;

// The rest of the file must be empty.
```

En el caso anterior, debe colocar este archivo `package-info.java` dentro de la carpeta del paquete de Java `com.example.foo` .

Campo de golf

El enlace a otros Javadocs se realiza con la etiqueta `@link` :

```
/**
 * You can link to the javadoc of an already imported class using {@link ClassName}.
 *
 * You can also use the fully-qualified name, if the class is not already imported:
 *  {@link some.other.ClassName}
 *
 * You can link to members (fields or methods) of a class like so:
 *  {@link ClassName#someMethod()}
 *  {@link ClassName#someMethodWithParameters(int, String)}
 *  {@link ClassName#someField}
 *  {@link #someMethodInThisClass()} - used to link to members in the current class
 *
 * You can add a label to a linked javadoc like so:
 *  {@link ClassName#someMethod() link text}
 */
```

You can link to the javadoc of an already imported class using ClassName.

You can also use the fully-qualified name, if the class is not already imported: some.other.ClassName

You can link to members (fields or methods) of a class like so:

ClassName.someMethod()

ClassName.someMethodWithParameters(int, String)

ClassName.someField

someMethodInThisClass() - used to link to members in the current class

You can add a label to a linked javadoc like so: link text

Con la etiqueta `@see` puede agregar elementos a la sección *Vea también* . Como `@param` o `@return` el lugar donde aparecen no es relevante. La especificación dice que debes escribirlo después de `@return` .

```
/**
 * This method has a nice explanation but you might found further
 * information at the bottom.
 *
 * @see ClassName#someMethod()
 */
```

This method has a nice explanation but you might found furthe

See Also:

ClassName.someMethod()

Si desea agregar **enlaces a recursos externos,** solo puede usar la etiqueta HTML `<a>` . Puede usarlo en línea en cualquier lugar o dentro de las etiquetas `@link` y `@see` .

```
/**
 * Wondering how this works? You might want
 * to check this <a href="http://stackoverflow.com/">great service</a>.
 *
 * @see <a href="http://stackoverflow.com/">Stack Overflow</a>
 */
```

Wondering how this works? You might want to check this great service.

See Also:
Stack Overflow

Construyendo Javadocs desde la línea de comando

Muchos IDE proporcionan soporte para generar HTML desde Javadocs automáticamente; Algunas herramientas de compilación (Maven y Gradle , por ejemplo) también tienen complementos que pueden manejar la creación de HTML.

Sin embargo, estas herramientas no son necesarias para generar el HTML Javadoc; Esto se puede hacer utilizando la herramienta de línea de comandos `javadoc` .

El uso más básico de la herramienta es:

```
javadoc JavaFile.java
```

Lo que generará HTML a partir de los comentarios de Javadoc en `JavaFile.java` .

Un uso más práctico de la herramienta de línea de comandos, que leerá recursivamente todos los archivos java en `[source-directory]` , creará la documentación para `[package.name]` y todos los subpaquetes, y colocará el HTML generado en el `[docs-directory]` es:

```
javadoc -d [docs-directory] -subpackages -sourcepath [source-directory] [package.name]
```

Documentación del código en línea

Aparte del código de documentación Javadoc se puede documentar en línea.

Los comentarios de una sola línea se inician con `//` y se pueden colocar después de una declaración en la misma línea, pero no antes.

```
public void method() {

  //single line comment
  someMethodCall(); //single line comment after statement

}
```

Los comentarios multilínea se definen entre `/*` y `*/` . Pueden abarcar varias líneas e incluso pueden colocarse entre sentencias.

```
public void method(Object object) {

  /*
    multi
    line
    comment
  */
  object/*inner-line-comment*/.method();
```

```
}
```

Los JavaDocs son una forma especial de comentarios de varias líneas, comenzando con `/**` .

Como demasiados comentarios en línea pueden disminuir la legibilidad del código, se deben usar de forma dispersa en caso de que el código no se explique lo suficiente o la decisión de diseño no sea obvia.

Un caso de uso adicional para los comentarios de una sola línea es el uso de TAG, que son palabras clave cortas y basadas en convenciones. Algunos entornos de desarrollo reconocen ciertas convenciones para dichos comentarios individuales. Ejemplos comunes son

- `//TODO`
- `//FIXME`

O emitir referencias, es decir, para Jira.

- `//PRJ-1234`

Fragmentos de código dentro de la documentación.

La forma canónica de escribir código dentro de la documentación es con la construcción `{@code }` . Si tiene código multilínea dentro de `<pre></pre>` .

```
/**
 * The Class TestUtils.
 * <p>
 * This is an {@code inline("code example")}.
 * <p>
 * You should wrap it in pre tags when writing multiline code.
 * <pre>{@code
 *  Example example1 = new FirstLineExample();
 *  example1.butYouCanHaveMoreThanOneLine();
 * }</pre>
 * <p>
 * Thanks for reading.
 */
class TestUtils {
```

A veces es posible que necesite poner un código complejo dentro del comentario javadoc. El signo `@` es especialmente problemático. El uso de la etiqueta `<code>` antigua junto con la construcción `{@literal }` resuelve el problema.

```
/**
 * Usage:
 * <pre><code>
 * class SomethingTest {
 * {@literal @}Rule
 *  public SingleTestRule singleTestRule = new SingleTestRule("test1");
 *
 * {@literal @}Test
 *  public void test1() {
 *      // only this test will be executed
 *  }
```

```
 *
 *   ...
 * }
 * </code></pre>
 */
class SingleTestRule implements TestRule { }
```

Lea Documentando el código de Java en línea:
https://riptutorial.com/es/java/topic/140/documentando-el-codigo-de-java

Capítulo 57: E / S de consola

Examples

Lectura de entrada de usuario desde la consola.

Utilizando `BufferedReader` :

```
System.out.println("Please type your name and press Enter.");

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
try {
    String name = reader.readLine();
    System.out.println("Hello, " + name + "!");
} catch(IOException e) {
    System.out.println("An error occurred: " + e.getMessage());
}
```

Las siguientes importaciones son necesarias para este código:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

Utilizando `Scanner` :

Java SE 5

```
System.out.println("Please type your name and press Enter");

Scanner scanner = new Scanner(System.in);
String name = scanner.nextLine();

System.out.println("Hello, " + name + "!");
```

La siguiente importación es necesaria para este ejemplo:

```
import java.util.Scanner;
```

Para leer más de una línea, invoque `scanner.nextLine()` repetidamente:

```
System.out.println("Please enter your first and your last name, on separate lines.");

Scanner scanner = new Scanner(System.in);
String firstName = scanner.nextLine();
String lastName = scanner.nextLine();

System.out.println("Hello, " + firstName + " " + lastName + "!");
```

Hay dos métodos para obtener `Strings` , `next()` y `nextLine()` . `next()` devuelve el texto hasta el primer espacio (también conocido como "token"), y `nextLine()` devuelve todo el texto que el usuario ingresó hasta presionar intro.

`Scanner` también proporciona métodos de utilidad para leer tipos de datos que no sean `String` . Éstos incluyen:

```
scanner.nextByte();
scanner.nextShort();
scanner.nextInt();
scanner.nextLong();
scanner.nextFloat();
scanner.nextDouble();
scanner.nextBigInteger();
scanner.nextBigDecimal();
```

El prefijo de cualquiera de estos métodos con `has` (como en `hasNextLine()` , `hasNextInt()`) devuelve `true` si el flujo tiene más del tipo de solicitud. Nota: estos métodos bloquearán el programa si la entrada no es del tipo solicitado (por ejemplo, al escribir "a" para `nextInt()`). Puede usar un `try {} catch() {}` para evitar esto (ver: Excepciones)

```
Scanner scanner = new Scanner(System.in); //Create the scanner
scanner.useLocale(Locale.US); //Set number format excepted
System.out.println("Please input a float, decimal separator is .");
if (scanner.hasNextFloat()){ //Check if it is a float
    float fValue = scanner.nextFloat(); //retrive the value directly as float
    System.out.println(fValue + " is a float");
}else{
    String sValue = scanner.next(); //We can not retrive as float
    System.out.println(sValue + " is not a float");
}
```

Utilizando `System.console` :

Java SE 6

```
String name = System.console().readLine("Please type your name and press Enter%n");

System.out.printf("Hello, %s!", name);

//To read passwords (without echoing as in unix terminal)
char[] password = System.console().readPassword();
```

Ventajas :

- Los métodos de lectura están sincronizados.
- Se puede usar la sintaxis de cadena de formato

Nota : Esto solo funcionará si el programa se ejecuta desde una línea de comando real sin redirigir las secuencias de entrada y salida estándar. No funciona cuando el programa se ejecuta desde ciertos IDE, como Eclipse. Para el código que funciona dentro de los IDE y con la

redirección de flujos, vea los otros ejemplos.

Implementar el comportamiento básico de la línea de comandos

Para los prototipos básicos o el comportamiento básico de la línea de comandos, el siguiente bucle es útil.

```
public class ExampleCli {

    private static final String CLI_LINE   = "example-cli>"; //console like string

    private static final String CMD_QUIT   = "quit";    //string for exiting the program
    private static final String CMD_HELLO  = "hello";    //string for printing "Hello World!"
on the screen
    private static final String CMD_ANSWER = "answer";    //string for printing 42 on the
screen

    public static void main(String[] args) {
        ExampleCli claimCli = new ExampleCli();    // creates an object of this class

        try {
            claimCli.start();    //calls the start function to do the work like console
        }
        catch (IOException e) {
            e.printStackTrace();    //prints the exception log if it is failed to do get the
user input or something like that
        }
    }

    private void start() throws IOException {
        String cmd = "";

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        while (!cmd.equals(CMD_QUIT)) {    // terminates console if user input is "quit"
            System.out.print(CLI_LINE);    //prints the console-like string

            cmd = reader.readLine();    //takes input from user. user input should be started
with "hello",  "answer" or "quit"
            String[] cmdArr = cmd.split(" ");

            if (cmdArr[0].equals(CMD_HELLO)) {    //executes when user input starts with
"hello"
                hello(cmdArr);
            }
            else if (cmdArr[0].equals(CMD_ANSWER)) {    //executes when user input starts with
"answer"
                answer(cmdArr);
            }
        }
    }

    // prints "Hello World!" on the screen if user input starts with "hello"
    private void hello(String[] cmdArr) {
        System.out.println("Hello World!");
    }

    // prints "42" on the screen if user input starts with "answer"
    private void answer(String[] cmdArr) {
        System.out.println("42");
```

```
    }
}
```

Alineación de cuerdas en consola.

El método `PrintWriter.format` (llamado a través de `System.out.format`) se puede utilizar para imprimir cadenas alineadas en la consola. El método recibe una `String` con la información de formato y una serie de objetos para formatear:

```
String rowsStrings[] = new String[] {"1",
                                     "1234",
                                     "1234567",
                                     "123456789"};

String column1Format = "%-3s";    // min 3 characters, left aligned
String column2Format = "%-5.8s";  // min 5 and max 8 characters, left aligned
String column3Format = "%6.6s";   // fixed size 6 characters, right aligned
String formatInfo = column1Format + " " + column2Format + " " + column3Format;

for(int i = 0; i < rowsStrings.length; i++) {
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);
    System.out.println();
}
```

Salida:

```
1   1          1
1234 1234    1234
1234567 1234567 123456
123456789 12345678 123456
```

El uso de cadenas de formato con permisos de tamaño fijo para imprimir las cadenas en una apariencia similar a una tabla con columnas de tamaño fijo:

```
String rowsStrings[] = new String[] {"1",
                                     "1234",
                                     "1234567",
                                     "123456789"};

String column1Format = "%-3.3s";  // fixed size 3 characters, left aligned
String column2Format = "%-8.8s";  // fixed size 8 characters, left aligned
String column3Format = "%6.6s";   // fixed size 6 characters, right aligned
String formatInfo = column1Format + " " + column2Format + " " + column3Format;

for(int i = 0; i < rowsStrings.length; i++) {
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);
    System.out.println();
}
```

Salida:

```
1   1             1
123 1234       1234
```

```
123 1234567  123456
123 12345678 123456
```

Ejemplos de cadenas de formato

- `%s` : solo una cadena sin formato
- `%5s` : formatee la cadena con un **mínimo** de 5 caracteres; si la cadena es más corta, se **rellenará** con 5 caracteres y se alineará a la **derecha**
- `%-5s` : formatee la cadena con un **mínimo** de 5 caracteres; si la cadena es más corta, se **rellenará** con 5 caracteres y se alineará a la **izquierda**
- `%5.10s` : formatee la cadena con un **mínimo** de 5 caracteres y un **máximo** de 10 caracteres; si la cadena es más corta que 5, se **rellenará** con 5 caracteres y se alineará a la **derecha** ; si la cadena es más larga que 10, se **truncará** a 10 caracteres y se alineará a la **derecha**
- `%-5.5s` : formatee la cadena con un tamaño **fijo** de 5 caracteres (el mínimo y el máximo son iguales); si la cadena es más corta que 5, se **rellenará** con 5 caracteres y se alineará a la **izquierda** ; si la cadena es más larga que 5, se **truncará** a 5 caracteres y se alineará a la **izquierda**

Lea E / S de consola en línea: https://riptutorial.com/es/java/topic/126/e---s-de-consola

Capítulo 58: Ediciones, versiones, lanzamientos y distribuciones de Java

Examples

Diferencias entre las distribuciones Java SE JRE o Java SE JDK

Las versiones de Sun / Oracle de Java SE vienen en dos formas: JRE y JDK. En términos simples, los JRE admiten la ejecución de aplicaciones Java y los JDK también admiten el desarrollo de Java.

Java Runtime Environment

Las distribuciones de Java Runtime Environment o JRE consisten en el conjunto de bibliotecas y herramientas necesarias para ejecutar y administrar aplicaciones Java. Las herramientas en un JRE moderno típico incluyen:

- El comando `java` para ejecutar un programa Java en una JVM (Java Virtual Machine)
- El comando `jjs` para ejecutar el motor de Javascript de Nashorn.
- El comando `keytool` para manipular los almacenes de claves de Java.
- El comando `policytool` para editar las políticas de seguridad de sandbox.
- Las herramientas `pack200` y `unpack200` para empaquetar y desempaquetar el archivo "pack200" para el despliegue web.
- Los `orbd` , `rmid` , `rmiregistry` y `tnameserv` que admiten aplicaciones Java CORBA y RMI.

Los instaladores de "Desktop JRE" incluyen un complemento de Java adecuado para algunos navegadores web. Esto se deja deliberadamente fuera de "Server JRE" installers.linux syscall read benchmarku

A partir de la actualización 6 de Java 7, los instaladores de JRE han incluido JavaFX (versión 2.2 o posterior).

Kit de desarrollo de Java

Un kit de desarrollo de Java o una distribución de JDK incluye las herramientas JRE y herramientas adicionales para desarrollar software de Java. Las herramientas adicionales típicamente incluyen:

- El comando `javac` , que compila el código fuente de Java (".java") en archivos de código de bytes (".class").
- Las herramientas para crear archivos JAR como `jar` y `jarsigner`
- Herramientas de desarrollo tales como:
  - `appletviewer` para ejecutar applets
  - `idlj` el IDL CORBA al compilador de Java

    - `javah` el generador de `javah` JNI
    - `native2ascii` para la conversión de `native2ascii` de caracteres del código fuente de Java
    - `schemagen` el generador de esquemas de Java a XML (parte de JAXB)
    - `serialver` generar la cadena de versión de Java Object Serialization.
    - `wsgen` herramientas de soporte `wsgen` y `wsimport` para JAX-WS
- Herramientas de diagnóstico tales como:
    - `jdb` el depurador de Java básico
    - `jmap` y `jhat` para descargar y analizar un montón de Java.
    - `jstack` para obtener un volcado de pila de hilos.
    - `javap` para examinar archivos ".class".
- Gestión de aplicaciones y herramientas de monitoreo tales como:
    - `jconsole` una consola de gestión,
    - `jstat` , `jstatd` , `jinfo` y `jps` para monitoreo de aplicaciones

Una instalación típica de Sun / Oracle JDK también incluye un archivo ZIP con el código fuente de las bibliotecas de Java. Antes de Java 6, este era el único código fuente de Java disponible públicamente.

Desde Java 6 en adelante, el código fuente completo de OpenJDK está disponible para su descarga desde el sitio de OpenJDK. Normalmente no se incluye en los paquetes JDK (Linux), pero está disponible como un paquete separado.

¿Cuál es la diferencia entre Oracle Hotspot y OpenJDK?

Ortogonal a la dicotomía JRE versus JDK, hay dos tipos de versión de Java que están ampliamente disponibles:

- Las versiones de Oracle Hotspot son las que se descargan de los sitios de descarga de Oracle.
- Las versiones de OpenJDK son las que se crean (normalmente por proveedores de terceros) a partir de los repositorios de origen de OpenJDK.

En términos funcionales, hay poca diferencia entre una versión de Hotspot y una versión de OpenJDK. Hay algunas características "empresariales" adicionales en Hotspot que los clientes de Oracle (de pago) de Java pueden habilitar, pero aparte de eso, la misma tecnología está presente tanto en Hotspot como en OpenJDK.

Otra ventaja de Hotspot sobre OpenJDK es que las versiones de parches para Hotspot tienden a estar disponibles un poco antes. Esto también depende de cuán ágil sea su proveedor de OpenJDK; por ejemplo, cuánto tiempo tarda el equipo de compilación de una distribución de Linux en preparar y realizar un control de calidad en una nueva compilación de OpenJDK y colocarla en sus repositorios públicos.

La otra cara es que las versiones de Hotspot no están disponibles en los repositorios de paquetes para la mayoría de las distribuciones de Linux. Esto significa que mantener su software Java actualizado en una máquina con Linux generalmente es más trabajo si usa Hotspot.

Diferencias entre Java EE, Java SE, Java ME y JavaFX

La tecnología Java es tanto un lenguaje de programación como una plataforma. El lenguaje de programación Java es un lenguaje de alto nivel orientado a objetos que tiene una sintaxis y un estilo particulares. Una plataforma Java es un entorno particular en el que se ejecutan las aplicaciones de lenguaje de programación Java.

Hay varias plataformas de Java. Muchos desarrolladores, incluso los desarrolladores de lenguajes de programación Java de larga data, no entienden cómo las diferentes plataformas se relacionan entre sí.

Las plataformas de lenguaje de programación Java

Hay cuatro plataformas del lenguaje de programación Java:

- Plataforma Java, Edición Estándar (Java SE)
- Plataforma Java, Edición Enterprise (Java EE)
- Plataforma Java, Micro Edition (Java ME)
- Java FX

Todas las plataformas de Java consisten en una Máquina Virtual de Java (VM) y una interfaz de programación de aplicaciones (API). La máquina virtual de Java es un programa, para una plataforma de hardware y software en particular, que ejecuta aplicaciones de tecnología Java. Una API es una colección de componentes de software que puede utilizar para crear otros componentes de software o aplicaciones. Cada plataforma Java proporciona una máquina virtual y una API, y esto permite que las aplicaciones escritas para esa plataforma se ejecuten en cualquier sistema compatible con todas las ventajas del lenguaje de programación Java: independencia de la plataforma, potencia, estabilidad, facilidad de desarrollo y seguridad.

Java SE

Cuando la mayoría de la gente piensa en el lenguaje de programación Java, piensa en la API de Java SE. La API de Java SE proporciona la funcionalidad principal del lenguaje de programación Java. Define todo, desde los tipos y objetos básicos del lenguaje de programación Java hasta las clases de alto nivel que se utilizan para redes, seguridad, acceso a bases de datos, desarrollo de interfaz gráfica de usuario (GUI) y análisis de XML.

Además de la API central, la plataforma Java SE consta de una máquina virtual, herramientas de desarrollo, tecnologías de implementación y otras bibliotecas de clase y kits de herramientas que se utilizan comúnmente en las aplicaciones de tecnología Java.

Java EE

La plataforma Java EE está construida sobre la plataforma Java SE. La plataforma Java EE proporciona una API y un entorno de tiempo de ejecución para desarrollar y ejecutar aplicaciones de red a gran escala, de múltiples niveles, escalables, confiables y seguras.

Java ME

La plataforma Java ME proporciona una API y una máquina virtual de tamaño reducido para ejecutar aplicaciones de lenguaje de programación Java en dispositivos pequeños, como teléfonos móviles. La API es un subconjunto de la API de Java SE, junto con bibliotecas de clases especiales útiles para el desarrollo de aplicaciones de dispositivos pequeños. Las aplicaciones Java ME son a menudo clientes de los servicios de la plataforma Java EE.

Java FX

La tecnología Java FX es una plataforma para crear aplicaciones de Internet enriquecidas escritas en Java FX ScriptTM. Java FX Script es un lenguaje declarativo de tipo estático que se compila en el bytecode de la tecnología Java, que luego se puede ejecutar en una máquina virtual Java. Las aplicaciones escritas para la plataforma Java FX pueden incluir y vincular clases de lenguaje de programación Java y pueden ser clientes de los servicios de la plataforma Java EE.

- Tomado de la documentación de Oracle

Versiones de Java SE

Historial de versiones de Java SE

La siguiente tabla proporciona la línea de tiempo para las principales versiones significativas de la plataforma Java SE.

Java SE versión [1]	Nombre clave	Fin de la vida (libre [2])	Fecha de lanzamiento
Java SE 9 (acceso temprano)	*Ninguna*	futuro	2017-07-27 (estimado)
Java SE 8	*Ninguna*	futuro	2014-03-18
Java SE 7	Delfín	2015-04-14	2011-07-28
Java SE 6	Mustango	2013-04-16	2006-12-23

Java SE versión [1]	Nombre clave	Fin de la vida (libre [2])	Fecha de lanzamiento
Java SE 5	Tigre	2009-11-04	2004-10-04
Java SE 1.4.2	Mantis	antes del 2009-11-04	2003-06-26
Java SE 1.4.1	Tolva / Saltamontes	antes del 2009-11-04	2002-09-16
Java SE 1.4	Esmerejón	antes del 2009-11-04	2002-02-06
Java SE 1.3.1	Mariquita	antes del 2009-11-04	2001-05-17
Java SE 1.3	Cernícalo	antes del 2009-11-04	2000-05-08
Java SE 1.2	Patio de recreo	antes del 2009-11-04	1998-12-08
Java SE 1.1	Bengala	antes del 2009-11-04	1997-02-19
Java SE 1.0	Roble	antes del 2009-11-04	1996-01-21

Notas al pie:

1. Los enlaces son para copias en línea de la documentación de las versiones respectivas en el sitio web de Oracle. La documentación de muchas versiones anteriores ya no está en línea, aunque normalmente se puede descargar desde los Archivos Java de Oracle.

2. La mayoría de las versiones históricas de Java SE han pasado sus fechas oficiales de "fin de vida". Cuando una versión de Java supera este hito, Oracle deja de proporcionarle actualizaciones gratuitas. Las actualizaciones todavía están disponibles para los clientes con contratos de soporte.

Fuente:

- Fechas de lanzamiento de JDK por Roedy Green de Canadian Mind Products

Aspectos destacados de la versión de Java SE

Versión de Java SE	Reflejos
Java SE 8	Expresiones lambda y corrientes inspiradas en MapReduce. El motor de Javascript de Nashorn. Anotaciones sobre tipos y anotaciones repetidas. Extensiones aritméticas sin firmar. Nuevas API de fecha y hora. Bibliotecas JNI enlazadas estáticamente. Lanzador de JavaFX. Eliminación de PermGen.
Java SE 7	Conmutadores de cadenas, *try-with-resource* , el diamante (`<>`), mejoras literales numéricas y mejoras de manejo / reingreso de excepciones. Mejoras en la biblioteca de concurrencia. Soporte mejorado para sistemas de archivos nativos. Timsort. ECC algoritmos criptográficos. Compatibilidad con gráficos 2D mejorados (GPU). Anotaciones enchufables.
Java SE 6	Mejoras significativas en el rendimiento de la plataforma JVM y Swing. Lenguaje de scripting API y motor Javascript de Mozilla Rhino. JDBC 4.0. API del compilador. JAXB 2.0. Soporte de servicios web (JAX-WS)
Java SE 5	Genéricos, anotaciones, boxeo automático, clases de `enum` , varargs, mejorados `for` bucles e importaciones estáticas. Especificación del modelo de memoria Java. Swing y mejoras de RMI. Adición del paquete `java.util.concurrent.*` Y el `Scanner` .
Java SE 1.4	La palabra clave de `assert` . Clases de expresiones regulares. Encadenamiento de excepciones. API de NIO: E / S, `Buffer` y `Channel` sin bloqueo. `java.util.logging.*` API. API de E / S de imágenes. XML integrado y XSLT (JAXP). Seguridad integrada y criptografía (JCE, JSSE, JAAS). Java Web Start integrado. API de preferencias.
Java SE 1.3	HotSpot JVM incluido. Integración CORBA / RMI. Nombre de Java y la interfaz de directorio (JNDI). Marco de depuración (JPDA). API de JavaSound. API de proxy.
Java SE 1.2	La palabra clave `strictfp` . Swing APIs. El plugin de Java (para navegadores web). Interoperabilidad CORBA. Marco de colecciones.
Java SE 1.1	Clases de interior. Reflexión. JDBC. RMI. Unicode / streams de caracteres. Apoyo a la internacionalización. Revisión del modelo de evento AWT. JavaBeans.

Fuente:

- Wikipedia: historia de la versión de Java

Lea Ediciones, versiones, lanzamientos y distribuciones de Java en línea: https://riptutorial.com/es/java/topic/8973/ediciones--versiones--lanzamientos-y-distribuciones-de-java

Capítulo 59: Ejecutor, ExecutorService y grupos de subprocesos

Introducción

La interfaz de Executor en Java proporciona una forma de desacoplar el envío de tareas de la mecánica de cómo se ejecutará cada tarea, incluidos los detalles del uso de subprocesos, la programación, etc. Normalmente se utiliza un Ejecutor en lugar de crear subprocesos explícitamente. Con los Ejecutores, los desarrolladores no tendrán que reescribir significativamente su código para poder ajustar fácilmente la política de ejecución de tareas de su programa.

Observaciones

Escollos

- Cuando programa una tarea para ejecución repetida, dependiendo de la ScheduledExecutorService utilizada, su tarea podría suspenderse de cualquier ejecución posterior, si una ejecución de su tarea provoca una excepción que no se maneja. Ver Madre F ** k el ScheduledExecutorService!

Examples

Fuego y olvido - Tareas ejecutables

Los ejecutores aceptan un `java.lang.Runnable` que contiene código (potencialmente computacional o de otro modo pesado o pesado) para ejecutarse en otro Thread.

El uso sería:

```
Executor exec = anExecutor;
exec.execute(new Runnable() {
    @Override public void run() {
        //offloaded work, no need to get result back
    }
});
```

Tenga en cuenta que con este ejecutor, no tiene medios para recuperar ningún valor calculado. Con Java 8, uno puede utilizar lambdas para acortar el ejemplo de código.

Java SE 8

```
Executor exec = anExecutor;
exec.execute(() -> {
    //offloaded work, no need to get result back
});
```

ThreadPoolExecutor

Un Ejecutor común que se usa es el `ThreadPoolExecutor` , que se encarga del manejo de Thread. Puede configurar la cantidad mínima de subprocesos que el ejecutor siempre debe mantener cuando no hay mucho que hacer (se denomina tamaño del núcleo) y un tamaño máximo de subprocesos en el que la agrupación puede crecer, si hay más trabajo por hacer. Una vez que la carga de trabajo disminuye, el Grupo reduce lentamente el número de subprocesos nuevamente hasta que alcanza el tamaño mínimo.

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(
    1,                                      // keep at least one thread ready,
                                            // even if no Runnables are executed
    5,                                      // at most five Runnables/Threads
                                            // executed in parallel
    1, TimeUnit.MINUTES,                    // idle Threads terminated after one
                                            // minute, when min Pool size exceeded
    new ArrayBlockingQueue<Runnable>(10)); // outstanding Runnables are kept here

pool.execute(new Runnable() {
    @Override public void run() {
        //code to run
    }
});
```

Nota: si configura `ThreadPoolExecutor` con una cola *ilimitada* , entonces el recuento de subprocesos no excederá `corePoolSize` ya que los nuevos subprocesos solo se crean si la cola está llena:

ThreadPoolExecutor con todos los parámetros:

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

de JavaDoc

> Si hay más de corePoolSize pero menos de maximumPoolSize se están ejecutando, se creará un nuevo thread solo si la cola está llena.

Ventajas:

1. El tamaño de BlockingQueue se puede controlar y los escenarios de memoria insuficiente se pueden evitar. El rendimiento de la aplicación no se degradará con el tamaño limitado de la cola delimitada.

2. Puede usar políticas existentes o crear nuevas políticas de manejo de rechazo.

    1. En el ThreadPoolExecutor.AbortPolicy predeterminado, el controlador lanza una excepción RejectedExecutionException en tiempo de ejecución cuando se rechaza.

    2. En `ThreadPoolExecutor.CallerRunsPolicy` , el hilo que invoca a sí mismo ejecuta la tarea. Esto proporciona un mecanismo de control de retroalimentación simple que reducirá la

velocidad a la que se envían las nuevas tareas.

3. En `ThreadPoolExecutor.DiscardPolicy` , una tarea que no se puede ejecutar simplemente se elimina.

4. En `ThreadPoolExecutor.DiscardOldestPolicy` , si el ejecutor no se apaga, la tarea en la cabecera de la cola de trabajo se descarta, y luego se reintenta la ejecución (lo que puede fallar nuevamente, lo que hace que se repita).

3. `ThreadFactory` puede configurar `ThreadFactory` personalizado, lo cual es útil:

    1. Para establecer un nombre de hilo más descriptivo
    2. Para establecer el estado del daemon de hilo
    3. Para establecer la prioridad del hilo

Aquí hay un ejemplo de cómo usar ThreadPoolExecutor

Recuperando valor de cómputo - Callable

Si su cómputo produce algún valor de retorno que luego se requiere, una simple tarea Ejecutable no es suficiente. Para tales casos, puede usar `ExecutorService.submit( Callable <T>)` que devuelve un valor una vez que se completa la ejecución.

El Servicio devolverá un `Future` que puede utilizar para recuperar el resultado de la ejecución de la tarea.

```
// Submit a callable for execution
ExecutorService pool = anExecutorService;
Future<Integer> future = pool.submit(new Callable<Integer>() {
    @Override public Integer call() {
        //do some computation
        return new Random().nextInt();
    }
});
// ... perform other tasks while future is executed in a different thread
```

Cuando necesite obtener el resultado del futuro, llame a `future.get()`

- Espera indefinidamente a que el futuro termine con un resultado.

```
try {
    // Blocks current thread until future is completed
    Integer result = future.get();
catch (InterruptedException || ExecutionException e) {
    // handle appropriately
}
```

- Espere a que el futuro termine, pero no más de lo especificado.

```
try {
    // Blocks current thread for a maximum of 500 milliseconds.
    // If the future finishes before that, result is returned,
```

```
        // otherwise TimeoutException is thrown.
        Integer result = future.get(500, TimeUnit.MILLISECONDS);
    catch (InterruptedException || ExecutionException || TimeoutException e) {
        // handle appropriately
    }
```

Si ya no es necesario el resultado de una tarea programada o en ejecución, puede llamar a `Future.cancel(boolean)` para cancelarla.

- Llamar a `cancel(false)` solo eliminará la tarea de la cola de tareas que se ejecutarán.
- Llamar a `cancel(true)` *también* interrumpirá la tarea si se está ejecutando actualmente.

Programar tareas para que se ejecuten a una hora determinada, después de un retraso o repetidamente

La clase `ScheduledExecutorService` proporciona métodos para programar tareas únicas o repetidas de varias maneras. El siguiente ejemplo de código supone que el `pool` se ha declarado e inicializado de la siguiente manera:

```
ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);
```

Además de los métodos normales de `ExecutorService` , la API `ScheduledExecutorService` agrega 4 métodos que programan tareas y devuelven objetos `ScheduledFuture` . Este último se puede utilizar para recuperar resultados (en algunos casos) y cancelar tareas.

Iniciar una tarea después de un retraso fijo

El siguiente ejemplo programa una tarea para comenzar después de diez minutos.

```
ScheduledFuture<Integer> future = pool.schedule(new Callable<>() {
        @Override public Integer call() {
            // do something
            return 42;
        }
    },
    10, TimeUnit.MINUTES);
```

Comenzando tareas a una tasa fija

El siguiente ejemplo programa una tarea para comenzar después de diez minutos, y luego repetidamente a una velocidad de una vez cada minuto.

```
ScheduledFuture<?> future = pool.scheduleAtFixedRate(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

La ejecución de la tarea continuará de acuerdo con el cronograma hasta que la `pool` se cierre, el `future` se cancele o una de las tareas encuentre una excepción.

Se garantiza que las tareas programadas por una llamada `scheduledAtFixedRate` AtFixedRate no se superpondrán en el tiempo. Si una tarea lleva más tiempo que el período prescrito, las ejecuciones de la siguiente tarea y las posteriores pueden comenzar tarde.

Comenzando tareas con un retraso fijo

El siguiente ejemplo programa una tarea para comenzar después de diez minutos, y luego repetidamente con un retraso de un minuto entre el final de una tarea y el inicio de la siguiente.

```
ScheduledFuture<?> future = pool.scheduleWithFixedDelay(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

La ejecución de la tarea continuará de acuerdo con el cronograma hasta que la `pool` se cierre, el `future` se cancele o una de las tareas encuentre una excepción.

Manejar Ejecución Rechazada

Si

1. intenta enviar tareas a un Ejecutor de apagado o
2. la cola está saturada (solo es posible con las encuadernadas) y se ha alcanzado el número máximo de subprocesos,

`RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor)` .

El comportamiento predeterminado es que recibirá una excepción RejectedExecutionException en la persona que llama. Pero hay más comportamientos predefinidos disponibles:

- **ThreadPoolExecutor.AbortPolicy** (por defecto, lanzará REE)
- **ThreadPoolExecutor.CallerRunsPolicy** (ejecuta la tarea en el subproceso de la persona que llama, *bloqueándolo*)
- **ThreadPoolExecutor.DiscardPolicy** (descartar tarea silenciosamente)
- **ThreadPoolExecutor.DiscardOldestPolicy** (descarta silenciosamente la tarea **más antigua** en la cola y reintenta la ejecución de la nueva tarea)

Puedes establecerlos usando uno de los constructores de ThreadPool:

```
public ThreadPoolExecutor(int corePoolSize,
                      int maximumPoolSize,
                      long keepAliveTime,
                      TimeUnit unit,
                      BlockingQueue<Runnable> workQueue,
                      RejectedExecutionHandler handler) // <--
```

```
public ThreadPoolExecutor(int corePoolSize,
                      int maximumPoolSize,
                      long keepAliveTime,
                      TimeUnit unit,
                      BlockingQueue<Runnable> workQueue,
                      ThreadFactory threadFactory,
                      RejectedExecutionHandler handler) // <--
```

También puede implementar su propio comportamiento al extender la interfaz RejectedExecutionHandler :

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
```

diferencias de manejo de excepciones de submit () vs execute ()

Generalmente, el comando de ejecución () se usa para las llamadas de disparo y olvido (sin necesidad de analizar el resultado) y el comando de envío () se usa para analizar el resultado del objeto futuro.

Debemos ser conscientes de la diferencia clave de los mecanismos de manejo de excepciones entre estos dos comandos.

El marco se traga las excepciones de submit () si no las atrapó.

Código de ejemplo para entender la diferencia:

Caso 1: envíe el comando Ejecutable con ejecutar (), que informa la Excepción.

```
import java.util.concurrent.*;
import java.util.*;

public class ExecuteSubmitDemo {
    public ExecuteSubmitDemo() {
        System.out.println("creating service");
        ExecutorService service = Executors.newFixedThreadPool(2);
        //ExtendedExecutor service = new ExtendedExecutor();
        for (int i = 0; i < 2; i++){
            service.execute(new Runnable(){
                 public void run(){
                    int a = 4, b = 0;
                    System.out.println("a and b=" + a + ":" + b);
                    System.out.println("a/b:" + (a / b));
                    System.out.println("Thread Name in Runnable after divide by
zero:"+Thread.currentThread().getName());
                 }
            });
        }
        service.shutdown();
    }
    public static void main(String args[]){
        ExecuteSubmitDemo demo = new ExecuteSubmitDemo();
    }
}
```

```
class ExtendedExecutor extends ThreadPoolExecutor {

   public ExtendedExecutor() {
       super(1, 1, 60, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(100));
   }
   // ...
   protected void afterExecute(Runnable r, Throwable t) {
     super.afterExecute(r, t);
     if (t == null && r instanceof Future<?>) {
       try {
         Object result = ((Future<?>) r).get();
       } catch (CancellationException ce) {
           t = ce;
       } catch (ExecutionException ee) {
           t = ee.getCause();
       } catch (InterruptedException ie) {
           Thread.currentThread().interrupt(); // ignore/reset
       }
     }
     if (t != null)
       System.out.println(t);
   }
 }
```

salida:

```
creating service
a and b=4:0
a and b=4:0
Exception in thread "pool-1-thread-1" Exception in thread "pool-1-thread-2"
java.lang.ArithmeticException: / by zero
        at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
        at java.lang.Thread.run(Thread.java:744)
java.lang.ArithmeticException: / by zero
        at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
        at java.lang.Thread.run(Thread.java:744)
```

Caso 2: Reemplace execute () con submit (): `service.submit(new Runnable(){` En este caso, Framework se traga las excepciones ya que el método run () no las detectó explícitamente.

salida:

```
creating service
a and b=4:0
a and b=4:0
```

Caso 3: Cambie el nuevoFixedThreadPool a ExtendedExecutor

```
//ExecutorService service = Executors.newFixedThreadPool(2);
 ExtendedExecutor service = new ExtendedExecutor();
```

salida:

```
creating service
a and b=4:0
java.lang.ArithmeticException: / by zero
a and b=4:0
java.lang.ArithmeticException: / by zero
```

He demostrado este ejemplo para cubrir dos temas: Use su ThreadPoolExecutor personalizado y maneje Exectpion con ThreadPoolExecutor personalizado.

Otra solución simple al problema anterior: cuando está utilizando el comando ExecutorService & submit normal, obtenga el objeto Future del comando submit () call call get () en Future. Capte las tres excepciones, que se han citado en la implementación del método afterExecute. Ventaja de ThreadPoolExecutor personalizado sobre este enfoque: tiene que manejar el mecanismo de manejo de excepciones en un solo lugar: ThreadPoolExecutor personalizado.

Casos de uso para diferentes tipos de construcciones de concurrencia.

1. EjecutorServicio

   `ExecutorService executor = Executors.newFixedThreadPool(50);`

   Es simple y fácil de usar. Oculta los detalles de bajo nivel de `ThreadPoolExecutor` .

   Prefiero este cuando el número de tareas `Callable/Runnable` es pequeño en número y la acumulación de tareas en una cola ilimitada no aumenta la memoria y degrada el rendimiento del sistema. Si tiene restricciones de `CPU/Memory` , prefiero usar `ThreadPoolExecutor` con restricciones de capacidad y `RejectedExecutionHandler` para manejar el rechazo de tareas.

2. CountDownLatch

   `CountDownLatch` se inicializará con un recuento determinado. Este conteo es disminuido por las llamadas al método `countDown()` . Los hilos que esperan que este conteo llegue a cero pueden llamar a uno de los métodos `await()` . La llamada `await()` bloquea el hilo hasta que el conteo llega a cero. *Esta clase permite que un subproceso java espere hasta que otro conjunto de subprocesos complete sus tareas.*

   Casos de uso:

   1. Lograr el máximo paralelismo: a veces queremos iniciar una serie de subprocesos al mismo tiempo para lograr el máximo paralelismo
   2. Espere N hilos para completar antes de iniciar la ejecución
   3. Detección de interbloqueo.

3. ThreadPoolExecutor : Proporciona más control. Si la aplicación está restringida por la cantidad de tareas pendientes ejecutables / recuperables, puede usar la cola delimitada configurando la capacidad máxima. Una vez que la cola alcanza su capacidad máxima, puede definir RejectionHandler. Java proporciona cuatro tipos de políticas `RejectedExecutionHandler`

.

1. `ThreadPoolExecutor.AbortPolicy` , el controlador lanza una `ThreadPoolExecutor.AbortPolicy` RejectedExecutionException en tiempo de ejecución al rechazarla.

2. ThreadPoolExecutor.CallerRunsPolicy`, el hilo que invoca a sí mismo ejecuta la tarea. Esto proporciona un mecanismo de control de retroalimentación simple que reducirá la velocidad a la que se envían las nuevas tareas.

3. En `ThreadPoolExecutor.DiscardPolicy` , una tarea que no se puede ejecutar simplemente se elimina.

4. `ThreadPoolExecutor.DiscardOldestPolicy` , si el ejecutor no se apaga, la tarea en la cabecera de la cola de trabajo se descarta y, a continuación, se vuelve a intentar la ejecución (lo que puede fallar de nuevo, haciendo que esto se repita).

Si desea simular `CountDownLatch` comportamiento, puede utilizar `invokeAll()` método.

4. Un mecanismo más que no ha citado es ForkJoinPool

   El `ForkJoinPool` se agregó a Java en Java 7. El `ForkJoinPool` es similar al Java `ExecutorService` pero con una diferencia. `ForkJoinPool` facilita que las tareas dividan su trabajo en tareas más pequeñas que luego se envían también a `ForkJoinPool` . El robo de tareas ocurre en `ForkJoinPool` cuando los subprocesos de trabajo libres roban tareas de la cola de subprocesos de trabajo ocupado.

   Java 8 ha introducido una API más en ExecutorService para crear un grupo de robo de trabajo. No tiene que crear `RecursiveTask` y `RecursiveAction` pero aún puede usar `ForkJoinPool` .

   ```
   public static ExecutorService newWorkStealingPool()
   ```

   > Crea un grupo de subprocesos de robo de trabajo utilizando todos los procesadores disponibles como su nivel de paralelismo objetivo.

   Por defecto, tomará el número de núcleos de CPU como parámetro.

Todos estos cuatro mecanismos son complementarios entre sí. Dependiendo del nivel de granularidad que desee controlar, debe elegir los correctos.

Espere a que se completen todas las tareas en ExecutorService

Echemos un vistazo a varias opciones para esperar a que se completen las tareas enviadas a Executor

1. ExecutorService `invokeAll()`

   > Ejecuta las tareas dadas, devolviendo una lista de futuros que mantienen su

estado y resultados cuando todo está completado.

Ejemplo:

```
import java.util.concurrent.*;
import java.util.*;

public class InvokeAllDemo{
    public InvokeAllDemo(){
        System.out.println("creating service");
        ExecutorService service =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        List<MyCallable> futureList = new ArrayList<MyCallable>();
        for (int i = 0; i < 10; i++){
            MyCallable myCallable = new MyCallable((long)i);
            futureList.add(myCallable);
        }
        System.out.println("Start");
        try{
            List<Future<Long>> futures = service.invokeAll(futureList);
        } catch(Exception err){
            err.printStackTrace();
        }
        System.out.println("Completed");
        service.shutdown();
    }
    public static void main(String args[]){
        InvokeAllDemo demo = new InvokeAllDemo();
    }
    class MyCallable implements Callable<Long>{
        Long id = 0L;
        public MyCallable(Long val){
            this.id = val;
        }
        public Long call(){
            // Add your business logic
            return id;
        }
    }
}
```

2. CountDownLatch

   Una ayuda de sincronización que permite que uno o más subprocesos esperen hasta que se complete un conjunto de operaciones que se están realizando en otros subprocesos.

   Un **CountDownLatch** se inicializa con un recuento dado. El bloque de métodos de espera hasta que el conteo actual llegue a cero debido a invocaciones del método `countDown()` , después de lo cual se liberan todos los subprocesos en espera y cualquier invocación posterior de espera regresa de inmediato. Este es un fenómeno de un solo disparo: el conteo no se puede reiniciar. Si necesita una versión que restablezca la cuenta, considere usar un **CyclicBarrier** .

3. ForkJoinPool o `newWorkStealingPool()` en Ejecutores

4. Iterar a través de todos los objetos `Future` creados después de enviar a `ExecutorService`

5. Forma recomendada de apagar desde la página de documentación de Oracle de ExecutorService :

```
void shutdownAndAwaitTermination(ExecutorService pool) {
    pool.shutdown(); // Disable new tasks from being submitted
    try {
      // Wait a while for existing tasks to terminate
      if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
        pool.shutdownNow(); // Cancel currently executing tasks
        // Wait a while for tasks to respond to being cancelled
        if (!pool.awaitTermination(60, TimeUnit.SECONDS))
            System.err.println("Pool did not terminate");
      }
    } catch (InterruptedException ie) {
      // (Re-)Cancel if current thread also interrupted
      pool.shutdownNow();
      // Preserve interrupt status
      Thread.currentThread().interrupt();
    }
```

`shutdown()`: inicia un cierre ordenado en el que se ejecutan las tareas enviadas anteriormente, pero no se aceptarán nuevas tareas.

`shutdownNow()`: intenta detener todas las tareas en ejecución activa, detiene el procesamiento de las tareas en espera y devuelve una lista de las tareas que estaban pendientes de ejecución.

En el ejemplo anterior, si sus tareas tardan más tiempo en completarse, puede cambiar si la condición se convierte en condición condicional.

Reemplazar

```
if (!pool.awaitTermination(60, TimeUnit.SECONDS))
```

con

```
while(!pool.awaitTermination(60, TimeUnit.SECONDS)) {
  Thread.sleep(60000);
```

}

Casos de uso para diferentes tipos de servicios de ejecución

Los ejecutores devuelven diferentes tipos de ThreadPools para satisfacer necesidades específicas.

1. `public static ExecutorService newSingleThreadExecutor()`

   Crea un Ejecutor que utiliza un único subproceso de trabajo que opera en una cola ilimitada

Hay una diferencia entre `newFixedThreadPool(1)` y `newSingleThreadExecutor()` como dice el documento java para este último:

> A diferencia del newFixedThreadPool (1) equivalente, se garantiza que el ejecutor devuelto no se puede volver a configurar para utilizar subprocesos adicionales.

Lo que significa que un `newFixedThreadPool` se puede reconfigurar más adelante en el programa por: `((ThreadPoolExecutor) fixedThreadPool).setMaximumPoolSize(10)` Esto no es posible para `newSingleThreadExecutor`

Casos de uso:

1. Desea ejecutar las tareas enviadas en una secuencia.
2. Solo necesitas un hilo para manejar toda tu solicitud

Contras:

1. La cola no enlazada es dañina

2. `public static ExecutorService newFixedThreadPool(int nThreads)`

> Crea un grupo de subprocesos que reutiliza un número fijo de subprocesos que operan en una cola compartida ilimitada. En cualquier momento, como máximo, las hebras nThreads serán tareas de procesamiento activas. Si se envían tareas adicionales cuando todos los subprocesos están activos, esperarán en la cola hasta que haya un subproceso disponible

Casos de uso:

1. Uso efectivo de los núcleos disponibles. Configure `nThreads` como
   `Runtime.getRuntime().availableProcessors() nThreads`
   `Runtime.getRuntime().availableProcessors()`
2. Cuando decide que el número de subprocesos no debe exceder un número en el grupo de subprocesos

Contras:

1. La cola no unida es dañina.

3. `public static ExecutorService newCachedThreadPool()`

> Crea un grupo de subprocesos que crea nuevos subprocesos según sea necesario, pero reutilizará los subprocesos construidos previamente cuando estén disponibles.

Casos de uso:

1. Para tareas asíncronas de corta duración.

Contras:

1. La cola no unida es dañina.
2. Cada nueva tarea creará un nuevo hilo si todos los hilos existentes están ocupados. Si la tarea está tomando una larga duración, se crearán más hilos, lo que degradará el rendimiento del sistema. Alternativa en este caso: `newFixedThreadPool`

4. `public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`

   Crea un grupo de subprocesos que puede programar comandos para que se ejecuten después de un retraso determinado o para que se ejecuten periódicamente.

   Casos de uso:

   1. Manejo de eventos recurrentes con retrasos, lo que ocurrirá en el futuro en cierto intervalo de tiempo

   Contras:

   1. La cola no unida es dañina.

5. `public static ExecutorService newWorkStealingPool()`

   Crea un grupo de subprocesos de robo de trabajo utilizando todos los procesadores disponibles como su nivel de paralelismo objetivo

   Casos de uso:

   1. Para dividir y vencer tipo de problemas.
   2. Uso efectivo de hilos inactivos. Subprocesos inactivos roba tareas de subprocesos ocupados.

   Contras:

   1. El tamaño de la cola sin límite es perjudicial.

Puede ver un inconveniente común en todos estos ExecutorService: cola ilimitada. Esto será abordado con ThreadPoolExecutor

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

Con `ThreadPoolExecutor` , puedes

1. Controlar el tamaño de la agrupación de hilos dinámicamente
2. Establecer la capacidad para `BlockingQueue`
3. Define `RejectionExecutionHander` cuando la cola está llena
4. `CustomThreadFactory` para agregar alguna funcionalidad adicional durante la creación de Thread `(public Thread newThread(Runnable r)`

Uso de grupos de subprocesos

Grupos de subprocesos se utilizan principalmente métodos de llamada en `ExecutorService` .

Se pueden usar los siguientes métodos para enviar trabajo para ejecución:

Método	Descripción
`submit`	Ejecuta el trabajo enviado y devuelve un futuro que puede usarse para obtener el resultado.
`execute`	Ejecute la tarea en algún momento en el futuro sin obtener ningún valor de retorno
`invokeAll`	Ejecuta una lista de tareas y devuelve una lista de futuros
`invokeAny`	Ejecuta todo pero devuelve solo el resultado de uno que ha sido exitoso (sin excepciones)

Una vez que haya terminado con el grupo de subprocesos, puede llamar a `shutdown()` para terminar el grupo de subprocesos. Esto ejecuta todas las tareas pendientes. Para esperar a que se ejecuten todas las tareas, puede recorrer `awaitTermination` o `isShutdown()` .

Lea Ejecutor, ExecutorService y grupos de subprocesos en línea:
https://riptutorial.com/es/java/topic/143/ejecutor--executorservice-y-grupos-de-subprocesos

Capítulo 60: El classpath

Introducción

La ruta de clase enumera los lugares donde el tiempo de ejecución de Java debe buscar clases y recursos. El compilador de Java también utiliza la ruta de clase para encontrar dependencias compiladas previamente y externas.

Observaciones

Carga de clases de Java

La JVM (Java Virtual Machine) cargará las clases a medida que se requieran las clases (esto se denomina carga lenta). Las ubicaciones de las clases que se utilizarán se especifican en tres lugares:

1. Los requeridos por la plataforma Java se cargan primero, como los de la biblioteca de clases de Java y sus dependencias.
2. Las clases de extensión se cargan a continuación (es decir, aquellas en `jre/lib/ext/`)
3. Luego se cargan las clases definidas por el usuario a través del classpath.

Las clases se cargan utilizando clases que son subtipos de `java.lang.ClassLoader` . Esto se describe con más detalle en este tema: Cargadores de clases .

Classpath

El classpath es un parámetro utilizado por la JVM o compilador que especifica las ubicaciones de las clases y los paquetes definidos por el usuario. Esto se puede establecer en la línea de comandos como en la mayoría de estos ejemplos o mediante una variable de entorno (`CLASSPATH`)

Examples

Diferentes formas de especificar el classpath.

Hay tres formas de configurar el classpath.

1. Se puede configurar utilizando la variable de entorno `CLASSPATH` :

```
set CLASSPATH=...          # Windows and csh
export CLASSPATH=...       # Unix ksh/bash
```

2. Se puede configurar en la línea de comando de la siguiente manera

```
java -classpath ...
javac -classpath ...
```

Tenga en cuenta que la `-classpath` (o `-cp`) tiene prioridad sobre la variable de entorno `CLASSPATH` .

3. La ruta de `Class-Path` para un archivo JAR ejecutable se especifica mediante el elemento `Class-Path` en `MANIFEST.MF` :

```
Class-Path: jar1-name jar2-name directory-name/jar3-name
```

Tenga en cuenta que esto solo se aplica cuando el archivo JAR se ejecuta así:

```
java -jar some.jar ...
```

En este modo de ejecución, la opción `-classpath` y la variable de entorno CLASSPATH se ignorarán, incluso si el archivo JAR no tiene ningún elemento `Class-Path` .

Si no se especifica ninguna ruta de clase, entonces la ruta de `java -jar` predeterminada es el archivo JAR seleccionado cuando se utiliza `java -jar` , o el directorio actual de lo contrario.

Relacionado:

- https://docs.oracle.com/javase/tutorial/deployment/jar/downman.html
- http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html

Agregando todos los JARs en un directorio a la ruta de clase

Si desea agregar todos los archivos JAR en el directorio a la ruta de clase, puede hacer esto de manera concisa usando la sintaxis de comodín de la ruta de clase; por ejemplo:

```
someFolder/*
```

Esto le indica a la JVM que agregue todos los archivos JAR y ZIP en el directorio `someFolder` a la ruta de `someFolder` . Esta sintaxis se puede usar en un argumento `-cp` , una variable de entorno `CLASSPATH` o un atributo `Class-Path` en el archivo de manifiesto de un archivo JAR ejecutable. Consulte *Configuración de la ruta de clase:* comodines de *ruta de clase* para ver ejemplos y advertencias.

Notas:

1. Los comodines de Classpath se introdujeron por primera vez en Java 6. Las versiones anteriores de Java no tratan a "*" como un comodín.
2. No puede poner otros caracteres antes o después de " *"; por ejemplo, "someFolder /* .jar" no es un comodín.
3. Un comodín coincide solo con archivos con el sufijo ".jar" o ".JAR". Los archivos ZIP se ignoran, al igual que los archivos JAR con sufijos diferentes.
4. Un comodín solo coincide con los archivos JAR en el directorio, no en sus subdirectorios.
5. Cuando un grupo de archivos JAR coincide con una entrada de comodín, no se especifica su orden relativo en la ruta de clase.

Sintaxis de ruta de clase

La ruta de clase es una secuencia de entradas que son nombres de ruta de directorio, nombres de ruta de archivos JAR o ZIP o especificaciones de comodín JAR / ZIP.

- Para una ruta de clase especificada en la línea de comando (por ejemplo, `-classpath`) o como una variable de entorno, las entradas deben estar separadas con `;` (punto y coma) en Windows, o `:` (dos puntos) en otras plataformas (Linux, UNIX, MacOSX, etc.).
- Para el elemento `Class-Path` en `MANIFEST.MF` un archivo JAR, use un solo espacio para separar las entradas.

A veces es necesario incrustar un espacio en una entrada classpath

- Cuando se especifica la ruta de clase en la línea de comando, es simplemente una cuestión de usar la comilla de shell apropiada. Por ejemplo:

  ```
  export CLASSPATH="/home/user/My JAR Files/foo.jar:second.jar"
  ```

  (Los detalles pueden depender del comando shell que utilice).

- Cuando la ruta de clase se especifica en un archivo JAR, un archivo "MANIFEST.MF", se debe utilizar la codificación de URL.

  ```
  Class-Path: /home/user/My%20JAR%20Files/foo.jar second.jar
  ```

Classpath dinámico

A veces, solo agregar todos los archivos JAR desde una carpeta no es suficiente, por ejemplo, cuando tiene un código nativo y necesita seleccionar un subconjunto de archivos JAR. En este caso, necesitas dos métodos `main()` . El primero construye un cargador de clases y luego usa este cargador de clases para llamar al segundo `main()` .

Este es un ejemplo que selecciona el JAR nativo SWT correcto para su plataforma, agrega todos los JAR de su aplicación y luego invoca el método `main()` real: Cree una aplicación SWT Java multiplataforma

Cargar un recurso desde el classpath

Puede ser útil cargar un recurso (imagen, archivo de texto, propiedades, KeyStore, ...) que está empaquetado dentro de un JAR. Para este propósito, podemos usar los `Class` y `ClassLoader` s.

Supongamos que tenemos la siguiente estructura de proyecto:

```
program.jar
|
\-com
  \-project
    |
```

```
    |-file.txt
    \-Test.class
```

Y queremos acceder al contenido de `file.txt` desde la clase `Test` . Podemos hacerlo preguntando al cargador de clases:

```
InputStream is = Test.class.getClassLoader().getResourceAsStream("com/project/file.txt");
```

Al usar el cargador de clases, necesitamos especificar la ruta de acceso completa de nuestro recurso (cada paquete).

O alternativamente, podemos preguntar directamente al objeto de la clase Test.

```
InputStream is = Test.class.getResourceAsStream("file.txt");
```

Usando el objeto de clase, la ruta es relativa a la clase en sí. `Test.class` estar nuestro `Test.class` en el paquete `com.project` , al igual que `file.txt` , no necesitamos especificar ninguna ruta.

Sin embargo, podemos usar rutas absolutas desde el objeto de clase, de esta manera:

```
 is = Test.class.getResourceAsStream("/com/project/file.txt");
```

Asignación de nombres de clases a rutas de acceso

La cadena de herramientas estándar de Java (y las herramientas de terceros diseñadas para interactuar con ellas) tienen reglas específicas para asignar los nombres de las clases a las rutas de los archivos y otros recursos que los representan.

Las asignaciones son las siguientes

- Para las clases en el paquete predeterminado, las rutas de acceso son nombres de archivos simples.
- Para las clases en un paquete con nombre, los componentes del nombre del paquete se asignan a directorios.
- Para las clases anidadas e internas nombradas, el componente de nombre de archivo se forma al unir los nombres de clase con un carácter `$` .
- Para las clases internas anónimas, los números se utilizan en lugar de los nombres.

Esto se ilustra en la siguiente tabla:

Nombre de la clase	Fuente de acceso	Nombre de archivo
`SomeClass`	`SomeClass.java`	`SomeClass.class`
`com.example.SomeClass`	`com/example/SomeClass.java`	`com/example/SomeClass.class`
`SomeClass.Inner`	(en `SomeClass.java`)	`SomeClass$Inner.class`
`SomeClass` clases internas	(en `SomeClass.java`)	`SomeClass$1.class` ,

Nombre de la clase	Fuente de acceso	Nombre de archivo
de anon		`SomeClass$2.class` , etc.

Qué significa el classpath: cómo funcionan las búsquedas

El propósito del classpath es decirle a una JVM dónde encontrar clases y otros recursos. El significado de la ruta de clase y el proceso de búsqueda están entrelazados.

El classpath es una forma de ruta de búsqueda que especifica una secuencia de *ubicaciones* para buscar recursos. En una ruta de clase estándar, estos lugares son un directorio en el sistema de archivos del host, un archivo JAR o un archivo ZIP. En cada caso, la ubicación es la raíz de un *espacio* de *nombres* que se buscará.

El procedimiento estándar para buscar una clase en el classpath es el siguiente:

1. Asigne el nombre de la clase a un archivo de clase relativo nombre de ruta `RP` . La asignación de nombres de clase a nombres de archivo de clase se describe en otra parte.

2. Para cada entrada `E` en el classpath:

    - Si la entrada es un directorio de sistema de archivos:
        - Resuelva el `RP` relativo a `E` para dar un `AP` nombre de ruta absoluto.
        - Probar si `AP` es una ruta para un archivo existente.
        - Si es así, cargue la clase de ese archivo
    - Si la entrada es un archivo JAR o ZIP:
        - Busque `RP` en el índice de archivos JAR / ZIP.
        - Si existe la entrada del archivo JAR / ZIP correspondiente, cargue la clase de esa entrada.

El procedimiento para buscar un recurso en la ruta de clase depende de si la ruta del recurso es absoluta o relativa. Para una ruta de recursos absoluta, el procedimiento es el anterior. Para una ruta de recursos relativa resuelta utilizando `Class.getResource` o `Class.getResourceAsStream` , la ruta para el paquete de clases se añade antes de la búsqueda.

(Tenga en cuenta que estos son los procedimientos implementados por los cargadores de clases Java estándar. Un cargador de clases personalizado podría realizar la búsqueda de manera diferente).

La ruta de clases de arranque

Los cargadores de clases de Java normales buscan clases primero en la ruta de clase de rutina de carga, antes de verificar las extensiones y la ruta de clase de la aplicación. De forma predeterminada, la ruta de clase de arranque consta del archivo "rt.jar" y algunos otros archivos JAR importantes que se suministran con la instalación de JRE. Estos proporcionan todas las clases en la biblioteca de clases estándar de Java SE, junto con varias clases de implementación "internas".

En circunstancias normales, no necesita preocuparse por esto. De forma predeterminada, los

comandos como `java` , `javac` , etc. usarán las versiones apropiadas de las bibliotecas de tiempo de ejecución.

Muy ocasionalmente, es necesario anular el comportamiento normal del tiempo de ejecución de Java utilizando una versión alternativa de una clase en las bibliotecas estándar. Por ejemplo, puede encontrar un error "show stopper" en las bibliotecas de tiempo de ejecución que no puede solucionar por medios normales. En tal situación, es posible crear un archivo JAR que contenga la clase modificada y luego agregarlo a la ruta de clases de arranque que inicia la JVM.

El comando `java` proporciona las siguientes opciones `-X` para modificar la ruta de clase de arranque:

- `-Xbootclasspath:<path>` reemplaza la `-Xbootclasspath:<path>` inicio actual con la ruta proporcionada.
- `-Xbootclasspath/a:<path>` agrega la ruta de acceso proporcionada a la ruta de clase de arranque actual.
- `-Xbootclasspath/p:<path>` antepasa la ruta proporcionada a la `-Xbootclasspath/p:<path>` clase de arranque actual.

Tenga en cuenta que cuando utiliza las opciones de la ruta de inicio para reemplazar o anular una clase de Java (etcétera), técnicamente está modificando Java. *Puede haber* implicaciones de licencia si luego distribuye su código. (Consulte los términos y condiciones de la licencia binaria de Java ... y consulte a un abogado).

Lea El classpath en línea: https://riptutorial.com/es/java/topic/3720/el-classpath

Capítulo 61: El Comando de Java - 'java' y 'javaw'

Sintaxis

- `java [ <opt> ... ] <class-name> [ <argument> ... ]`
- `java [ <opt> ... ] -jar <jar-file-pathname> [ <argument> ... ]`

Observaciones

El comando `java` se utiliza para ejecutar una aplicación Java desde la línea de comandos. Está disponible como parte de cualquier Java SE JRE o JDK.

En los sistemas Windows hay dos variantes del comando `java` :

- La variante de `java` inicia la aplicación en una nueva ventana de consola.
- La variante `javaw` inicia la aplicación sin crear una nueva ventana de consola.

En otros sistemas (por ejemplo, Linux, Mac OSX, UNIX) solo se proporciona el comando `java` y no se abre una nueva ventana de consola.

El símbolo `<opt>` en la sintaxis denota una opción en la línea de comando `java` . Los temas "Opciones de Java" y "Opciones de tamaño de pila y pila" cubren las opciones más utilizadas. Otros están cubiertos en el tema Banderas JVM .

Examples

Ejecutando un archivo JAR ejecutable

Los archivos JAR ejecutables son la forma más sencilla de ensamblar código Java en un solo archivo que se puede ejecutar. * (Nota editorial: la creación de archivos JAR debe estar cubierta por un tema separado.) *

Suponiendo que tiene un archivo JAR ejecutable con la ruta de acceso `<jar-path>` , debería poder ejecutarlo de la siguiente manera:

```
java -jar <jar-path>
```

Si el comando requiere argumentos de línea de comandos, agréguelos después de `<jar-path>` . Por ejemplo:

```
java -jar <jar-path> arg1 arg2 arg3
```

Si necesita proporcionar opciones de JVM adicionales en la línea de comandos de `java` , deben ir

antes que la opción `-jar` . Tenga en cuenta que una opción `-cp` / `-classpath` se ignorará si utiliza `-jar` . La ruta de clase de la aplicación está determinada por el archivo JAR manifiesto.

Ejecutando aplicaciones Java a través de una clase "principal"

Cuando una aplicación no se ha empaquetado como un JAR ejecutable, debe proporcionar el nombre de una clase de punto de entrada en la línea de comandos de `java` .

Ejecutando la clase HelloWorld

El ejemplo "HelloWorld" se describe en Crear un nuevo programa Java . Consiste en una clase única llamada `HelloWorld` que satisface los requisitos para un punto de entrada.

Suponiendo que el archivo "HelloWorld.class" (compilado) está en el directorio actual, se puede iniciar de la siguiente manera:

```
java HelloWorld
```

Algunas cosas importantes a tener en cuenta son:

- Debemos proporcionar el nombre de la clase: no el nombre de ruta para el archivo ".class" o el archivo ".java".
- Si la clase se declara en un paquete (como la mayoría de las clases de Java), entonces el nombre de la clase que suministramos al comando `java` debe ser el nombre completo de la clase. Por ejemplo, si se declara `SomeClass` en el paquete `com.example` , entonces el nombre completo de la clase será `com.example.SomeClass` .

Especificando un classpath

A menos que estemos usando la sintaxis del comando `java -jar` , el comando `java` busca la clase que se cargará buscando la ruta de clase; ver el classpath . El comando anterior se basa en que la ruta de clase predeterminada es (o incluye) el directorio actual. Podemos ser más explícitos sobre esto especificando la ruta de `-cp` que se usará usando la opción `-cp` .

```
java -cp . HelloWorld
```

Esto dice que hacer que el directorio actual (que es a lo que "." Se refiere) sea la única entrada en el classpath.

El `-cp` es una opción que es procesada por el comando `java` . Todas las opciones destinadas al comando `java` deben estar antes del nombre de clase. Cualquier cosa después de la clase se tratará como un argumento de línea de comando para la aplicación Java y se pasará a la aplicación en la `String[]` que se pasa al método `main` .

(Si no se proporciona la opción `-cp` , `java` usará la ruta de clase que proporciona la variable de entorno `CLASSPATH` . Si esa variable no está establecida o está vacía, `java` usa "." Como la ruta de clase predeterminada).

Clases de punto de entrada

Una clase de punto de entrada Java tiene un método `main` con la siguiente firma y modificadores:

```
public static void main(String[] args)
```

> Nota al margen: debido a cómo funcionan las matrices, también puede ser `(String args[])`

Cuando el comando `java` inicia la máquina virtual, carga las clases de punto de entrada especificadas e intenta encontrar las `main` . Si tiene éxito, los argumentos de la línea de comando se convierten en objetos de `String` Java y se ensamblan en una matriz. Si `main` se invoca así, la matriz *no* será `null` y no contendrá ninguna entrada `null` .

Un método de clase de punto de entrada válido debe hacer lo siguiente:

- Ser nombrado `main` (distingue entre mayúsculas y minúsculas)
- Sé `public` y `static`
- Tener un tipo de retorno `void`
- Tener un solo argumento con una `String[]` . El argumento debe estar presente y no se permite más de un argumento.
- Ser genérico: los parámetros de tipo no están permitidos.
- Tener una clase envolvente no genérica, de nivel superior (no anidada o interna)

Es convencional declarar la clase como `public` pero esto no es estrictamente necesario. Desde Java 5 en adelante, el tipo de argumento del método `main` puede ser una `String` varargs en lugar de una cadena de cadenas. Opcionalmente, `main` puede lanzar excepciones, y su parámetro puede tener cualquier nombre, pero convencionalmente es `args` .

Puntos de entrada de JavaFX

Desde Java 8 en adelante, el comando `java` también puede iniciar directamente una aplicación JavaFX. JavaFX está documentado en la etiqueta JavaFX , pero un punto de entrada de JavaFX debe hacer lo siguiente:

- Extender `javafx.application.Application`
- Sé `public` y no `abstract`
- No ser genérico o anidado
- Tener un constructor no-args `public` explícito o implícito

Solución de problemas del comando 'java'

Este ejemplo cubre errores comunes con el uso del comando 'java'.

"Comando no encontrado"

Si recibe un mensaje de error como:

```
java: command not found
```

cuando intenta ejecutar el comando `java` , esto significa que no hay un comando `java` en la ruta de búsqueda de comandos de su shell. La causa podría ser:

- no tienes instalado un JRE o JDK de Java,
- no ha actualizado la `PATH` entorno `PATH` (correctamente) en su archivo de inicialización de shell, o
- no ha "obtenido" el archivo de inicialización relevante en el shell actual.

Consulte "Instalación de Java" para conocer los pasos que debe seguir.

"No se pudo encontrar o cargar la clase principal"

Este mensaje de error es generado por el comando `java` si no ha podido encontrar / cargar la clase de punto de entrada que ha especificado. En términos generales, hay tres razones generales por las que esto puede suceder:

- Ha especificado una clase de punto de entrada que no existe.
- La clase existe, pero la has especificado incorrectamente.
- La clase existe y la ha especificado correctamente, pero Java no puede encontrarla porque la ruta de clase es incorrecta.

Aquí hay un procedimiento para diagnosticar y resolver el problema:

1. Averigüe el nombre completo de la clase de punto de entrada.

    - Si tiene un código fuente para una clase, entonces el nombre completo consiste en el nombre del paquete y el nombre de la clase simple. La instancia de la clase "Main" se declara en el paquete "com.example.myapp" y su nombre completo es "com.example.myapp.Main".
    - Si tiene un archivo de clase compilado, puede encontrar el nombre de la clase ejecutando `javap` en él.
    - Si el archivo de clase está en un directorio, puede inferir el nombre completo de la clase a partir de los nombres de directorio.
    - Si el archivo de clase está en un archivo JAR o ZIP, puede inferir el nombre completo de la clase a partir de la ruta del archivo en el archivo JAR o ZIP.

2. Mira el mensaje de error del comando `java` . El mensaje debe terminar con el nombre completo de la clase que `java` está tratando de usar.

    - Compruebe que coincida exactamente con el nombre de clase completo para la clase de punto de entrada.
    - No debe terminar con ".java" o ".class".
    - No debe contener barras o ningún otro carácter que no sea legal en un identificador de Java [1] .
    - La carcasa del nombre debe coincidir exactamente con el nombre completo de la clase.

3. Si está utilizando el nombre de clase correcto, asegúrese de que la clase esté realmente en la ruta de clase:

    - Calcula el nombre de ruta al que se asigna el nombre de clase; ver Asignar nombres de clases a rutas de acceso
    - Averigua cuál es el classpath; Vea este ejemplo: Diferentes maneras de especificar la ruta de clase.
    - Mire cada uno de los archivos JAR y ZIP en la ruta de clase para ver si contienen una clase con la ruta de acceso requerida.
    - Mire cada directorio para ver si la ruta de acceso se resuelve en un archivo dentro del directorio.

Si la comprobación del classpath a mano no encontró el problema, puede agregar las opciones `-Xdiag` y `-XshowSettings` . La primera lista todas las clases que se cargan, y la última imprime configuraciones que incluyen la ruta de clase efectiva para la JVM.

Finalmente, hay algunas causas *oscuras* para este problema:

- Un archivo JAR ejecutable con un atributo de `Main-Class` que especifica una clase que no existe.
- Un archivo JAR ejecutable con un atributo de `Class-Path` incorrecto.
- Si se equivoca [2] las opciones antes de que el nombre de la clase, el `java` comando puede tratar de interpretar uno de ellos como el nombre de clase.
- Si alguien ha ignorado las reglas de estilo de Java y ha usado identificadores de paquete o clase que difieren solo en letras mayúsculas, y está ejecutando en una plataforma que trata las letras mayúsculas en los nombres de archivo como no significativas.
- Problemas con homoglifos en los nombres de clase en el código o en la línea de comando.

"El método principal no se encuentra en la clase <nombre>"

Este problema ocurre cuando el comando `java` puede encontrar y cargar la clase que usted nominó, pero luego no puede encontrar un método de punto de entrada.

Hay tres posibles explicaciones:

- Si está intentando ejecutar un archivo JAR ejecutable, entonces el manifiesto de JAR tiene un atributo "Clase principal" incorrecto que especifica una clase que no es una clase de punto de entrada válida.
- Le ha dicho al comando `java` una clase que no es una clase de punto de entrada.
- La clase de punto de entrada es incorrecta; ver Clases de punto de entrada para más información.

Otros recursos

- ¿Qué significa "no se pudo encontrar o cargar la clase principal"?
- http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html

---

1 - Desde Java 8 y `java` posteriores, el comando `java` asignará de manera útil un separador de nombre de archivo ("/" o "") a un punto (."). Sin embargo, este comportamiento no está documentado en las páginas del manual.

2 - Un caso muy oscuro es si copia y pega un comando de un documento formateado donde el editor de texto ha usado un "guión largo" en lugar de un guión normal.

Ejecutando una aplicación Java con dependencias de biblioteca

Esta es una continuación de los ejemplos de "clase principal" y "JAR ejecutable" .

Las aplicaciones Java típicas consisten en un código específico de la aplicación y varios códigos de biblioteca reutilizables que ha implementado o que ha sido implementado por terceros. Estas últimas se conocen comúnmente como dependencias de bibliotecas y, por lo general, se empaquetan como archivos JAR.

Java es un lenguaje enlazado dinámicamente. Cuando ejecuta una aplicación Java con dependencias de biblioteca, la JVM necesita saber dónde están las dependencias para poder cargar las clases según sea necesario. En términos generales, hay dos maneras de lidiar con esto:

- La aplicación y sus dependencias se pueden volver a empaquetar en un solo archivo JAR que contiene todas las clases y recursos necesarios.
- Se le puede decir a la JVM dónde encontrar los archivos JAR dependientes a través del classpath de tiempo de ejecución.

Para un archivo JAR ejecutable, la ruta de clase de tiempo de ejecución se especifica mediante el atributo de manifiesto "Class-Path". *(Nota editorial: Esto se debe describir en un Tema separado en el comando `jar`).* De lo contrario, la ruta de `-cp` tiempo de ejecución debe suministrarse mediante la opción `-cp` o la variable de entorno `CLASSPATH` .

Por ejemplo, supongamos que tenemos una aplicación Java en el archivo "myApp.jar" cuya clase de punto de entrada es `com.example.MyApp` . Supongamos también que la aplicación depende de los archivos JAR de la biblioteca "lib / library1.jar" y "lib / library2.jar". Podríamos iniciar la aplicación usando el comando `java` como sigue en una línea de comando:

```
$ # Alternative 1 (preferred)
$ java -cp myApp.jar:lib/library1.jar:lib/library2.jar com.example.MyApp

$ # Alternative 2
$ export CLASSPATH=myApp.jar:lib/library1.jar:lib/library2.jar
$ java com.example.MyApp
```

(En Windows, usaría `;` lugar de `:` como el separador de classpath, y establecería la variable `CLASSPATH` (local) usando `set` lugar de `export`).

Si bien un desarrollador de Java se sentiría cómodo con eso, no es "fácil de usar". Por lo tanto, es una práctica común escribir un script de shell simple (o un archivo por lotes de Windows) para ocultar los detalles que el usuario no necesita conocer. Por ejemplo, si coloca el siguiente script de shell en un archivo llamado "myApp", lo convierte en ejecutable y lo coloca en un directorio en

la ruta de búsqueda de comandos:

```
#!/bin/bash
# The 'myApp' wrapper script

export DIR=/usr/libexec/myApp
export CLASSPATH=$DIR/myApp.jar:$DIR/lib/library1.jar:$DIR/lib/library2.jar
java com.example.MyApp
```

entonces podrías ejecutarlo de la siguiente manera:

```
$ myApp arg1 arg2 ...
```

Cualquier argumento en la línea de comando se pasará a la aplicación Java a través de la expansión `"$@"` . (Puedes hacer algo similar con un archivo por lotes de Windows, aunque la sintaxis es diferente).

Espacios y otros caracteres especiales en argumentos.

En primer lugar, el problema de manejar espacios en argumentos NO es realmente un problema de Java. Más bien, es un problema que debe ser manejado por el comando shell que está utilizando cuando ejecuta un programa Java.

Como ejemplo, supongamos que tenemos el siguiente programa simple que imprime el tamaño de un archivo:

```
import java.io.File;

public class PrintFileSizes {

    public static void main(String[] args) {
        for (String name: args) {
            File file = new File(name);
            System.out.println("Size of '" + file + "' is " + file.size());
        }
    }
}
```

Ahora supongamos que queremos imprimir el tamaño de un archivo cuyo nombre de ruta tiene espacios en él; Por ejemplo, `/home/steve/Test File.txt` . Si ejecutamos el comando así:

```
$ java PrintFileSizes /home/steve/Test File.txt
```

el shell no sabrá que `/home/steve/Test File.txt` es en realidad un nombre de ruta. En su lugar, pasará 2 argumentos distintos a la aplicación Java, que intentará encontrar sus respectivos tamaños de archivo, y fallará porque los archivos con esas rutas (probablemente) no existen.

Soluciones usando un shell POSIX

Las carcasas POSIX incluyen `sh` , así como derivados, como `bash` y `ksh` . Si está utilizando uno de

estos shells, puede resolver el problema *citando* el argumento.

```
$ java PrintFileSizes "/home/steve/Test File.txt"
```

Las comillas dobles alrededor de la ruta de acceso le indican al shell que se debe pasar como un solo argumento. Las citas serán eliminadas cuando esto suceda. Hay un par de otras maneras de hacer esto:

```
$ java PrintFileSizes '/home/steve/Test File.txt'
```

Las comillas simples (rectas) se tratan como comillas dobles, excepto que también suprimen varias expansiones dentro del argumento.

```
$ java PrintFileSizes /home/steve/Test\ File.txt
```

Una barra invertida escapa al siguiente espacio y hace que no se interprete como un separador de argumentos.

Para obtener una documentación más completa, incluidas las descripciones de cómo tratar otros caracteres especiales en los argumentos, consulte el tema de cita en la documentación de Bash .

Solución para Windows

El problema fundamental para Windows es que, a nivel del sistema operativo, los argumentos se pasan a un proceso secundario como una sola cadena (fuente). Esto significa que la responsabilidad final de analizar (o volver a analizar) la línea de comandos recae en el programa o en sus bibliotecas de tiempo de ejecución. Hay mucha inconsistencia.

En el caso de Java, para acortar una larga historia corta:

- Puede colocar comillas dobles alrededor de un argumento en un comando `java` , y eso le permitirá pasar argumentos con espacios en ellos.
- Aparentemente, el comando `java` sí está analizando la cadena de comandos, y lo hace más o menos correcto
- Sin embargo, cuando intenta combinar esto con el uso de `SET` y la sustitución de variables en un archivo por lotes, se vuelve realmente complicado si se eliminan las comillas dobles.
- El shell `cmd.exe` aparentemente tiene otros mecanismos de escape; por ejemplo, duplicar las comillas dobles y usar `^` escapes.

Para más detalles, consulte la documentación de Batch-File .

Opciones de Java

El comando `java` soporta una amplia gama de opciones:

- Todas las opciones comienzan con un único guión o signo menos (`-`): no se admite la convención de GNU / Linux de usar `--` para opciones "largas".
- Las opciones deben aparecer antes del `<classname>` o del `-jar <jarfile>` para ser reconocido. Cualquier argumento posterior a ellos se tratará como argumentos que se pasarán a la aplicación Java que se está ejecutando.
- Las opciones que no comienzan con `-X` o `-XX` son opciones estándar. Puede confiar en todas las implementaciones de Java [1] para admitir cualquier opción estándar.
- Las opciones que comienzan con `-X` son opciones no estándar y pueden retirarse de una versión de Java a la siguiente.
- Las opciones que comienzan con `-XX` son opciones avanzadas y también se pueden retirar.

Configurando las propiedades del sistema con `-D`

La opción `-D<property>=<value>` se usa para establecer una propiedad en el objeto `Properties` del sistema. Este parámetro se puede repetir para establecer diferentes propiedades.

Opciones de memoria, pila y recolector de basura

Las opciones principales para controlar los tamaños de pila y pila se documentan en Configuración de los tamaños de pila, PermGen y Pila . *(Nota editorial: las opciones del recolector de basura se deben describir en el mismo tema).*

Habilitar y deshabilitar aserciones

El `-ea` y `-da` opciones respectivamente habilitar y deshabilitar Java `assert` comprobación:

- Todas las comprobaciones de aserción están deshabilitadas por defecto.
- La opción `-ea` permite verificar todas las aserciones.
- El `-ea:<packagename>...` habilita la verificación de aserciones en un paquete *y todos los subpaquetes* .
- El `-ea:<classname>...` habilita la verificación de aserciones en una clase.
- La opción `-da` desactiva la comprobación de todas las aserciones
- El `-da:<packagename>...` deshabilita la verificación de aserciones en un paquete *y todos los subpaquetes* .
- El `-da:<classname>...` deshabilita la verificación de aserciones en una clase.
- La opción `-esa` permite verificar todas las clases del sistema.
- La opción `-dsa` desactiva la comprobación de todas las clases del sistema.

Las opciones se pueden combinar. Por ejemplo.

```
$ # Enable all assertion checking in non-system classes
$ java -ea -dsa MyApp
```

```
$ # Enable assertions for all classes in a package except for one.
$ java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat MyApp
```

Tenga en cuenta que habilitar la verificación de afirmaciones puede alterar el comportamiento de una programación Java.

- Es probable que la aplicación sea más lenta en general.
- Puede hacer que los métodos específicos demoren más en ejecutarse, lo que podría cambiar el tiempo de los subprocesos en una aplicación de subprocesos múltiples.
- Puede introducir relaciones fortuitas *antes de* que desaparezcan las anomalías de la memoria.
- Un implementado incorrectamente `assert` declaración podría tener efectos secundarios no deseados.

Seleccionando el tipo de máquina virtual

Los `-client` y `-server` opciones le permiten seleccionar entre dos formas diferentes de la HotSpot VM:

- El formulario "cliente" está optimizado para las aplicaciones de usuario y ofrece un inicio más rápido.
- El formulario "servidor" está optimizado para aplicaciones de larga ejecución. La estadística de captura lleva más tiempo durante el "calentamiento" de JVM, lo que permite al compilador JIT hacer un mejor trabajo de optimización del código nativo.

De forma predeterminada, la JVM se ejecutará en modo de 64 bits si es posible, dependiendo de las capacidades de la plataforma. Las opciones `-d32` y `-d64` permiten seleccionar el modo explícitamente.

---

1 - Verifique el manual oficial del comando `java` . A veces, una opción *estándar* se describe como "sujeto a cambio".

Lea El Comando de Java - 'java' y 'javaw' en línea: https://riptutorial.com/es/java/topic/5791/el-comando-de-java----java--y--javaw-

Capítulo 62: Encapsulacion

Introducción

Imagine que tiene una clase con algunas variables bastante importantes y que fueron configuradas (por otros programadores desde su código) a valores inaceptables. Su código trajo errores en su código. Como solución, en OOP, permite que el estado de un objeto (almacenado en sus variables) se modifique solo a través de métodos. Ocultar el estado de un objeto y proporcionar toda la interacción a través de los métodos de un objeto se conoce como encapsulación de datos.

Observaciones

Es mucho más fácil comenzar marcando una variable como `private` y exponerla si es necesario que ocultar una variable que ya es `public` .

Hay una excepción en la que la encapsulación puede no ser beneficiosa: estructuras de datos "simples" (clases cuyo único propósito es mantener variables).

```
public class DumbData {
    public String name;
    public int timeStamp;
    public int value;
}
```

En este caso, la interfaz de la clase *es* la información que contiene.

Tenga en cuenta que las variables marcadas como `final` pueden marcarse como `public` sin violar la encapsulación porque no se pueden cambiar después de establecerlas.

Examples

Encapsulamiento para mantener invariantes.

Hay dos partes de una clase: la interfaz y la implementación.

La interfaz es la funcionalidad expuesta de la clase. Sus métodos y variables públicas forman parte de la interfaz.

La implementación es el funcionamiento interno de una clase. Otras clases no deberían necesitar conocer la implementación de una clase.

La encapsulación se refiere a la práctica de ocultar la implementación de una clase de cualquier usuario de esa clase. Esto permite que la clase haga suposiciones sobre su estado interno.

Por ejemplo, toma esta clase que representa un ángulo:

```
public class Angle {

    private double angleInDegrees;
    private double angleInRadians;

    public static Angle angleFromDegrees(double degrees){
        Angle a = new Angle();
        a.angleInDegrees = degrees;
        a.angleInRadians = Math.PI*degrees/180;
        return a;
    }

    public static Angle angleFromRadians(double radians){
        Angle a = new Angle();
        a.angleInRadians = radians;
        a.angleInDegrees = radians*180/Math.PI;
        return a;
    }

    public double getDegrees(){
        return angleInDegrees;
    }

    public double getRadians(){
        return angleInRadians;
    }

    public void setDegrees(double degrees){
        this.angleInDegrees = degrees;
        this.angleInRadians = Math.PI*degrees/180;
    }

    public void setRadians(double radians){
        this.angleInRadians = radians;
        this.angleInDegrees = radians*180/Math.PI;
    }
    private Angle(){}
}
```

Esta clase se basa en un supuesto básico (o *invariante*): **angleInDegrees y angleInRadians siempre están sincronizados** . Si los miembros de la clase fueran públicos, no habría garantías de que las dos representaciones de ángulos estén correlacionadas.

Encapsulamiento para reducir el acoplamiento.

La encapsulación le permite realizar cambios internos en una clase sin afectar ningún código que llame a la clase. Esto reduce el *acoplamiento* , o cuánto una clase dada depende de la implementación de otra clase.

Por ejemplo, cambiemos la implementación de la clase Angle del ejemplo anterior:

```
public class Angle {

    private double angleInDegrees;

    public static Angle angleFromDegrees(double degrees){
        Angle a = new Angle();
```

```
        a.angleInDegrees = degrees;
        return a;
    }

    public static Angle angleFromRadians(double radians){
        Angle a = new Angle();
        a.angleInDegrees = radians*180/Math.PI;
        return a;
    }

    public double getDegrees(){
        return angleInDegrees;
    }

    public double getRadians(){
        return angleInDegrees*Math.PI / 180;
    }

    public void setDegrees(double degrees){
        this.angleInDegrees = degrees;
    }

    public void setRadians(double radians){
        this.angleInDegrees = radians*180/Math.PI;
    }

    private Angle(){}
}
```

La implementación de esta clase ha cambiado para que solo almacene una representación del ángulo y calcule el otro ángulo cuando sea necesario.

Sin embargo, **la implementación cambió, pero la interfaz no** . Si una clase que llama se basó en acceder al método angleInRadians, tendría que cambiarse para usar la nueva versión de `Angle` . Las clases que llaman no deberían preocuparse por la representación interna de una clase.

Lea Encapsulacion en línea: https://riptutorial.com/es/java/topic/1295/encapsulacion

Capítulo 63: Enchufes

Introducción

Un zócalo es un punto final de un enlace de comunicación de dos vías entre dos programas que se ejecutan en la red.

Examples

Leer de socket

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try (
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out =
        new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream()));
    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in))
) {
    //Use the socket
}
```

Lea Enchufes en línea: https://riptutorial.com/es/java/topic/9918/enchufes

Capítulo 64: Enum a partir del número

Introducción

Java no permite que el nombre de enumeración comience con un número como 100A, 25K. En ese caso, podemos agregar el código con _ (guión bajo) o cualquier patrón permitido y verificarlo.

Examples

Enum con nombre al principio

```
public enum BookCode {
    _10A("Simon Haykin", "Communication System"),
    _42B("Stefan Hakins", "A Brief History of Time"),
    E1("Sedra Smith", "Electronics Circuits");

    private String author;
    private String title;

    BookCode(String author, String title) {
        this.author = author;
        this.title = title;
    }

    public String getName() {
        String name = name();
        if (name.charAt(0) == '_') {
            name = name.substring(1, name.length());
        }
        return name;
    }

    public static BookCode of(String code) {
        if (Character.isDigit(code.charAt(0))) {
            code = "_" + code;
        }
        return BookCode.valueOf(code);
    }
}
```

Lea Enum a partir del número en línea: https://riptutorial.com/es/java/topic/10719/enum-a-partir-del-numero

Capítulo 65: Enums

Introducción

Las enumeraciones de Java (declaradas usando la palabra clave `enum`) son sintaxis abreviada para cantidades considerables de constantes de una sola clase.

Sintaxis

- Enumeración [pública / protegida / privada] Enum_name {// Declarar una nueva enumeración.
- ENUM_CONSTANT_1 [, ENUM_CONSTANT_2 ...]; // Declara las constantes de enumeración. Esta debe ser la primera línea dentro de la enumeración y debe estar separada por comas, con un punto y coma al final.
- ENUM_CONSTANT_1 (param) [, ENUM_CONSTANT_2 (param) ...]; // Declarar constantes de enumeración con parámetros. Los tipos de parámetros deben coincidir con el constructor.
- ENUM_CONSTANT_1 {...} [, ENUM_CONSTANT_2 {...} ...]; // Declarar constantes de enumeración con métodos sobrescritos. Esto debe hacerse si la enumeración contiene métodos abstractos; Todos estos métodos deben ser implementados.
- ENUM_CONSTANT.name () // Devuelve una cadena con el nombre de la enumeración constante.
- ENUM_CONSTANT.ordinal () // Devuelve el ordinal de esta constante de enumeración, su posición en su declaración enum, donde a la constante inicial se le asigna un ordinal de cero.
- Enum_name.values () // Devuelve una nueva matriz (de tipo Enum_name []) que contiene cada constante de esa enumeración cada vez que se llama.
- Enum_name.valueOf ("ENUM_CONSTANT") // El inverso de ENUM_CONSTANT.name () - devuelve la constante de enumeración con el nombre dado.
- Enum.valueOf (Enum_name.class, "ENUM_CONSTANT") // Un sinónimo del anterior: El inverso de ENUM_CONSTANT.name () - devuelve la enumeración constante con el nombre dado.

Observaciones

Restricciones

Las enumeraciones siempre extienden `java.lang.Enum` , por lo que es imposible que una enumeración extienda una clase. Sin embargo, pueden implementar muchas interfaces.

consejos y trucos

Debido a su representación especializada, hay mapas y conjuntos más eficientes que se pueden

usar con las enumeraciones como sus claves. Estos a menudo se ejecutan más rápido que sus contrapartes no especializadas.

Examples

Declarar y usar una enumeración básica

Enum puede ser considerado como sintaxis de azúcar para una clase sellada que solo se crea una instancia de varias veces conocidas en tiempo de compilación para definir un conjunto de constantes.

Una enumeración simple para enumerar las diferentes temporadas se declararía de la siguiente manera:

```
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

Si bien las constantes de enumeración no necesariamente tienen que estar en mayúsculas, es una convención de Java que los nombres de las constantes están en mayúsculas, con palabras separadas por guiones bajos.

Puedes declarar un Enum en su propio archivo:

```
/**
 * This enum is declared in the Season.java file.
*/
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

Pero también puedes declararlo dentro de otra clase:

```
 public class Day {

    private Season season;

    public String getSeason() {
        return season.name();
    }

    public void setSeason(String season) {
        this.season = Season.valueOf(season);
    }

    /**
```

```
     * This enum is declared inside the Day.java file and
     * cannot be accessed outside because it's declared as private.
     */
    private enum Season {
        WINTER,
        SPRING,
        SUMMER,
        FALL
    }

}
```

Finalmente, no puede declarar un Enum dentro de un cuerpo de método o constructor:

```
public class Day {

    /**
     * Constructor
    */
    public Day() {
        // Illegal. Compilation error
        enum Season {
            WINTER,
            SPRING,
            SUMMER,
            FALL
        }
    }

    public void aSimpleMethod() {
        // Legal. You can declare a primitive (or an Object) inside a method. Compile!
        int primitiveInt = 42;

        // Illegal. Compilation error.
        enum Season {
            WINTER,
            SPRING,
            SUMMER,
            FALL
        }

        Season season = Season.SPRING;
    }

}
```

No se permiten constantes de enumeración duplicadas:

```
public enum Season {
    WINTER,
    WINTER, //Compile Time Error : Duplicate Constants
    SPRING,
    SUMMER,
    FALL
}
```

Cada constante de enumeración es **`public`** , **`static`** y **`final`** por defecto. Como todas las constantes son `static` , se puede acceder directamente usando el nombre de enumeración.

Las constantes de enumeración se pueden pasar como parámetros del método:

```
public static void display(Season s) {
    System.out.println(s.name());  // name() is a built-in method that gets the exact name of
the enum constant
}

display(Season.WINTER);  // Prints out "WINTER"
```

Puede obtener una matriz de las constantes de enumeración utilizando el método `values()` . Se garantiza que los valores están en orden de declaración en la matriz devuelta:

```
Season[] seasons = Season.values();
```

Nota: este método asigna una nueva matriz de valores cada vez que se llama.

Para iterar sobre las constantes de enumeración:

```
public static void enumIterate() {
    for (Season s : Season.values()) {
        System.out.println(s.name());
    }
}
```

Puede utilizar enumeraciones en una instrucción de `switch` :

```
public static void enumSwitchExample(Season s) {
    switch(s) {
        case WINTER:
            System.out.println("It's pretty cold");
            break;
        case SPRING:
            System.out.println("It's warming up");
            break;
        case SUMMER:
            System.out.println("It's pretty hot");
            break;
        case FALL:
            System.out.println("It's cooling down");
            break;
    }
}
```

También puedes comparar constantes de enumeración usando `==` :

```
Season.FALL == Season.WINTER    // false
Season.SPRING == Season.SPRING  // true
```

Otra forma de comparar las constantes de enumeración es utilizando `equals()` como se muestra a continuación, lo que se considera una mala práctica, ya que puede caer fácilmente en las dificultades de la siguiente manera:

```
Season.FALL.equals(Season.FALL); // true
Season.FALL.equals(Season.WINTER); // false
Season.FALL.equals("FALL"); // false and no compiler error
```

Además, aunque el conjunto de instancias en la `enum` no se puede cambiar en tiempo de ejecución, las instancias en sí mismas no son inherentemente inmutables porque, como cualquier otra clase, una `enum` puede contener campos mutables como se muestra a continuación.

```
public enum MutableExample {
    A,
    B;

    private int count = 0;

    public void increment() {
        count++;
    }

    public void print() {
        System.out.println("The count of " + name() + " is " + count);
    }
}

// Usage:
MutableExample.A.print();       // Outputs 0
MutableExample.A.increment();
MutableExample.A.print();       // Outputs 1 -- we've changed a field
MutableExample.B.print();       // Outputs 0 -- another instance remains unchanged
```

Sin embargo, una buena práctica es hacer que las instancias de `enum` inmutables, es decir, cuando no tienen campos adicionales o todos estos campos están marcados como `final` y son inmutables. Esto asegurará que durante toda la vida de la aplicación, una `enum` no perderá memoria y que es seguro usar sus instancias en todos los subprocesos.

Enums implícitamente implementa `Serializable` y `Comparable` porque la clase `Enum` hace:

```
public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable
```

Enums con constructores

Una `enum` no puede tener un constructor público; sin embargo, los constructores privados son aceptables (los constructores para enums son de paquete privado por defecto):

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25); // usual names for US coins
    // note that the above parentheses and the constructor arguments match
```

```
    private int value;

    Coin(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

int p = Coin.NICKEL.getValue(); // the int value will be 5
```

Se recomienda mantener todos los campos privados y proporcionar métodos de obtención, ya que hay un número finito de instancias para una enumeración.

Si tuvieras que implementar un `Enum` como `class` , se vería así:

```
public class Coin<T extends Coin<T>> implements Comparable<T>, Serializable{
    public static final Coin PENNY = new Coin(1);
    public static final Coin NICKEL = new Coin(5);
    public static final Coin DIME = new Coin(10);
    public static final Coin QUARTER = new Coin(25);

    private int value;

    private Coin(int value){
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

int p = Coin.NICKEL.getValue(); // the int value will be 5
```

Las constantes de enumeración son técnicamente mutables, por lo que se podría agregar un setter para cambiar la estructura interna de una constante de enumeración. Sin embargo, esto se considera muy mala práctica y debe evitarse.

La mejor práctica es hacer que los campos de Enum sean inmutables, con `final` :

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value){
        this.value = value;
    }

    ...

}
```

Puede definir múltiples constructores en la misma enumeración. Cuando lo hace, los argumentos que pasa en su declaración de enumeración deciden a qué constructor se llama:

```
public enum Coin {
    PENNY(1, true), NICKEL(5, false), DIME(10), QUARTER(25);

    private final int value;
    private final boolean isCopperColored;

    Coin(int value){
        this(value, false);
    }

    Coin(int value, boolean isCopperColored){
        this.value = value;
        this.isCopperColored = isCopperColored;
    }

    ...

}
```

Nota: Todos los campos de enumeración no primitivos deberían implementar `Serializable` porque la clase `Enum` sí lo hace.

Utilizando métodos y bloques estáticos.

Una enumeración puede contener un método, al igual que cualquier clase. Para ver cómo funciona esto, declararemos una enumeración como esta:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;
}
```

Tengamos un método que devuelva la enumeración en la dirección opuesta:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;

    public Direction getOpposite(){
        switch (this){
            case NORTH:
                return SOUTH;
            case SOUTH:
                return NORTH;
            case WEST:
                return EAST;
            case EAST:
                return WEST;
            default: //This will never happen
                return null;
        }
    }
}
```

Esto se puede mejorar aún más mediante el uso de campos y bloques de inicialización estáticos:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;

    private Direction opposite;

    public Direction getOpposite(){
        return opposite;
    }

    static {
        NORTH.opposite = SOUTH;
        SOUTH.opposite = NORTH;
        WEST.opposite = EAST;
        EAST.opposite = WEST;
    }
}
```

En este ejemplo, la dirección opuesta se almacena en un campo de instancia privada `opposite` , que se inicializa de forma estática la primera vez que se utiliza una `Direction` . En este caso particular (debido a que el `NORTH` referencia al `SOUTH` y viceversa), no podemos usar Enums con constructores aquí (los Constructores `NORTH(SOUTH), SOUTH(NORTH), EAST(WEST), WEST(EAST)` serían más elegantes y permitirían `opposite` a ser declarado `final` , pero sería auto-referencial y por lo tanto no está permitido).

Implementa interfaz

Esta es una `enum` que también es una función invocable que prueba las entradas de `String` contra patrones de expresión regular precompilados.

```
import java.util.function.Predicate;
import java.util.regex.Pattern;

enum RegEx implements Predicate<String> {
    UPPER("[A-Z]+"), LOWER("[a-z]+"), NUMERIC("[+-]?[0-9]+");

    private final Pattern pattern;

    private RegEx(final String pattern) {
        this.pattern = Pattern.compile(pattern);
    }

    @Override
    public boolean test(final String input) {
        return this.pattern.matcher(input).matches();
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(RegEx.UPPER.test("ABC"));
        System.out.println(RegEx.LOWER.test("abc"));
        System.out.println(RegEx.NUMERIC.test("+111"));
    }
}
```

Cada miembro de la enumeración también puede implementar el método:

```
import java.util.function.Predicate;

enum Acceptor implements Predicate<String> {
    NULL {
        @Override
        public boolean test(String s) { return s == null; }
    },
    EMPTY {
        @Override
        public boolean test(String s) { return s.equals(""); }
    },
    NULL_OR_EMPTY {
        @Override
        public boolean test(String s) { return NULL.test(s) || EMPTY.test(s); }
    };
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Acceptor.NULL.test(null));  // true
        System.out.println(Acceptor.EMPTY.test(""));   // true
        System.out.println(Acceptor.NULL_OR_EMPTY.test(" ")); // false
    }
}
```

Patrón de polimorfismo enum

Cuando un método necesita aceptar un conjunto "extensible" de valores de `enum` , el programador puede aplicar polimorfismo como en una `class` normal creando una interfaz que se usará en cualquier lugar donde se usen las `enum` :

```
public interface ExtensibleEnum {
    String name();
}
```

De esta manera, cualquier `enum` etiquetada por (implementando) la interfaz se puede usar como un parámetro, lo que permite al programador crear una cantidad variable de `enum` que será aceptada por el método. Esto puede ser útil, por ejemplo, en las API donde hay una `enum` predeterminada (no modificable) y el usuario de estas API desea "extender" la `enum` con más valores.

Un conjunto de valores de enumeración predeterminados se puede definir de la siguiente manera:

```
public enum DefaultValues implements ExtensibleEnum {
    VALUE_ONE, VALUE_TWO;
}
```

Los valores adicionales se pueden definir así:

```
public enum ExtendedValues implements ExtensibleEnum {
    VALUE_THREE, VALUE_FOUR;
```

```
}
```

Ejemplo que muestra cómo usar las enumeraciones: observe cómo `printEnum()` acepta valores de ambos tipos de `enum` :

```
private void printEnum(ExtensibleEnum val) {
    System.out.println(val.name());
}

printEnum(DefaultValues.VALUE_ONE);    // VALUE_ONE
printEnum(DefaultValues.VALUE_TWO);    // VALUE_TWO
printEnum(ExtendedValues.VALUE_THREE); // VALUE_THREE
printEnum(ExtendedValues.VALUE_FOUR);  // VALUE_FOUR
```

Nota: este patrón no le impide redefinir los valores de enumeración, que ya están definidos en una enumeración, en otra enumeración. Estos valores de enumeración serían diferentes instancias entonces. Además, no es posible utilizar switch-on-enum ya que todo lo que tenemos es la interfaz, no la `enum` real.

Enums con métodos abstractos

Las enumeraciones pueden definir métodos abstractos, que cada miembro de la `enum` debe implementar.

```
enum Action {
    DODGE {
        public boolean execute(Player player) {
            return player.isAttacking();
        }
    },
    ATTACK {
        public boolean execute(Player player) {
            return player.hasWeapon();
        }
    },
    JUMP {
        public boolean execute(Player player) {
            return player.getCoordinates().equals(new Coordinates(0, 0));
        }
    };

    public abstract boolean execute(Player player);
}
```

Esto permite que cada miembro de la enumeración defina su propio comportamiento para una operación dada, sin tener que activar tipos en un método en la definición de nivel superior.

Tenga en cuenta que este patrón es una forma corta de lo que normalmente se logra utilizando polimorfismo y / o implementación de interfaces.

Documentando enumeraciones

No siempre el nombre de la `enum` es lo suficientemente claro como para ser entendido. Para

documentar una `enum` , use javadoc estándar:

```
/**
 * United States coins
 */
public enum Coins {

    /**
     * One-cent coin, commonly known as a penny,
     * is a unit of currency equaling one-hundredth
     * of a United States dollar
     */
    PENNY(1),

    /**
     * A nickel is a five-cent coin equaling
     * five-hundredth of a United States dollar
     */
    NICKEL(5),

    /**
     * The dime is a ten-cent coin refers to
     * one tenth of a United States dollar
     */
    DIME(10),

    /**
     * The quarter is a US coin worth 25 cents,
     * one-fourth of a United States dollar
     */
    QUARTER(25);

    private int value;

    Coins(int value){
        this.value = value;
    }

    public int getValue(){
        return value;
    }
}
```

Obtener los valores de una enumeración

Cada clase de enumeración contiene un método estático implícito llamado `values()` . Este método devuelve una matriz que contiene todos los valores de esa enumeración. Puede utilizar este método para iterar sobre los valores. Sin embargo, es importante tener en cuenta que este método devuelve una **nueva** matriz cada vez que se llama.

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

    /**
    * Print out all the values in this enum.
    */
    public static void printAllDays() {
```

```
        for(Day day : Day.values()) {
            System.out.println(day.name());
        }
    }
}
```

Si necesita un `Set` también puede usar `EnumSet.allOf(Day.class)` .

Enum como un parámetro de tipo limitado

Al escribir una clase con genéricos en java, es posible asegurarse de que el parámetro type sea una enumeración. Como todas las enumeraciones amplían la clase `Enum` , se puede usar la siguiente sintaxis.

```
public class Holder<T extends Enum<T>> {
    public final T value;

    public Holder(T init) {
        this.value = init;
    }
}
```

En este ejemplo, el tipo `T` *debe* ser una enumeración.

Obtener enumeración constante por nombre

Digamos que tenemos una enumeración `DayOfWeek` :

```
enum DayOfWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}
```

Una enumeración se compila con un método static `valueOf()` incorporado que se puede usar para buscar una constante por su nombre:

```
String dayName = DayOfWeek.SUNDAY.name();
assert dayName.equals("SUNDAY");

DayOfWeek day = DayOfWeek.valueOf(dayName);
assert day == DayOfWeek.SUNDAY;
```

Esto también es posible utilizando un tipo de enumeración dinámica:

```
Class<DayOfWeek> enumType = DayOfWeek.class;
DayOfWeek day = Enum.valueOf(enumType, "SUNDAY");
assert day == DayOfWeek.SUNDAY;
```

Ambos de estos métodos `valueOf()` lanzarán una `IllegalArgumentException` si la enumeración especificada no tiene una constante con un nombre coincidente.

La biblioteca de Guava proporciona un método auxiliar `Enums.getIfPresent()` que devuelve un

Guava `Optional` para eliminar el manejo de excepciones explícitas:

```
DayOfWeek defaultDay = DayOfWeek.SUNDAY;
DayOfWeek day = Enums.valueOf(DayOfWeek.class, "INVALID").or(defaultDay);
assert day == DayOfWeek.SUNDAY;
```

Implementar el patrón Singleton con una enumeración de un solo elemento.

Las constantes de enumeración se crean instancias cuando se hace referencia a una enumeración por primera vez. Por lo tanto, eso permite implementar el patrón de diseño del software Singleton con una enumeración de un solo elemento.

```
public enum Attendant {

    INSTANCE;

    private Attendant() {
        // perform some initialization routine
    }

    public void sayHello() {
        System.out.println("Hello!");
    }
}


public class Main {

    public static void main(String... args) {
        Attendant.INSTANCE.sayHello();// instantiated at this point
    }
}
```

Según el libro "Effective Java" de Joshua Bloch, una enumeración de un solo elemento es la mejor manera de implementar un singleton. Este enfoque tiene las siguientes ventajas:

- seguridad del hilo
- garantía de instanciación única
- serialización fuera de la caja

Y como se muestra en la sección implementa la interfaz, este singleton también puede implementar una o más interfaces.

Enumerar con propiedades (campos)

En caso de que queramos usar `enum` con más información y no solo valores constantes, queremos poder comparar dos enumeraciones.

Considere el siguiente ejemplo:

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
```

```
    private final int value;

    Coin(int value){
        this.value = value;
    }

    public boolean isGreaterThan(Coin other){
        return this.value > other.value;
    }

}
```

Aquí definimos un `Enum` llamado `Coin` que representa su valor. Con el método `isGreaterThan` podemos comparar dos `enum` s:

```
Coin penny = Coin.PENNY;
Coin dime = Coin.DIME;

System.out.println(penny.isGreaterThan(dime)); // prints: false
System.out.println(dime.isGreaterThan(penny)); // prints: true
```

Convertir enum a cadena

A veces quieres convertir tu enumeración en una Cadena, hay dos formas de hacerlo.

Supongamos que tenemos:

```
public enum Fruit {
    APPLE, ORANGE, STRAWBERRY, BANANA, LEMON, GRAPE_FRUIT;
}
```

Entonces, ¿cómo convertimos algo como `Fruit.APPLE` a `"APPLE"` ?

Convertir usando `name()`

`name()` es un método interno en `enum` que devuelve la representación de `String` de la enumeración, la `String` retorno representa ***exactamente*** cómo se definió el valor de enumeración.

Por ejemplo:

```
System.out.println(Fruit.BANANA.name());       // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.name()); // "GRAPE_FRUIT"
```

Convertir utilizando `toString()`

`toString()` está, *por defecto* , reemplazado para tener el mismo comportamiento que `name()`

Sin embargo, es probable que los *desarrolladores* `toString()` para que imprima una `String` más fácil de usar

> No uses `toString()` si quieres verificar tu código, `name()` es mucho más estable para eso. Solo use `toString()` cuando vaya a dar salida al valor de logs o stdout o algo así

Por defecto:

```
System.out.println(Fruit.BANANA.toString());      // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "GRAPE_FRUIT"
```

Ejemplo de ser anulado

```
System.out.println(Fruit.BANANA.toString());      // "Banana"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "Grape Fruit"
```

Enumeración específica del cuerpo

En una `enum` es posible definir un comportamiento específico para una constante particular de la `enum` que anula el comportamiento predeterminado de la `enum` , esta técnica se conoce como *cuerpo específico constante* .

Supongamos que tres estudiantes de piano, John, Ben y Luke, se definen en una `enum` llamada `PianoClass` , de la siguiente manera:

```
    enum PianoClass {
    JOHN, BEN, LUKE;
        public String getSex() {
            return "Male";
        }
        public String getLevel() {
            return "Beginner";
        }
    }
```

Y un día llegan otros dos estudiantes, Rita y Tom, con un sexo (femenino) y un nivel (intermedio) que no coinciden con los anteriores:

```
    enum PianoClass2 {
    JOHN, BEN, LUKE, RITA, TOM;
        public String getSex() {
            return "Male"; // issue, Rita is a female
        }
        public String getLevel() {
            return "Beginner"; // issue, Tom is an intermediate student
        }
    }
```

de modo que simplemente agregar los nuevos alumnos a la declaración constante, como sigue, no es correcto:

```
PianoClass2 tom = PianoClass2.TOM;
PianoClass2 rita = PianoClass2.RITA;
System.out.println(tom.getLevel()); // prints Beginner -> wrong Tom's not a beginner
System.out.println(rita.getSex()); // prints Male -> wrong Rita's not a male
```

Es posible definir un comportamiento específico para cada una de las constantes, Rita y Tom, que anula el comportamiento predeterminado de `PianoClass2` la siguiente manera:

```
enum PianoClass3 {
    JOHN, BEN, LUKE,
    RITA {
        @Override
        public String getSex() {
            return "Female";
        }
    },
    TOM {
        @Override
        public String getLevel() {
            return "Intermediate";
        }
    };
    public String getSex() {
        return "Male";
    }
    public String getLevel() {
        return "Beginner";
    }
}
```

y ahora el nivel de Tom y el sexo de Rita son como deberían ser:

```
PianoClass3 tom = PianoClass3.TOM;
PianoClass3 rita = PianoClass3.RITA;
System.out.println(tom.getLevel()); // prints Intermediate
System.out.println(rita.getSex()); // prints Female
```

Otra forma de definir el cuerpo específico del contenido es mediante el uso de constructor, por ejemplo:

```
enum Friend {
    MAT("Male"),
    JOHN("Male"),
    JANE("Female");

    private String gender;

    Friend(String gender) {
        this.gender = gender;
    }

    public String getGender() {
        return this.gender;
```

```
    }
}
```

y uso:

```
Friend mat = Friend.MAT;
Friend john = Friend.JOHN;
Friend jane = Friend.JANE;
System.out.println(mat.getGender());     // Male
System.out.println(john.getGender());    // Male
System.out.println(jane.getGender());    // Female
```

Enumeración cero instancia

```
enum Util {
    /* No instances */;

    public static int clamp(int min, int max, int i) {
        return Math.min(Math.max(i, min), max);
    }

    // other utility methods...
}
```

Del mismo modo que `enum` se puede utilizar para singletons (clases de 1 instancia), se puede usar para clases de utilidad (0 clases de instancia). Solo asegúrese de terminar la lista (vacía) de constantes de enumeración con un `;` .

Consulte la pregunta Zero instance enum contra constructores privados para evitar la creación de instancias para una discusión sobre pros y contras en comparación con constructores privados.

Enums con campos estáticos

Si se requiere que su clase de enumeración tenga campos estáticos, tenga en cuenta que se crean **después de** los propios valores de enumeración. Eso significa que, el siguiente código resultará en una `NullPointerException` :

```
enum Example {
    ONE(1), TWO(2);

    static Map<String, Integer> integers = new HashMap<>();

    private Example(int value) {
        integers.put(this.name(), value);
    }
}
```

Una posible forma de solucionar esto:

```
enum Example {
    ONE(1), TWO(2);
```

```
    static Map<String, Integer> integers;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }
}
```

No inicialice el campo estático:

```
enum Example {
    ONE(1), TWO(2);

    // after initialisisation integers is null!!
    static Map<String, Integer> integers = null;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }
    // !!this may lead to null poiner exception!!
    public int getValue(){
        return (Example.integers.get(this.name()));
    }
}
```

inicialisis

- crear los valores de enumeración
  - como efecto secundario putValue () llamado que inicializa enteros
- se establecen los valores estáticos
  - enteros = nulo; // se ejecuta después de las enumeraciones por lo que el contenido de los enteros se pierde

Comparar y Contiene para los valores Enum

Enums contiene solo constantes y puede compararse directamente con `==` . Por lo tanto, solo se necesita una verificación de referencia, no es necesario utilizar el método `.equals` . Además, si `.equals` usa incorrectamente, puede generar la `NullPointerException` mientras que ese no es el caso con `==` check.

```
enum Day {
    GOOD, AVERAGE, WORST;
}
```

```
public class Test {

    public static void main(String[] args) {
        Day day = null;

        if (day.equals(Day.GOOD)) {//NullPointerException!
            System.out.println("Good Day!");
        }

        if (day == Day.GOOD) {//Always use == to compare enum
            System.out.println("Good Day!");
        }

    }
}
```

Para agrupar, complementar, `EnumSet` valores de enumeración, tenemos la clase `EnumSet` que contiene diferentes métodos.

- `EnumSet#range` : para obtener un subconjunto de enumeración por rango definido por dos puntos finales
- `EnumSet#of` : Conjunto de enumeraciones específicas sin ningún rango. Múltiples sobrecargas `of` métodos están ahí.
- `EnumSet#complementOf` : Conjunto de enumeración que es el complemento de los valores de enumeración proporcionados en el parámetro del método

```
enum Page {
   A1, A2, A3, A4, A5, A6, A7, A8, A9, A10
}

public class Test {

  public static void main(String[] args) {
      EnumSet<Page> range = EnumSet.range(Page.A1, Page.A5);

      if (range.contains(Page.A4)) {
          System.out.println("Range contains A4");
      }

      EnumSet<Page> of = EnumSet.of(Page.A1, Page.A5, Page.A3);

      if (of.contains(Page.A1)) {
          System.out.println("Of contains A1");
      }
  }
}
```

Lea Enums en línea: https://riptutorial.com/es/java/topic/155/enums

Capítulo 66: Errores comunes de Java

Introducción

Este tema describe algunos de los errores comunes que cometen los principiantes en Java.

Esto incluye cualquier error común en el uso del lenguaje Java o la comprensión del entorno de tiempo de ejecución.

Los errores asociados con API específicas se pueden describir en temas específicos de esas API. Las cuerdas son un caso especial; están cubiertos en la especificación del lenguaje Java. Los detalles que no sean errores comunes se pueden describir en este tema en Cadenas .

Examples

Pitfall: utilizando == para comparar objetos de envoltorios primitivos, como Integer

(Este escollo se aplica por igual a todos los tipos de envoltorios primitivos, pero lo ilustraremos para `Integer` e `int` .)

Cuando se trabaja con objetos `Integer` , es tentador usar `==` para comparar valores, porque eso es lo que haría con los valores `int` . Y en algunos casos esto parecerá funcionar:

```
Integer int1_1 = Integer.valueOf("1");
Integer int1_2 = Integer.valueOf(1);

System.out.println("int1_1 == int1_2: " + (int1_1 == int1_2));          // true
System.out.println("int1_1 equals int1_2: " + int1_1.equals(int1_2));   // true
```

Aquí creamos dos objetos `Integer` con el valor `1` y los comparamos (en este caso creamos uno de una `String` y uno de un literal `int` . Hay otras alternativas). Además, observamos que los dos métodos de comparación (`==` y `equals`) son ambos `true` .

Este comportamiento cambia cuando elegimos diferentes valores:

```
Integer int2_1 = Integer.valueOf("1000");
Integer int2_2 = Integer.valueOf(1000);

System.out.println("int2_1 == int2_2: " + (int2_1 == int2_2));          // false
System.out.println("int2_1 equals int2_2: " + int2_1.equals(int2_2));   // true
```

En este caso, solo la comparación de `equals` produce el resultado correcto.

La razón de esta diferencia en el comportamiento es que la JVM mantiene un caché de objetos `Integer` para el rango de -128 a 127. (El valor superior se puede anular con la propiedad del sistema "java.lang.Integer.IntegerCache.high" o la Argumento de JVM "-XX: AutoBoxCacheMax =

tamaño"). Para los valores en este rango, `Integer.valueOf()` devolverá el valor almacenado en caché en lugar de crear uno nuevo.

Por lo tanto, en el primer ejemplo, las llamadas `Integer.valueOf(1)` y `Integer.valueOf("1")` devolvieron la misma instancia de `Integer` caché. Por el contrario, en el segundo ejemplo, `Integer.valueOf(1000)` y `Integer.valueOf("1000")` crearon y devolvieron nuevos objetos `Integer` .

El operador `==` para tipos de referencia prueba la igualdad de referencia (es decir, el mismo objeto). Por lo tanto, en el primer ejemplo `int1_1 == int1_2` es `true` porque las referencias son las mismas. En el segundo ejemplo `int2_1 == int2_2` es falso porque las referencias son diferentes.

Pitfall: olvidarse de liberar recursos

Cada vez que un programa abre un recurso, como un archivo o una conexión de red, es importante liberar el recurso una vez que haya terminado de usarlo. Se debe tener la misma precaución si se lanzara alguna excepción durante las operaciones con dichos recursos. Se podría argumentar que `FileInputStream` tiene un finalizador que invoca el método `close()` en un evento de recolección de basura; sin embargo, dado que no podemos estar seguros de cuándo se iniciará un ciclo de recolección de basura, la secuencia de entrada puede consumir recursos de computadora por un período de tiempo indefinido. El recurso se debe cerrar en una sección de `finally` de un bloque try-catch:

Java SE 7

```
private static void printFileJava6() throws IOException {
    FileInputStream input;
    try {
        input = new FileInputStream("file.txt");
        int data = input.read();
        while (data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    } finally {
        if (input != null) {
            input.close();
        }
    }
}
```

Desde Java 7 hay una declaración realmente útil y ordenada introducida en Java 7 particularmente para este caso, llamada try-with-resources:

Java SE 7

```
private static void printFileJava7() throws IOException {
    try (FileInputStream input = new FileInputStream("file.txt")) {
        int data = input.read();
        while (data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
```

```
}
```

La sentencia *try-con-recursos* se puede utilizar con cualquier objeto que implemente la `Closeable` o `AutoCloseable` interfaz. Asegura que cada recurso se cierre al final de la declaración. La diferencia entre las dos interfaces es que el método `close()` de `Closeable` lanza una `IOException` que debe manejarse de alguna manera.

En los casos en los que el recurso ya se ha abierto pero debe cerrarse de manera segura después de su uso, se puede asignar a una variable local dentro de try-with-resources

Java SE 7

```
private static void printFileJava7(InputStream extResource) throws IOException {
    try (InputStream input = extResource) {
        ... //access resource
    }
}
```

La variable de recurso local creada en el constructor try-with-resources es efectivamente final.

Trampa: fugas de memoria

Java gestiona la memoria automáticamente. No es necesario liberar la memoria manualmente. La memoria de un objeto en el montón puede ser liberada por un recolector de basura cuando el objeto ya no es *accesible* por un hilo vivo.

Sin embargo, puede evitar que se libere la memoria, permitiendo que se pueda acceder a objetos que ya no son necesarios. Ya sea que llame a esto una pérdida de memoria o una tasa de paquetes de memoria, el resultado es el mismo: un aumento innecesario en la memoria asignada.

Las fugas de memoria en Java pueden ocurrir de varias maneras, pero la razón más común son las referencias eternas de objetos, porque el recolector de basura no puede eliminar objetos del montón mientras todavía hay referencias a ellos.

Campos estáticos

Uno puede crear una referencia de este tipo definiendo la clase con un campo `static` contiene alguna colección de objetos, y olvidando establecer ese campo `static` en `null` después de que la colección ya no sea necesaria. `static` campos `static` se consideran raíces GC y nunca se recopilan. Otro problema son las fugas en la memoria no de pila cuando se utiliza JNI .

Fuga del cargador de clases

De lejos, sin embargo, el tipo más insidioso de pérdida de memoria es la pérdida del cargador de clases . Un cargador de clases contiene una referencia a cada clase que ha cargado, y cada clase tiene una referencia a su cargador de clases. Cada objeto tiene una referencia a su clase también. Por lo tanto, si incluso un *solo* objeto de una clase cargada por un cargador de clases no es basura, no se puede recopilar una sola clase que ese cargador de clases haya cargado. Como cada clase también hace referencia a sus campos estáticos, tampoco se pueden recopilar.

Fuga de acumulación El ejemplo de fuga de acumulación podría ser similar al siguiente:

```
final ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);
final Deque<BigDecimal> numbers = new LinkedBlockingDeque<>();
final BigDecimal divisor = new BigDecimal(51);

scheduledExecutorService.scheduleAtFixedRate(() -> {
    BigDecimal number = numbers.peekLast();
    if (number != null && number.remainder(divisor).byteValue() == 0) {
        System.out.println("Number: " + number);
        System.out.println("Deque size: " + numbers.size());
    }
}, 10, 10, TimeUnit.MILLISECONDS);

scheduledExecutorService.scheduleAtFixedRate(() -> {
    numbers.add(new BigDecimal(System.currentTimeMillis()));
}, 10, 10, TimeUnit.MILLISECONDS);

try {
    scheduledExecutorService.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Este ejemplo crea dos tareas programadas. La primera tarea toma el último número de un deque llamado `numbers` , y, si el número es divisible por 51, imprime el número y el tamaño del deque. La segunda tarea pone números en el deque. Ambas tareas se programan a una velocidad fija y se ejecutan cada 10 ms.

Si se ejecuta el código, verá que el tamaño del deque está aumentando permanentemente. Esto eventualmente hará que el deque se llene con objetos que consumen toda la memoria disponible del montón.

Para evitar esto y preservar la semántica de este programa, podemos usar un método diferente para tomar números del deque: `pollLast` . Al contrario del método `peekLast` , `pollLast` devuelve el elemento y lo elimina del deque, mientras que `peekLast` solo devuelve el último elemento.

Pitfall: usando == para comparar cadenas

Un error común para los principiantes de Java es usar el operador `==` para probar si dos cadenas son iguales. Por ejemplo:

```
public class Hello {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0] == "hello") {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}
```

Se supone que el programa anterior prueba el primer argumento de la línea de comando e imprime diferentes mensajes cuando no es la palabra "hola". Pero el problema es que no funcionará. Ese programa producirá "¿Te sientes malhumorado hoy?" no importa cuál sea el primer argumento de la línea de comando.

En este caso particular, la `String` "hola" se coloca en el grupo de cadenas mientras que la `String` args [0] reside en el montón. Esto significa que hay dos objetos que representan el mismo literal, cada uno con su referencia. Dado que `==` prueba las referencias, no la igualdad real, la comparación producirá un falso la mayoría de las veces. Esto no significa que siempre lo hará.

Cuando utiliza `==` para probar cadenas, lo que realmente está probando es si dos objetos de `String` son el mismo objeto de Java. Desafortunadamente, eso no es lo que significa la igualdad de cadenas en Java. De hecho, la forma correcta de probar cadenas es usar el método `equals(Object)` . Para un par de cadenas, generalmente queremos probar si están formadas por los mismos caracteres en el mismo orden.

```
public class Hello2 {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0].equals("hello")) {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}
```

Pero en realidad se pone peor. El problema es que `==` *dará* la respuesta esperada en algunas circunstancias. Por ejemplo

```
public class Test1 {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";
        if (s1 == s2) {
            System.out.println("same");
        } else {
            System.out.println("different");
        }
    }
}
```

Curiosamente, esto imprimirá "igual", aunque estamos probando las cadenas de manera incorrecta. ¿Porqué es eso? Debido a que la Especificación del lenguaje Java (Sección 3.10.5: Literales de cadenas) estipula que dos cadenas >> literales << consistentes en los mismos caracteres serán representadas por el mismo objeto Java. Por lo tanto, la prueba `==` dará verdadero para literales iguales. (Los literales de cadena se "internan" y se agregan a un "grupo de cadenas" compartido cuando se carga su código, pero eso es en realidad un detalle de implementación).

Para agregar a la confusión, la especificación del lenguaje Java también estipula que cuando se

tiene una expresión constante en tiempo de compilación que concatena dos literales de cadena, es equivalente a un solo literal. Así:

```
    public class Test1 {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hel" + "lo";
        String s3 = " mum";
        if (s1 == s2) {
            System.out.println("1. same");
        } else {
            System.out.println("1. different");
        }
        if (s1 + s3 == "hello mum") {
            System.out.println("2. same");
        } else {
            System.out.println("2. different");
        }
    }
}
```

Esto dará salida a "1. igual" y "2. diferente". En el primer caso, la expresión `+` se evalúa en tiempo de compilación y comparamos un objeto `String` consigo mismo. En el segundo caso, se evalúa en tiempo de ejecución y comparamos dos objetos `String` diferentes

En resumen, usar `==` para probar cadenas en Java es casi siempre incorrecto, pero no se garantiza que dé la respuesta incorrecta.

Pitfall: probar un archivo antes de intentar abrirlo.

Algunas personas recomiendan que aplique varias pruebas a un archivo antes de intentar abrirlo para proporcionar un mejor diagnóstico o evitar tratar con excepciones. Por ejemplo, este método intenta verificar si la `path` corresponde a un archivo legible:

```
public static File getValidatedFile(String path) throws IOException {
    File f = new File(path);
    if (!f.exists()) throw new IOException("Error: not found: " + path);
    if (!f.isFile()) throw new IOException("Error: Is a directory: " + path);
    if (!f.canRead()) throw new IOException("Error: cannot read file: " + path);
    return f;
}
```

Puedes usar el método anterior como este:

```
File f = null;
try {
    f = getValidatedFile("somefile");
} catch (IOException ex) {
    System.err.println(ex.getMessage());
    return;
}
try (InputStream is = new FileInputStream(file)) {
    // Read data etc.
}
```

El primer problema está en la firma para `FileInputStream(File)` porque el compilador seguirá insistiendo en que `IOException` aquí, o más arriba en la pila.

El segundo problema es que las comprobaciones realizadas por `getValidatedFile` no garantizan que `FileInputStream` tendrá éxito.

- Condiciones de la carrera: otro hilo o un proceso separado podría cambiar el nombre del archivo, eliminar el archivo o eliminar el acceso de lectura después de que `getValidatedFile` el `getValidatedFile` . Eso llevaría a una `IOException` "simple" sin el mensaje personalizado.
- Hay casos de borde no cubiertos por esas pruebas. Por ejemplo, en un sistema con SELinux en modo "de cumplimiento", un intento de leer un archivo puede fallar a pesar de que `canRead()` devuelva `true` .

El tercer problema es que las pruebas son ineficientes. Por ejemplo, las llamadas `exists` , `isFile` y `canRead` harán cada una syscall para realizar la verificación requerida. Luego se hace otro syscall para abrir el archivo, que repite las mismas comprobaciones detrás de escena.

En resumen, los métodos como `getValidatedFile` son erróneos. Es mejor simplemente intentar abrir el archivo y manejar la excepción:

```
try (InputStream is = new FileInputStream("somefile")) {
    // Read data etc.
} catch (IOException ex) {
    System.err.println("IO Error processing 'somefile': " + ex.getMessage());
    return;
}
```

Si desea distinguir los errores de E / S que se producen al abrir y leer, puede usar un try / catch anidado. Si desea producir mejores diagnósticos para fallas abiertas, puede realizar las `isFile` `exists` , `isFile` y `canRead` en el controlador.

Pitfall: pensar las variables como objetos

Ninguna variable Java representa un objeto.

```
String foo;   // NOT AN OBJECT
```

Ninguna matriz de Java contiene objetos.

```
String bar[] = new String[100];  // No member is an object.
```

Si piensa erróneamente que las variables son objetos, el comportamiento real del lenguaje Java lo sorprenderá.

- Para las variables de Java que tienen un tipo primitivo (como `int` o `float`), la variable contiene una copia del valor. Todas las copias de un valor primitivo son indistinguibles; es decir, solo hay un valor `int` para el número uno. Los valores primitivos no son objetos y no se comportan como objetos.

- Para las variables de Java que tienen un tipo de referencia (ya sea una clase o un tipo de matriz), la variable contiene una referencia. Todas las copias de una referencia son indistinguibles. Las referencias pueden apuntar a objetos, o pueden ser `null` que significa que no apuntan a ningún objeto. Sin embargo, no son objetos y no se comportan como objetos.

Las variables no son objetos en ningún caso, y no contienen objetos en ninguno de los casos. Pueden contener *referencias a objetos* , pero eso es decir algo diferente.

Clase de ejemplo

Los ejemplos que siguen utilizan esta clase, que representa un punto en el espacio 2D.

```
public final class MutableLocation {
   public int x;
   public int y;

   public MutableLocation(int x, int y) {
       this.x = x;
       this.y = y;
   }

   public boolean equals(Object other) {
       if (!(other instanceof MutableLocation) {
           return false;
       }
       MutableLocation that = (MutableLocation) other;
       return this.x == that.x && this.y == that.y;
   }
}
```

Una instancia de esta clase es un objeto que tiene dos campos `x` y `y` que tienen el tipo `int` .

Podemos tener muchas instancias de la clase `MutableLocation` . Algunos representarán las mismas ubicaciones en el espacio 2D; Es decir, los valores respectivos de `x` y `y` coincidirán. Otros representarán diferentes lugares.

Múltiples variables pueden apuntar al mismo objeto.

```
MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);
```

En lo anterior, hemos declarado tres variables `here` , `there` y en `elsewhere` que pueden contener referencias a objetos `MutableLocation` .

Si (incorrectamente) piensa que estas variables son objetos, entonces es probable que malinterprete las afirmaciones diciendo:

1. Copie la ubicación "[1, 2]" `here`
2. Copie la ubicación "[1, 2]" `there`

3. Copie la ubicación "[1, 2]" a `elsewhere`

A partir de eso, es probable que deduzca que tenemos tres objetos independientes en las tres variables. De hecho *solo* hay *dos objetos creados* por el anterior. Las variables `here` y `there` realidad se refieren al mismo objeto.

Podemos demostrar esto. Suponiendo las declaraciones de variables como anteriormente:

```
System.out.println("BEFORE: here.x is " + here.x + ", there.x is " + there.x +
                   "elsewhere.x is " + elsewhere.x);
here.x = 42;
System.out.println("AFTER: here.x is " + here.x + ", there.x is " + there.x +
                   "elsewhere.x is " + elsewhere.x);
```

Esto dará salida a lo siguiente:

```
BEFORE: here.x is 1, there.x is 1, elsewhere.x is 1
AFTER: here.x is 42, there.x is 42, elsewhere.x is 1
```

Le asignamos un nuevo valor a `here.x` y cambió el valor que vemos a través de `there.x` . Se están refiriendo al mismo objeto. Pero el valor que vemos a través de `elsewhere.x` no ha cambiado, por lo que en `elsewhere` debe referirse a un objeto diferente.

Si una variable era un objeto, entonces la asignación `here.x = 42` no cambiaría `there.x` .

El operador de igualdad NO prueba que dos objetos son iguales

La aplicación del operador de igualdad (`==`) a los valores de referencia comprueba si los valores se refieren al mismo objeto. *No* prueba si dos (diferentes) objetos son "iguales" en el sentido intuitivo.

```
MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);

if (here == there) {
    System.out.println("here is there");
}
if (here == elsewhere) {
    System.out.println("here is elsewhere");
}
```

Esto imprimirá "aquí está ahí", pero no imprimirá "aquí está en otra parte". (Las referencias `here` y en `elsewhere` son para dos objetos distintos).

Por el contrario, si llamamos al método `equals(Object)` que implementamos anteriormente, vamos a probar si dos instancias de `MutableLocation` tienen una ubicación igual.

```
if (here.equals(there)) {
```

```
    System.out.println("here equals there");
 }
 if (here.equals(elsewhere)) {
    System.out.println("here equals elsewhere");
 }
```

Esto imprimirá ambos mensajes. En particular, `here.equals(elsewhere)` devuelve `true` porque los criterios semánticos que elegimos para la igualdad de dos objetos `MutableLocation` se han cumplido.

Las llamadas a métodos NO pasan objetos en absoluto

Las llamadas al método Java usan el *paso por valor* [1] para pasar argumentos y devolver un resultado.

Cuando pasa un valor de referencia a un método, en realidad está pasando una referencia a un objeto *por valor* , lo que significa que está creando una copia de la referencia del objeto.

Siempre que ambas referencias de objetos sigan apuntando al mismo objeto, puede modificar ese objeto de cualquiera de las dos referencias, y esto es lo que causa confusión para algunos.

Sin embargo, *no* está pasando un objeto por referencia [2.] La distinción es que si la copia de referencia del objeto se modifica para apuntar a otro objeto, la referencia del objeto original seguirá apuntando al objeto original.

```
void f(MutableLocation foo) {
    foo = new MutableLocation(3, 4);   // Point local foo at a different object.
}

void g() {
    MutableLocation foo = MutableLocation(1, 2);
    f(foo);
    System.out.println("foo.x is " + foo.x); // Prints "foo.x is 1".
}
```

Tampoco estás pasando una copia del objeto.

```
void f(MutableLocation foo) {
    foo.x = 42;
}

void g() {
    MutableLocation foo = new MutableLocation(0, 0);
    f(foo);
    System.out.println("foo.x is " + foo.x); // Prints "foo.x is 42"
}
```

---

1 - En idiomas como Python y Ruby, el término "pasar compartiendo" se prefiere para "pasar por valor" de un objeto / referencia.

2 - El término "pasar por referencia" o "llamada por referencia" tiene un significado muy específico en la terminología del lenguaje de programación. En efecto, significa que pasa la dirección *de una variable o un elemento de matriz* , de

modo que cuando el método llamado asigna un nuevo valor al argumento formal, cambia el valor en la variable original. Java no soporta esto. Para obtener una descripción más completa de los diferentes mecanismos para pasar parámetros, consulte https://en.wikipedia.org/wiki/Evaluation_strategy .

Pitfall: combinación de asignación y efectos secundarios

Ocasionalmente vemos preguntas de Java de StackOverflow (y preguntas de C o C ++) que preguntan algo como esto:

```
i += a[i++] + b[i--];
```

evalúa a ... para algunos estados iniciales conocidos de `i` , `a` y `b` .

Generalmente hablando:

- para Java, la respuesta siempre se especifica [1] , pero no es obvia y, a menudo, difícil de entender
- para C y C ++ la respuesta a menudo no se especifica.

Tales ejemplos se utilizan a menudo en exámenes o entrevistas de trabajo como un intento de ver si el estudiante o el entrevistado entiende cómo funciona realmente la evaluación de expresiones en el lenguaje de programación Java. Esto es posiblemente legítimo como una "prueba de conocimiento", pero eso no significa que debas hacer esto en un programa real.

Para ilustrar, el siguiente ejemplo aparentemente simple ha aparecido varias veces en preguntas de StackOverflow (como esta). En algunos casos, aparece como un error genuino en el código de alguien.

```
int a = 1;
a = a++;
System.out.println(a);    // What does this print.
```

La mayoría de los programadores (incluidos los expertos en Java) que leen esas declaraciones *rápidamente* dirían que produce `2` . De hecho, produce `1` . Para una explicación detallada de por qué, lea esta Respuesta .

Sin embargo la comida para llevar real a partir de esto y ejemplos similares es que *cualquier* declaración de Java que *tanto* le asigna *y* los efectos secundarios de la misma variable va a ser *en el mejor de* difícil de entender, y *en* el *peor* francamente engañosa. Debes evitar escribir código como este.

---

1 - módulo problemas potenciales con el modelo de memoria de Java si las variables u objetos son visibles a otros hilos.

Pitfall: no entender que String es una clase inmutable

Los nuevos programadores de Java a menudo olvidan, o no comprenden completamente, que la clase `String` Java es inmutable. Esto conduce a problemas como el del siguiente ejemplo:

```
public class Shout {
    public static void main(String[] args) {
        for (String s : args) {
            s.toUpperCase();
            System.out.print(s);
            System.out.print(" ");
        }
        System.out.println();
    }
}
```

Se supone que el código anterior imprime los argumentos de la línea de comandos en mayúsculas. Desafortunadamente, no funciona, el caso de los argumentos no se cambia. El problema es esta afirmación:

```
s.toUpperCase();
```

Podría pensar que llamar a `toUpperCase()` cambiará `s` a una cadena en mayúsculas. No lo hace No se puede `String` objetos de `String` son inmutables. No se pueden cambiar.

En realidad, el método `toUpperCase()` *devuelve* un objeto `String` que es una versión en mayúsculas del `String` que lo llamas. Probablemente este será un nuevo objeto `String` , pero si `s` ya estaba todo en mayúsculas, el resultado podría ser la cadena existente.

Por lo tanto, para utilizar este método de manera efectiva, debe usar el objeto devuelto por la llamada al método; por ejemplo:

```
s = s.toUpperCase();
```

De hecho, la regla de "las cadenas nunca cambian" se aplica a todos los métodos de `String` . Si recuerdas eso, entonces puedes evitar toda una categoría de errores de principiantes.

Lea Errores comunes de Java en línea: https://riptutorial.com/es/java/topic/4388/errores-comunes-de-java

Capítulo 67: Errores de Java - Nulls y NullPointerException

Observaciones

El valor `null` es el valor predeterminado para un valor no inicializado de un campo cuyo tipo es un tipo de referencia.

`NullPointerException` (o NPE) es la excepción que se produce cuando intenta realizar una operación inapropiada en la referencia de objeto `null` . Tales operaciones incluyen:

- llamando a un método de instancia en un objeto de destino `null` ,
- accediendo a un campo de un objeto objetivo `null` ,
- intentando indexar un objeto de matriz `null` o acceder a su longitud,
- usando una referencia de objeto `null` como mutex en un bloque `synchronized` ,
- echando una referencia de objeto `null` ,
- Unboxing una referencia de objeto `null` , y
- lanzando una referencia de objeto `null` .

Las causas de raíz más comunes para NPEs:

- olvidando inicializar un campo con un tipo de referencia,
- olvidarse de inicializar elementos de una matriz de un tipo de referencia, o
- no probar los resultados de ciertos métodos de API que se *especifican* como devueltos `null` en ciertas circunstancias.

Ejemplos de métodos comúnmente usados que devuelven `null` incluyen:

- El método de `get(key)` en la API de `Map` devolverá un `null` si lo llama con una clave que no tiene una asignación.
- Los `getResource(path)` y `getResourceAsStream(path)` en las API de `ClassLoader` y `Class` devolverán el `null` si no se puede encontrar el recurso.
- El método `get()` en la API de `Reference` devolverá un `null` si el recolector de basura ha borrado la referencia.
- Varios métodos `getXxxx` en las API del servlet de Java EE devolverán un `null` si intenta recuperar un parámetro de solicitud, una sesión o un atributo de sesión inexistentes, etc.

Existen estrategias para evitar las NPE no deseadas, como probar explícitamente la existencia de valores `null` o el uso de la "Notación Yoda", pero estas estrategias a menudo tienen el resultado indeseable de *ocultar* problemas en su código que realmente deberían solucionarse.

Examples

Trampa: el uso innecesario de envolturas primitivas puede llevar a

NullPointerExceptions

A veces, los programadores que son nuevos Java usarán tipos primitivos y envoltorios de manera intercambiable. Esto puede llevar a problemas. Considera este ejemplo:

```
public class MyRecord {
    public int a, b;
    public Integer c, d;
}

...
MyRecord record = new MyRecord();
record.a = 1;               // OK
record.b = record.b + 1;    // OK
record.c = 1;               // OK
record.d = record.d + 1;    // throws a NullPointerException
```

Nuestra clase [1] de `MyRecord` basa en la inicialización predeterminada para inicializar los valores en sus campos. Por lo tanto, cuando `new` un registro, los `a` y `b` campos se ponen a cero, y los `c` y `d` campos se establecerán en `null` .

Cuando intentamos usar los campos inicializados predeterminados, vemos que los campos `int` funcionan todo el tiempo, pero los campos de `Integer` funcionan en algunos casos y no en otros. Específicamente, en el caso de que falle (con `d`), lo que sucede es que la expresión del lado derecho intenta desempaquetar una referencia `null` , y eso es lo que hace que se lance la excepción `NullPointerException` .

Hay un par de maneras de ver esto:

- Si los campos `c` y `d` deben ser envoltorios primitivos, entonces no debemos confiar en la inicialización predeterminada, o deberíamos estar comprobando que no `null` . Para anterior es el enfoque correcto a *menos que* haya un significado definido para los campos en estado `null` .
- Si los campos no necesitan ser envoltorios primitivos, entonces es un error hacerlos envoltorios primitivos. Además de este problema, los envoltorios primitivos tienen gastos generales adicionales en relación con los tipos primitivos.

La lección aquí es no usar tipos de envoltorios primitivos a menos que realmente lo necesite.

---

1 - Esta clase no es un ejemplo de buenas prácticas de codificación. Por ejemplo, una clase bien diseñada no tendría campos públicos. Sin embargo, ese no es el punto de este ejemplo.

Pitfall - Usar nulo para representar una matriz o colección vacía

Algunos programadores piensan que es una buena idea ahorrar espacio utilizando un `null` para representar una matriz o colección vacía. Si bien es cierto que puede ahorrar una pequeña cantidad de espacio, la otra cara es que hace que su código sea más complicado y más frágil. Compara estas dos versiones de un método para sumar una matriz:

La primera versión es cómo normalmente codificarías el método:

```
/**
 * Sum the values in an array of integers.
 * @arg values the array to be summed
 * @return the sum
 **/
public int sum(int[] values) {
    int sum = 0;
    for (int value : values) {
        sum += value;
    }
    return sum;
}
```

La segunda versión es cómo debe codificar el método si tiene la costumbre de usar `null` para representar una matriz vacía.

```
/**
 * Sum the values in an array of integers.
 * @arg values the array to be summed, or null.
 * @return the sum, or zero if the array is null.
 **/
public int sum(int[] values) {
    int sum = 0;
    if (values != null) {
        for (int value : values) {
            sum += value;
        }
    }
    return sum;
}
```

Como puedes ver, el código es un poco más complicado. Esto es directamente atribuible a la decisión de usar `null` de esta manera.

Ahora considere si esta matriz que podría ser `null` se usa en muchos lugares. En cada lugar donde lo use, debe considerar si necesita realizar una prueba de `null` . Si pierde una prueba `null` que debe estar allí, se arriesga a una `NullPointerException` . Por lo tanto, la estrategia de utilizar `null` de esta manera hace que su aplicación sea más frágil; Es decir, más vulnerable a las consecuencias de los errores de programación.

La lección aquí es usar matrices vacías y listas vacías cuando eso es lo que quieres decir.

```
int[] values = new int[0];                      // always empty
List<Integer> list = new ArrayList();           // initially empty
List<Integer> list = Collections.emptyList();   // always empty
```

La sobrecarga de espacio es pequeña, y hay otras formas de minimizarlo si esto es algo que vale la pena hacer.

Pitfall - "Haciendo buenos" nulos inesperados

En StackOverflow, a menudo vemos código como este en Respuestas:

```
public String joinStrings(String a, String b) {
    if (a == null) {
        a = "";
    }
    if (b == null) {
        b = "";
    }
    return a + ": " + b;
}
```

A menudo, esto se acompaña con una afirmación que es "la mejor práctica" para realizar una prueba `null` como esta para evitar `NullPointerException` .

¿Es la mejor práctica? En resumen: No.

Hay algunas suposiciones subyacentes que deben ser cuestionadas antes de que podamos decir si es una buena idea hacer esto en nuestras `joinStrings` :

¿Qué significa que "a" o "b" sean nulos?

Un valor de `String` puede ser cero o más caracteres, por lo que ya tenemos una forma de representar una cadena vacía. ¿ `null` significa algo diferente a `""` ? Si no, es problemático tener dos formas de representar una cadena vacía.

¿El nulo vino de una variable no inicializada?

Un `null` puede provenir de un campo no inicializado o de un elemento de matriz no inicializado. El valor podría no estar inicializado por diseño o por accidente. Si fue por accidente este es un error.

¿El nulo representa un "no se sabe" o "valor perdido"?

A veces una `null` puede tener un significado genuino; por ejemplo, que el valor real de una variable es desconocido o no está disponible o es "opcional". En Java 8, la clase `Optional` proporciona una mejor manera de expresar eso.

Si se trata de un error (o un error de diseño), ¿debemos "solucionarlo"?

Una interpretación del código es que estamos "arreglando" un `null` inesperado al usar una cadena vacía en su lugar. ¿Es la estrategia correcta? ¿Sería mejor dejar que la `NullPointerException` suceda, y luego capturar la excepción más arriba en la pila y registrarla como un error?

El problema con "hacer el bien" es que puede ocultar el problema o dificultar su diagnóstico.

¿Es esto eficiente / bueno para la calidad del código?

Si el enfoque de "hacer el bien" se usa constantemente, su código contendrá muchas pruebas nulas "defensivas". Esto va a hacer que sea más largo y más difícil de leer. Además, todas estas pruebas y "hacer el bien" pueden afectar el rendimiento de su aplicación.

En resumen

Si `null` es un valor significativo, entonces la prueba para el caso `null` es el enfoque correcto. El corolario es que si un valor `null` es significativo, entonces esto debe documentarse claramente en los javadocs de cualquier método que acepte el valor `null` o lo devuelva.

De lo contrario, es una mejor idea tratar un `null` inesperado como un error de programación y dejar que se produzca la `NullPointerException` para que el desarrollador sepa que hay un problema en el código.

Pitfall - Devolver nulo en lugar de lanzar una excepción

Algunos programadores de Java tienen una aversión general a lanzar o propagar excepciones. Esto conduce a un código como el siguiente:

```
public Reader getReader(String pathname) {
    try {
        return new BufferedReader(FileReader(pathname));
    } catch (IOException ex) {
        System.out.println("Open failed: " + ex.getMessage());
        return null;
    }
```

}

Entonces, ¿cuál es el problema con eso?

El problema es que `getReader` está devolviendo un `null` como un valor especial para indicar que no se pudo abrir el `Reader` . Ahora se debe probar el valor devuelto para ver si es `null` antes de usarlo. Si se omite la prueba, el resultado será una `NullPointerException` .

En realidad hay tres problemas aquí:

1. La `IOException` fue capturada demasiado pronto.
2. La estructura de este código significa que existe el riesgo de que se filtre un recurso.
3. Luego se usó un `null` porque no había un `Reader` "real" disponible para devolver.

De hecho, asumiendo que la excepción debía ser detectada temprano de esta manera, había un par de alternativas para devolver `null` :

1. Sería posible implementar una clase `NullReader` ; por ejemplo, una en la que las operaciones de API se comportan como si el lector ya estuviera en la posición de "final de archivo".
2. Con Java 8, sería posible declarar `getReader` como retornando un `Optional<Reader>` .

Pitfall: no se comprueba si un flujo de E / S ni siquiera se inicializa al cerrarlo

Para evitar pérdidas de memoria, no se debe olvidar cerrar una secuencia de entrada o una secuencia de salida cuyo trabajo se haya realizado. Esto generalmente se hace con una declaración `try` - `catch` - `finally` sin la parte `catch` :

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        out.close();
    }
}
```

Si bien el código anterior puede parecer inocente, tiene un defecto que puede hacer que la depuración sea imposible. Si la línea en la `out` se inicializa (`out = new FileOutputStream(filename)`) lanza una excepción, entonces `out` será `null` cuando `out.close()` se ejecuta, lo que resulta en una desagradable `NullPointerException` !

Para evitar esto, simplemente asegúrese de que el flujo no sea `null` antes de intentar cerrarlo.

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        if (out != null)
            out.close();
    }
}
```

Un enfoque aún mejor es `try` -con-recursos, ya que se va a cerrar automáticamente la corriente con una probabilidad de 0 a lanzar una NPE sin la necesidad de un `finally` de bloquear.

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    try (FileOutputStream out = new FileOutputStream(filename)) {
        for(; count > 0; count--)
            out.write(0);
    }
}
```

Pitfall: uso de la "notación Yoda" para evitar la excepción NullPointerException

Un montón de código de ejemplo publicado en StackOverflow incluye fragmentos como este:

```
if ("A".equals(someString)) {
```

```
    // do something
}
```

Esto "evita" o "evita" una posible `someString` `NullPointerException` en el caso de que `someString` sea `null` . Además, es discutible que

```
    "A".equals(someString)
```

es mejor que:

```
    someString != null && someString.equals("A")
```

(Es más conciso y, en algunas circunstancias, podría ser más eficiente. Sin embargo, como argumentamos a continuación, la concisión podría ser negativa).

Sin embargo, el verdadero escollo es usar la prueba de Yoda **para evitar `NullPointerExceptions`** como una cuestión de hábito.

Cuando escribes `"A".equals(someString)` realidad estás "haciendo bien" el caso en el que sucede que `someString` son `null` . Pero como lo explica otro ejemplo (Pitfall - "Hacer buenos" nulos inesperados), "Hacer buenos" valores `null` puede ser perjudicial por una variedad de razones.

Esto significa que las condiciones de Yoda no son "mejores prácticas" [1] . A menos que se espere el `null` , es mejor dejar que ocurra la `NullPointerException` para que pueda obtener una falla de prueba de la unidad (o un informe de error). Eso le permite encontrar y corregir el error que provocó que apareciera un `null` inesperado / no deseado.

Las condiciones de Yoda solo deben usarse en los casos en que se *espera* el `null` porque el objeto que está probando proviene de una API que está *documentada* como que devuelve un `null` . Y podría decirse que podría ser mejor usar una de las formas menos bonitas que expresan la prueba porque eso ayuda a resaltar la prueba `null` para alguien que está revisando su código.

---

1 - Según Wikipedia : *"Las mejores prácticas de codificación son un conjunto de reglas informales que la comunidad de desarrollo de software ha aprendido a lo largo del tiempo y puede ayudar a mejorar la calidad del software".* . El uso de la notación Yoda no logra esto. En muchas situaciones, empeora el código.

Lea Errores de Java - Nulls y NullPointerException en línea:
https://riptutorial.com/es/java/topic/5680/errores-de-java---nulls-y-nullpointerexception

Capítulo 68: Errores de Java - Problemas de rendimiento

Introducción

Este tema describe una serie de "errores" (es decir, los errores que cometen los programadores java novatos) que se relacionan con el rendimiento de la aplicación Java.

Observaciones

Este tema describe algunas prácticas de codificación "micro" de Java que son ineficientes. En la mayoría de los casos, las ineficiencias son relativamente pequeñas, pero aún así vale la pena evitarlas.

Examples

Pitfall - Los gastos generales de crear mensajes de registro

`TRACE` niveles de registro `TRACE` y `DEBUG` están ahí para poder transmitir detalles sobre el funcionamiento del código dado en tiempo de ejecución. Por lo general, se recomienda establecer el nivel de registro por encima de estos, sin embargo, se debe tener cuidado con estas afirmaciones para que no afecten el rendimiento, incluso cuando aparentemente están "desactivadas".

Considere esta declaración de registro:

```
// Processing a request of some kind, logging the parameters
LOG.debug("Request coming from " + myInetAddress.toString()
          + " parameters: " + Arrays.toString(veryLongParamArray));
```

Incluso cuando el nivel de registro se establece en `INFO` , los argumentos pasados a `debug()` se evaluarán en cada ejecución de la línea. Esto hace que consuma innecesariamente en varios aspectos:

- Concatenación de `String` : se crearán múltiples instancias de `String`
- `InetAddress` podría incluso hacer una búsqueda de DNS.
- El `veryLongParamArray` puede ser muy largo: crear una cadena a partir de ella consume memoria, lleva tiempo

Solución

La mayoría de los marcos de registro proporcionan medios para crear mensajes de registro utilizando cadenas de arreglos y referencias de objetos. El mensaje de registro se evaluará solo si

el mensaje está realmente registrado. Ejemplo:

```
// No toString() evaluation, no string concatenation if debug is disabled
LOG.debug("Request coming from {} parameters: {}", myInetAddress, parameters));
```

Esto funciona muy bien siempre que todos los parámetros se puedan convertir en cadenas usando String.valueOf (Object) . Si la composición del mensaje de registro es más compleja, el nivel de registro se puede verificar antes del registro:

```
if (LOG.isDebugEnabled()) {
    // Argument expression evaluated only when DEBUG is enabled
    LOG.debug("Request coming from {}, parameters: {}", myInetAddress,
              Arrays.toString(veryLongParamArray);
}
```

Aquí, `LOG.debug()` con el costoso `Arrays.toString(Obect[])` se procesa solo cuando `DEBUG` está realmente habilitado.

Pitfall - La concatenación de cadenas en un bucle no se escala

Considere el siguiente código como una ilustración:

```
public String joinWords(List<String> words) {
    String message = "";
    for (String word : words) {
        message = message + " " + word;
    }
    return message;
}
```

Desafortunadamente, este código es ineficiente si la lista de `words` es larga. La raíz del problema es esta declaración:

```
message = message + " " + word;
```

Para cada iteración de bucle, esta declaración crea una nueva cadena de `message` contiene una copia de todos los caracteres en la cadena de `message` original con caracteres adicionales añadidos. Esto genera una gran cantidad de cadenas temporales, y hace una gran cantidad de copias.

Cuando analizamos `joinWords` , asumiendo que hay N palabras con una longitud promedio de M, encontramos que se crean cadenas temporales O (N) y se copiarán caracteres O (MN^2) en el proceso. El componente N^2 es particularmente preocupante.

El enfoque recomendado para este tipo de problema [1] es utilizar un `StringBuilder` lugar de la concatenación de cadenas de la siguiente manera:

```
public String joinWords2(List<String> words) {
    StringBuilder message = new StringBuilder();
    for (String word : words) {
```

```
        message.append(" ").append(word);
    }
    return message.toString();
}
```

El análisis de `joinWords2` debe tener en cuenta los gastos generales de "hacer crecer" la matriz de respaldo `StringBuilder` que contiene los caracteres del constructor. Sin embargo, resulta que la cantidad de nuevos objetos creados es O (logN) y que la cantidad de caracteres copiados es O (MN). El último incluye caracteres copiados en la llamada final `toString()` .

(Puede ser posible sintonizar esto aún más, creando el `StringBuilder` con la capacidad correcta para comenzar. Sin embargo, la complejidad general sigue siendo la misma).

Volviendo al método original de `joinWords` , resulta que la declaración crítica será optimizada por un compilador típico de Java a algo como esto:

```
  StringBuilder tmp = new StringBuilder();
  tmp.append(message).append(" ").append(word);
  message = tmp.toString();
```

Sin embargo, el compilador de Java no "levantará" el `StringBuilder` fuera del bucle, como hicimos a mano en el código para `joinWords2` .

Referencia:

- "¿El operador String '+' de Java en un bucle es lento?"

---

1 - En Java 8 y `Joiner` posteriores, la clase `Joiner` puede usarse para resolver este problema en particular. Sin embargo, de eso no se trata *realmente* este ejemplo.

Pitfall: el uso de 'nuevo' para crear instancias de contenedor primitivas es ineficiente

El lenguaje Java le permite usar lo `new` para crear instancias `Integer` , `Boolean` , etc., pero generalmente es una mala idea. Es mejor usar el autoboxing (Java 5 y posterior) o el método `valueOf` .

```
Integer i1 = new Integer(1);       // BAD
Integer i2 = 2;                    // BEST (autoboxing)
Integer i3 = Integer.valueOf(3);   // OK
```

La razón por la que el uso de un `new Integer(int)` explícitamente es una mala idea es que crea un nuevo objeto (a menos que el compilador JIT lo optimice). Por el contrario, cuando se usa el autoboxing o una llamada explícita a `valueOf` , el tiempo de ejecución de Java intentará reutilizar un objeto `Integer` desde un caché de objetos preexistentes. Cada vez que el tiempo de ejecución tiene un "hit" de caché, evita la creación de un objeto. Esto también ahorra memoria del montón y reduce los gastos generales del GC causados por la rotación de objetos.

Notas:

1. En las implementaciones recientes de Java, el autoboxing se implementa llamando a `valueOf` , y hay cachés para `Boolean` , `Byte` , `Short` , `Integer` , `Long` y `Character` .
2. El comportamiento de almacenamiento en caché para los tipos integrales está ordenado por la especificación del lenguaje Java.

Pitfall - Llamar 'nueva cadena (String)' es ineficiente

Usar una `new String(String)` para duplicar una cadena es ineficiente y casi siempre es innecesario.

- Los objetos de cadena son inmutables, por lo que no es necesario copiarlos para protegerse contra los cambios.
- En algunas versiones anteriores de Java, los objetos `String` pueden compartir matrices de respaldo con otros objetos `String` . En esas versiones, es posible perder memoria creando una subcadena (pequeña) de una cadena (grande) y reteniéndola. Sin embargo, a partir de Java 7 en adelante, las matrices de respaldo de `String` no se comparten.

En ausencia de cualquier beneficio tangible, llamar `new String(String)` es simplemente un desperdicio:

- Hacer la copia lleva tiempo de CPU.
- La copia utiliza más memoria, lo que aumenta la huella de memoria de la aplicación y / o aumenta los gastos generales del GC.
- Las operaciones como `equals(Object)` y `hashCode()` pueden ser más lentas si se copian los objetos String.

Pitfall - Calling System.gc () es ineficiente

Es (casi siempre) una mala idea llamar a `System.gc()` .

El javadoc para el método `gc()` especifica lo siguiente:

> "Llamar al método `gc` sugiere que la Máquina Virtual de Java haga un esfuerzo por reciclar los objetos no utilizados para que la memoria que ocupan actualmente esté disponible para una reutilización rápida. Cuando el control regresa de la llamada al método, la Máquina Virtual de Java ha hecho un mejor esfuerzo para reclamar espacio de todos los objetos descartados ".

Hay un par de puntos importantes que se pueden extraer de esto:

1. El uso de la palabra "sugiere" en lugar de (decir) "dice" significa que la JVM es libre de ignorar la sugerencia. El comportamiento predeterminado de la JVM (lanzamientos recientes) es seguir la sugerencia, pero esto puede `-XX:+DisableExplicitGC` configurando `-XX:+DisableExplicitGC` cuando se `-XX:+DisableExplicitGC` la JVM.

2. La frase "un mejor esfuerzo para recuperar espacio de todos los objetos descartados" implica que llamar a `gc` activará una recolección de basura "completa".

Entonces, ¿por qué es una mala idea llamar a `System.gc()` ?

En primer lugar, ejecutar una recolección de basura completa es costoso. Un GC completo implica visitar y "marcar" todos los objetos a los que todavía se puede acceder; Es decir, todo objeto que no sea basura. Si dispara esto cuando no hay mucha basura que recoger, entonces el GC hace mucho trabajo por un beneficio relativamente pequeño.

En segundo lugar, una recolección de basura completa puede perturbar las propiedades de "localidad" de los objetos que no se recolectan. Los objetos que se asignan por el mismo subproceso casi al mismo tiempo tienden a asignarse juntos en la memoria. Esto es bueno. Es probable que los objetos que se asignan al mismo tiempo estén relacionados; es decir, hacer referencia entre sí. Si su aplicación utiliza esas referencias, es probable que el acceso a la memoria sea más rápido debido a los diversos efectos de la memoria y el almacenamiento en caché de la página. Desafortunadamente, una colección de basura completa tiende a mover objetos, de modo que los objetos que una vez estuvieron cerca ahora están más separados.

Tercero, la ejecución de una recolección de basura completa puede hacer que su aplicación se detenga hasta que se complete la recolección. Mientras esto suceda, su aplicación no responderá.

De hecho, la mejor estrategia es dejar que la JVM decida cuándo ejecutar el GC y qué tipo de colección ejecutar. Si no interfiere, la JVM elegirá un tiempo y un tipo de colección que optimice el rendimiento o minimice los tiempos de pausa del GC.

Al principio dijimos "... (casi siempre) una mala idea ...". De hecho, hay un par de escenarios en los que *podría* ser una buena idea:

1. Si está implementando una prueba unitaria para algún código que es sensible a la recolección de basura (por ejemplo, algo que involucra finalizadores o referencias débiles / blandas / fantasmas), entonces puede ser necesario llamar a `System.gc()` .
2. En algunas aplicaciones interactivas, puede haber puntos particulares en el tiempo donde el usuario no se preocupará si hay una pausa de recolección de basura. Un ejemplo es un juego donde hay pausas naturales en el "juego"; por ejemplo, cuando se carga un nuevo nivel.

Pitfall - El uso excesivo de tipos de envoltorios primitivos es ineficiente

Considera estas dos piezas de código:

```
int a = 1000;
int b = a + 1;
```

y

```
Integer a = 1000;
Integer b = a + 1;
```

Pregunta: ¿Qué versión es más eficiente?

Respuesta: Las dos versiones parecen casi idénticas, pero la primera es mucho más eficiente que la segunda.

La segunda versión está utilizando una representación para los números que ocupa más espacio, y se basa en el boxeo automático y el boxeo automático entre bastidores. De hecho, la segunda versión es directamente equivalente al siguiente código:

```
Integer a = Integer.valueOf(1000);               // box 1000
Integer b = Integer.valueOf(a.intValue() + 1);   // unbox 1000, add 1, box 1001
```

Comparando esto con la otra versión que usa `int` , hay claramente tres llamadas de método adicionales cuando se usa `Integer` . En el caso de `valueOf` , cada llamada creará e inicializará un nuevo objeto `Integer` . Es probable que todo este trabajo extra de boxeo y desempaquetado haga que la segunda versión sea un orden de magnitud más lenta que la primera.

Además de eso, la segunda versión está asignando objetos en el montón en cada llamada `valueOf` . Si bien la utilización del espacio es específica de la plataforma, es probable que se encuentre en la región de 16 bytes para cada objeto `Integer` . En contraste, la versión `int` necesita cero espacio de almacenamiento adicional, asumiendo que `a` y `b` son variables locales.

Otra razón importante por la que los primitivos son más rápidos que sus equivalentes en caja es la forma en que sus respectivos tipos de matrices se presentan en la memoria.

Si toma `int[]` y `Integer[]` como ejemplo, en el caso de `int[]` los *valores* `int` se establecen de forma contigua en la memoria. Pero en el caso de un `Integer[]` no son los valores que se presentan, sino las referencias (punteros) a los objetos `Integer` , que a su vez contienen los valores `int` reales.

Además de ser un nivel adicional de direccionamiento indirecto, este puede ser un gran tanque cuando se trata de la localidad de caché cuando se itera sobre los valores. En el caso de un `int[]` la CPU podría obtener todos los valores de la matriz, en su caché a la vez, porque son contiguos en la memoria. Pero en el caso de un `Integer[]` la CPU potencialmente tiene que hacer una recuperación de memoria adicional para cada elemento, ya que la matriz solo contiene referencias a los valores reales.

En resumen, el uso de tipos primitivos de envoltura es relativamente costoso tanto en recursos de CPU como de memoria. Su uso innecesario es eficiente.

Pitfall - Iterar las claves de un mapa puede ser ineficiente

El siguiente código de ejemplo es más lento de lo que debe ser:

```
Map<String, String> map = new HashMap<>();
for (String key : map.keySet()) {
    String value = map.get(key);
    // Do something with key and value
}
```

Esto se debe a que requiere una búsqueda en el mapa (el método `get()`) para cada clave en el mapa. Es posible que esta búsqueda no sea eficiente (en un HashMap, implica llamar a `hashCode` en la clave, luego buscar el depósito correcto en las estructuras de datos internas y, a veces, incluso llamar a `equals`). En un mapa grande, esto puede no ser una sobrecarga trivial.

La forma correcta de evitar esto es iterar en las entradas del mapa, que se detalla en el tema Colecciones.

Pitfall: el uso de size () para comprobar si una colección está vacía es ineficiente.

El Java Collections Framework proporciona dos métodos relacionados para todos los objetos de la `Collection` :

- `size()` devuelve el número de entradas en una `Collection` , y
- `isEmpty()` método `isEmpty()` devuelve verdadero si (y solo si) la `Collection` está vacía.

Ambos métodos se pueden utilizar para probar el vacío de la colección. Por ejemplo:

```
Collection<String> strings = new ArrayList<>();
boolean isEmpty_wrong = strings.size() == 0; // Avoid this
boolean isEmpty = strings.isEmpty();         // Best
```

Si bien estos enfoques parecen iguales, algunas implementaciones de colección no almacenan el tamaño. Para tal colección, la implementación de `size()` necesita calcular el tamaño cada vez que se llama. Por ejemplo:

- Una clase de lista vinculada simple (pero no el `java.util.LinkedList`) puede necesitar atravesar la lista para contar los elementos.
- La clase `ConcurrentHashMap` necesita sumar las entradas en todos los "segmentos" del mapa.
- Una implementación perezosa de una colección podría necesitar realizar la colección completa en la memoria para contar los elementos.

Por el contrario, un método `isEmpty()` solo necesita probar si hay *al menos un* elemento en la colección. Esto no implica contar los elementos.

Si bien `size() == 0` no siempre es menos eficiente que `isEmpty()` , es inconcebible que un `isEmpty()` implementado correctamente sea menos eficiente que `size() == 0` . Por `isEmpty()` tanto, se prefiere `isEmpty()` .

Pitfall - Problemas de eficiencia con expresiones regulares

La coincidencia de expresiones regulares es una herramienta poderosa (en Java y en otros contextos) pero tiene algunos inconvenientes. Uno de estos que las expresiones regulares tiende a ser bastante caro.

Las instancias de Pattern y Matcher deben ser reutilizadas

Considere el siguiente ejemplo:

```
/**
 * Test if all strings in a list consist of English letters and numbers.
 * @param strings the list to be checked
 * @return 'true' if an only if all strings satisfy the criteria
 * @throws NullPointerException if 'strings' is 'null' or a 'null' element.
 */
public boolean allAlphanumeric(List<String> strings) {
    for (String s : strings) {
        if (!s.matches("[A-Za-z0-9]*")) {
            return false;
        }
    }
    return true;
}
```

Este código es correcto, pero es ineficiente. El problema está en la `matches(...)` llamada. Bajo el capó, `s.matches("[A-Za-z0-9]*")` es equivalente a esto:

```
Pattern.matches(s, "[A-Za-z0-9]*")
```

que a su vez es equivalente a

```
Pattern.compile("[A-Za-z0-9]*").matcher(s).matches()
```

La `Pattern.compile("[A-Za-z0-9]*")` analiza la expresión regular, la analiza y construye un objeto `Pattern` que contiene la estructura de datos que utilizará el motor de expresiones regulares. Este es un cálculo no trivial. Luego se crea un objeto `Matcher` para envolver el argumento `s` . Finalmente, llamamos `match()` para hacer la coincidencia de patrón real.

El problema es que todo este trabajo se repite para cada iteración de bucle. La solución es reestructurar el código de la siguiente manera:

```
private static Pattern ALPHA_NUMERIC = Pattern.compile("[A-Za-z0-9]*");

public boolean allAlphanumeric(List<String> strings) {
    Matcher matcher = ALPHA_NUMERIC.matcher("");
    for (String s : strings) {
        matcher.reset(s);
        if (!matcher.matches()) {
            return false;
        }
    }
    return true;
}
```

Tenga en cuenta que el javadoc para los estados de `Pattern` :

> Las instancias de esta clase son inmutables y son seguras para el uso de múltiples subprocesos simultáneos. Las instancias de la clase `Matcher` no son seguras para tal uso.

No uses match () cuando deberías usar find ()

Supongamos que desea probar si una cadena `s` contiene tres o más dígitos seguidos. Usted puede expresar esto de varias maneras, incluyendo:

```
if (s.matches(".*[0-9]{3}.*")) {
    System.out.println("matches");
}
```

o

```
if (Pattern.compile("[0-9]{3}").matcher(s).find()) {
    System.out.println("matches");
}
```

El primero es más conciso, pero también es probable que sea menos eficiente. A primera vista, la primera versión intentará hacer coincidir toda la cadena con el patrón. Además, dado que ". *" Es un patrón "codicioso", es probable que el emparejador del patrón avance "con entusiasmo" hasta el final de la cadena y retroceda hasta que encuentre una coincidencia.

Por el contrario, la segunda versión buscará de izquierda a derecha y dejará de buscar tan pronto como encuentre los 3 dígitos seguidos.

Usar alternativas más eficientes a las expresiones regulares.

Las expresiones regulares son una herramienta poderosa, pero no deberían ser su única herramienta. Muchas tareas se pueden hacer de manera más eficiente de otras maneras. Por ejemplo:

```
Pattern.compile("ABC").matcher(s).find()
```

hace lo mismo que:

```
s.contains("ABC")
```

Excepto que este último es mucho más eficiente. (Incluso si puede amortizar el costo de compilar la expresión regular).

A menudo, la forma no regex es más complicada. Por ejemplo, la prueba realizada por la `matches()` llama al método `allAlplanumeric` anterior `allAlplanumeric` puede reescribirse como:

```
public boolean matches(String s) {
    for (char c : s) {
        if ((c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z') ||
            (c >= '0' && c <= '9')) {
             return false;
        }
```

```
    }
    return true;
}
```

Ahora es más código que usar un `Matcher` , pero también va a ser mucho más rápido.

Retroceso catastrófico

(Esto es potencialmente un problema con todas las implementaciones de expresiones regulares, pero lo mencionaremos aquí porque es un escollo para `Pattern` uso del `Pattern`).

Considere este ejemplo (artificial):

```
Pattern pat = Pattern.compile("(A+)+B");
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB").matches());
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAC").matches());
```

La primera `println` llamada imprimir rápidamente `true` . El segundo imprimirá `false` . Finalmente. De hecho, si experimenta con el código anterior, verá que cada vez que agregue una `A` antes de la `C` , el tiempo se duplicará.

Este es el comportamiento es un ejemplo de *retroceso catastrófico* . El motor de coincidencia de patrones que implementa la coincidencia de expresiones regulares está probando infructuosamente todas las formas *posibles en* que el patrón *podría* coincidir.

Veamos lo que realmente significa `(A+)+B` Superficialmente, parece decir "uno o más caracteres `A` seguidos de un valor `B` ", pero en realidad dice uno o más grupos, cada uno de los cuales consta de uno o más caracteres `A` Así por ejemplo:

- 'AB' solo coincide de una manera: '(A) B'
- 'AAB' coincide de dos maneras: '(AA) B' o '(A) (A) B`
- 'AAAB' coincide de cuatro maneras: '(AAA) B' o '(AA) (A) B `or '(A)(AA)B` o '(A) (A) (A) B`
- y así

En otras palabras, el número de posibles coincidencias es $2^{N,}$ donde N es el número de caracteres `A`

El ejemplo anterior está claramente diseñado, pero los patrones que muestran este tipo de características de rendimiento (es decir, `O(2^N)` u `O(N^K)` para un `K` grande aparecen con frecuencia cuando se utilizan expresiones regulares mal consideradas. Hay algunos remedios estándar:

- Evite anidar patrones repetitivos dentro de otros patrones repetitivos.
- Evite utilizar demasiados patrones de repetición.
- Use la repetición sin retroceso según sea apropiado.
- No utilice expresiones regulares para tareas de análisis complicadas. (Escriba un analizador adecuado en su lugar.)

Finalmente, tenga cuidado con las situaciones en las que un usuario o un cliente de API puede suministrar una cadena de expresiones regulares con características patológicas. Eso puede

llevar a una "denegación de servicio" accidental o deliberada.

Referencias:

- La etiqueta de expresiones regulares , en particular http://www.riptutorial.com/regex/topic/259/getting-started-with-regular-expressions/977/backtracking#t=20161001033913136116 3 y http://www.riptutorial.com/expresiones regex / topic / 259 / getting-started-with-regular-expressions / 4527 / cuando-no-debe-usar-expresiones-regulares # t = 20161001033959356491 3
- "Regex Performance" de Jeff Atwood.
- "Cómo matar a Java con una expresión regular" por Andreas Haufler.

Pitfall - Interning Strings para que puedas usar == es una mala idea

Cuando algunos programadores ven este consejo:

> "Probar cadenas usando `==` es incorrecto (a menos que las cadenas estén internadas)"

su reacción inicial es aplicar cadenas internas para que puedan usar `==` . (Después de todo `==` es más rápido que llamar a `String.equals(...)` , ¿no es así?)

Este es el enfoque equivocado, desde una serie de perspectivas:

Fragilidad

En primer lugar, solo puede usar de forma segura `==` si sabe que *todos* los objetos `String` que está probando han sido internados. El JLS garantiza que los literales de cadena en su código fuente se habrán internado. Sin embargo, ninguna de las API de Java SE estándar garantiza devolver cadenas internadas, aparte de `String.intern(String)` . Si pierde solo una fuente de objetos `String` que no han sido internados, su aplicación no será confiable. Esa falta de fiabilidad se manifestará como falsos negativos en lugar de excepciones que pueden dificultar su detección.

Costos de usar 'intern ()'

Bajo el capó, el internado funciona manteniendo una tabla hash que contiene objetos `String` previamente internados. Se utiliza algún tipo de mecanismo de referencia débil para que la tabla hash de internado no se convierta en una fuga de almacenamiento. Mientras que la tabla hash se implementa en código nativo (a diferencia de `HashMap` , `HashTable` y así sucesivamente), los `intern` llamadas son todavía relativamente costoso en términos de CPU y de memoria utilizadas.

Este costo debe compararse con el ahorro que obtendremos utilizando `==` lugar de `equals` . De hecho, no vamos a interrumpir el equilibrio a menos que cada cadena internada se compare unas pocas veces "varias veces".

(Aparte: las pocas situaciones en las que vale la pena realizar una pasantía tienden a ser reducir la huella de memoria de una aplicación donde las mismas cadenas se repiten muchas veces, *y* esas cadenas tienen una larga vida útil).

El impacto en la recolección de basura.

Además de los costos directos de CPU y memoria descritos anteriormente, las cadenas internas afectan el rendimiento del recolector de basura.

Para las versiones de Java anteriores a Java 7, las cadenas internas se mantienen en el espacio "PermGen" que se recopila con poca frecuencia. Si es necesario recopilar PermGen, esto (normalmente) activa una recolección de basura completa. Si el espacio de PermGen se llena completamente, la JVM se bloquea, incluso si había espacio libre en los espacios de almacenamiento dinámico normales.

En Java 7, el grupo de cadenas se movió de "PermGen" al montón normal. Sin embargo, la tabla hash seguirá siendo una estructura de datos de larga duración, lo que hará que las cadenas internas sean de larga duración. (Incluso si los objetos de cadena internados se asignaran en el espacio del Edén, probablemente se promoverían antes de ser recolectados).

Por lo tanto, en todos los casos, internar una cadena prolongará su tiempo de vida en relación con una cadena normal. Eso aumentará los gastos generales de recolección de basura durante la vida útil de la JVM.

El segundo problema es que la tabla hash necesita usar un mecanismo de referencia débil de algún tipo para evitar que la cadena pierda la memoria interna. Pero tal mecanismo es más trabajo para el recolector de basura.

Estos gastos generales de recolección de basura son difíciles de cuantificar, pero existen pocas dudas de que existan. Si usas `intern` extensivamente, podrían ser importantes.

El tamaño de hashtable del grupo de cadenas

De acuerdo con esta fuente , desde Java 6 en adelante, el conjunto de cadenas se implementa como una tabla hash de tamaño fijo con cadenas para tratar con las cadenas que se agrupan en el mismo grupo. En las primeras versiones de Java 6, la tabla hash tenía un tamaño constante (cableado). Se agregó un parámetro de ajuste (`-XX:StringTableSize`) como una actualización de mitad de vida a Java 6. Luego, en una actualización de mitad de vida a Java 7, el tamaño predeterminado de la agrupación se cambió de `1009` a `60013` .

La conclusión es que si tiene la intención de utilizar `intern` en su código, es *recomendable* elegir una versión de Java en la que el tamaño de la tabla hash sea ajustable y asegurarse de que ajusta el tamaño de manera adecuada. De lo contrario, el rendimiento del `intern` puede degradarse a medida que el grupo se hace más grande.

Interning como un potencial vector de denegación de servicio

El algoritmo de hashcode para cadenas es bien conocido. Si internas cadenas proporcionadas por usuarios o aplicaciones malintencionados, esto podría usarse como parte de un ataque de

denegación de servicio (DoS). Si el agente malicioso dispone que todas las cadenas que proporciona tienen el mismo código hash, esto podría llevar a una tabla hash no balanceada y un rendimiento `O(N)` para el `intern` ... donde `N` es el número de cadenas colisionadas.

(Hay formas más simples / más efectivas de lanzar un ataque DoS contra un servicio. Sin embargo, este vector podría usarse si el objetivo del ataque DoS es romper la seguridad o evadir las defensas DoS de primera línea).

Pitfall - Las lecturas / escrituras pequeñas en flujos no almacenados en búfer son ineficientes

Considere el siguiente código para copiar un archivo a otro:

```
import java.io.*;

public class FileCopy {

    public static void main(String[] args) throws Exception {
        try (InputStream is = new FileInputStream(args[0]);
             OutputStream os = new FileOutputStream(args[1])) {
           int octet;
           while ((octet = is.read()) != -1) {
               os.write(octet);
           }
        }
    }
}
```

(Hemos deliberado omitido la verificación normal de los argumentos, el informe de errores, etc., ya que no son relevantes para el *punto* de este ejemplo).

Si compila el código anterior y lo utiliza para copiar un archivo enorme, notará que es muy lento. De hecho, será al menos un par de órdenes de magnitud más lento que las utilidades estándar de copia de archivos del sistema operativo.

(¡ *Añada las mediciones de rendimiento reales aquí!*)

La razón principal por la que el ejemplo anterior es lento (en el caso del archivo grande) es que está realizando lecturas de un byte y escrituras de un byte en flujos de bytes sin búfer. La forma sencilla de mejorar el rendimiento es envolver las secuencias con secuencias almacenadas en búfer. Por ejemplo:

```
import java.io.*;

public class FileCopy {

    public static void main(String[] args) throws Exception {
        try (InputStream is = new BufferedInputStream(
                      new FileInputStream(args[0]));
             OutputStream os = new BufferedOutputStream(
                      new FileOutputStream(args[1]))) {
           int octet;
           while ((octet = is.read()) != -1) {
```

```
                os.write(octet);
            }
        }
    }
}
```

Estos pequeños cambios mejorarán la velocidad de copia de datos en *al menos* un par de órdenes de magnitud, dependiendo de diversos factores relacionados con la plataforma. Las envolturas de flujo almacenadas en búfer hacen que los datos se lean y escriban en trozos más grandes. Ambas instancias tienen buffers implementados como matrices de bytes.

- Con `is` , los datos se leen del archivo en el búfer unos pocos kilobytes a la vez. Cuando se llama a `read()` , la implementación normalmente devolverá un byte desde el búfer. Solo se leerá de la secuencia de entrada subyacente si el búfer se ha vaciado.
- El comportamiento para `os` es análogo. Las llamadas a `os.write(int)` escriben bytes únicos en el búfer. Los datos solo se escriben en el flujo de salida cuando el búfer está lleno, o cuando `os` se vacía o se cierra.

¿Qué pasa con las corrientes basadas en caracteres?

Como debe saber, Java I / O proporciona diferentes API para leer y escribir datos binarios y de texto.

- `InputStream` y `OutputStream` son las API base para E / S binarias basadas en flujo
- `Reader` y `Writer` son las API básicas para la E / S de texto basada en flujo.

Para texto de E / S, `BufferedReader` y `BufferedWriter` son los equivalentes de `BufferedInputStream` y `BufferedOutputStream` .

¿Por qué los flujos amortiguados hacen tanta diferencia?

La verdadera razón por la que las transmisiones en búfer ayudan al rendimiento es la forma en que una aplicación habla con el sistema operativo:

- El método Java en una aplicación Java o las llamadas a procedimientos nativos en las bibliotecas de tiempo de ejecución nativas de la JVM son rápidas. Por lo general, toman un par de instrucciones de la máquina y tienen un impacto mínimo en el rendimiento.
- Por el contrario, las llamadas en tiempo de ejecución de JVM al sistema operativo no son rápidas. Implican algo conocido como "syscall". El patrón típico para un syscall es el siguiente:

    1. Ponga los argumentos de syscall en los registros.
    2. Ejecutar una instrucción de trampa SYSENTER.
    3. El manejador de trampas cambió a un estado privilegiado y cambia las asignaciones de memoria virtual. Luego se envía al código para manejar el syscall específico.
    4. El controlador syscall comprueba los argumentos, teniendo cuidado de que no se le

diga que acceda a la memoria que el proceso del usuario no debería ver.
5. Se realiza el trabajo específico de syscall. En el caso de un syscall de `read` , esto puede implicar:
    1. verificar que haya datos para leer en la posición actual del descriptor de archivo
    2. llamar al controlador del sistema de archivos para obtener los datos requeridos del disco (o donde sea que estén almacenados) en el caché del búfer,
    3. copiar datos desde la memoria caché del búfer a la dirección proporcionada por la JVM
    4. Ajuste de la posición del descriptor de archivos de Pointstream
6. Regreso del syscall. Esto implica cambiar de nuevo las asignaciones de VM y cambiar de estado privilegiado.

Como se puede imaginar, realizar una sola llamada puede miles de instrucciones de la máquina. De manera conservadora, *al menos* dos órdenes de magnitud más largos que una llamada de método regular. (Probablemente tres o más).

Teniendo en cuenta esto, la razón por la que los flujos de búferes hacen una gran diferencia es que reducen drásticamente el número de syscalls. En lugar de hacer un syscall para cada llamada de `read()` , la secuencia de entrada almacenada en el búfer lee una gran cantidad de datos en un búfer según se requiera. La mayoría de las llamadas de `read()` en el flujo almacenado en búfer realizan algunas comprobaciones de límites simples y devuelven un `byte` que se leyó anteriormente. Un razonamiento similar se aplica en el caso del flujo de salida, y también en los casos del flujo de caracteres.

(Algunas personas piensan que el rendimiento de E / S en búfer proviene de la falta de coincidencia entre el tamaño de la solicitud de lectura y el tamaño de un bloque de disco, la latencia de rotación del disco y cosas así. De hecho, un sistema operativo moderno utiliza una serie de estrategias para garantizar que la aplicación *normalmente* no necesita esperar por el disco. Esta no es la explicación real.)

¿Las transmisiones amortiguadas son siempre una victoria?

No siempre. Las transmisiones en búfer son definitivamente una ganancia si su aplicación va a hacer muchas "pequeñas" lecturas o escrituras. Sin embargo, si su aplicación solo necesita realizar lecturas grandes o escrituras a / desde un `byte[]` grande `byte[]` o `char[]` , las secuencias con búfer no le brindarán beneficios reales. De hecho, incluso podría haber una (pequeña) penalización de rendimiento.

¿Es esta la forma más rápida de copiar un archivo en Java?

No, no lo es. Cuando utiliza las API basadas en flujo de Java para copiar un archivo, incurre en el costo de al menos una copia extra de memoria a memoria de los datos. Es posible evitar esto si utiliza las API de NIO `ByteBuffer` y `Channel` . (*Agregue un enlace a un ejemplo separado aquí*) .

Lea Errores de Java - Problemas de rendimiento en línea:
https://riptutorial.com/es/java/topic/5455/errores-de-java---problemas-de-rendimiento

Capítulo 69: Errores de Java - sintaxis de lenguaje

Introducción

Varios usos indebidos del lenguaje de programación Java pueden llevar a cabo un programa para generar resultados incorrectos a pesar de haber sido compilados correctamente. El propósito principal de este tema es enumerar las dificultades comunes con sus causas y proponer la forma correcta de evitar caer en tales problemas.

Observaciones

Este tema trata sobre aspectos específicos de la sintaxis del lenguaje Java que son propensos a errores o que no deben usarse de ciertas maneras.

Examples

Pitfall - Ignorar la visibilidad del método

Incluso los desarrolladores Java experimentados tienden a pensar que Java tiene solo tres modificadores de protección. ¡El lenguaje en realidad tiene cuatro! El nivel de visibilidad del **paquete privado** (también conocido como predeterminado) a menudo se olvida.

Debes prestar atención a los métodos que haces públicos. Los métodos públicos en una aplicación son la API visible de la aplicación. Esto debería ser lo más pequeño y compacto posible, especialmente si está escribiendo una biblioteca reutilizable (vea también el principio SOLID). Es importante considerar de manera similar la visibilidad de todos los métodos, y usar solo el acceso privado protegido o en paquetes cuando sea apropiado.

Cuando declara métodos que deberían ser **privados** como públicos, expone los detalles de la implementación interna de la clase.

Un corolario de esto es que solo prueba por unidad los métodos públicos de su clase; de hecho, **solo** puede probar métodos públicos. Es una mala práctica aumentar la visibilidad de los métodos privados solo para poder ejecutar pruebas unitarias contra esos métodos. La prueba de métodos públicos que llaman a los métodos con una visibilidad más restrictiva debería ser suficiente para probar una API completa. **Nunca se** debe ampliar su API con los métodos más comunes sólo para permitir las pruebas unitarias.

Pitfall - Falta un 'break' en un caso de 'cambio'

Estos problemas de Java pueden ser muy embarazosos y, a veces, no se han descubierto hasta que se ejecutan en producción. El comportamiento fallido en las declaraciones de cambio suele ser útil; sin embargo, faltar una palabra clave de "ruptura" cuando no se desea tal comportamiento

puede llevar a resultados desastrosos. Si ha olvidado poner un "descanso" en el "caso 0" en el ejemplo de código a continuación, el programa escribirá "Cero" seguido de "Uno", ya que el flujo de control que se encuentra aquí pasará por toda la declaración del "interruptor" hasta que Alcanza un “descanso”. Por ejemplo:

```
public static void switchCasePrimer() {
        int caseIndex = 0;
        switch (caseIndex) {
            case 0:
                System.out.println("Zero");
            case 1:
                System.out.println("One");
                break;
            case 2:
                System.out.println("Two");
                break;
            default:
                System.out.println("Default");
        }
}
```

En la mayoría de los casos, la solución más limpia sería utilizar interfaces y mover el código con un comportamiento específico a implementaciones separadas (*composición sobre herencia*)

Si una declaración de cambio es inevitable, se recomienda documentar los avances "esperados" si se producen. De esa manera, le demuestra a los demás desarrolladores que está al tanto del salto faltante y que este es el comportamiento esperado.

```
switch(caseIndex) {
    [...]
    case 2:
        System.out.println("Two");
        // fallthrough
    default:
        System.out.println("Default");
```

Pitfall - punto y coma mal colocados y llaves faltantes

Este es un error que causa confusión real para los principiantes de Java, al menos la primera vez que lo hacen. En lugar de escribir esto:

```
if (feeling == HAPPY)
    System.out.println("Smile");
else
    System.out.println("Frown");
```

accidentalmente escriben esto:

```
if (feeling == HAPPY);
    System.out.println("Smile");
else
    System.out.println("Frown");
```

y se desconciertan cuando el compilador de Java les dice que la `else` está fuera de lugar. El compilador de Java interpreta lo anterior de la siguiente manera:

```
if (feeling == HAPPY)
    /*empty statement*/ ;
System.out.println("Smile");   // This is unconditional
else                           // This is misplaced.  A statement cannot
                               // start with 'else'
System.out.println("Frown");
```

En otros casos, no habrá errores de compilación, pero el código no hará lo que pretende el programador. Por ejemplo:

```
for (int i = 0; i < 5; i++);
    System.out.println("Hello");
```

sólo imprime "Hola" una vez. Una vez más, el punto y coma falso significa que el cuerpo del bucle `for` es una declaración vacía. Eso significa que la llamada `println` que sigue es incondicional.

Otra variación:

```
for (int i = 0; i < 5; i++);
    System.out.println("The number is " + i);
```

Esto dará un error de "No se puede encontrar el símbolo" para `i` . La presencia del punto y coma espurio significa que la llamada de `println` está intentando usar `i` fuera de su alcance.

En esos ejemplos, hay una solución directa: simplemente elimine el punto y coma no esencial. Sin embargo, hay algunas lecciones más profundas que se pueden extraer de estos ejemplos:

1. El punto y coma en Java no es "ruido sintáctico". La presencia o ausencia de un punto y coma puede cambiar el significado de su programa. No solo los agregue al final de cada línea.

2. No confíes en la sangría de tu código. En el lenguaje Java, el compilador ignora los espacios en blanco adicionales al principio de una línea.

3. Utilice un indentador automático. Todos los IDE y muchos editores de texto simples entienden cómo sangrar correctamente el código Java.

4. Esta es la lección más importante. Siga las últimas pautas de estilo de Java y ponga llaves alrededor de las declaraciones "then" y "else" y la declaración del cuerpo de un bucle. La abrazadera abierta (`{`) no debe estar en una nueva línea.

Si el programador siguiera las reglas de estilo, el ejemplo `if` con puntos y coma fuera de lugar se vería así:

```
if (feeling == HAPPY); {
    System.out.println("Smile");
} else {
```

```
    System.out.println("Frown");
}
```

Eso se ve extraño para un ojo experimentado. Si autodentara ese código, probablemente se vería así:

```
if (feeling == HAPPY); {
                           System.out.println("Smile");
                       } else {
                           System.out.println("Frown");
                       }
```

Lo que debería destacarse como mal incluso para un principiante.

Trampa: omitir llaves: los problemas de "colgar si" y "colgar de otra manera"

La última versión de la guía de estilo Java de Oracle exige que las declaraciones "entonces" y "más" en una instrucción `if` siempre deben estar entre "llaves" o "llaves". Reglas similares se aplican a los cuerpos de varias declaraciones de bucle.

```
if (a) {            // <- open brace
    doSomething();
    doSomeMore();
}                   // <- close brace
```

Esto no es realmente requerido por la sintaxis del lenguaje Java. De hecho, si la parte "entonces" de una declaración `if` es una sola declaración, es legal omitir las llaves

```
if (a)
    doSomething();
```

o incluso

```
if (a) doSomething();
```

Sin embargo, hay peligros en ignorar las reglas de estilo de Java y omitir las llaves. Específicamente, aumenta significativamente el riesgo de que el código con sangría defectuosa se lea mal.

El problema "que cuelga si":

Considere el código de ejemplo de arriba, reescrito sin llaves.

```
if (a)
   doSomething();
   doSomeMore();
```

Este código *parece decir* que las llamadas a `doSomething` y `doSomeMore` se producen tanto *si y sólo si* `a` es `true` . De hecho, el código está sangrado incorrectamente. La especificación del lenguaje

Java que la llamada `doSomeMore()` es una declaración separada que sigue a la instrucción `if` . La sangría correcta es la siguiente:

```
if (a)
   doSomething();
doSomeMore();
```

El problema de "colgar más"

Un segundo problema aparece cuando agregamos `else` a la mezcla. Considere el siguiente ejemplo con llaves faltantes.

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
else
   doZ();
```

El código anterior *parece* `doZ` que `doZ` se llamará cuando `a` sea `false` . De hecho, la sangría es incorrecta una vez más. La sangría correcta para el código es:

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
   else
      doZ();
```

Si el código se escribiera de acuerdo con las reglas de estilo de Java, en realidad se vería así:

```
if (a) {
   if (b) {
      doX();
   } else if (c) {
      doY();
   } else {
      doZ();
   }
}
```

Para ilustrar por qué es mejor eso, suponga que accidentalmente ha malgastado el código. Podría terminar con algo como esto:

```
if (a) {
   if (b) {
      doX();
   } else if (c) {
      doY();
} else {
   doZ();
}
```

```
if (a) {
    if (b) {
       doX();
    } else if (c) {
       doY();
} else {
    doZ();
    }
```

```
}                                   }
```

Pero en ambos casos, el código mal escrito "se ve mal" a los ojos de un programador Java experimentado.

Trampa: sobrecargar en lugar de anular

Considere el siguiente ejemplo:

```
public final class Person {
    private final String firstName;
    private final String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = (firstName == null) ? "" : firstName;
        this.lastName = (lastName == null) ? "" : lastName;
    }

    public boolean equals(String other) {
        if (!(other instanceof Person)) {
            return false;
        }
        Person p = (Person) other;
        return firstName.equals(p.firstName) &&
                lastName.equals(p.lastName);
    }

    public int hashcode() {
        return firstName.hashCode() + 31 * lastName.hashCode();
    }
}
```

Este código no va a comportarse como se espera. El problema es que los métodos `equals` y `hashcode` para `Person` no anulan los métodos estándar definidos por `Object` .

- El método `equals` tiene la firma incorrecta. Se debe declarar como `equals(Object)` no `equals(String)` .
- El método `hashcode` tiene el nombre equivocado. Debe ser `hashCode()` (tenga en cuenta la **C** mayúscula).

Estos errores significan que hemos declarado sobrecargas accidentales, y no se utilizarán si `Person` se usa en un contexto polimórfico.

Sin embargo, hay una manera simple de lidiar con esto (desde Java 5 en adelante). Use la anotación `@Override` siempre que *pretenda que* su método sea una anulación:

Java SE 5

```
public final class Person {
    ...

    @Override
    public boolean equals(String other) {
        ....
```

```
    }

    @Override
    public hashcode() {
        ....
    }
}
```

Cuando añadimos un `@Override` anotación a una declaración de método, el compilador comprobará que el método *no* anula (o implementar) un método declarado en una superclase o interfaz. Entonces, en el ejemplo anterior, el compilador nos dará dos errores de compilación, que deberían ser suficientes para alertarnos sobre el error.

Pitfall - Octales literales

Considere el siguiente fragmento de código:

```
// Print the sum of the numbers 1 to 10
int count = 0;
for (int i = 1; i < 010; i++) {    // Mistake here ....
    count = count + i;
}
System.out.println("The sum of 1 to 10 is " + count);
```

Un principiante de Java podría sorprenderse al saber que el programa anterior imprime la respuesta incorrecta. En realidad imprime la suma de los números del 1 al 8.

El motivo es que el compilador de Java interpreta un literal entero que comienza con el dígito cero ('0') como un literal octal, no un literal decimal, como cabría esperar. Por lo tanto, `010` es el número octal 10, que es 8 en decimal.

Pitfall - Declarar clases con los mismos nombres que las clases estándar

A veces, los programadores que son nuevos en Java cometen el error de definir una clase con un nombre que es el mismo que una clase muy utilizada. Por ejemplo:

```
package com.example;

/**
 * My string utilities
 */
public class String {
    ....
}
```

Entonces se preguntan por qué reciben errores inesperados. Por ejemplo:

```
package com.example;

public class Test {
    public static void main(String[] args) {
        System.out.println("Hello world!");
```

```
    }
}
```

Si compilas y luego intentas ejecutar las clases anteriores, obtendrás un error:

```
$ javac com/example/*.java
$ java com.example.Test
Error: Main method not found in class test.Test, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Alguien que mire el código de la clase `Test` vería la declaración de `main` y miraría su firma y se preguntaría de qué se está quejando el comando `java` . Pero, de hecho, el comando `java` está diciendo la verdad.

Cuando declaramos una versión de `String` en el mismo paquete que `Test` , esta versión tiene prioridad sobre la importación automática de `java.lang.String` . Por lo tanto, la firma del método `Test.main` es en realidad

```
void main(com.example.String[] args)
```

en lugar de

```
void main(java.lang.String[] args)
```

y el `java` comando no reconocerá *que* como un método de punto de entrada.

Lección: no defina clases que tengan el mismo nombre que las clases existentes en `java.lang` u otras clases comúnmente utilizadas en la biblioteca de Java SE. Si haces eso, te estás abriendo para todo tipo de errores oscuros.

Pitfall - Usar '==' para probar un booleano

A veces, un nuevo programador de Java escribirá código así:

```
public void check(boolean ok) {
    if (ok == true) {           // Note 'ok == true'
        System.out.println("It is OK");
    }
}
```

Un programador experimentado lo consideraría torpe y querría reescribirlo como:

```
public void check(boolean ok) {
    if (ok) {
       System.out.println("It is OK");
    }
}
```

Sin embargo, hay más errores con `ok == true` que simple torpeza. Considera esta variación:

```
public void check(boolean ok) {
    if (ok = true) {            // Oooops!
        System.out.println("It is OK");
    }
}
```

Aquí el programador ha escrito mal `==` como `=` ... y ahora el código tiene un error sutil. La expresión `x = true` asigna incondicionalmente `true` a `x` y luego se evalúa como `true` . En otras palabras, el método de `check` ahora imprimirá "Está bien", sin importar cuál fue el parámetro.

La lección aquí es dejar el hábito de usar `== false` y `== true` . Además de ser detallados, hacen que su codificación sea más propensa a errores.

Nota: Una posible alternativa a `ok == true` que evita el escollo es usar las condiciones de Yoda ; es decir, ponga el literal en el lado izquierdo del operador relacional, como en `true == ok` . Esto funciona, pero la mayoría de los programadores probablemente estarían de acuerdo en que las condiciones de Yoda son extrañas. Ciertamente, `ok` (o `!ok`) es más conciso y más natural.

Pitfall: las importaciones de comodines pueden hacer que su código sea frágil

Considere el siguiente ejemplo parcial:

```
import com.example.somelib.*;
import com.acme.otherlib.*;

public class Test {
    private Context x = new Context();   // from com.example.somelib
    ...
}
```

Supongamos que la primera vez que desarrolló el código contra la versión 1.0 de `somelib` y la versión 1.0 de `otherlib` . Luego, en algún momento posterior, deberá actualizar sus dependencias a versiones posteriores, y decide utilizar la versión 2.0 de `otherlib` . Supongamos también que uno de los cambios que hicieron en `otherlib` entre 1.0 y 2.0 fue agregar una clase de `Context` .

Ahora, cuando recompiles la `Test` , obtendrás un error de compilación que te dice que el `Context` es una importación ambigua.

Si está familiarizado con el código base, esto probablemente sea solo un inconveniente menor. Si no es así, entonces tiene trabajo que hacer para solucionar este problema, aquí y potencialmente en otro lugar.

El problema aquí es la importación de comodines. Por un lado, el uso de comodines puede hacer que sus clases sean un poco más cortas. Por otra parte:

- Los cambios compatibles hacia arriba en otras partes de su base de código, bibliotecas estándar de Java o bibliotecas de terceros pueden provocar errores de compilación.

- La legibilidad sufre. A menos que esté utilizando un IDE, puede ser difícil determinar cuál de las importaciones de comodín está generando una clase con nombre.

La lección es que es una mala idea usar las importaciones de comodines en el código que debe ser de larga duración. Las importaciones específicas (sin comodines) no son mucho esfuerzo de mantener si usa un IDE, y el esfuerzo vale la pena.

Pitfall: Usar 'assert' para validar argumentos o entradas de usuario

Una pregunta que ocasionalmente en StackOverflow es si es apropiado usar `assert` para validar los argumentos proporcionados a un método, o incluso las entradas proporcionadas por el usuario.

La respuesta simple es que no es apropiado.

Mejores alternativas incluyen:

- Lanzar una excepción IllegalArgument utilizando un código personalizado.
- Usando los métodos de las condiciones `Preconditions` disponibles en la biblioteca de Google Guava.
- Usando los métodos `Validate` disponibles en la biblioteca Lang3 de Apache Commons.

Esto es lo que la Especificación del lenguaje Java (JLS 14.10, para Java 8) aconseja sobre este asunto:

> Normalmente, la verificación de afirmaciones se habilita durante el desarrollo y las pruebas del programa, y se deshabilita para la implementación, para mejorar el rendimiento.
>
> Debido a que las aserciones pueden estar deshabilitadas, los programas no deben asumir que las expresiones contenidas en las aserciones serán evaluadas. Por lo tanto, estas expresiones booleanas generalmente deben estar libres de efectos secundarios. La evaluación de tal expresión booleana no debería afectar ningún estado que sea visible después de que se complete la evaluación. No es ilegal que una expresión booleana contenida en una aserción tenga un efecto secundario, pero generalmente es inapropiado, ya que podría causar que el comportamiento del programa varíe dependiendo de si las aserciones estaban habilitadas o deshabilitadas.
>
> A la luz de esto, las aserciones no deben usarse para la verificación de argumentos en métodos públicos. La verificación de argumentos generalmente es parte del contrato de un método, y este contrato debe ser confirmado si las aserciones están habilitadas o deshabilitadas.
>
> Un problema secundario con el uso de aserciones para la verificación de argumentos es que los argumentos erróneos deberían resultar en una excepción de tiempo de ejecución apropiada (como `IllegalArgumentException` , `ArrayIndexOutOfBoundsException` o `NullPointerException`). Un fallo de aserción no arrojará una excepción apropiada. Nuevamente, no es ilegal usar aserciones para la verificación de argumentos en métodos públicos, pero en general es inapropiado. Se pretende que `AssertionError`

nunca se detecte, pero es posible hacerlo, por lo tanto, las reglas para las declaraciones de prueba deben tratar las aserciones que aparecen en un bloque de prueba de manera similar al tratamiento actual de las declaraciones de lanzamiento.

El escollo de los objetos nulos de autoenvasado en primitivos

```
public class Foobar {
    public static void main(String[] args) {

        // example:
        Boolean ignore = null;
        if (ignore == false) {
            System.out.println("Do not ignore!");
        }
    }
}
```

El escollo aquí es que `null` se compara con `false` . Como estamos comparando un `boolean` primitivo con un `Boolean` , Java intenta *desempaquetar* el `Object Boolean` en un equivalente primitivo, listo para la comparación. Sin embargo, dado que ese valor es `null` , se lanza una `NullPointerException` .

Java es incapaz de comparar tipos primitivos contra valores `null` , lo que provoca una `NullPointerException` en tiempo de ejecución. Considere el caso primitivo de la condición `false == null` ; esto generaría un error de *tiempo de compilación de* `incomparable types: int and <null>` .

Lea Errores de Java - sintaxis de lenguaje en línea:
https://riptutorial.com/es/java/topic/5382/errores-de-java---sintaxis-de-lenguaje

Capítulo 70: Errores de Java - Uso de excepciones

Introducción

Varios usos indebidos del lenguaje de programación Java pueden llevar a cabo un programa para generar resultados incorrectos a pesar de haber sido compilados correctamente. El propósito principal de este tema es enumerar las **dificultades** comunes relacionadas con el **manejo de excepciones** y proponer la forma correcta de evitar tales dificultades.

Examples

Pitfall - Ignorar o aplastar excepciones

Este ejemplo trata sobre ignorar deliberadamente o "aplastar" las excepciones. O para ser más precisos, se trata de cómo capturar y manejar una excepción de una manera que la ignora. Sin embargo, antes de describir cómo hacer esto, primero debemos señalar que las excepciones de aplastamiento generalmente no son la forma correcta de tratarlas.

Las excepciones generalmente se lanzan (por algo) para notificar a otras partes del programa que ha ocurrido algún evento significativo (es decir, "excepcional"). En general (aunque no siempre) una excepción significa que algo ha salido mal. Si codifica su programa para eliminar la excepción, existe la posibilidad de que el problema vuelva a aparecer en otra forma. Para empeorar las cosas, cuando aplastas la excepción, estás desechando la información en el objeto de excepción y su seguimiento de pila asociado. Es probable que sea más difícil averiguar cuál fue la fuente original del problema.

En la práctica, el aplastamiento de excepciones ocurre con frecuencia cuando utiliza la función de corrección automática de un IDE para "arreglar" un error de compilación causado por una excepción no controlada. Por ejemplo, puede ver código como este:

```
try {
    inputStream = new FileInputStream("someFile");
} catch (IOException e) {
    /* add exception handling code here */
}
```

Claramente, el programador ha aceptado la sugerencia del IDE para que desaparezca el error de compilación, pero la sugerencia fue inapropiada. (Si el archivo ha fallado, lo más probable es que el programa haga algo al respecto. Con la "corrección" anterior, es probable que el programa falle más adelante; por ejemplo, con una `NullPointerException` porque `inputStream` ahora es `null`).

Dicho esto, aquí hay un ejemplo de aplastar deliberadamente una excepción. (Para los fines del argumento, supongamos que hemos determinado que una interrupción al mostrar la autofoto es inofensiva). El comentario le dice al lector que eliminamos la excepción deliberadamente y por

qué lo hicimos.

```
try {
    selfie.show();
} catch (InterruptedException e) {
    // It doesn't matter if showing the selfie is interrupted.
}
```

Otra forma convencional de resaltar que estamos aplastando *deliberadamente* una excepción sin decir por qué es indicar esto con el nombre de la variable de excepción, como este:

```
try {
    selfie.show();
} catch (InterruptedException ignored) {  }
```

Algunos IDE (como IntelliJ IDEA) no mostrarán una advertencia sobre el bloque catch vacío si el nombre de la variable se establece en `ignored` .

Pitfall: captura de Throwable, Exception, Error o RuntimeException

Un patrón de pensamiento común que los programadores inexpertos Java es que las excepciones son "un problema" o "una carga" y la mejor manera de lidiar con esto es que coger todos [1] tan pronto como sea posible. Esto lleva a un código como este:

```
....
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (Exception ex) {
    System.out.println("Could not open file " + fileName);
}
```

El código anterior tiene un defecto significativo. La `catch` realidad va a atrapar más excepciones de las que espera el programador. Supongamos que el valor del nombre de `fileName` es `null` , debido a un error en otra parte de la aplicación. Esto hará que el constructor `FileInputStream` lance una `NullPointerException` . El controlador detectará esto e informará al usuario:

```
    Could not open file null
```

que es inútil y confuso. Peor aún, supongamos que fue el código de "procesar la entrada" el que lanzó la excepción inesperada (activada o desactivada). Ahora el usuario recibirá el mensaje engañoso para un problema que no se produjo al abrir el archivo y es posible que no esté relacionado con la E / S en absoluto.

La raíz del problema es que el programador ha codificado un controlador para `Exception` . Esto es casi siempre un error:

- La `Exception` captura capturará todas las excepciones marcadas, y la mayoría de las excepciones no marcadas también.
- La captura de `RuntimeException` detectará la mayoría de las excepciones no verificadas.

- El `Error` captura detectará las excepciones sin marcar que señalan errores internos de JVM. Estos errores generalmente no son recuperables y no deben ser detectados.
- La captura de `Throwable` capturará todas las excepciones posibles.

El problema con la captura de un conjunto demasiado amplio de excepciones es que el controlador generalmente no puede manejarlas todas adecuadamente. En el caso de la `Exception` etc., es difícil para el programador predecir lo que *podría* detectarse; es decir, qué esperar.

En general, la solución correcta es lidiar con las excepciones que *se* lanzan. Por ejemplo, puedes atraparlos y manejarlos in situ:

```
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (FileNotFoundException ex) {
    System.out.println("Could not open file " + fileName);
}
```

o puede declararlos como `thrown` por el método adjunto.

Hay muy pocas situaciones en las que la captura de `Exception` es apropiada. El único que surge comúnmente es algo como esto:

```
public static void main(String[] args) {
    try {
        // do stuff
    } catch (Exception ex) {
        System.err.println("Unfortunately an error has occurred. " +
                           "Please report this to X Y Z");
        // Write stacktrace to a log file.
        System.exit(1);
    }
}
```

Aquí realmente queremos lidiar con todas las excepciones, por lo que capturar `Exception` (o incluso `Throwable`) es correcto.

---

1 - También conocido como Pokemon Exception Handling .

Pitfall - Lanzar Throwable, Exception, Error o RuntimeException

Si bien la captura de las `Exception` `Throwable` , `Exception` , `Error` y `RuntimeException` es mala, lanzarlas es aún peor.

El problema básico es que cuando su aplicación necesita manejar excepciones, la presencia de las excepciones de nivel superior hace que sea difícil discriminar entre diferentes condiciones de error. Por ejemplo

```
try {
    InputStream is = new FileInputStream(someFile);  // could throw IOException
```

```
    ...
    if (somethingBad) {
        throw new Exception();  // WRONG
    }
} catch (IOException ex) {
    System.err.println("cannot open ...");
} catch (Exception ex) {
    System.err.println("something bad happened");  // WRONG
}
```

El problema es que debido a que lanzamos una instancia de `Exception` , nos vemos obligados a atraparla. Sin embargo, como se describe en otro ejemplo, la captura de `Exception` es mala. En esta situación, se vuelve difícil discriminar entre el caso "esperado" de una `Exception` que se lanza si `somethingBad` es una `true` , y el caso inesperado en el que realmente detectamos una excepción no comprobada, como `NullPointerException` .

Si se permite que la excepción de nivel superior se propague, nos encontramos con otros problemas:

- Ahora tenemos que recordar todas las diferentes razones por las que lanzamos el nivel superior y discriminarlos / manejarlos.
- En el caso de `Exception` y `Throwable` , también necesitamos agregar estas excepciones a la cláusula de `throws` de métodos si queremos que la excepción se propague. Esto es problemático, como se describe a continuación.

En resumen, no tirar estas excepciones. Lanzar una excepción más específica que describa con más detalle el "evento excepcional" que ha ocurrido. Si lo necesita, defina y use una clase de excepción personalizada.

Declarar Throwable o Exception en los "lanzamientos" de un método es problemático.

Es tentador reemplazar una larga lista de excepciones lanzadas en la cláusula de `throws` un método con `Exception` o incluso `Throwable. Esta es una mala idea:

1. Obliga al llamante a manejar (o propagar) la `Exception` .
2. Ya no podemos confiar en el compilador para informarnos sobre excepciones comprobadas específicas que deben manejarse.
3. Manejar la `Exception` adecuadamente es difícil. Es difícil saber qué excepciones reales pueden detectarse, y si no sabe qué podría detectarse, es difícil saber qué estrategia de recuperación es adecuada.
4. El manejo de `Throwable` es aún más difícil, ya que ahora también tiene que hacer frente a fallas potenciales de las que nunca debería recuperarse.

Este consejo significa que se deben evitar ciertos otros patrones. Por ejemplo:

```
try {
    doSomething();
} catch (Exception ex) {
```

```
    report(ex);
    throw ex;
}
```

Lo anterior intenta registrar todas las excepciones a medida que pasan, sin manejarlas definitivamente. Desafortunadamente, antes de Java 7, el `throw ex;` La declaración hizo que el compilador pensara que se podía lanzar cualquier `Exception` . Eso podría forzarlo a declarar el método de cierre como `throws Exception` . Desde Java 7 en adelante, el compilador sabe que el conjunto de excepciones que podrían ser (relanzadas) es menor.

Pitfall - Catching InterruptedException

Como ya se señaló en otros escollos, capturar todas las excepciones usando

```
try {
    // Some code
} catch (Exception) {
    // Some error handling
}
```

Viene con muchos problemas diferentes. Pero un problema importante es que puede provocar interbloqueos ya que rompe el sistema de interrupción al escribir aplicaciones de subprocesos múltiples.

Si inicia un hilo, por lo general, también debe poder detenerlo bruscamente por varios motivos.

```
Thread t = new Thread(new Runnable() {
    public void run() {
         while (true) {
             //Do something indefinetely
         }
    }
}

t.start();

//Do something else

// The thread should be canceld if it is still active.
// A Better way to solve this is with a shared variable that is tested
// regularily by the thread for a clean exit, but for this example we try to
// forcibly interrupt this thread.
if (t.isAlive()) {
   t.interrupt();
   t.join();
}

//Continue with program
```

El `t.interrupt()` generará una InterruptedException en ese hilo, que está destinado a cerrar el hilo. Pero, ¿qué pasa si el subproceso necesita limpiar algunos recursos antes de que se detenga por completo? Para esto puede atrapar la excepción interrumpida y hacer algo de limpieza.

```
 Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                //Do something indefinetely
            }
        } catch (InterruptedException ex) {
            //Do some quick cleanup

            // In this case a simple return would do.
            // But if you are not 100% sure that the thread ends after
            // catching the InterruptedException you will need to raise another
            // one for the layers surrounding this code.
            Thread.currentThread().interrupt();
        }
    }
}
```

Pero si tiene una expresión de captura en su código, la InterruptedException también será detectada por ella y la interrupción no continuará. Lo que en este caso podría provocar un punto muerto, ya que el subproceso principal espera indefinidamente a que este thead se detenga con `t.join()` .

```
 Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                try {
                    //Do something indefinetely
                }
                catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        } catch (InterruptedException ex) {
            // Dead code as the interrupt exception was already caught in
            // the inner try-catch
            Thread.currentThread().interrupt();
        }
    }
}
```

Por lo tanto, es mejor capturar Excepciones individualmente, pero si insistes en usar un catch-all, al menos debes capturar la InterruptedException individualmente de antemano.

```
Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                try {
                    //Do something indefinetely
                } catch (InterruptedException ex) {
                    throw ex; //Send it up in the chain
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
```

```
        } catch (InterruptedException ex) {
            // Some quick cleanup code

            Thread.currentThread().interrupt();
        }
    }
}
```

Pitfall - Uso de excepciones para el control de flujo normal

Hay un mantra que algunos expertos de Java suelen recitar:

> "Las excepciones solo deben usarse en casos excepcionales".

(Por ejemplo: http://programmers.stackexchange.com/questions/184654)

La esencia de esto es que es una mala idea (en Java) usar excepciones y el manejo de excepciones para implementar el control de flujo normal. Por ejemplo, compare estas dos formas de tratar un parámetro que podría ser nulo.

```
public String truncateWordOrNull(String word, int maxLength) {
    if (word == null) {
        return "";
    } else {
        return word.substring(0, Math.min(word.length(), maxLength));
    }
}

public String truncateWordOrNull(String word, int maxLength) {
    try {
        return word.substring(0, Math.min(word.length(), maxLength));
    } catch (NullPointerException ex) {
        return "";
    }
}
```

En este ejemplo, estamos (por diseño) tratando el caso donde la `word` es `null` como si fuera una palabra vacía. Las dos versiones se ocupan de `null` ya sea utilizando convencional *si ... más* y / o *intentan ... atrapar* . ¿Cómo deberíamos decidir qué versión es mejor?

El primer criterio es la legibilidad. Si bien la legibilidad es difícil de cuantificar objetivamente, la mayoría de los programadores estarían de acuerdo en que el significado esencial de la primera versión es más fácil de discernir. De hecho, con el fin de entender realmente la segunda forma, es necesario entender que una `NullPointerException` no puede ser lanzada por los `Math.min` o `String.substring` métodos.

El segundo criterio es la eficiencia. En las versiones de Java anteriores a Java 8, la segunda versión es significativamente más lenta (órdenes de magnitud) que la primera versión. En particular, la construcción de un objeto de excepción implica la captura y el registro de los cuadros de pila, en caso de que se requiera el seguimiento de la pila.

Por otro lado, hay muchas situaciones donde el uso de excepciones es más legible, más eficiente

y (a veces) más correcto que usar el código condicional para tratar eventos "excepcionales". De hecho, hay situaciones raras en las que es necesario usarlas para eventos "no excepcionales"; Es decir, eventos que ocurren con relativa frecuencia. Para este último, vale la pena buscar formas de reducir los gastos generales de creación de objetos de excepción.

Pitfall - Stacktraces excesivos o inapropiados

Una de las cosas más molestas que pueden hacer los programadores es dispersar las llamadas a `printStackTrace()` largo de su código.

El problema es que `printStackTrace()` escribirá el seguimiento de pila en la salida estándar.

- Para una aplicación que está destinada a usuarios finales que no son programadores de Java, un seguimiento de pila no es informativo en el mejor de los casos, y alarmante en el peor.
- Para una aplicación del lado del servidor, es probable que nadie vea la salida estándar.

Una idea mejor es no llamar directamente a `printStackTrace` , o si lo llama, hágalo de manera que el seguimiento de la pila se escriba en un archivo de registro o de error en lugar de hacerlo en la consola del usuario final.

Una forma de hacer esto es usar un marco de registro y pasar el objeto de excepción como un parámetro del evento de registro. Sin embargo, incluso el registro de la excepción puede ser perjudicial si se realiza de forma poco prudente. Considera lo siguiente:

```
public void method1() throws SomeException {
    try {
        method2();
        // Do something
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method1", ex);
        throw ex;
    }
}

public void method2() throws SomeException {
    try {
        // Do something else
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method2", ex);
        throw ex;
    }
}
```

Si la excepción se produce en el `method2` , es probable que vea dos copias del mismo seguimiento de pila en el archivo de registro, que corresponde al mismo error.

En resumen, registre la excepción o vuelva a lanzarla más (posiblemente envuelta con otra excepción). No hagas ambas cosas.

Pitfall - Subclasificando directamente `Throwable`

`Throwable` tiene dos subclases directas, `Exception` y `Error` . Si bien es posible crear una nueva clase que extienda `Throwable` directamente, esto no es aconsejable ya que muchas aplicaciones suponen que solo existe `Exception` y `Error` .

Más aún, no hay ningún beneficio práctico en la subclasificación directa de `Throwable` , ya que la clase resultante es, en efecto, simplemente una excepción comprobada. La `Exception` subclases en cambio resultará en el mismo comportamiento, pero transmitirá más claramente su intención.

Lea Errores de Java - Uso de excepciones en línea:
https://riptutorial.com/es/java/topic/5381/errores-de-java---uso-de-excepciones

Capítulo 71: Escáner

Sintaxis

- Escáner escáner = nuevo escáner (fuente de origen);
- Escáner escáner = nuevo escáner (System.in);

Parámetros

Parámetro	Detalles
Fuente	La fuente puede ser una de String, File o cualquier tipo de InputStream

Observaciones

La clase `Scanner` se introdujo en Java 5. El método `reset()` se agregó en Java 6, y se agregaron un par de nuevos constructores en Java 7 para la interoperabilidad con la (entonces) nueva interfaz `Path` .

Examples

Lectura de la entrada del sistema utilizando el escáner

```
Scanner scanner = new Scanner(System.in); //Scanner obj to read System input
String inputTaken = new String();
while (true) {
    String input = scanner.nextLine(); // reading one line of input
    if (input.matches("\\s+"))         // if it matches spaces/tabs, stop reading
        break;
    inputTaken += input + " ";
}
System.out.println(inputTaken);
```

El objeto del escáner se inicializa para leer la entrada desde el teclado. Por lo tanto, para la siguiente entrada del teclado, producirá la salida como `Reading from keyboard`

```
Reading
from
keyboard
  //space
```

Lectura de entrada de archivo utilizando escáner

```
Scanner scanner = null;
try {
```

```
    scanner = new Scanner(new File("Names.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (Exception e) {
    System.err.println("Exception occurred!");
} finally {
    if (scanner != null)
        scanner.close();
}
```

Aquí se crea un objeto `Scanner` pasando un objeto `File` que contiene el nombre de un archivo de texto como entrada. Este archivo de texto se abrirá con el objeto Archivo y se leerá con el objeto del escáner en las siguientes líneas. `scanner.hasNext()` verificará si hay una próxima línea de datos en el archivo de texto. Combinar que con un `while` de bucle le permitirá iterar a través de cada línea de datos en el `Names.txt` archivo. Para recuperar los datos en sí, podemos usar métodos como `nextLine()` , `nextInt()` , `nextBoolean()` , etc. En el ejemplo anterior, se usa `scanner.nextLine()` . `nextLine()` hace referencia a la siguiente línea en un archivo de texto, y su combinación con un objeto de `scanner` permite imprimir el contenido de la línea. Para cerrar un objeto de escáner, usaría `.close()` .

Al usar try con recursos (desde Java 7 en adelante), el código mencionado anteriormente puede escribirse con elegancia de la siguiente manera.

```
try (Scanner scanner = new Scanner(new File("Names.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (Exception e) {
    System.err.println("Exception occurred!");
}
```

Lee toda la entrada como una cadena usando un escáner

Puede usar `Scanner` para leer todo el texto de la entrada como una Cadena, usando `\Z` (entrada completa) como delimitador. Por ejemplo, esto se puede usar para leer todo el texto en un archivo de texto en una línea:

```
String content = new Scanner(new File("filename")).useDelimiter("\\Z").next();
System.out.println(content);
```

Recuerde que tendrá que cerrar el Escáner, así como también capturar la `IoException` esto puede `IoException` , como se describe en el ejemplo Leyendo la entrada del archivo usando el Escáner .

Usando delimitadores personalizados

Puede usar delimitadores personalizados (expresiones regulares) con Scanner, con `.useDelimiter(",")` , para determinar cómo se lee la entrada. Esto funciona de manera similar a `String.split(...)` . Por ejemplo, puede usar el `Scanner` para leer de una lista de valores separados por comas en una cadena:

```
Scanner scanner = null;
try{
    scanner = new Scanner("i,like,unicorns").useDelimiter(",");;
    while(scanner.hasNext()){
        System.out.println(scanner.next());
    }
}catch(Exception e){
    e.printStackTrace();
}finally{
    if (scanner != null)
        scanner.close();
}
```

Esto le permitirá leer cada elemento en la entrada individualmente. Tenga en cuenta que **no** debe utilizar este para analizar los datos CSV, en cambio, utilizar una biblioteca adecuada analizador CSV, consulte CSV analizador para Java para otras posibilidades.

Patrón general que realiza las tareas más frecuentes.

Lo siguiente es cómo usar correctamente la clase `java.util.Scanner` para leer interactivamente la entrada del usuario de `System.in` correctamente (a veces se denomina `stdin` , especialmente en C, C ++ y otros idiomas, así como en Unix y Linux). Demuestra de manera idiomática las cosas más comunes que se requieren hacer.

```
package com.stackoverflow.scanner;

import javax.annotation.Nonnull;
import java.math.BigInteger;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.*;
import java.util.regex.Pattern;

import static java.lang.String.format;

public class ScannerExample
{
    private static final Set<String> EXIT_COMMANDS;
    private static final Set<String> HELP_COMMANDS;
    private static final Pattern DATE_PATTERN;
    private static final String HELP_MESSAGE;

    static
    {
        final SortedSet<String> ecmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        ecmds.addAll(Arrays.asList("exit", "done", "quit", "end", "fino"));
        EXIT_COMMANDS = Collections.unmodifiableSortedSet(ecmds);
        final SortedSet<String> hcmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        hcmds.addAll(Arrays.asList("help", "helpi", "?"));
        HELP_COMMANDS = Collections.unmodifiableSet(hcmds);
        DATE_PATTERN = Pattern.compile("\\d{4}([-\\/])\\d{2}\\1\\d{2}"); //
http://regex101.com/r/xB8dR3/1
        HELP_MESSAGE = format("Please enter some data or enter one of the following commands
to exit %s", EXIT_COMMANDS);
    }

    /**
```

```
     * Using exceptions to control execution flow is always bad.
     * That is why this is encapsulated in a method, this is done this
     * way specifically so as not to introduce any external libraries
     * so that this is a completely self contained example.
     * @param s possible url
     * @return true if s represents a valid url, false otherwise
     */
    private static boolean isValidURL(@Nonnull final String s)
    {
        try { new URL(s); return true; }
        catch (final MalformedURLException e) { return false; }
    }

    private static void output(@Nonnull final String format, @Nonnull final Object... args)
    {
        System.out.println(format(format, args));
    }

    public static void main(final String[] args)
    {
        final Scanner sis = new Scanner(System.in);
        output(HELP_MESSAGE);
        while (sis.hasNext())
        {
            if (sis.hasNextInt())
            {
                final int next = sis.nextInt();
                output("You entered an Integer = %d", next);
            }
            else if (sis.hasNextLong())
            {
                final long next = sis.nextLong();
                output("You entered a Long = %d", next);
            }
            else if (sis.hasNextDouble())
            {
                final double next = sis.nextDouble();
                output("You entered a Double = %f", next);
            }
            else if (sis.hasNext("\\d+"))
            {
                final BigInteger next = sis.nextBigInteger();
                output("You entered a BigInteger = %s", next);
            }
            else if (sis.hasNextBoolean())
            {
                final boolean next = sis.nextBoolean();
                output("You entered a Boolean representation = %s", next);
            }
            else if (sis.hasNext(DATE_PATTERN))
            {
                final String next = sis.next(DATE_PATTERN);
                output("You entered a Date representation = %s", next);
            }
            else // unclassified
            {
                final String next = sis.next();
                if (isValidURL(next))
                {
                    output("You entered a valid URL = %s", next);
                }
```

```
                else
                {
                    if (EXIT_COMMANDS.contains(next))
                    {
                        output("Exit command %s issued, exiting!", next);
                        break;
                    }
                    else if (HELP_COMMANDS.contains(next)) { output(HELP_MESSAGE); }
                    else { output("You entered an unclassified String = %s", next); }
                }
            }
        }
        /*
           This will close the underlying Readable, in this case System.in, and free those
resources.
           You will not be to read from System.in anymore after this you call .close().
           If you wanted to use System.in for something else, then don't close the Scanner.
        */
        sis.close();
        System.exit(0);
    }
}
```

Lee un int desde la línea de comando

```
import java.util.Scanner;

Scanner s = new Scanner(System.in);
int number = s.nextInt();
```

Si desea leer un int desde la línea de comandos, solo use este fragmento. En primer lugar, debe crear un objeto Escáner, que escucha System.in, que es la línea de comandos de forma predeterminada, cuando inicia el programa desde la línea de comandos. Después de eso, con la ayuda del objeto Escáner, lee el primer int que el usuario pasa a la línea de comando y lo almacena en el número de variable. Ahora puedes hacer lo que quieras con ese int almacenado.

Cerrar cuidadosamente un escáner

puede suceder que use un escáner con el System.in como parámetro para el constructor, entonces debe tener en cuenta que al cerrar el escáner se cerrará también el InputStream, lo que hará que cada vez que intente leer la entrada (o cualquier otro) objeto de escáner) lanzará una `java.util.NoSuchElementException` o una `java.lang.IllegalStateException`

ejemplo:

```
    Scanner sc1 = new Scanner(System.in);
    Scanner sc2 = new Scanner(System.in);
    int x1 = sc1.nextInt();
    sc1.close();
    // java.util.NoSuchElementException
    int x2 = sc2.nextInt();
    // java.lang.IllegalStateException
    x2 = sc1.nextInt();
```

Lea Escáner en línea: https://riptutorial.com/es/java/topic/551/escaner

Capítulo 72: Escogiendo Colecciones

Introducción

Java ofrece una amplia variedad de colecciones. Elegir qué colección usar puede ser complicado. Consulte la sección de ejemplos para ver un diagrama de flujo fácil de seguir para elegir la colección adecuada para el trabajo.

Examples

Diagrama de flujo de colecciones Java

Utilice el siguiente diagrama de flujo para elegir la Colección correcta para el trabajo.

Este diagrama de flujo se basó en [http://i.stack.imgur.com/aSDsG.png) .

Lea Escogiendo Colecciones en línea: https://riptutorial.com/es/java/topic/10846/escogiendo-colecciones

Capítulo 73: Escritor Buffered

Sintaxis

- nuevo BufferedWriter (Writer); // El constructor por defecto
- BufferedWriter.write (int c); // escribe un solo caracter
- BufferedWriter.write (String str); // escribe una cadena
- BufferedWriter.newLine (); // Escribe un separador de linea
- BufferedWriter.close (); // Cierra el BufferedWriter

Observaciones

- Si intenta escribir desde un `BufferedWriter` (utilizando `BufferedWriter.write()`) después de cerrar el `BufferedWriter` (utilizando `BufferedWriter.close()`), se emitirá una `IOException` .
- El constructor `BufferedWriter(Writer)` NO lanza una `IOException` . Sin embargo, el `FileWriter(File)` lanza una `FileNotFoundException` , que extiende la `IOException` . Por lo tanto, la captura de la `IOException` también detectará la `FileNotFoundException` , nunca se necesita una segunda instrucción catch a menos que planee hacer algo diferente con la `FileNotFoundException` .

Examples

Escribe una línea de texto a archivo

Este código escribe la cadena en un archivo. Es importante cerrar el escritor, así que esto se hace en un `finally` de bloque.

```
public void writeLineToFile(String str) throws IOException {
  File file = new File("file.txt");
  BufferedWriter bw = null;
  try {
    bw = new BufferedWriter(new FileWriter(file));
    bw.write(str);
  } finally {
    if (bw != null) {
      bw.close();
    }
  }
}
```

También tenga en cuenta que `write(String s)` no coloca el carácter de nueva línea después de que se haya escrito la cadena. Para ponerlo use el método `newLine()` .

Java SE 7

Java 7 agrega el paquete `java.nio.file` y try-with-resources :

```
public void writeLineToFile(String str) throws IOException {
    Path path = Paths.get("file.txt");
    try (BufferedWriter bw = Files.newBufferedWriter(path)) {
        bw.write(str);
    }
}
```

Lea Escritor Buffered en línea: https://riptutorial.com/es/java/topic/3063/escritor-buffered

Capítulo 74: Estructuras de control básicas

Observaciones

Todas las estructuras de control, a menos que se indique lo contrario, utilizan declaraciones de bloque . Estos se denotan mediante llaves `{}` .

Esto difiere de los enunciados normales , que *no* requieren llaves, pero también vienen con una advertencia rígida en que solo se considerará la línea que *sigue inmediatamente al* enunciado anterior.

Por lo tanto, es perfectamente válido escribir cualquiera de estas estructuras de control sin llaves, siempre y cuando solo *una* declaración siga el principio, pero está ***totalmente desaconsejada*** , ya que puede llevar a implementaciones defectuosas o códigos rotos.

Ejemplo:

```
// valid, but discouraged
Scanner scan = new Scanner(System.in);
int val = scan.nextInt();
if(val % 2 == 0)
    System.out.println("Val was even!");


// invalid; will not compile
// note the misleading indentation here
for(int i = 0; i < 10; i++)
    System.out.println(i);
    System.out.println("i is currently: " + i);
```

Examples

Si / Else Si / Else Control

```
if (i < 2) {
  System.out.println("i is less than 2");
} else if (i > 2) {
  System.out.println("i is more than 2");
} else {
  System.out.println("i is not less than 2, and not more than 2");
}
```

El bloque `if` solo se ejecutará cuando `i` sea 1 o menos.

La condición `else if` se comprueba solo si todas las condiciones anteriores (en las anteriores construcciones `else if` , y la primaria `if` construye) se han probado como `false` . En este ejemplo, la condición `else if` solo se verificará si `i` es mayor o igual a 2.

Si su resultado es `true` , su bloque se ejecuta, y cualquier `else if` y de lo `else` construye después

de que se omita.

Si ninguna de las condiciones `if` y `else if` han sido probadas como `true` , se ejecutará el bloque `else` al final.

Para loops

```
for (int i = 0; i < 100; i++) {
    System.out.println(i);
}
```

Los tres componentes del bucle `for` (separados por `;`) son la declaración / inicialización de la variable (aquí `int i = 0`), la condición (aquí `i < 100`) y la declaración de incremento (aquí `i++`). La declaración de variables se realiza una vez como si se colocara justo dentro de `{` en la primera ejecución. Luego se comprueba la condición, si es `true` el cuerpo del bucle se ejecutará, si es `false` el bucle se detendrá. Suponiendo que el bucle continúe, el cuerpo se ejecutará y, finalmente, cuando se alcance el `}` la instrucción de incremento se ejecutará justo antes de que se compruebe nuevamente la condición.

Las llaves son opcionales (puede una línea con un punto y coma) si el bucle contiene una sola instrucción. Pero, siempre se recomienda utilizar llaves para evitar malentendidos y errores.

Los componentes de bucle `for` son opcionales. Si su lógica de negocios contiene una de estas partes, puede omitir el componente correspondiente de su bucle `for` .

```
int i = obj.getLastestValue(); // i value is fetched from a method

for (; i < 100; i++) { // here initialization is not done
    System.out.println(i);
}
```

La estructura `for (;;) { function-body }` es igual a un bucle `while (true)` .

`Nested For Loops`

Cualquier instrucción de bucle que tenga otra instrucción de bucle dentro de un bucle anidado llamado. La misma forma para que el bucle tenga más bucle interno se llama 'anidado para bucle'.

```
    for(;;){
        //Outer Loop Statements
        for(;;){
            //Inner Loop Statements
        }
        //Outer Loop Statements
    }
```

Se puede demostrar que Nested for loop imprime números en forma de triángulo.

```
for(int i=9;i>0;i--){//Outer Loop
    System.out.println();
    for(int k=i;k>0;k--){//Inner Loop -1
```

```
        System.out.print(" ");
    }
    for(int j=i;j<=9;j++){//Inner Loop -2
        System.out.print(" "+j);
    }
 }
```

Mientras bucles

```
int i = 0;
while (i < 100) { // condition gets checked BEFORE the loop body executes
    System.out.println(i);
    i++;
}
```

A `while` bucle se ejecuta siempre que la condición dentro de los paréntesis es `true` . Esto también se denomina estructura de "bucle de prueba previa" porque la declaración condicional debe cumplirse antes de que se realice el cuerpo del bucle principal cada vez.

Las llaves son opcionales si el bucle contiene solo una declaración, pero algunas convenciones de estilo de codificación prefieren tener las llaves independientemente.

hacer ... mientras bucle

El bucle `do...while` while difiere de otros bucles en que se garantiza que se ejecutará **al menos una vez** . También se denomina estructura de "bucle posterior a la prueba" porque la declaración condicional se realiza después del cuerpo del bucle principal.

```
int i = 0;
do {
    i++;
    System.out.println(i);
} while (i < 100); // Condition gets checked AFTER the content of the loop executes.
```

En este ejemplo, el bucle se ejecutará hasta que se imprima el número `100` (aunque la condición sea `i < 100` y no `i <= 100`), porque la condición del bucle se evalúa *después de que se* ejecuta el bucle.

Con la garantía de al menos una ejecución, es posible declarar variables fuera del bucle e inicializarlas dentro.

```
String theWord;
Scanner scan = new Scanner(System.in);
do {
    theWord = scan.nextLine();
} while (!theWord.equals("Bird"));

System.out.println(theWord);
```

En este contexto, la `theWord` se define fuera del bucle, pero como se garantiza que tiene un valor basado en su flujo natural, la `theWord` se inicializará.

Para cada

Java SE 5

Con Java 5 y superior, se pueden usar bucles para cada uno, también conocidos como bucles for mejorados:

```
List strings = new ArrayList();

strings.add("This");
strings.add("is");
strings.add("a for-each loop");

for (String string : strings) {
    System.out.println(string);
}
```

Para cada bucle se puede usar para iterar sobre arreglos e implementaciones de la interfaz de `Iterable` , el último incluye clases de colecciones , como `List` o `Set` .

La variable de bucle puede ser de cualquier tipo que sea asignable desde el tipo de fuente.

La variable de bucle para un bucle mejorado para `Iterable<T>` o `T[]` puede ser de tipo `S` , si

- `T extends S`
- tanto `T` como `S` son tipos primitivos y se pueden asignar sin una conversión
- `S` es un tipo primitivo y `T` puede convertirse en un tipo asignable a `S` después de la conversión de unboxing.
- `T` es un tipo primitivo y se puede convertir a `S` mediante la conversión autoboxing.

Ejemplos:

```
T elements = ...
for (S s : elements) {
}
```

T	S	Compila
En t[]	largo	sí
largo[]	En t	no
`Iterable<Byte>`	largo	sí
`Iterable<String>`	CharSequence	sí
`Iterable<CharSequence>`	Cuerda	no
En t[]	Largo	no
En t[]	Entero	sí

Si / Else

```
int i = 2;
if (i < 2) {
  System.out.println("i is less than 2");
} else {
  System.out.println("i is greater than 2");
}
```

Una instrucción `if` ejecuta el código condicionalmente según el resultado de la condición entre paréntesis. Cuando la condición entre paréntesis es verdadera, entrará al bloque de la instrucción if que está definida por llaves como `{` y `}` . el soporte de apertura hasta el soporte de cierre es el alcance de la sentencia if.

El bloque `else` es opcional y puede omitirse. Se ejecuta si la instrucción `if` es `false` y no se ejecuta si la instrucción `if` es verdadera. En ese caso, `if` ejecuta la instrucción.

Ver también: ternario si

Cambiar la declaración

La instrucción `switch` es la declaración de rama de múltiples vías de Java. Se utiliza para reemplazar el largo de las cadenas `if` , `else if` , `else` , y hacerlas más legibles. Sin embargo, a diferencia de las afirmaciones `if` , uno no puede usar desigualdades; Cada valor debe ser definido concretamente.

Hay tres componentes críticos para la instrucción `switch` :

- `case` : Este es el valor que se evalúa para determinar la equivalencia con el argumento de la instrucción `switch` .
- `default` : esta es una expresión opcional, de alcance general, en caso de que ninguna de las declaraciones de `case` evalúe como `true` .
- Finalización abrupta de la declaración del `case` ; generalmente se `break` : esto es necesario para evitar la evaluación no deseada de declaraciones de `case` adicionales.

Con la excepción de `continue` , es posible usar cualquier declaración que cause la finalización brusca de una declaración . Esto incluye:

- `break`
- `return`
- `throw`

En el siguiente ejemplo, una declaración de `switch` típica se escribe con cuatro casos posibles, incluido el `default` .

```
Scanner scan = new Scanner(System.in);
int i = scan.nextInt();
switch (i) {
    case 0:
        System.out.println("i is zero");
        break;
```

```
    case 1:
        System.out.println("i is one");
        break;
    case 2:
        System.out.println("i is two");
        break;
    default:
        System.out.println("i is less than zero or greater than two");
}
```

Al omitir una `break` o cualquier declaración que pudiera completarse de manera abrupta, podemos aprovechar lo que se conoce como casos "directos", que evalúan varios valores. Esto se puede usar para crear rangos para que un valor sea exitoso, pero aún no es tan flexible como las desigualdades.

```
Scanner scan = new Scanner(System.in);
int foo = scan.nextInt();
switch(foo) {
    case 1:
        System.out.println("I'm equal or greater than one");
    case 2:
    case 3:
        System.out.println("I'm one, two, or three");
        break;
    default:
        System.out.println("I'm not either one, two, or three");
}
```

En caso de `foo == 1` la salida será:

```
I'm equal or greater than one
I'm one, two, or three
```

En caso de `foo == 3` la salida será:

```
I'm one, two, or three
```

Java SE 5

La instrucción de cambio también se puede utilizar con `enum` s.

```
enum Option {
    BLUE_PILL,
    RED_PILL
}

public void takeOne(Option option) {
    switch(option) {
        case BLUE_PILL:
            System.out.println("Story ends, wake up, believe whatever you want.");
            break;
        case RED_PILL:
            System.out.println("I show you how deep the rabbit hole goes.");
            break;
```

```
    }
}
```

Java SE 7

La instrucción `switch` también se puede usar con `String` s.

```
public void rhymingGame(String phrase) {
    switch (phrase) {
        case "apples and pears":
            System.out.println("Stairs");
            break;
        case "lorry":
            System.out.println("truck");
            break;
        default:
            System.out.println("Don't know any more");
    }
}
```

Operador ternario

A veces hay que verificar una condición y establecer el valor de una variable.

Por ej.

```
String name;

if (A > B) {
    name = "Billy";
} else {
    name = "Jimmy";
}
```

Esto se puede escribir fácilmente en una línea como

```
String name = A > B ? "Billy" : "Jimmy";
```

El valor de la variable se establece en el valor inmediatamente después de la condición, si la condición es verdadera. Si la condición es falsa, se le dará el segundo valor a la variable.

Descanso

La instrucción `break` termina un bucle (como `for` , `while`) o la evaluación de una instrucción switch .

Lazo:

```
while(true) {
   if(someCondition == 5) {
       break;
   }
```

```
}
```

El bucle en el ejemplo se ejecutaría para siempre. Pero cuando `someCondition` es igual a `5` en algún punto de ejecución, entonces el bucle termina.

Si varios bucles se conectan en cascada, solo la mayoría de los bucles internos terminan utilizando `break` .

Intenta ... Atrapa ... Finalmente

La estructura de control `try { ... } catch ( ... ) { ... }` se utiliza para manejar las excepciones .

```
String age_input = "abc";
try {
    int age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("You can vote!");
    } else {
        System.out.println("Sorry, you can't vote yet.");
    }
} catch (NumberFormatException ex) {
    System.err.println("Invalid input.  '" + age_input + "' is not a valid integer.");
}
```

Esto imprimiría:

> Entrada inválida. 'abc' no es un número entero válido.

Se puede añadir una cláusula `finally` después de la `catch` . La cláusula `finally` siempre se ejecutaría, independientemente de si se lanzó una excepción.

`try { ... } catch ( ... ) { ... } finally { ... }`

```
String age_input = "abc";
try {
    int age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("You can vote!");
    } else {
        System.out.println("Sorry, you can't vote yet.");
    }
} catch (NumberFormatException ex) {
    System.err.println("Invalid input.  '" + age_input + "' is not a valid integer.");
} finally {
    System.out.println("This code will always be run, even if an exception is thrown");
}
```

Esto imprimiría:

> Entrada inválida. 'abc' no es un número entero válido.
> Este código siempre se ejecutará, incluso si se lanza una excepción

Pausa anidada / continuar

Es posible `break` / `continue` a un bucle externo usando declaraciones de etiqueta:

```
outerloop:
for(...) {
    innerloop:
    for(...) {
        if(condition1)
            break outerloop;

        if(condition2)
            continue innerloop; // equivalent to: continue;
    }
}
```

No hay otro uso para las etiquetas en Java.

Continuar declaración en Java

La instrucción continue se usa para omitir los pasos restantes en la iteración actual y comenzar con la siguiente iteración de bucle. El control pasa de la instrucción de `continue` al valor de paso (incremento o decremento), si lo hay.

```
String[] programmers = {"Adrian", "Paul", "John", "Harry"};

    //john is not printed out
    for (String name : programmers) {
        if (name.equals("John"))
            continue;
        System.out.println(name);
    }
```

La instrucción de `continue` también puede hacer que el control del programa cambie al valor de paso (si lo hay) de un bucle con nombre:

```
Outer: // The name of the outermost loop is kept here as 'Outer'
for(int i = 0; i < 5; )
{
    for(int j = 0; j < 5; j++)
    {
        continue Outer;
    }
}
```

Lea Estructuras de control básicas en línea: https://riptutorial.com/es/java/topic/118/estructuras-de-control-basicas

Capítulo 75: Evaluación XML XPath

Observaciones

Las expresiones XPath se utilizan para navegar y seleccionar uno o más nodos dentro de un documento del árbol XML, como seleccionar un elemento o atributo determinado.

Consulte esta recomendación del W3C para obtener una referencia en este idioma.

Examples

Evaluando un NodeList en un documento XML

Dado el siguiente documento XML:

```
<documentation>
    <tags>
        <tag name="Java">
            <topic name="Regular expressions">
                <example>Matching groups</example>
                <example>Escaping metacharacters</example>
            </topic>
            <topic name="Arrays">
                <example>Looping over arrays</example>
                <example>Converting an array to a list</example>
            </topic>
        </tag>
        <tag name="Android">
            <topic name="Building Android projects">
                <example>Building an Android application using Gradle</example>
                <example>Building an Android application using Maven</example>
            </topic>
            <topic name="Layout resources">
                <example>Including layout resources</example>
                <example>Supporting multiple device screens</example>
            </topic>
        </tag>
    </tags>
</documentation>
```

Lo siguiente recupera todos los nodos de `example` para la etiqueta de Java (use este método si solo evalúa XPath en el XML una vez. Vea otro ejemplo para cuando se evalúan múltiples llamadas de XPath en el mismo archivo XML):

```
XPathFactory xPathFactory = XPathFactory.newInstance();
XPath xPath = xPathFactory.newXPath(); //Make new XPath
InputSource inputSource = new InputSource("path/to/xml.xml"); //Specify XML file path

NodeList javaExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Java']//example", inputSource,
XPathConstants.NODESET); //Evaluate the XPath
...
```

Analizar múltiples expresiones XPath en un solo XML

Usando el mismo ejemplo que **Evaluating a NodeList en un documento XML** , aquí es cómo podría hacer múltiples llamadas XPath de manera eficiente:

Dado el siguiente documento XML:

```
<documentation>
    <tags>
        <tag name="Java">
            <topic name="Regular expressions">
                <example>Matching groups</example>
                <example>Escaping metacharacters</example>
            </topic>
            <topic name="Arrays">
                <example>Looping over arrays</example>
                <example>Converting an array to a list</example>
            </topic>
        </tag>
        <tag name="Android">
            <topic name="Building Android projects">
                <example>Building an Android application using Gradle</example>
                <example>Building an Android application using Maven</example>
            </topic>
            <topic name="Layout resources">
                <example>Including layout resources</example>
                <example>Supporting multiple device screens</example>
            </topic>
        </tag>
    </tags>
</documentation>
```

Así es como usaría XPath para evaluar múltiples expresiones en un documento:

```
XPath xPath = XPathFactory.newInstance().newXPath(); //Make new XPath
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //Specify XML file path

NodeList javaExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Java']//example", doc, XPathConstants.NODESET);
//Evaluate the XPath
xPath.reset(); //Resets the xPath so it can be used again
NodeList androidExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Android']//example", doc,
XPathConstants.NODESET); //Evaluate the XPath

...
```

Analizar una sola expresión XPath varias veces en un XML

En este caso, desea tener la expresión compilada antes de las evaluaciones, para que cada llamada a `evaluate` no `compile` la misma expresión. La sintaxis simple sería:

```
XPath xPath = XPathFactory.newInstance().newXPath(); //Make new XPath
XPathExpression exp = xPath.compile("/documentation/tags/tag[@name='Java']//example");
```

```
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //Specify XML file path

NodeList javaExampleNodes = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //Evaluate
the XPath from the already-compiled expression

NodeList javaExampleNodes2 = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //Do it
again
```

En general, dos llamadas a `XPathExpression.evaluate()` serán mucho más eficientes que dos llamadas a `XPath.evaluate()` .

Lea Evaluación XML XPath en línea: https://riptutorial.com/es/java/topic/4148/evaluacion-xml-xpath

Capítulo 76: Examen de la unidad

Introducción

Las pruebas unitarias son una parte integral del desarrollo impulsado por pruebas, y una característica importante para construir cualquier aplicación robusta. En Java, las pruebas unitarias se realizan casi exclusivamente utilizando bibliotecas y marcos externos, la mayoría de los cuales tienen su propia etiqueta de documentación. Este talón sirve como medio para presentar al lector las herramientas disponibles y su documentación respectiva.

Observaciones

Marcos de prueba unitaria

Hay numerosos marcos disponibles para pruebas de unidad dentro de Java. La opción más popular por el momento es JUnit. Está documentado bajo lo siguiente:

JUIT

JUnit4 - *Etiqueta propuesta para las características de JUnit4; aún no implementado* .

Existen otros marcos de prueba de unidad, y tienen documentación disponible:

Prueba

Herramientas de prueba unitaria

Hay varias otras herramientas utilizadas para la prueba de la unidad:

Mockito - marco burlón ; Permite que los objetos sean imitados. Útil para imitar el comportamiento **esperado** de una unidad externa dentro de la prueba de una unidad dada, para no vincular el comportamiento de la unidad externa a las pruebas de la unidad dada.

JBehave - BDD Framework. Permite vincular las pruebas a los comportamientos de los usuarios (lo que permite la validación de requisitos / escenarios) *No hay etiquetas de documentos disponibles en el momento de la escritura; Aquí hay un enlace externo* .

Examples

¿Qué es la prueba unitaria?

Esto es un poco de una cartilla. En su mayoría se incluye porque la documentación se ve obligada a tener un ejemplo, incluso si está pensado como un artículo de código auxiliar. Si ya

conoce los conceptos básicos de la prueba de unidad, siéntase libre de saltar a los comentarios, donde se mencionan los marcos específicos.

La prueba de unidad es asegurar que un módulo dado se comporte como se espera. En aplicaciones a gran escala, garantizar la ejecución adecuada de los módulos en el vacío es una parte integral para garantizar la fidelidad de la aplicación.

Considere el siguiente pseudo-ejemplo (trivial):

```
public class Example {
  public static void main (String args[]) {
    new Example();
  }

  // Application-level test.
  public Example() {
    Consumer c = new Consumer();
    System.out.println("VALUE = " + c.getVal());
  }

  // Your Module.
  class Consumer {
    private Capitalizer c;

    public Consumer() {
      c = new Capitalizer();
    }

    public String getVal() {
      return c.getVal();
    }
  }

  // Another team's module.
  class Capitalizer {
    private DataReader dr;

    public Capitalizer() {
      dr = new DataReader();
    }

    public String getVal() {
      return dr.readVal().toUpperCase();
    }
  }

  // Another team's module.
  class DataReader {
    public String readVal() {
      // Refers to a file somewhere in your application deployment, or
      // perhaps retrieved over a deployment-specific network.
      File f;
      String s = "data";
      // ... Read data from f into s ...
      return s;
    }
  }
}
```

Entonces este ejemplo es trivial; `DataReader` obtiene los datos de un archivo, los pasa al `Capitalizer` , que convierte todos los caracteres a mayúsculas, que luego pasa al `Consumer` . Pero el `DataReader` está fuertemente vinculado a nuestro entorno de aplicación, por lo que aplazamos las pruebas de esta cadena hasta que estemos listos para implementar una versión de prueba.

Ahora, supongamos que en algún momento del lanzamiento de una versión, por razones desconocidas, el método `getVal()` en `Capitalizer` cambió de devolver una cadena `toUpperCase()` a una `toLowerCase()` :

```
// Another team's module.
class Capitalizer {
  ...

  public String getVal() {
    return dr.readVal().toLowerCase();
  }
}
```

Claramente, esto rompe el comportamiento esperado. Pero, debido a los arduos procesos involucrados en la ejecución del `DataReader` , no lo notaremos hasta nuestra próxima implementación de prueba. Entonces, los días / semanas / meses pasan con este error en nuestro sistema, y luego el gerente de producto ve esto y se dirige instantáneamente a usted, el líder del equipo asociado con el `Consumer` . "¿Por qué está sucediendo esto? ¿Qué cambiaron?" Obviamente, no tienes ni idea. No tienes idea de lo que está pasando. No cambiaste ningún código que debería estar tocando esto; ¿Por qué se rompe de repente?

Finalmente, después de la discusión entre los equipos y la colaboración, el problema se rastrea y el problema se resuelve. Pero, plantea la pregunta; ¿Cómo pudo haberse evitado esto?

Hay dos cosas obvias:

Las pruebas deben ser automatizadas

Nuestra confianza en las pruebas manuales hace que este error pase inadvertido durante demasiado tiempo. Necesitamos una forma de automatizar el proceso bajo el cual los errores se introducen **instantáneamente** . No dentro de 5 semanas. No dentro de 5 días No dentro de 5 minutos. Ahora mismo.

Hay que apreciar que, en este ejemplo, he expresado un error **muy trivial** que se introdujo y pasó desapercibido. En una aplicación industrial, con docenas de módulos que se actualizan constantemente, estos pueden arrastrarse por todas partes. Arregla algo con un módulo, solo para darte cuenta de que el mismo comportamiento que "arreglaste" se confió de alguna manera en otra parte (ya sea interna o externamente).

Sin una validación rigurosa, las cosas entrarán en el sistema. Es posible que, si se descuida lo suficiente, esto resulte en mucho trabajo adicional al tratar de arreglar los cambios (y luego corregir esos arreglos, etc.), que un producto **aumente** en el trabajo restante a medida que se pone esfuerzo en ello. No quieres estar en esta situación.

Las pruebas deben ser de grano fino

El segundo problema observado en nuestro ejemplo anterior es la cantidad de tiempo que llevó rastrear el error. El gerente de producto le hizo un ping cuando los evaluadores lo notaron, usted investigó y encontró que el `Capitalizer` estaba devolviendo datos aparentemente malos, hizo un ping al equipo de `Capitalizer` con sus hallazgos, investigaron, etc., etc., etc.

El mismo punto que mencioné anteriormente sobre la cantidad y la dificultad de este ejemplo trivial se sostiene aquí. Obviamente, cualquier persona razonablemente versada en Java podría encontrar el problema introducido rápidamente. Pero a menudo es mucho, mucho más difícil rastrear y comunicar problemas. Tal vez el equipo de `Capitalizer` proporcionó un JAR sin ninguna fuente. Tal vez estén ubicados en el otro lado del mundo, y las horas de comunicación son muy limitadas (tal vez a correos electrónicos que se envían una vez al día). Puede dar lugar a errores que tardan semanas o más en rastrearse (y, nuevamente, podría haber varios de estos para una versión determinada).

Para mitigar esto, queremos que las pruebas rigurosas se realicen en el nivel más **preciso** posible (también desea que las pruebas generales aseguren que los módulos interactúen correctamente, pero ese no es nuestro punto focal aquí). Queremos especificar rigurosamente cómo funciona toda la funcionalidad orientada hacia el exterior (como mínimo), y probar esa funcionalidad.

Entrar en la prueba de la unidad

Imagínese si tuviéramos una prueba, asegurando específicamente que el método `getVal()` de `Capitalizer` devolvió una cadena en mayúscula para una cadena de entrada determinada. Además, imagine que la prueba se ejecutó incluso antes de que cometiéramos ningún código. El error introducido en el sistema (es decir, `toUpperCase()` se reemplaza con `toLowerCase()`) no causaría problemas porque el error nunca **se** introduciría en el sistema. Lo atraparíamos en una prueba, el desarrollador (con suerte) se daría cuenta de su error, y se alcanzaría una solución alternativa en cuanto a cómo introducir el efecto deseado.

Aquí se hacen algunas omisiones en cuanto a **cómo** implementar estas pruebas, pero están cubiertas en la documentación específica del marco (enlazada en los comentarios). Con suerte, esto sirve como un ejemplo de **por qué las** pruebas unitarias son importantes.

Lea Examen de la unidad en línea: https://riptutorial.com/es/java/topic/8155/examen-de-la-unidad

Capítulo 77: Excepciones y manejo de excepciones.

Introducción

Los objetos del tipo `Throwable` y sus subtipos pueden enviarse a la pila con la palabra clave `throw` y capturados con `try…catch` declaraciones `try…catch` .

Sintaxis

- void someMethod () emite SomeException {} // declaración de método, obliga a los llamadores a capturar si SomeException es un tipo de excepción comprobada
- tratar {

  ```
  someMethod(); //code that might throw an exception
  ```

  }
- captura (SomeException e) {

  ```
   System.out.println("SomeException was thrown!"); //code that will run if certain
  exception (SomeException) is thrown
  ```

  }
- finalmente {

  ```
   //code that will always run, whether try block finishes or not
  ```

  }

Examples

Atrapando una excepción con try-catch

Se puede capturar y manejar una excepción utilizando la declaración `try...catch` . (De hecho, las declaraciones de `try` toman otras formas, como se describe en otros ejemplos sobre `try...catch...finally` y `try-with-resources`).

Prueba-captura con un bloque de captura

La forma más simple se ve así:

```
try {
    doSomething();
} catch (SomeException e) {
    handle(e);
}
// next statement
```

El comportamiento de un simple `try...catch` es el siguiente:

- Se ejecutan las sentencias en el bloque `try` .
- Si las declaraciones en el bloque `try` no generan ninguna excepción, entonces el control pasa a la siguiente instrucción después del `try...catch` .
- Si se lanza una excepción dentro del bloque `try` .
    - El objeto de excepción se prueba para ver si es una instancia de `SomeException` o un subtipo.
    - Si es así, entonces la `catch` bloque *detectar* la excepción:
        - La variable `e` está vinculada al objeto de excepción.
        - Se ejecuta el código dentro del bloque `catch` .
        - Si ese código lanza una excepción, entonces la excepción recién lanzada se propaga en lugar de la original.
        - De lo contrario, el control pasa a la siguiente instrucción después del `try...catch` .
    - Si no lo es, la excepción original continúa propagándose.

Prueba-captura con múltiples capturas

Un `try...catch` también puede tener varios bloques de `catch` . Por ejemplo:

```
try {
    doSomething();
} catch (SomeException e) {
    handleOneWay(e)
} catch (SomeOtherException e) {
    handleAnotherWay(e);
}
// next statement
```

Si hay varios bloques de `catch` , se intentan uno por uno comenzando con el primero, hasta que se encuentra una coincidencia para la excepción. El controlador correspondiente se ejecuta (como anteriormente) y luego el control se pasa a la siguiente instrucción después de la instrucción `try...catch` . Los bloques de `catch` posteriores a los que coinciden siempre se omiten, *incluso si el código del manejador lanza una excepción* .

La estrategia de coincidencia "de arriba abajo" tiene consecuencias para los casos en que las excepciones en los bloques `catch` no son desunidas. Por ejemplo:

```
try {
    throw new RuntimeException("test");
} catch (Exception e) {
    System.out.println("Exception");
```

```
} catch (RuntimeException e) {
    System.out.println("RuntimeException");
}
```

Este fragmento de código generará "Exception" en lugar de "RuntimeException". Dado que `RuntimeException` es un subtipo de `Exception` , la primera `catch` (más general) coincidirá. La segunda `catch` (más específica) nunca será ejecutada.

La lección para aprender de esto es que los bloques de `catch` más específicos (en términos de los tipos de excepción) deben aparecer primero, y los más generales deben ser los últimos. (Algunos compiladores de Java le avisarán si una `catch` nunca puede ejecutarse, pero esto no es un error de compilación).

Bloques de captura multi-excepción

Java SE 7

A partir de Java SE 7, un solo bloque `catch` puede manejar una lista de excepciones no relacionadas. El tipo de excepción se enumera, separado por un símbolo de barra vertical (`|`). Por ejemplo:

```
try {
    doSomething();
} catch (SomeException | SomeOtherException e) {
    handleSomeException(e);
}
```

El comportamiento de una captura de múltiples excepciones es una extensión simple para el caso de excepción única. La `catch` coincide si la excepción lanzada coincide (al menos) con una de las excepciones enumeradas.

Hay alguna sutileza adicional en la especificación. El tipo de `e` es una *unión* sintética de los tipos de excepción en la lista. Cuando se utiliza el valor de `e` , su tipo estático es el supertipo menos común de la unión de tipos. Sin embargo, si `e` es rethrown dentro del bloque `catch` , los tipos de excepción que se lanzan son los tipos en la unión. Por ejemplo:

```
public void method() throws IOException, SQLException
    try {
        doSomething();
    } catch (IOException | SQLException e) {
        report(e);
        throw e;
    }
```

En lo anterior, `IOException` y `SQLException` son excepciones revisadas cuyo supertipo menos común es `Exception` . Esto significa que el método de `report` debe coincidir con el `report(Exception)` . Sin embargo, el compilador sabe que el `throw` puede lanzar solamente una `IOException` o un `SQLException` . Por lo tanto, el `method` se puede declarar como `throws IOException, SQLException` lugar de `throws Exception` . (Lo que es bueno: vea Pitfall - Lanzar Throwable, Exception, Error o RuntimeException).

Lanzar una excepción

El siguiente ejemplo muestra los conceptos básicos de lanzar una excepción:

```
public void checkNumber(int number) throws IllegalArgumentException {
    if (number < 0) {
        throw new IllegalArgumentException("Number must be positive: " + number);
    }
}
```

La excepción es lanzada en la tercera línea. Esta declaración se puede dividir en dos partes:

- `new IllegalArgumentException(...)` crea una instancia de la clase `IllegalArgumentException` , con un mensaje que describe el error que informa la excepción.
- `throw ...` es lanzar el objeto de excepción.

Cuando se lanza la excepción, hace que las declaraciones adjuntas *terminen de forma anormal* hasta que se *maneje* la excepción. Esto se describe en otros ejemplos.

Es una buena práctica crear y lanzar el objeto de excepción en una sola declaración, como se muestra arriba. También es una buena práctica incluir un mensaje de error significativo en la excepción para ayudar al programador a comprender la causa del problema. Sin embargo, este no es necesariamente el mensaje que debe mostrar al usuario final. (Para empezar, Java no tiene soporte directo para internacionalizar mensajes de excepción).

Hay un par de puntos más por hacer:

- Hemos declarado el `checkNumber` como `throws IllegalArgumentException` . Esto no fue estrictamente necesario, ya que `IllegalArgumentException` es una excepción comprobada; vea La jerarquía de excepciones de Java - Excepciones no verificadas y revisadas . Sin embargo, es una buena práctica hacer esto, y también incluir las excepciones lanzadas por los comentarios javadoc de un método.
- El código inmediatamente después de una declaración de `throw` es *inalcanzable* . Por lo tanto si escribimos esto:

  ```
  throw new IllegalArgumentException("it is bad");
  return;
  ```

  el compilador informaría un error de compilación para la declaración de `return` .

Encadenamiento de excepciones

Muchas excepciones estándar tienen un constructor con un segundo argumento de `cause` además del argumento de `message` convencional. La `cause` permite encadenar excepciones. Aquí hay un ejemplo.

Primero definimos una excepción no verificada que nuestra aplicación va a lanzar cuando

encuentra un error no recuperable. Tenga en cuenta que hemos incluido un constructor que acepta un argumento de `cause` .

```
public class AppErrorException extends RuntimeException {
    public AppErrorException() {
        super();
    }

    public AppErrorException(String message) {
        super(message);
    }

    public AppErrorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

A continuación, aquí hay un código que ilustra el encadenamiento de excepciones.

```
public String readFirstLine(String file) throws AppErrorException {
    try (Reader r = new BufferedReader(new FileReader(file))) {
        String line = r.readLine();
        if (line != null) {
            return line;
        } else {
            throw new AppErrorException("File is empty: " + file);
        }
    } catch (IOException ex) {
        throw new AppErrorException("Cannot read file: " + file, ex);
    }
}
```

El `throw` dentro del bloque `try` detecta un problema y lo informa a través de una excepción con un mensaje simple. Por el contrario, el `throw` dentro del bloque `catch` está manejando la `IOException` envolviéndolo en una nueva excepción (marcada). Sin embargo, no está tirando la excepción original. Al pasar la `IOException` como `cause` , la grabamos para que pueda imprimirse en el seguimiento de pila, como se explica en Creación y lectura de los registros de pila .

Excepciones personalizadas

En la mayoría de los casos, es más simple desde el punto de vista de diseño de código usar clases de `Exception` genéricas existentes al lanzar excepciones. Esto es especialmente cierto si solo necesita la excepción para llevar un simple mensaje de error. En ese caso, generalmente se prefiere RuntimeException , ya que no es una excepción marcada. Existen otras clases de excepción para las clases comunes de errores:

- UnsupportedOperationException : una determinada operación no es compatible
- IllegalArgumentException : se pasó un valor de parámetro no válido a un método
- IllegalStateException : su API ha alcanzado internamente una condición que nunca debería ocurrir, o que se produce como resultado de usar su API de forma no válida

Casos en los que **usted** desee utilizar una clase de excepción personalizada incluyen los siguientes:

- Está escribiendo una API o biblioteca para que otros la utilicen, y quiere permitir que los usuarios de su API puedan capturar y manejar específicamente las excepciones de su API y *poder diferenciar esas excepciones de otras excepciones más genéricas* .
- Está lanzando excepciones para un **tipo específico de error** en una parte de su programa, que desea detectar y manejar en otra parte de su programa, y desea poder diferenciar estos errores de otros errores más genéricos.

Puede crear sus propias excepciones personalizadas extendiendo `RuntimeException` para una excepción no verificada, o excepción comprobada extendiendo cualquier `Exception` *que no sea también subclase de RuntimeException* , porque:

> Las subclases de Excepción que no son también subclases de RuntimeException son excepciones comprobadas

```
public class StringTooLongException extends RuntimeException {
    // Exceptions can have methods and fields like other classes
    // those can be useful to communicate information to pieces of code catching
    // such an exception
    public final String value;
    public final int maximumLength;

    public StringTooLongException(String value, int maximumLength){
        super(String.format("String exceeds maximum Length of %s: %s", maximumLength, value));
        this.value = value;
        this.maximumLength = maximumLength;
    }
}
```

Se pueden usar solo como excepciones predefinidas:

```
void validateString(String value){
    if (value.length() > 30){
        throw new StringTooLongException(value, 30);
    }
}
```

Y los campos se pueden usar donde se captura y maneja la excepción:

```
void anotherMethod(String value){
    try {
        validateString(value);
    } catch(StringTooLongException e){
        System.out.println("The string '" + e.value +
                "' was longer than the max of " + e.maximumLength );
    }
}
```

Tenga en cuenta que, según la documentación de Java de Oracle :

> [...] Si es razonable esperar que un cliente se recupere de una excepción, conviértalo en una excepción comprobada. Si un cliente no puede hacer nada para recuperarse de la excepción, conviértalo en una excepción sin marcar.

Más:

- ¿Por qué RuntimeException no requiere un manejo de excepciones explícito?

La declaración de prueba con recursos

Java SE 7

Como lo ilustra el ejemplo de la sentencia try-catch-final , la limpieza de recursos usando una cláusula `finally` requiere una cantidad significativa de código de "placa de caldera" para implementar los casos de borde correctamente. Java 7 proporciona una forma mucho más sencilla de resolver este problema en la forma de la declaración *try-with-resources* .

¿Qué es un recurso?

Java 7 introdujo la interfaz `java.lang.AutoCloseable` para permitir que las clases se administren usando la declaración *try-with-resources* . Las instancias de clases que implementan `AutoCloseable` se conocen como *recursos* . Por lo general, estos deben eliminarse de manera oportuna en lugar de confiar en el recolector de basura para eliminarlos.

`AutoCloseable` interfaz `AutoCloseable` define un solo método:

```
public void close() throws Exception
```

Un método `close()` debe disponer del recurso de una manera apropiada. La especificación establece que debería ser seguro llamar al método en un recurso que ya se ha eliminado. Además, se *recomienda encarecidamente* a las clases que implementan el `Autocloseable` que declaren el método `close()` para lanzar una excepción más específica que la `Exception` , o ninguna excepción en absoluto.

Una amplia gama de clases e interfaces Java estándar implementan `AutoCloseable` . Éstos incluyen:

- `InputStream` , `OutputStream` y sus subclases
- `Reader` , `Writer` y sus subclases.
- `Socket` y `ServerSocket` y sus subclases
- `Channel` y sus subclases, y
- Las interfaces JDBC `Connection` , `Statement` y `ResultSet` y sus subclases.

La aplicación y las clases de terceros pueden hacer esto también.

La declaración básica de prueba con recursos.

La sintaxis de un *try-with-resources* se basa en las formas clásicas *try-catch* , *try-finally* y *try-catch-finally* . Aquí hay un ejemplo de una forma "básica"; Es decir, la forma sin `catch` o `finally` .

```
try (PrintStream stream = new PrintStream("hello.txt")) {
```

```
    stream.println("Hello world!");
}
```

Los recursos a administrar se declaran como variables en la sección `(...)` después de la cláusula `try` . En el ejemplo anterior, declaramos un `stream` variable de recurso y lo inicializamos a un `PrintStream` recién creado.

Una vez que las variables de recursos se han inicializado, se ejecuta el bloque `try` . Cuando esto se complete, se `stream.close()` automáticamente para garantizar que el recurso no se escape. Tenga en cuenta que la llamada `close()` ocurre sin importar cómo se complete el bloque.

Las declaraciones mejoradas de prueba con recursos

La instrucción *try-with-resources* puede mejorarse con `catch` bloqueos `catch` y `finally` , como con la sintaxis *try-catch-finally* pre-Java 7. El siguiente fragmento de código agrega un bloque `catch` a nuestro anterior para tratar con la `FileNotFoundException` que el constructor `PrintStream` puede lanzar:

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
} catch (FileNotFoundException ex) {
    System.err.println("Cannot open the file");
} finally {
    System.err.println("All done");
}
```

Si la inicialización del recurso o el bloque try lanzan la excepción, entonces se ejecutará el bloque `catch` . El bloque `finally` siempre se ejecutará, como con una sentencia convencional *try-catch-finally* .

Hay un par de cosas a tener en cuenta, sin embargo:

- La variable de recurso está *fuera de alcance* en los bloqueos de `catch` y `finally` .
- La limpieza de recursos se realizará antes de que la declaración intente coincidir con el bloque `catch` .
- Si la limpieza automática de recursos arrojó una excepción, entonces *podría* quedar atrapado en uno de los bloques de `catch` .

Gestionando múltiples recursos

Los fragmentos de código anteriores muestran un solo recurso que se está administrando. De hecho, *try-with-resources* puede administrar múltiples recursos en una sola declaración. Por ejemplo:

```
try (InputStream is = new FileInputStream(file1);
     OutputStream os = new FileOutputStream(file2)) {
    // Copy 'is' to 'os'
}
```

Esto se comporta como cabría esperar. Ambos `is` y `os` se cierran automáticamente al final del bloque `try` . Hay un par de puntos a tener en cuenta:

- Las inicializaciones se producen en el orden del código, y los inicializadores de variables de recursos posteriores pueden utilizar los valores de los anteriores.
- Todas las variables de recursos que se inicializaron correctamente se limpiarán.
- Las variables de recursos se limpian en *orden inverso a* sus declaraciones.

Por lo tanto, en el ejemplo anterior, `is` inicializa antes del sistema `os` y se limpia después de él, y `is` limpiará si hay una excepción al inicializar el sistema `os` .

Equivalencia de try-with-resource y clásico try-catch-finally

La especificación del lenguaje Java especifica el comportamiento de los formularios de *prueba con recursos* en términos de la declaración clásica de *prueba-captura-* final. (Consulte los detalles completos en el JLS.)

Por ejemplo, este *intento* básico *con el recurso* :

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}
```

se define como equivalente a este *try-catch-finally* :

```
// Note that the constructor is not part of the try-catch statement
PrintStream stream = new PrintStream("hello.txt");

// This variable is used to keep track of the primary exception thrown
// in the try statement. If an exception is thrown in the try block,
// any exception thrown by AutoCloseable.close() will be suppressed.
Throwable primaryException = null;

// The actual try block
try {
    stream.println("Hello world!");
} catch (Throwable t) {
    // If an exception is thrown, remember it for the finally block
    primaryException = t;
    throw t;
} finally {
    if (primaryException == null) {
        // If no exception was thrown so far, exceptions thrown in close() will
        // not be caught and therefore be passed on to the enclosing code.
        stream.close();
    } else {
        // If an exception has already been thrown, any exception thrown in
        // close() will be suppressed as it is likely to be related to the
        // previous exception. The suppressed exception can be retrieved
        // using primaryException.getSuppressed().
        try {
            stream.close();
        } catch (Throwable suppressedException) {
            primaryException.addSuppressed(suppressedException);
```

```
        }
    }
}
```

(El JLS especifica que las variables `t` y `primaryException` reales serán invisibles para el código Java normal).

La forma mejorada de *try-with-resources* se especifica como una equivalencia con la forma básica. Por ejemplo:

```
try (PrintStream stream = new PrintStream(fileName)) {
    stream.println("Hello world!");
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}
```

es equivalente a:

```
try {
    try (PrintStream stream = new PrintStream(fileName)) {
        stream.println("Hello world!");
    }
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}
```

Creación y lectura de stacktraces.

Cuando se crea un objeto de excepción (es decir, cuando se crea una `new`), el constructor `Throwable` captura información sobre el contexto en el que se creó la excepción. Más adelante, esta información se puede generar en forma de stacktrace, que puede usarse para ayudar a diagnosticar el problema que causó la excepción en primer lugar.

Imprimiendo un stacktrace

Imprimir un stacktrace es simplemente una cuestión de llamar al método `printStackTrace()` . Por ejemplo:

```
try {
    int a = 0;
    int b = 0;
    int c = a / b;
} catch (ArithmeticException ex) {
    // This prints the stacktrace to standard output
    ex.printStackTrace();
}
```

El método `printStackTrace()` sin argumentos se imprimirá en la salida estándar de la aplicación;

Es decir, el actual `System.out` . También hay `printStackTrace(PrintStream)` e `printStackTrace(PrintWriter)` que se imprimen en un `Stream` o `Writer` específico.

Notas:

1. El stacktrace no incluye los detalles de la excepción en sí. Puedes usar el método `toString()` para obtener esos detalles; p.ej

```
// Print exception and stacktrace
System.out.println(ex);
ex.printStackTrace();
```

2. La impresión Stacktrace debe usarse con moderación; ver Escapada - Stacktraces excesivos o inapropiados . A menudo es mejor usar un marco de registro y pasar el objeto de excepción que se va a registrar.

Entendiendo un stacktrace

Considere el siguiente programa simple que consiste en dos clases en dos archivos. (Hemos mostrado los nombres de archivo y los números de línea agregados con fines ilustrativos).

```
File: "Main.java"
1   public class Main {
2       public static void main(String[] args) {
3           new Test().foo();
4       }
5   }

File: "Test.java"
1   class Test {
2       public void foo() {
3           bar();
4       }
5
6       public int bar() {
7           int a = 1;
8           int b = 0;
9           return a / b;
10      }
```

Cuando estos archivos se compilan y ejecutan, obtendremos el siguiente resultado.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.bar(Test.java:9)
        at Test.foo(Test.java:3)
        at Main.main(Main.java:3)
```

Vamos a leer esta línea a la vez para averiguar lo que nos está diciendo.

La línea # 1 nos dice que el hilo llamado "main" ha terminado debido a una excepción no detectada. El nombre completo de la excepción es `java.lang.ArithmeticException` , y el mensaje de excepción es "/ by zero".

Si buscamos los javadocs para esta excepción, dice:

> Se lanza cuando ha ocurrido una condición aritmética excepcional. Por ejemplo, un entero "dividir por cero" lanza una instancia de esta clase.

De hecho, el mensaje "/ por cero" es un fuerte indicio de que la causa de la excepción es que algún código ha intentado dividir algo entre cero. ¿Pero que?

Las 3 líneas restantes son la traza de pila. Cada línea representa una llamada de método (o constructor) en la pila de llamadas, y cada una nos dice tres cosas:

- el nombre de la clase y el método que se estaba ejecutando,
- el nombre del archivo de código fuente,
- el número de línea del código fuente de la sentencia que se estaba ejecutando

Estas líneas de un seguimiento de pila se enumeran con el marco para la llamada actual en la parte superior. El cuadro superior en nuestro ejemplo anterior está en el método `Test.bar` y en la línea 9 del archivo Test.java. Esa es la siguiente línea:

```
    return a / b;
```

Si observamos un par de líneas en el archivo donde se inicializa `b` , es evidente que `b` tendrá el valor cero. Podemos decir sin ninguna duda que esta es la causa de la excepción.

Si quisiéramos ir más lejos, podemos ver que desde el StackTrace `bar()` fue llamado desde `foo()` en la línea 3 de Test.java, y que `foo()` a su vez fue llamado desde `Main.main()` .

Nota: Los nombres de clase y método en los marcos de pila son los nombres internos de las clases y métodos. Tendrá que reconocer los siguientes casos inusuales:

- Una clase anidada o interna se verá como "OuterClass $ InnerClass".
- Una clase interna anónima se verá como "Clase externa $ 1", "Clase externa $ 2", etc.
- Cuando se ejecuta el código en un constructor, el inicializador de campo de instancia o un bloque de inicializador de instancia, el nombre del método será "".
- Cuando se ejecuta el código en un inicializador de campo estático o en un bloque de inicializador estático, el nombre del método será "".

(En algunas versiones de Java, el código de formato stacktrace detectará y eliminará las secuencias repetidas de cuadros de pila, como puede ocurrir cuando una aplicación falla debido a una recursión excesiva).

Excepción de encadenamiento y stacktraces anidados.

Java SE 1.4

El encadenamiento de excepciones ocurre cuando un fragmento de código atrapa una excepción, y luego crea y lanza uno nuevo, pasando la primera excepción como la causa. Aquí hay un ejemplo:

```
File: Test,java
1   public class Test {
2      int foo() {
3           return 0 / 0;
4      }
5
6       public Test() {
7           try {
8               foo();
9           } catch (ArithmeticException ex) {
10              throw new RuntimeException("A bad thing happened", ex);
11          }
12      }
13
14      public static void main(String[] args) {
15          new Test();
16      }
17  }
```

Cuando la clase anterior se compila y ejecuta, obtenemos el siguiente seguimiento de pila:

```
Exception in thread "main" java.lang.RuntimeException: A bad thing happened
        at Test.<init>(Test.java:10)
        at Test.main(Test.java:15)
Caused by: java.lang.ArithmeticException: / by zero
        at Test.foo(Test.java:3)
        at Test.<init>(Test.java:8)
        ... 1 more
```

El stacktrace comienza con el nombre de la clase, el método y la pila de llamadas para la excepción que (en este caso) causó que la aplicación fallara. A esto le sigue una línea "Causado por:" que informa la excepción de `cause` . Se informa el nombre de la clase y el mensaje, seguidos de los marcos de pila de la excepción de causa. La traza finaliza con un "... N más" que indica que los últimos N cuadros son los mismos que para la excepción anterior.

El "Causado por:" solo se incluye en la salida cuando la `cause` la excepción principal no es `null`). Las excepciones se pueden encadenar indefinidamente, y en ese caso, el seguimiento de pila puede tener múltiples trazas de "Causado por:".

Nota: el mecanismo de `cause` solo se expuso en la API de `Throwable` en Java 1.4.0. Antes de eso, era necesario que la aplicación implementara el encadenamiento de excepciones utilizando un campo de excepción personalizado para representar la causa y un método personalizado de `printStackTrace` .

Capturando un stacktrace como una cadena

A veces, una aplicación necesita poder capturar un stacktrace como una `String` Java, para que pueda ser usada para otros propósitos. El enfoque general para hacer esto es crear un `OutputStream` o `Writer` temporal que escribe en un búfer en memoria y se lo pasa a `printStackTrace(...)` .

Las bibliotecas Apache Commons y Guava proporcionan métodos de utilidad para capturar un

stacktrace como una cadena:

```
org.apache.commons.lang.exception.ExceptionUtils.getStackTrace(Throwable)

com.google.common.base.Throwables.getStackTraceAsString(Throwable)
```

Si no puede usar bibliotecas de terceros en su base de código, entonces el siguiente método hace la tarea:

```
   /**
     * Returns the string representation of the stack trace.
     *
     * @param throwable the throwable
     * @return the string.
     */
    public static String stackTraceToString(Throwable throwable) {
        StringWriter stringWriter = new StringWriter();
        throwable.printStackTrace(new PrintWriter(stringWriter));
        return stringWriter.toString();
    }
```

Tenga en cuenta que si su intención es analizar el seguimiento de pila, es más sencillo utilizar `getStackTrace()` y `getCause()` que intentar analizar un seguimiento de pila.

Manipulación InterruptedException

`InterruptedException` es una bestia confusa: aparece en métodos aparentemente inocuos como `Thread.sleep()` , pero su manejo incorrecto conduce a un código difícil de administrar que se comporta mal en entornos concurrentes.

En su forma más básica, si se `Thread.interrupt()` una `InterruptedException` , significa alguien, en algún lugar, llamado `Thread.interrupt()` en el hilo en el que se está ejecutando su código. Es posible que esté inclinado a decir "¡Es mi código! ¡Nunca lo interrumpiré! " y por lo tanto hacer algo como esto:

```
// Bad. Don't do this.
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  // disregard
}
```

Pero esta es exactamente la manera incorrecta de manejar un evento "imposible" que ocurre. Si sabe que su aplicación nunca encontrará una `InterruptedException` , debe tratar dicho evento como una violación grave de las suposiciones de su programa y salir lo más rápido posible.

La forma correcta de manejar una interrupción "imposible" es así:

```
// When nothing will interrupt your code
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
```

```
  Thread.currentThread().interrupt();
  throw new AssertionError(e);
}
```

Esto hace dos cosas; primero restaura el estado de interrupción del hilo (como si la `InterruptedException` no se hubiera lanzado en primer lugar), y luego lanza un `AssertionError` indica que se han violado los invariantes básicos de su aplicación. Si sabe con certeza que nunca interrumpirá el hilo en el que se ejecuta este código es seguro, ya que nunca debe `catch` bloque `catch` .

El uso de la clase `Uninterruptibles` de Guava ayuda a simplificar este patrón; llamando a `Uninterruptibles.sleepUninterruptibly()` ignora el estado interrumpido de un hilo hasta que la duración del tiempo de espera (en ese momento se restaura para las llamadas posteriores para inspeccionar y lanzar su propia `InterruptedException`). Si sabe que nunca interrumpirá dicho código, esto evita con seguridad la necesidad de ajustar sus llamadas de reposo en un bloque try-catch.

Más a menudo, sin embargo, no puede garantizar que su hilo nunca será interrumpido. En particular, si está escribiendo un código que será ejecutado por un `Executor` o algún otro administrador de subprocesos, es fundamental que su código responda rápidamente a las interrupciones, de lo contrario su aplicación se atascará o incluso se interrumpirá.

En tales casos, lo mejor es generalmente permitir que la `InterruptedException` propague por la pila de llamadas, agregando una `throws InterruptedException` a cada método. Esto puede parecer un poco confuso, pero en realidad es una propiedad deseable: las firmas de su método ahora indican a las personas que llaman que responderá rápidamente a las interrupciones.

```
// Let the caller determine how to handle the interrupt if you're unsure
public void myLongRunningMethod() throws InterruptedException {
  ...
}
```

En casos limitados (p. Ej., Al anular un método que no `throw` ninguna excepción comprobada), puede restablecer el estado interrumpido sin generar una excepción, esperando que se ejecute cualquier código al lado para manejar la interrupción. Esto retrasa el manejo de la interrupción pero no la suprime por completo.

```
// Suppresses the exception but resets the interrupted state letting later code
// detect the interrupt and handle it properly.
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  Thread.currentThread().interrupt();
  return ...; // your expectations are still broken at this point - try not to do more work.
}
```

La jerarquía de excepciones de Java - Excepciones no verificadas y revisadas

Todas las excepciones de Java son instancias de clases en la jerarquía de clases de Excepción. Esto se puede representar de la siguiente manera:

- `java.lang.Throwable` : esta es la clase base para todas las clases de excepción. Sus métodos y constructores implementan una gama de funcionalidades comunes a todas las excepciones.
    - `java.lang.Exception` : esta es la superclase de todas las excepciones normales.
        - Diversas clases de excepción estándar y personalizadas.
        - `java.lang.RuntimeException` - Esta es la superclase de todas las excepciones normales que son *excepciones sin marcar* .
            - Varias clases de excepción de tiempo de ejecución estándar y personalizadas.
    - `java.lang.Error` : esta es la superclase de todas las excepciones de "error fatal".

Notas:

1. La distinción entre excepciones *marcadas* y *no marcadas* se describe a continuación.
2. La clase `Throwable` , `Exception` y `RuntimeException` debe tratar como `abstract` ; ver Pitfall - Lanzar Throwable, Exception, Error o RuntimeException .
3. La JVM lanza las excepciones de `Error` en situaciones en las que sería inseguro o imprudente que una aplicación intentara recuperarse.
4. No sería prudente declarar subtipos personalizados de `Throwable` . Las herramientas y bibliotecas de Java pueden asumir que `Error` y `Exception` son los únicos subtipos directos de `Throwable` , y se comportan mal si esa suposición es incorrecta.

Excepciones marcadas versus no verificadas

Una de las críticas de la compatibilidad con excepciones en algunos lenguajes de programación es que es difícil saber qué excepciones puede dar un método o procedimiento determinado. Dado que una excepción no controlada puede provocar que un programa se bloquee, esto puede hacer que las excepciones sean una fuente de fragilidad.

El lenguaje Java aborda esta preocupación con el mecanismo de excepción verificado. Primero, Java clasifica las excepciones en dos categorías:

- Las excepciones verificadas típicamente representan eventos anticipados con los que una aplicación debería poder lidiar. Por ejemplo, `IOException` y sus subtipos representan condiciones de error que pueden ocurrir en operaciones de E / S. Los ejemplos incluyen, el archivo se abre porque falla porque no existe un archivo o directorio, la red lee y escribe fallas porque una conexión de red se ha roto y así sucesivamente.

- Las excepciones no verificadas típicamente representan eventos no anticipados con los que una aplicación no puede lidiar. Estos suelen ser el resultado de un error en la aplicación.

(En lo siguiente, "lanzado" se refiere a cualquier excepción lanzada explícitamente (por una declaración de `throw`), o implícitamente (en una desreferencia fallida, tipo de conversión y así sucesivamente). De manera similar, "propagada" se refiere a una excepción que fue lanzada en una llamada anidada, y no atrapada dentro de esa llamada. El siguiente código de ejemplo lo ilustrará.)

La segunda parte del mecanismo de excepción comprobada es que existen restricciones en los métodos en los que se puede producir una excepción comprobada:

- Cuando una excepción marcada es lanzada o propagada en un método, *debe* ser capturada por el método o listada en la cláusula de `throws` del método. (El significado de la cláusula de `throws` se describe en este ejemplo).
- Cuando se lanza o se propaga una excepción marcada en un bloque de inicialización, se debe capturar el bloque.
- Una excepción comprobada no puede propagarse mediante una llamada de método en una expresión de inicialización de campo. (No hay manera de atrapar tal excepción.)

En resumen, una excepción marcada debe ser manejada o declarada.

Estas restricciones no se aplican a las excepciones sin marcar. Esto incluye todos los casos en los que se lanza una excepción implícitamente, ya que todos los casos arrojan excepciones no verificadas.

Ejemplos de excepciones verificadas

Estos fragmentos de código pretenden ilustrar las restricciones de excepción comprobadas. En cada caso, mostramos una versión del código con un error de compilación, y una segunda versión con el error corregido.

```
// This declares a custom checked exception.
public class MyException extends Exception {
    // constructors omitted.
}

// This declares a custom unchecked exception.
public class MyException2 extends RuntimeException {
    // constructors omitted.
}
```

El primer ejemplo muestra cómo las excepciones comprobadas lanzadas explícitamente se pueden declarar como "lanzadas" si no se deben manejar en el método.

```
// INCORRECT
public void methodThrowingCheckedException(boolean flag) {
    int i = 1 / 0;                // Compiles OK, throws ArithmeticException
    if (flag) {
        throw new MyException();  // Compilation error
    } else {
        throw new MyException2(); // Compiles OK
    }
}

// CORRECTED
public void methodThrowingCheckedException(boolean flag) throws MyException {
    int i = 1 / 0;                // Compiles OK, throws ArithmeticException
    if (flag) {
        throw new MyException();  // Compilation error
    } else {
        throw new MyException2(); // Compiles OK
    }
}
```

El segundo ejemplo muestra cómo se puede tratar una excepción comprobada propagada.

```
// INCORRECT
public void methodWithPropagatedCheckedException() {
    InputStream is = new FileInputStream("someFile.txt");  // Compilation error
    // FileInputStream throws IOException or a subclass if the file cannot
    // be opened.  IOException is a checked exception.
    ...
}

// CORRECTED (Version A)
public void methodWithPropagatedCheckedException() throws IOException {
    InputStream is = new FileInputStream("someFile.txt");
    ...
}

// CORRECTED (Version B)
public void methodWithPropagatedCheckedException() {
    try {
        InputStream is = new FileInputStream("someFile.txt");
        ...
    } catch (IOException ex) {
        System.out.println("Cannot open file: " + ex.getMessage());
    }
}
```

El último ejemplo muestra cómo lidiar con una excepción marcada en un inicializador de campo estático.

```
// INCORRECT
public class Test {
    private static final InputStream is =
            new FileInputStream("someFile.txt");  // Compilation error
}

// CORRECTED
public class Test {
    private static final InputStream is;
    static {
        InputStream tmp = null;
        try {
            tmp = new FileInputStream("someFile.txt");
        } catch (IOException ex) {
            System.out.println("Cannot open file: " + ex.getMessage());
        }
        is = tmp;
    }
}
```

Nótese que en este último caso, también tenemos que hacer frente a los problemas que `is` no se pueden asignar a más de una vez, y sin embargo, también tiene que ser asignado a, incluso en el caso de una excepción.

Introducción

Las excepciones son errores que ocurren cuando un programa se está ejecutando. Considere el

siguiente programa Java que divide dos enteros.

```
class Division {
    public static void main(String[] args) {

        int a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("Input two integers");

        a = input.nextInt();
        b = input.nextInt();

        result = a / b;

        System.out.println("Result = " + result);
    }
}
```

Ahora compilamos y ejecutamos el código anterior, y vemos el resultado de un intento de división por cero:

```
Input two integers
7 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Division.main(Disivion.java:14)
```

La división por cero es una operación no válida que produciría un valor que no se puede representar como un entero. Java se ocupa de esto *lanzando una* ***excepción*** . En este caso, la excepción es una instancia de la clase *ArithmeticException* .

Nota: el ejemplo sobre la creación y lectura de seguimientos de pila explica qué significa la salida después de los dos números.

La utilidad de una *excepción* es el control de flujo que permite. Sin usar excepciones, una solución típica a este problema puede ser verificar primero si `b == 0` :

```
class Division {
    public static void main(String[] args) {

        int a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("Input two integers");

        a = input.nextInt();
        b = input.nextInt();

        if (b == 0) {
            System.out.println("You cannot divide by zero.");
            return;
        }

        result = a / b;

        System.out.println("Result = " + result);
```

```
    }
}
```

Esto imprime el mensaje `You cannot divide by zero.` a la consola y abandona el programa de una manera elegante cuando el usuario intenta dividir por cero. Una forma equivalente de resolver este problema mediante el *manejo de excepciones* sería reemplazar el control `if` flow con un bloque `try-catch` :

```
...

a = input.nextInt();
b = input.nextInt();

try {
    result = a / b;
}
catch (ArithmeticException e) {
    System.out.println("An ArithmeticException occurred. Perhaps you tried to divide by
zero.");
    return;
}

...
```

Un bloque try catch se ejecuta de la siguiente manera:

1. Comienza a ejecutar el código en el bloque `try` .
2. Si se produce una *excepción* en el bloque try, abortar inmediatamente y comprobar para ver si esta excepción es *capturado* por la `catch` de bloque (en este caso, cuando la excepción es una instancia de `ArithmeticException`).
3. Si se *captura* la excepción, se asigna a la variable `e` y se ejecuta el bloque `catch` .
4. Si se completa el bloque `try` o `catch` (es decir, no se producen excepciones no detectadas durante la ejecución del código), continúe ejecutando el código debajo del bloque `try-catch` .

En general, se considera una buena práctica usar el *manejo de excepciones* como parte del control de flujo normal de una aplicación donde el comportamiento sería indefinido o inesperado. Por ejemplo, en lugar de devolver un `null` cuando falla un método, generalmente es una mejor práctica *lanzar una excepción* para que la aplicación que hace uso del método pueda definir su propio control de flujo para la situación mediante el *manejo* de *excepciones* del tipo ilustrado anteriormente. En cierto sentido, esto soluciona el problema de tener que devolver un *tipo* particular, ya que se pueden *lanzar* cualquiera de los múltiples tipos de *excepciones* para indicar el problema específico que ocurrió.

Para obtener más consejos sobre cómo y cómo no usar excepciones, consulte Desventajas de Java - Uso de excepciones

Declaraciones de retorno en el bloque try catch

Aunque es una mala práctica, es posible agregar varias declaraciones de devolución en un bloque de manejo de excepciones:

```
 public static int returnTest(int number){
    try{
        if(number%2 == 0) throw new Exception("Exception thrown");
        else return x;
    }
    catch(Exception e){
        return 3;
    }
    finally{
        return 7;
    }
}
```

Este método siempre devolverá 7 ya que el bloque finally asociado con el bloque try / catch se ejecuta antes de que se devuelva algo. Ahora, como finalmente ha `return 7;` , este valor reemplaza los valores de retorno de prueba / captura.

Si el bloque catch devuelve un valor primitivo y ese valor primitivo se cambia posteriormente en el bloque finally, se devolverá el valor devuelto en el bloque catch y se ignorarán los cambios del bloque finally.

El siguiente ejemplo imprimirá "0", no "1".

```
public class FinallyExample {

    public static void main(String[] args) {
        int n = returnTest(4);

        System.out.println(n);
    }

    public static int returnTest(int number) {

        int returnNumber = 0;

        try {
            if (number % 2 == 0)
                throw new Exception("Exception thrown");
            else
                return returnNumber;
        } catch (Exception e) {
            return returnNumber;
        } finally {
            returnNumber = 1;
        }
    }
}
```

Funciones avanzadas de excepciones

Este ejemplo cubre algunas características avanzadas y casos de uso para excepciones.

Examinando la pila de llamadas programáticamente

Java SE 1.4

El uso principal de los stacktraces de excepción es proporcionar información sobre un error de aplicación y su contexto para que el programador pueda diagnosticar y solucionar el problema. A veces se puede utilizar para otras cosas. Por ejemplo, una clase de `SecurityManager` debe examinar la pila de llamadas para decidir si el código que está realizando una llamada debe ser confiable.

Puede usar excepciones para examinar la pila de llamadas programáticamente de la siguiente manera:

```
Exception ex = new Exception();   // this captures the call stack
StackTraceElement[] frames = ex.getStackTrace();
System.out.println("This method is " + frames[0].getMethodName());
System.out.println("Called from method " + frames[1].getMethodName());
```

Hay algunas advertencias importantes sobre esto:

1. La información disponible en un `StackTraceElement` es limitada. No hay más información disponible de la que se muestra en `printStackTrace` . (Los valores de las variables locales en el marco no están disponibles.)

2. Los javadocs para `getStackTrace()` establecen que una JVM puede omitir marcos:

   > Algunas máquinas virtuales pueden, en algunas circunstancias, omitir uno o más marcos de pila de la traza de pila. En el caso extremo, se permite que una máquina virtual que no tenga información de rastreo de pila con respecto a este objeto de lanzamiento devuelva una matriz de longitud cero desde este método.

Optimizando la construcción de excepciones.

Como se mencionó en otra parte, construir una excepción es bastante costoso porque implica capturar y registrar información sobre todos los marcos de pila en el subproceso actual. A veces, sabemos que esa información nunca se utilizará para una excepción determinada; Por ejemplo, el stacktrace nunca se imprimirá. En ese caso, hay un truco de implementación que podemos usar en una excepción personalizada para hacer que la información no sea capturada.

La información del marco de pila necesaria para los stacktraces, se captura cuando los constructores de `Throwable` llaman al método `Throwable.fillInStackTrace()` . Este método es `public` , lo que significa que una subclase puede anularlo. El truco consiste en anular el método heredado de `Throwable` con uno que no hace nada; p.ej

```
public class MyException extends Exception {
    // constructors

    @Override
    public void fillInStackTrace() {
        // do nothing
    }
}
```

El problema con este enfoque es que una excepción que reemplaza a `fillInStackTrace()` nunca

puede capturar el seguimiento de pila, y es inútil en los escenarios en los que necesita uno.

Borrado o sustitución del trazo de pila.

Java SE 1.4

En algunas situaciones, el seguimiento de pila para una excepción creada de manera normal contiene información incorrecta o información que el desarrollador no quiere revelar al usuario. Para estos escenarios, `Throwable.setStackTrace` se puede usar para reemplazar la matriz de objetos `StackTraceElement` que contiene la información.

Por ejemplo, se puede usar lo siguiente para descartar la información de la pila de una excepción:

```
exception.setStackTrace(new StackTraceElement[0]);
```

Excepciones suprimidas

Java SE 7

Java 7 introdujo la construcción *try-with-resources* y el concepto asociado de supresión de excepciones. Considere el siguiente fragmento de código:

```
try (Writer w = new BufferedWriter(new FileWriter(someFilename))) {
    // do stuff
    int temp = 0 / 0;    // throws an ArithmeticException
}
```

Cuando se lanza la excepción, el `try` llamará a `close()` en la `w` lo que vaciará cualquier salida con búfer y luego cerrará el `FileWriter` . Pero, ¿qué sucede si se lanza una `IOException` al vaciar la salida?

Lo que sucede es que se *suprime* cualquier excepción que se lance al limpiar un recurso. La excepción se captura y se agrega a la lista de excepciones suprimidas de la excepción primaria. A continuación, el *try-with-resources* continuará con la limpieza de los otros recursos. Finalmente, la excepción primaria será devuelta.

Se produce un patrón similar si se produce una excepción durante la inicialización del recurso, o si el bloque `try` se completa normalmente. La primera excepción lanzada se convierte en la excepción principal, y las subsiguientes que surgen de la limpieza se suprimen.

Las excepciones suprimidas se pueden recuperar del objeto de excepción principal llamando a `getSuppressedExceptions` .

Las declaraciones try-finally y try-catch-finally

La declaración `try...catch...finally` combina el manejo de excepciones con el código de limpieza. El bloque `finally` contiene código que se ejecutará en todas las circunstancias. Esto los hace adecuados para la gestión de recursos y otros tipos de limpieza.

Intento-finalmente

Aquí hay un ejemplo de la forma más simple (`try...finally`):

```
try {
    doSomething();
} finally {
    cleanUp();
}
```

El comportamiento del `try...finally` es el siguiente:

- Se ejecuta el código en el bloque `try` .
- Si no se lanzó ninguna excepción en el bloque `try` :
    - Se ejecuta el código en el bloque `finally` .
    - Si el bloque `finally` lanza una excepción, esa excepción se propaga.
    - De lo contrario, el control pasa a la siguiente declaración después del `try...finally` .
- Si se lanzó una excepción en el bloque de prueba:
    - Se ejecuta el código en el bloque `finally` .
    - Si el bloque `finally` lanza una excepción, esa excepción se propaga.
    - De lo contrario, la excepción original continúa propagándose.

El código dentro de `finally` de bloque se ejecutará siempre. (Las únicas excepciones son si `System.exit(int)` se llama, o si los pánicos de JVM.) Así pues, un `finally` de bloque es el código correcto lugar que siempre necesita ser ejecutada; Por ejemplo, cerrando archivos y otros recursos o liberando bloqueos.

prueba-captura-finalmente

Nuestro segundo ejemplo muestra cómo `catch` y `finally` pueden usarse juntos. También ilustra que la limpieza de recursos no es sencilla.

```
// This code snippet writes the first line of a file to a string
String result = null;
Reader reader = null;
try {
    reader = new BufferedReader(new FileReader(fileName));
    result = reader.readLine();
} catch (IOException ex) {
    Logger.getLogger.warn("Unexpected IO error", ex);  // logging the exception
} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException ex) {
            // ignore / discard this exception
        }
    }
}
```

El conjunto completo de comportamientos (hipotéticos) de `try...catch...finally` en este ejemplo

es demasiado complicado de describir aquí. La versión simple es que el código en el bloque `finally` siempre será ejecutado.

Mirando esto desde la perspectiva de la gestión de recursos:

- Declaramos el "recurso" (es decir, la variable del `reader`) antes del bloque `try` para que esté dentro del alcance del bloque `finally` .
- Al colocar el `new FileReader(...)` , la `catch` puede manejar cualquier excepción `IOError` produce al abrir el archivo.
- Necesitamos un `reader.close()` en el bloque `finally` porque hay algunas rutas de excepción que no podemos interceptar ni en el bloque `try` ni en el bloque `catch` .
- Sin embargo, dado *que* se *podría* haber lanzado una excepción antes de que se inicializara el `reader` , también necesitamos una prueba `null` explícita.
- Finalmente, la llamada `reader.close()` podría (hipotéticamente) lanzar una excepción. No nos importa eso, pero si no detectamos la excepción en la fuente, tendremos que lidiar con eso más arriba en la pila de llamadas.

Java SE 7

Java 7 y versiones posteriores ofrecen una sintaxis alternativa de prueba con recursos que simplifica significativamente la limpieza de recursos.

La cláusula de 'tiros' en una declaración de método.

El mecanismo de *excepción comprobada* de Java requiere que el programador declare que ciertos métodos *podrían* generar excepciones comprobadas específicas. Esto se hace utilizando la cláusula de `throws` . Por ejemplo:

```
public class OddNumberException extends Exception { // a checked exception
}

public void checkEven(int number) throws OddNumberException {
    if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

La `throws OddNumberException` declara que una llamada a `checkEven` *podría* lanzar una excepción de tipo `OddNumberException` .

Una cláusula de `throws` puede declarar una lista de tipos y puede incluir excepciones no comprobadas, así como excepciones comprobadas.

```
public void checkEven(Double number)
        throws OddNumberException, ArithmeticException {
    if (!Double.isFinite(number)) {
        throw new ArithmeticException("INF or NaN");
    } else if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

¿Cuál es el punto de declarar excepciones no marcadas como lanzadas?

La cláusula de `throws` en una declaración de método tiene dos propósitos:

1. Le dice al compilador qué excepciones se lanzan para que el compilador pueda reportar las excepciones sin marcar (marcadas) como errores.

2. Le dice a un programador que está escribiendo un código que llama al método qué excepciones esperar. Para este propósito, a menudo hace que los sentidos incluyan excepciones no verificadas en una lista de `throws` .

Nota: la herramienta javadoc también utiliza la lista de `throws` al generar la documentación de la API y las sugerencias de métodos de "texto flotante" de un IDE típico.

Tiros y anulación de método.

La cláusula de los `throws` forma parte de la firma de un método con el propósito de anular el método. Un método de anulación se puede declarar con el mismo conjunto de excepciones marcadas que el método anulado, o con un subconjunto. Sin embargo, el método de anulación no puede agregar excepciones marcadas adicionales. Por ejemplo:

```
@Override
public void checkEven(int number) throws NullPointerException // OK—NullPointerException is an
unchecked exception
    ...

@Override
public void checkEven(Double number) throws OddNumberException // OK—identical to the
superclass
    ...

class PrimeNumberException extends OddNumberException {}
class NonEvenNumberException extends OddNumberException {}

@Override
public void checkEven(int number) throws PrimeNumberException, NonEvenNumberException //
OK—these are both subclasses

@Override
public void checkEven(Double number) throws IOExcepion         // ERROR
```

La razón de esta regla es que si un método anulado puede lanzar una excepción marcada que el método anulado no podría lanzar, eso interrumpiría la sustituibilidad del tipo.

Lea Excepciones y manejo de excepciones. en línea:
https://riptutorial.com/es/java/topic/89/excepciones-y-manejo-de-excepciones-

Capítulo 78: Expresiones

Introducción

Las expresiones en Java son la construcción primaria para hacer cálculos.

Observaciones

Para obtener una referencia sobre los operadores que se pueden usar en expresiones, consulte Operadores .

Examples

Precedencia del operador

Cuando una expresión contiene múltiples operadores, potencialmente puede leerse de diferentes maneras. Por ejemplo, la expresión matemática `1 + 2 x 3` se puede leer de dos maneras:

1. Suma `1` y `2` y multiplica el resultado por `3` . Esto da la respuesta `9` . Si agregáramos paréntesis, esto se vería como `( 1 + 2 ) x 3` .
2. Suma `1` al resultado de multiplicar `2` y `3` . Esto da la respuesta `7` . Si agregáramos paréntesis, esto se vería como `1 + ( 2 x 3 )` .

En matemáticas, la convención es leer la expresión de la segunda manera. La regla general es que la multiplicación y la división se realizan antes de la suma y la resta. Cuando se usa una notación matemática más avanzada, el significado es "evidente por sí mismo" (¡para un matemático capacitado!), O se agregan paréntesis para desambiguar. En cualquier caso, la efectividad de la notación para transmitir significado depende de la inteligencia y el conocimiento compartido de los matemáticos.

Java tiene las mismas reglas claras sobre cómo leer una expresión, en función de la *precedencia* de los operadores que se utilizan.

En general, a cada operador se le asigna un valor de *precedencia* ; Vea la tabla de abajo.

Por ejemplo:

```
1 + 2 * 3
```

La precedencia de `+` es menor que la precedencia de `*` , por lo que el resultado de la expresión es 7, no 9.

Descripción	Operadores / constructos (primarios)	Precedencia	Asociatividad
Índice Paréntesis Creación de instancias Acceso al campo Acceso a la matriz Invocación de método Referencia del método	nombre . nombre (expr) new primaria . nombre primaria [expr] primaria (expr, ...) primario :: nombre	15	De izquierda a derecha
Post Incremento	expr ++ , expr --	14	-
Pre incremento Unario Reparto [1]	++ expr, -- expr, + Expr, - expr, ~ expr, ! expr (tipo) expr	13	- De derecha a izquierda De derecha a izquierda
Multiplicativa	* /%	12	De izquierda a derecha
Aditivo	+ -	11	De izquierda a derecha
Cambio	<< >> >>>	10	De izquierda a derecha
Relacional	<> <=> = instanceof	9	De izquierda a derecha
Igualdad	==! =	8	De izquierda a derecha
Y a nivel de bit	Y	7	De izquierda a derecha
Exclusivo bitwise O	^	6	De izquierda a derecha
Bitwise inclusive O	|	5	De izquierda a derecha
Y lógico	&&	4	De izquierda a derecha
O lógico	||	3	De izquierda a

Descripción	Operadores / constructos (primarios)	Precedencia	Asociatividad
			derecha
Condicional [1]	? :	2	De derecha a izquierda
Asignación Lambda [1]	= * = / =% = + = - = << = >> = >>> = & = ^ = | = ->	1	De derecha a izquierda

[1] La precedencia de expresión Lambda es compleja, ya que también puede ocurrir después de una conversión, o como la tercera parte del operador ternario condicional.

Expresiones constantes

Una expresión constante es una expresión que produce un tipo primitivo o una cadena, y cuyo valor puede evaluarse en tiempo de compilación a un literal. La expresión debe evaluar sin lanzar una excepción, y debe estar compuesta de solo lo siguiente:

- Literales primitivos y de cuerdas.
- Tipo de conversión a tipos primitivos o `String` .
- Los siguientes operadores unarios: `+` , `-` , `~` y `!` .
- Los siguientes operadores binarios: `*` , `/` , `%` , `+` , `-` , `<<` , `>>` , `>>>` , `<` , `<=` , `>` , `>=` , `==` `!=` , `&` , `^` , `|` , `&&` y `||` .
- ¿El operador condicional ternario `?` `:` .
- Expresiones constantes entre paréntesis.
- Nombres simples que se refieren a variables constantes. (Una variable constante es una variable declarada como `final` donde la expresión inicializadora es en sí misma una expresión constante).
- Nombres calificados del formulario `<TypeName>` `.` `<Identifier>` que se refiere a variables constantes.

Tenga en cuenta que la lista anterior *excluye* `++` y `--` , los operadores de asignación, `class` y `instanceof` , llamadas de método y referencias a variables generales o campos.

Las expresiones constantes de tipo `String` resultar en una "internados" `String` , y operaciones de punto flotante en expresiones constantes se evalúan con la semántica FP-estrictas.

Usos para expresiones constantes

Las expresiones constantes se pueden usar (casi) en cualquier lugar donde se pueda usar una expresión normal. Sin embargo, tienen un significado especial en los siguientes contextos.

Las expresiones constantes son necesarias para las etiquetas de casos en las declaraciones de cambio. Por ejemplo:

```
switch (someValue) {
case 1 + 1:             // OK
case Math.min(2, 3):    // Error - not a constant expression
    doSomething();
}
```

Cuando la expresión en el lado derecho de una asignación es una expresión constante, la asignación puede realizar una conversión de estrechamiento primitivo. Esto se permite siempre que el valor de la expresión constante esté dentro del rango del tipo en el lado izquierdo. (Ver JLS 5.1.3 y 5.2) Por ejemplo:

```
byte b1 = 1 + 1;              // OK - primitive narrowing conversion.
byte b2 = 127 + 1;            // Error - out of range
byte b3 = b1 + 1;             // Error - not a constant expession
byte b4 = (byte) (b1 + 1);    // OK
```

Cuando se usa una expresión constante como condición en un `do` , `while` o `for` , entonces afecta el análisis de legibilidad. Por ejemplo:

```
while (false) {
    doSomething();            // Error - statenent not reachable
}
boolean flag = false;
while (flag) {
    doSomething();            // OK
}
```

(Tenga en cuenta que esto no se aplica a las sentencias `if` . El compilador de Java permite que se pueda acceder al bloque `then` o `else` de una sentencia `if` . Este es el análogo de Java de la compilación condicional en C y C ++).

Finalmente, `static final` campos `static final` en una clase o interfaz con inicializadores de expresiones constantes se inicializan con entusiasmo. Por lo tanto, se garantiza que estas constantes se observarán en el estado inicializado, incluso cuando hay un ciclo en el gráfico de dependencia de inicialización de clase.

Para obtener más información, consulte JLS 15.28. Expresiones constantes .

Orden de evaluación de expresiones

Las expresiones de Java se evalúan siguiendo las siguientes reglas:

- Los operandos se evalúan de izquierda a derecha.
- Los operandos de un operador se evalúan antes que el operador.
- Los operadores son evaluados de acuerdo a la precedencia del operador.

- Las listas de argumentos se evalúan de izquierda a derecha.

Ejemplo simple

En el siguiente ejemplo:

```
int i = method1() + method2();
```

El orden de evaluación es:

1. El operando izquierdo de `=` operator se evalúa en la dirección de `i` .
2. Se evalúa el operando izquierdo del operador `+` (`method1()`).
3. Se evalúa el operando derecho del operador `+` (`method2()`).
4. Se evalúa la operación `+` .
5. Se evalúa la operación `=` , asignando el resultado de la suma a `i` .

Tenga en cuenta que si los efectos de las llamadas son observables, podrá observar que la llamada a `method1` ocurre antes de la llamada a `method2` .

Ejemplo con un operador que tiene un efecto secundario

En el siguiente ejemplo:

```
int i = 1;
intArray[i] = ++i + 1;
```

El orden de evaluación es:

1. Se evalúa el operando izquierdo de `=` operator. Esto da la dirección de `intArray[1]` .
2. Se evalúa el pre-incremento. Esto agrega `1` a `i` , y evalúa a `2` .
3. El operando de la derecha del `+` es evaluado.
4. La operación `+` se evalúa como: `2 + 1` -> `3` .
5. Se evalúa la operación `=` , asignando `3` a `intArray[1]` .

Tenga en cuenta que dado que el operando de la izquierda de `=` se evalúa primero, no está influenciado por el efecto secundario de la subexpresión `++i` .

Referencia:

- JLS 15.7 - Orden de evaluación

Conceptos básicos de expresión

Las expresiones en Java son la construcción primaria para hacer cálculos. Aquí hay unos ejemplos:

```
1                   // A simple literal is an expression
1 + 2               // A simple expression that adds two numbers
```

```
(i + j) / k        // An expression with multiple operations
(flag) ? c : d     // An expression using the "conditional" operator
(String) s         // A type-cast is an expression
obj.test()         // A method call is an expression
new Object()       // Creation of an object is an expression
new int[]          // Creation of an object is an expression
```

En general, una expresión consta de las siguientes formas:

- Nombres de expresiones que consisten en:
  - Identificadores simples; por ejemplo, `someIdentifier`
  - Identificadores calificados; por ejemplo, `MyClass.someField`
- Primarias que consisten en:
  - Literales; por ejemplo, `1` , `1.0` , `'X'` , `"hello"` , `false` y `null`
  - Expresiones literales de clase; por ejemplo, `MyClass.class`
  - `this` y `<TypeName> . this`
  - Expresiones entre paréntesis; por ejemplo `( a + b )`
  - Expresiones de creación de instancia de clase; por ejemplo, `new MyClass(1, 2, 3)`
  - Expresiones de creación de instancia de matriz; por ejemplo, `new int[3]`
  - Expresiones de acceso al campo; por ejemplo, `obj.someField` o `this.someField`
  - Expresiones de acceso a la matriz; por ejemplo, `vector[21]`
  - Invocaciones de método; por ejemplo, `obj.doIt(1, 2, 3)`
  - Referencias de métodos (Java 8 y posteriores); por ejemplo, `MyClass::doIt`
- Expresiones de operador unario; por ejemplo `!a i++`
- Expresiones de operador binarias; por ejemplo, `a + b` u `obj == null`
- Expresiones del operador ternario; por ejemplo `(obj == null) ? 1 : obj.getCount()`
- Expresiones Lambda (Java 8 y posteriores); por ejemplo, `obj -> obj.getCount()`

Los detalles de las diferentes formas de expresiones se pueden encontrar en otros temas.

- El tema Operadores cubre expresiones de operador unarias, binarias y ternarias.
- El tema de expresiones Lambda cubre expresiones lambda y expresiones de referencia de métodos.
- El tema Clases y objetos abarca expresiones de creación de instancia de clase.
- El tema Arrays cubre expresiones de acceso a la matriz y expresiones de creación de instancia de matriz.
- El tema Literales cubre los diferentes tipos de expresiones literales.

El tipo de expresión

En la mayoría de los casos, una expresión tiene un tipo estático que se puede determinar en tiempo de compilación mediante el examen y sus subexpresiones. Estos se conocen como expresiones *independientes* .

Sin embargo, (en Java 8 y posteriores) los siguientes tipos de expresiones pueden ser *expresiones poli* :

- Expresiones entre paréntesis

- Expresiones de creación de instancia de clase
- Expresiones de invocación de métodos
- Expresiones de referencia del método
- Expresiones condicionales
- Expresiones lambda

Cuando una expresión es una expresión poli, su tipo puede verse influido por el *tipo de destino de* la expresión; Es decir, para qué se está utilizando.

El valor de una expresión

El valor de una expresión es la asignación compatible con su tipo. La excepción a esto es cuando ha ocurrido la *contaminación del montón* ; por ejemplo, porque las advertencias de "conversión insegura" se han suprimido o ignorado (de manera inadecuada).

Declaraciones de Expresión

A diferencia de muchos otros lenguajes, Java generalmente no permite que las expresiones se usen como declaraciones. Por ejemplo:

```
public void compute(int i, int j) {
    i + j;   // ERROR
}
```

Dado que el resultado de evaluar una expresión como no se puede usar, y como no puede afectar la ejecución del programa de ninguna otra manera, los diseñadores de Java tomaron la posición de que tal uso es un error o está equivocado.

Sin embargo, esto no se aplica a todas las expresiones. Un subconjunto de expresiones son (de hecho) legales como declaraciones. El conjunto comprende:

- Expresión de asignación, incluidas las asignaciones de *operaciones y se convierte en* .
- Expresiones pre y post incremento y decremento.
- Las llamadas de método (`void` o no `void`).
- Expresiones de creación de instancia de clase.

Lea Expresiones en línea: https://riptutorial.com/es/java/topic/8167/expresiones

Capítulo 79: Expresiones lambda

Introducción

Las expresiones Lambda proporcionan una forma clara y concisa de implementar una interfaz de un solo método utilizando una expresión. Le permiten reducir la cantidad de código que tiene que crear y mantener. Aunque son similares a las clases anónimas, no tienen información de tipo por sí mismas. La inferencia de tipos debe suceder.

Las referencias de métodos implementan interfaces funcionales utilizando métodos existentes en lugar de expresiones. Pertenecen a la familia lambda también.

Sintaxis

- () -> {return expresión; } // Zero-arity con el cuerpo de la función para devolver un valor.
- () -> expresión // taquigrafía para la declaración anterior; No hay punto y coma para las expresiones.
- () -> {function-body} // Efecto lateral en la expresión lambda para realizar operaciones.
- parámetroName -> expresión // expresión lambda de una aridad. En las expresiones lambda con un solo argumento, se puede eliminar el paréntesis.
- (Escriba ParameterName, Escriba secondParameterName, ...) -> expresión // lambda evaluando una expresión con los parámetros listados a la izquierda
- (nombre de parámetro, segundo Nombre de parámetro, ...) -> expresión // Taquigrafía que elimina los tipos de parámetros para los nombres de parámetros. Solo se puede usar en contextos que pueden ser deducidos por el compilador donde el tamaño de lista de parámetros dado coincide con uno (y solo uno) del tamaño de las interfaces funcionales esperadas.

Examples

Usando expresiones Lambda para ordenar una colección

Listas de clasificación

Antes de Java 8, era necesario implementar la interfaz `java.util.Comparator` con una clase anónima (o nombrada) al ordenar una lista [1] :

Java SE 1.2

```
List<Person> people = ...
Collections.sort(
    people,
    new Comparator<Person>() {
        public int compare(Person p1, Person p2){
```

```
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
);
```

A partir de Java 8, la clase anónima se puede reemplazar con una expresión lambda. Tenga en cuenta que los tipos para los parámetros `p1` y `p2` se pueden omitir, ya que el compilador los deducirá automáticamente:

```
Collections.sort(
    people,
    (p1, p2) -> p1.getFirstName().compareTo(p2.getFirstName())
);
```

El ejemplo se puede simplificar utilizando `Comparator.comparing` y referencias de métodos expresadas con el símbolo `::` (dos puntos).

```
Collections.sort(
    people,
    Comparator.comparing(Person::getFirstName)
);
```

Una importación estática nos permite expresar esto de manera más concisa, pero es discutible si esto mejora la legibilidad general:

```
import static java.util.Collections.sort;
import static java.util.Comparator.comparing;
//...
sort(people, comparing(Person::getFirstName));
```

Los comparadores construidos de esta manera también se pueden encadenar juntos. Por ejemplo, después de comparar personas por su primer nombre, si hay personas con el mismo nombre, el método `thenComparing` también se compara por apellido:

```
sort(people, comparing(Person::getFirstName).thenComparing(Person::getLastName));
```

1: tenga en cuenta que Collections.sort (...) solo funciona en colecciones que son subtipos de `List` . Las API de `Set` y `Collection` no implican ningún orden de los elementos.

Clasificando mapas

Puede ordenar las entradas de un `HashMap` por valor de una manera similar. (Tenga en cuenta que debe utilizarse un `LinkedHashMap` como destino. Las claves en un `HashMap` normal no están ordenadas).

```
Map<String, Integer> map = new HashMap();  // ... or any other Map class
// populate the map
map = map.entrySet()
```

```
    .stream()
    .sorted(Map.Entry.<String, Integer>comparingByValue())
    .collect(Collectors.toMap(k -> k.getKey(), v -> v.getValue(),
                              (k, v) -> k, LinkedHashMap::new));
```


Interfaces funcionales

Lambdas solo puede operar en una interfaz funcional, que es una interfaz con solo un método abstracto. Las interfaces funcionales pueden tener cualquier número de métodos `default` o `static` . (Por esta razón, a veces se les conoce como interfaces de método abstracto único o interfaces SAM).

```
interface Foo1 {
    void bar();
}

interface Foo2 {
    int bar(boolean baz);
}

interface Foo3 {
    String bar(Object baz, int mink);
}

interface Foo4 {
    default String bar() { // default so not counted
        return "baz";
    }
    void quux();
}
```

Al declarar una interfaz funcional, se puede agregar la anotación `@FunctionalInterface` . Esto no tiene ningún efecto especial, pero se generará un error de compilación si esta anotación se aplica a una interfaz que no es funcional, por lo que actúa como un recordatorio de que la interfaz no debe cambiarse.

```
@FunctionalInterface
interface Foo5 {
    void bar();
}

@FunctionalInterface
interface BlankFoo1 extends Foo3 { // inherits abstract method from Foo3
}

@FunctionalInterface
interface Foo6 {
    void bar();
    boolean equals(Object obj); // overrides one of Object's method so not counted
}
```

A la inversa, esta **no** es una interfaz funcional, ya que tiene más de **un** método **abstracto** :

```
interface BadFoo {
    void bar();
    void quux(); // <-- Second method prevents lambda: which one should
                 // be considered as lambda?
}
```

Esto **tampoco es** una interfaz funcional, ya que no tiene ningún método:

```
interface BlankFoo2 { }
```

Tome nota de lo siguiente. Supongamos que tienes

```
interface Parent { public int parentMethod(); }
```

y

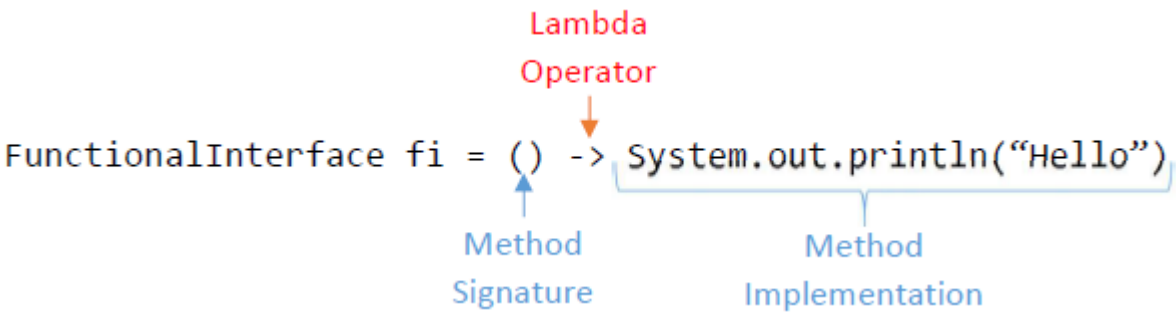
```
interface Child extends Parent { public int ChildMethod(); }
```

Entonces, `Child` **no puede** ser una interfaz funcional ya que tiene dos métodos especificados.

Java 8 también proporciona una serie de interfaces funcionales genéricas en el paquete `java.util.function` . Por ejemplo, la interfaz integrada `Predicate<T>` envuelve un solo método que ingresa un valor de tipo `T` y devuelve un valor `boolean` .

Expresiones lambda

La estructura básica de una expresión Lambda es:



`fi` entonces tendrá una instancia de singleton de una clase, similar a una clase anónima, que implementa `FunctionalInterface` y donde la definición de un método es `{ System.out.println("Hello"); }` . En otras palabras, lo anterior es mayormente equivalente a:

```
FunctionalInterface fi = new FunctionalInterface() {
    @Override
    public void theOneMethod() {
        System.out.println("Hello");
    }
};
```

La lambda solo es "mayormente equivalente" a la clase anónima porque en lambda, el significado de expresiones como `this` , `super` o `toString()` referencia a la clase dentro de la cual se realiza la asignación, no al objeto recién creado.

No puede especificar el nombre del método cuando usa un lambda, pero no debería necesitarlo, ya que una interfaz funcional solo debe tener un método abstracto, por lo que Java anula ese método.

En los casos en que el tipo de la lambda no es seguro, (por ejemplo, los métodos sobrecargados) puede agregar una conversión a la lambda para decirle al compilador cuál debería ser su tipo, así:

```
Object fooHolder = (Foo1) () -> System.out.println("Hello");
System.out.println(fooHolder instanceof Foo1); // returns true
```

Si el método único de la interfaz funcional toma parámetros, los nombres formales locales de estos deberían aparecer entre los paréntesis de la lambda. No es necesario declarar el tipo de parámetro o retorno, ya que estos se toman de la interfaz (aunque no es un error declarar los tipos de parámetros si lo desea). Por lo tanto, estos dos ejemplos son equivalentes:

```
Foo2 longFoo = new Foo2() {
    @Override
    public int bar(boolean baz) {
        return baz ? 1 : 0;
    }
};
Foo2 shortFoo = (x) -> { return x ? 1 : 0; };
```

Los paréntesis alrededor del argumento se pueden omitir si la función solo tiene un argumento:

```
Foo2 np = x -> { return x ? 1 : 0; }; // okay
Foo3 np2 = x, y -> x.toString() + y // not okay
```

Devoluciones implícitas

Si el código colocado dentro de un lambda es una *expresión de* Java en lugar de una *declaración* , se trata como un método que devuelve el valor de la expresión. Por lo tanto, los siguientes dos son equivalentes:

```
IntUnaryOperator addOneShort = (x) -> (x + 1);
IntUnaryOperator addOneLong = (x) -> { return (x + 1); };
```

Acceso a variables locales (cierres de valor)

Dado que las lambdas son una abreviatura sintáctica para las clases anónimas, siguen las mismas reglas para acceder a las variables locales en el ámbito adjunto; Las variables deben ser

tratadas como `final` y no modificadas dentro de la lambda.

```
IntUnaryOperator makeAdder(int amount) {
    return (x) -> (x + amount); // Legal even though amount will go out of scope
                                // because amount is not modified
}

IntUnaryOperator makeAccumulator(int value) {
    return (x) -> { value += x; return value; }; // Will not compile
}
```

Si es necesario envolver una variable cambiante de esta manera, se debe usar un objeto normal que mantenga una copia de la variable. Lea más en Java Closures con expresiones lambda.

Aceptando Lambdas

Debido a que una lambda es una implementación de una interfaz, no se necesita hacer nada especial para que un método acepte una lambda: cualquier función que tome una interfaz funcional también puede aceptar una lambda.

```
public void passMeALambda(Foo1 f) {
    f.bar();
}
passMeALambda(() -> System.out.println("Lambda called"));
```

El tipo de expresión lambda

Una expresión lambda, por sí misma, no tiene un tipo específico. Si bien es cierto que los tipos y la cantidad de parámetros, junto con el tipo de un valor de retorno pueden transmitir cierta información de tipo, dicha información solo restringirá a qué tipos se puede asignar. La lambda recibe un tipo cuando se asigna a un tipo de interfaz funcional de una de las siguientes maneras:

- Asignación directa a un tipo funcional, por ejemplo, `myPredicate = s -> s.isEmpty()`
- Pasándolo como un parámetro que tiene un tipo funcional, por ejemplo, `stream.filter(s -> s.isEmpty())`
- Devolviéndolo desde una función que devuelve un tipo funcional, por ejemplo, `return s -> s.isEmpty()`
- Convertirlo en un tipo funcional, por ejemplo `(Predicate<String>) s -> s.isEmpty()`

Hasta que se realice una asignación de este tipo a un tipo funcional, la lambda no tiene un tipo definido. Para ilustrar, considere la expresión lambda `o -> o.isEmpty()` . La misma expresión lambda se puede asignar a muchos tipos funcionales diferentes:

```
Predicate<String> javaStringPred = o -> o.isEmpty();
Function<String, Boolean> javaFunc = o -> o.isEmpty();
Predicate<List> javaListPred = o -> o.isEmpty();
Consumer<String> javaStringConsumer = o -> o.isEmpty(); // return value is ignored!
com.google.common.base.Predicate<String> guavaPredicate = o -> o.isEmpty();
```

Ahora que están asignados, los ejemplos que se muestran son de tipos completamente diferentes, aunque las expresiones lambda tienen el mismo aspecto y no pueden asignarse entre sí.

Referencias del método

Las referencias de método permiten que los métodos estáticos o de instancia predefinidos que se adhieran a una interfaz funcional compatible se pasen como argumentos en lugar de una expresión lambda anónima.

Supongamos que tenemos un modelo:

```
class Person {
    private final String name;
    private final String surname;

    public Person(String name, String surname){
        this.name = name;
        this.surname = surname;
    }

    public String getName(){ return name; }
    public String getSurname(){ return surname; }
}

List<Person> people = getSomePeople();
```

Referencia del método de instancia (a una instancia arbitraria)

```
people.stream().map(Person::getName)
```

La lambda equivalente:

```
people.stream().map(person -> person.getName())
```

En este ejemplo, se está pasando una referencia de método al método de instancia `getName()` de tipo `Person` . Como se sabe que es del tipo de colección, se invocará el método en la instancia (conocido más adelante).

Referencia del método de instancia (a una instancia específica)

```
people.forEach(System.out::println);
```

Dado que `System.out` es una instancia de `PrintStream` , una referencia de método a esta instancia

específica se pasa como un argumento.

La lambda equivalente:

```
people.forEach(person -> System.out.println(person));
```

Referencia de método estático

También para transformar flujos podemos aplicar referencias a métodos estáticos:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
numbers.stream().map(String::valueOf)
```

Este ejemplo pasa una referencia al método static `valueOf()` en el tipo `String` . Por lo tanto, el objeto de instancia en la colección se pasa como un argumento a `valueOf()` .

La lambda equivalente:

```
 numbers.stream().map(num -> String.valueOf(num))
```

Referencia a un constructor

```
List<String> strings = Arrays.asList("1", "2", "3");
strings.stream().map(Integer::new)
```

Lea Recopilar elementos de un flujo en una colección para ver cómo recopilar elementos para recopilar.

El constructor de argumento de cadena única del tipo `Integer` se está utilizando aquí, para construir un entero dado la cadena proporcionada como el argumento. En este caso, siempre que la cadena represente un número, el flujo se asignará a enteros. La lambda equivalente:

```
strings.stream().map(s -> new Integer(s));
```

Hoja de trucos

Formato de referencia del método	Código	Equivalente lambda
Método estático	`TypeName::method`	`(args) -> TypeName.method(args)`
Método no estático (en caso de *)	`instance::method`	`(args) -> instance.method(args)`

Formato de referencia del método	Código	Equivalente lambda
Método no estático (sin instancia)	`TypeName::method`	`(instance, args) -> instance.method(args)`
Constructor **	`TypeName::new`	`(args) -> new TypeName(args)`
Constructor de arrays	`TypeName[]::new`	`(int size) -> new TypeName[size]`

---

* `instance` puede ser cualquier expresión que evalúe una referencia a una instancia, por ejemplo, `getInstance()::method` , `this::method`

** Si `TypeName` es una clase interna no estática, la referencia del constructor solo es válida dentro del alcance de una instancia de clase externa

Implementando multiples interfaces

A veces es posible que desee tener una expresión lambda implementando más de una interfaz. Esto es principalmente útil con interfaces de marcadores (como java.io.Serializable) ya que no agregan métodos abstractos.

Por ejemplo, desea crear un `TreeSet` con un `Comparator` personalizado y luego serializarlo y enviarlo a través de la red. El enfoque trivial:

```
TreeSet<Long> ts = new TreeSet<>((x, y) -> Long.compare(y, x));
```

no funciona ya que la lambda para el comparador no implementa `Serializable` . Puede solucionar este problema utilizando tipos de intersección y especificando explícitamente que este lambda debe ser serializable:

```
TreeSet<Long> ts = new TreeSet<>(
    (Comparator<Long> & Serializable) (x, y) -> Long.compare(y, x));
```

Si usa con frecuencia tipos de intersección (por ejemplo, si usa un marco como Apache Spark, donde casi todo tiene que ser serializable), puede crear interfaces vacías y usarlas en su código:

```
public interface SerializableComparator extends Comparator<Long>, Serializable {}

public class CustomTreeSet {
  public CustomTreeSet(SerializableComparator comparator) {}
}
```

De esta manera, se garantiza que el comparador pasado será serializable.

Lambdas y patrón de ejecución

Hay varios buenos ejemplos del uso de lambdas como FunctionalInterface en escenarios simples.

Un caso de uso bastante común que puede ser mejorado por las lambdas es lo que se denomina patrón de ejecución. En este patrón, tiene un conjunto de código de configuración / desmontaje estándar que se necesita para múltiples escenarios que rodean el código específico de caso de uso. Algunos ejemplos comunes de esto son los archivos io, database io, try / catch blocks.

```
interface DataProcessor {
    void process( Connection connection ) throws SQLException;;
}

public void doProcessing( DataProcessor processor ) throws SQLException{
    try (Connection connection = DBUtil.getDatabaseConnection();) {
        processor.process(connection);
        connection.commit();
    }
}
```

Luego, para llamar a este método con un lambda, podría verse así:

```
public static void updateMyDAO(MyVO vo) throws DatabaseException {
    doProcessing((Connection conn) -> MyDAO.update(conn, ObjectMapper.map(vo)));
}
```

Esto no se limita a las operaciones de E / S. Se puede aplicar a cualquier escenario en el que se apliquen tareas similares de configuración / desmontaje con pequeñas variaciones. El principal beneficio de este patrón es la reutilización del código y la aplicación de DRY (no se repita).

Usando lambda expresión con su propia interfaz funcional

Lambdas está diseñado para proporcionar código de implementación en línea para interfaces de un solo método y la capacidad de pasarlas como lo hemos estado haciendo con las variables normales. Los llamamos interfaz funcional.

Por ejemplo, escribir un Runnable en una clase anónima y comenzar un Thread se ve así:

```
//Old way
new Thread(
        new Runnable(){
            public void run(){
                System.out.println("run logic...");
            }
        }
).start();

//lambdas, from Java 8
new Thread(
        ()-> System.out.println("run logic...")
).start();
```

Ahora, en línea con lo anterior, digamos que tiene alguna interfaz personalizada:

```
interface TwoArgInterface {
    int operate(int a, int b);
}
```

¿Cómo utiliza lambda para implementar esta interfaz en su código? Igual que el ejemplo de Runnable que se muestra arriba. Vea el programa del conductor a continuación:

```
public class CustomLambda {
    public static void main(String[] args) {

        TwoArgInterface plusOperation = (a, b) -> a + b;
        TwoArgInterface divideOperation = (a,b)->{
            if (b==0) throw new IllegalArgumentException("Divisor can not be 0");
            return a/b;
        };

        System.out.println("Plus operation of 3 and 5 is: " + plusOperation.operate(3, 5));
        System.out.println("Divide operation 50 by 25 is: " + divideOperation.operate(50,
25));

    }
}
```

`return 'solo regresa de la lambda, no del método externo

El método de `return` solo regresa de la lambda, no del método externo.

¡Cuidado, esto es *diferente* de Scala y Kotlin!

```
void threeTimes(IntConsumer r) {
  for (int i = 0; i < 3; i++) {
    r.accept(i);
  }
}

void demo() {
  threeTimes(i -> {
    System.out.println(i);
    return; // Return from lambda to threeTimes only!
  });
}
```

Esto puede llevar a un comportamiento inesperado al intentar escribir construcciones de lenguaje propias, como en las construcciones integradas, como `for` bucles de `return` comporta de manera diferente:

```
void demo2() {
  for (int i = 0; i < 3; i++) {
    System.out.println(i);
    return; // Return from 'demo2' entirely
  }
}
```

En Scala y Kotlin, `demo` y `demo2` solo imprimirían `0` . Pero esto *no* es más consistente. El enfoque de Java es consistente con la refactorización y el uso de clases: el `return` en el código en la parte superior y el siguiente código se comporta de la misma manera:

```
void demo3() {
```

```
  threeTimes(new MyIntConsumer());
}

class MyIntConsumer implements IntConsumer {
  public void accept(int i) {
    System.out.println(i);
    return;
  }
}
```

Por lo tanto, el Java `return` es más coherente con los métodos de clase y refactorización, pero menos con el `for` y `while` las órdenes internas, éstas siguen siendo especial.

Debido a esto, los siguientes dos son equivalentes en Java:

```
IntStream.range(1, 4)
    .map(x -> x * x)
    .forEach(System.out::println);
IntStream.range(1, 4)
    .map(x -> { return x * x; })
    .forEach(System.out::println);
```

Además, el uso de try-with-resources es seguro en Java:

```
class Resource implements AutoCloseable {
  public void close() { System.out.println("close()"); }
}

void executeAround(Consumer<Resource> f) {
  try (Resource r = new Resource()) {
    System.out.print("before ");
    f.accept(r);
    System.out.print("after ");
  }
}

void demo4() {
  executeAround(r -> {
    System.out.print("accept() ");
    return; // Does not return from demo4, but frees the resource.
  });
}
```

se imprimirá `before accept() after close()` . En la semántica de Scala y Kotlin, el intento con recursos no se cerraría, pero se imprimiría `before accept()` solamente.

Cierres de Java con expresiones lambda.

Un cierre lambda se crea cuando una expresión lambda hace referencia a las variables de un ámbito de distribución (global o local). Las reglas para hacer esto son las mismas que para los métodos en línea y las clases anónimas.

Las variables locales de un ámbito que se usan dentro de un lambda tienen que ser `final` . Con Java 8 (la versión más antigua que admite lambdas), no necesitan ser *declarados* `final` en el

contexto externo, sino que deben ser tratados de esa manera. Por ejemplo:

```
int n = 0; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
```

Esto es legal siempre que no se cambie el valor de la variable `n` . Si intenta cambiar la variable, dentro o fuera de la lambda, obtendrá el siguiente error de compilación:

> "las variables locales referenciadas desde una expresión lambda deben ser *finales* o *efectivamente finales* ".

Por ejemplo:

```
int n = 0;
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
n++; // Will generate an error.
```

Si es necesario usar una variable cambiante dentro de un lambda, el enfoque normal es declarar una copia `final` de la variable y usar la copia. Por ejemplo

```
int n = 0;
final int k = n; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = k;
    // do something
};
n++;      // Now will not generate an error
r.run();  // Will run with i = 0 because k was 0 when the lambda was created
```

Naturalmente, el cuerpo de la lambda no ve los cambios en la variable original.

Tenga en cuenta que Java no admite cierres verdaderos. Un lambda de Java no se puede crear de manera que le permita ver los cambios en el entorno en el que se creó la instancia. Si desea implementar un cierre que observe o realice cambios en su entorno, debe simularlo utilizando una clase regular. Por ejemplo:

```
// Does not compile ...
public IntUnaryOperator createAccumulator() {
    int value = 0;
    IntUnaryOperator accumulate = (x) -> { value += x; return value; };
    return accumulate;
}
```

El ejemplo anterior no se compilará por razones discutidas previamente. Podemos solucionar el error de compilación de la siguiente manera:

```
// Compiles, but is incorrect ...
public class AccumulatorGenerator {
    private int value = 0;

    public IntUnaryOperator createAccumulator() {
        IntUnaryOperator accumulate = (x) -> { value += x; return value; };
        return accumulate;
    }
}
```

El problema es que esto rompe el contrato de diseño para la interfaz `IntUnaryOperator` que establece que las instancias deben ser funcionales y sin estado. Si dicho cierre se pasa a funciones integradas que aceptan objetos funcionales, es probable que cause bloqueos o comportamientos erróneos. Los cierres que encapsulan el estado mutable deben implementarse como clases regulares. Por ejemplo.

```
// Correct ...
public class Accumulator {
   private int value = 0;

   public int accumulate(int x) {
      value += x;
      return value;
   }
}
```

Lambda - Ejemplo de oyente

Oyente anónimo de clase

Antes de Java 8, es muy común que se use una clase anónima para manejar el evento de clic de un JButton, como se muestra en el siguiente código. Este ejemplo muestra cómo implementar una escucha anónima dentro del alcance de `btn.addActionListener` .

```
JButton btn = new JButton("My Button");
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button was pressed");
    }
});
```

Oyente lambda

Debido a que la interfaz `ActionListener` define solo un método `actionPerformed()` , es una interfaz funcional que significa que hay un lugar para usar expresiones Lambda para reemplazar el código repetitivo. El ejemplo anterior se puede reescribir usando expresiones Lambda de la siguiente manera:

```
JButton btn = new JButton("My Button");
btn.addActionListener(e -> {
    System.out.println("Button was pressed");
});
```

Estilo tradicional al estilo Lambda.

Forma tradicional

```
interface MathOperation{
    boolean unaryOperation(int num);
}

public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = new MathOperation() {
            @Override
            public boolean unaryOperation(int num) {
                return num%2 == 0;
            }
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}
```

Estilo lambda

1. Eliminar el nombre de la clase y el cuerpo de la interfaz funcional.

```
public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = (int num) -> {
            return num%2 == 0;
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}
```

2. Declaración de tipo opcional

```
MathOperation isEven = (num) -> {
    return num%2 == 0;
};
```

3. Paréntesis opcional alrededor del parámetro, si es un solo parámetro

```
MathOperation isEven = num -> {
    return num%2 == 0;
};
```

4. Tirantes opcionales, si solo hay una línea en el cuerpo de la función
5. Palabra clave de retorno opcional, si solo hay una línea en el cuerpo de la función

```
MathOperation isEven = num -> num%2 == 0;
```

Lambdas y utilización de memoria.

Como los lambdas de Java son cierres, pueden "capturar" los valores de las variables en el ámbito léxico adjunto. Si bien no todos los lambdas capturan nada, los lambdas simples como `s -> s.length()` no capturan nada y se llaman *sin estado* , la captura de los lambdas requiere un objeto temporal para contener las variables capturadas. En este fragmento de código, la lambda `() -> j` es una lambda de captura, y puede hacer que se asigne un objeto cuando se evalúa:

```
public static void main(String[] args) throws Exception {
    for (int i = 0; i < 1000000000; i++) {
        int j = i;
        doSomethingWithLambda(() -> j);
    }
}
```

Aunque podría no ser inmediatamente obvio, ya que la `new` palabra clave no aparece en ningún lugar del fragmento de código, este código puede crear 1,000,000,000 objetos separados para representar las instancias de la expresión `() -> j` lambda. Sin embargo, también se debe tener en cuenta que las versiones futuras de Java [1] pueden ser capaces de optimizar esto para que *en el tiempo de ejecución* las instancias de lambda se reutilizaran o se representaran de alguna otra manera.

---

1 - Por ejemplo, Java 9 introduce una fase de "enlace" opcional a la secuencia de compilación de Java que brindará la oportunidad de realizar optimizaciones globales como esta.

Usar expresiones y predicados lambda para obtener un determinado valor (es) de una lista

A partir de Java 8, puede utilizar expresiones y predicados lambda.

Ejemplo: Use una expresión lambda y un predicado para obtener un determinado valor de una lista. En este ejemplo, todas las personas se imprimirán con el hecho si tienen 18 años o más o no.

Clase de persona:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int getAge() { return age; }
    public String getName() { return name; }
}
```

El Predicado de la interfaz incorporada de los paquetes java.util.function.Predicate es una interfaz

funcional con un método de `boolean test(T t)` .

Ejemplo de uso:

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class LambdaExample {
    public static void main(String[] args) {
        List<Person> personList = new ArrayList<Person>();
        personList.add(new Person("Jeroen", 20));
        personList.add(new Person("Jack", 5));
        personList.add(new Person("Lisa", 19));

        print(personList, p -> p.getAge() >= 18);
    }

    private static void print(List<Person> personList, Predicate<Person> checker) {
        for (Person person : personList) {
            if (checker.test(person)) {
                System.out.print(person + " matches your expression.");
            } else {
                System.out.println(person  + " doesn't match your expression.");
            }
        }
    }
}
```

The `print(personList, p -> p.getAge() >= 18);` el método toma una expresión lambda (porque el Predicado se usa como un parámetro) donde puede definir la expresión que se necesita. El método de prueba del verificador comprueba si esta expresión es correcta o no: `checker.test(person)` .

Puede cambiar esto fácilmente a otra cosa, por ejemplo, para `print(personList, p -> p.getName().startsWith("J"));` . Esto verificará si el nombre de la persona comienza con una "J".

Lea Expresiones lambda en línea: https://riptutorial.com/es/java/topic/91/expresiones-lambda

Capítulo 80: Expresiones regulares

Introducción

Una expresión regular es una secuencia especial de caracteres que ayuda a hacer coincidir o encontrar otras cadenas o conjuntos de cadenas, utilizando una sintaxis especializada contenida en un patrón. Java tiene soporte para el uso de expresiones regulares a través del paquete `java.util.regex` . Este tema es presentar y ayudar a los desarrolladores a comprender más con ejemplos sobre cómo se deben usar las expresiones regulares en Java.

Sintaxis

- Pattern patternName = Pattern.compile (regex);
- Matcher MatcherName = patternName.matcher (textToSearch);
- matcherName.matches () // Devuelve true si el textToSearch coincide exactamente con la expresión regular
- matcherName.find () // Busca en textToSearch la primera instancia de una subcadena que coincida con la expresión regular. Las llamadas subsiguientes buscarán el resto de la cadena.
- matcherName.group (groupNum) // Devuelve la subcadena dentro de un grupo de captura
- matcherName.group (groupName) // Devuelve la subcadena dentro de un grupo de captura con nombre (Java 7+)

Observaciones

Importaciones

Deberá agregar las siguientes importaciones antes de poder usar Regex:

```
import java.util.regex.Matcher
import java.util.regex.Pattern
```

Escollos

En java, una barra diagonal inversa se escapa con una barra diagonal inversa doble, por lo que una barra diagonal inversa en la cadena de expresiones regulares debe ingresarse como una barra diagonal inversa doble. Si necesita escapar de una barra diagonal inversa doble (para hacer coincidir una barra diagonal inversa única con la expresión regular, debe ingresarla como una barra diagonal inversa cuádruple).

Símbolos importantes explicados

Personaje	Descripción
`*`	Coincidir con el carácter o subexpresión anterior 0 o más veces
`+`	Coincidir con el carácter o subexpresión anterior 1 o más veces
`?`	Coincidir con el carácter o subexpresión anterior 0 o 1 veces

Otras lecturas

El tema de expresiones regulares contiene más información sobre expresiones regulares.

Examples

Utilizando grupos de captura

Si necesita extraer una parte de la cadena de la cadena de entrada, podemos usar **grupos** de **captura** de expresiones regulares.

Para este ejemplo, comenzaremos con una expresión regular de número de teléfono simple:

```
\d{3}-\d{3}-\d{4}
```

Si se agregan paréntesis a la expresión regular, cada conjunto de paréntesis se considera un *grupo de captura* . En este caso, estamos usando lo que se llama grupos de captura numerados:

```
(\d{3})-(\d{3})-(\d{4})
^-----^ ^-----^ ^-----^
Group 1 Group 2 Group 3
```

Antes de que podamos usarlo en Java, no debemos olvidar seguir las reglas de las Cadenas, escapando de las barras invertidas, resultando en el siguiente patrón:

```
"(\\d{3})-(\\d{3})-(\\d{4})"
```

Primero necesitamos compilar el patrón de expresiones regulares para hacer un `Pattern` y luego necesitamos un `Matcher` para hacer coincidir nuestra cadena de entrada con el patrón:

```
Pattern phonePattern = Pattern.compile("(\\d{3})-(\\d{3})-(\\d{4})");
Matcher phoneMatcher = phonePattern.matcher("abcd800-555-1234wxyz");
```

A continuación, el Matcher debe encontrar la primera subsecuencia que coincida con la expresión

regular:

```
phoneMatcher.find();
```

Ahora, usando el método de grupo, podemos extraer los datos de la cadena:

```
String number = phoneMatcher.group(0); //"800-555-1234" (Group 0 is everything the regex
matched)
String aCode = phoneMatcher.group(1); //"800"
String threeDigit = phoneMatcher.group(2); //"555"
String fourDigit = phoneMatcher.group(3); //"1234"
```

Nota: `Matcher.group()` puede usarse en lugar de `Matcher.group(0)` .

Java SE 7

Java 7 introdujo los grupos de captura nombrados. Los grupos de captura nombrados funcionan igual que los grupos de captura numerados (pero con un nombre en lugar de un número), aunque hay cambios leves de sintaxis. El uso de grupos de captura con nombre mejora la legibilidad.

Podemos alterar el código anterior para usar grupos nombrados:

```
(?<AreaCode>\d{3})-(\d{3})-(\d{4})
^----------------^ ^-----^ ^-----^
AreaCode           Group 2 Group 3
```

Para obtener los contenidos de "AreaCode", podemos usar:

```
String aCode = phoneMatcher.group("AreaCode"); //"800"
```

Usando expresiones regulares con comportamiento personalizado compilando el patrón con banderas

Un `Pattern` se puede compilar con indicadores, si la expresión regular se utiliza como una `String` literal, use modificadores en línea:

```
Pattern pattern = Pattern.compile("foo.", Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
pattern.matcher("FOO\n").matches(); // Is true.

/* Had the regex not been compiled case insensitively and singlelined,
 * it would fail because FOO does not match /foo/ and \n (newline)
 * does not match /./.
 */

Pattern anotherPattern = Pattern.compile("(?si)foo");
anotherPattern.matcher("FOO\n").matches(); // Is true.

"foOt".replaceAll("(?si)foo", "ca"); // Returns "cat".
```

Personajes de escape

Generalmente

Para usar caracteres específicos de expresiones regulares (`?+|` Etc.) en su significado literal, deben escaparse. En expresiones regulares comunes esto se hace mediante una barra invertida `\` . Sin embargo, como tiene un significado especial en Java Strings, debes usar una doble barra invertida `\\` .

Estos dos ejemplos no funcionarán:

```
"???".replaceAll ("?", "!"); //java.util.regex.PatternSyntaxException
"???".replaceAll ("\?", "!"); //Invalid escape sequence
```

Este ejemplo funciona

```
"???".replaceAll ("\\?", "!"); //"!!!"
```

División de una cadena delimitada por tuberías

Esto no devuelve el resultado esperado:

```
"a|b".split ("|"); // [a, |, b]
```

Esto devuelve el resultado esperado:

```
"a|b".split ("\\|"); // [a, b]
```

Escape de barra invertida `\`

Esto dará un error:

```
"\\".matches("\\"); // PatternSyntaxException
"\\".matches("\\\"); // Syntax Error
```

Esto funciona:

```
"\\".matches("\\\\"); // true
```

Coincidencia con un regex literal.

Si necesita hacer coincidir caracteres que forman parte de la sintaxis de expresiones regulares, puede marcar todo o parte del patrón como un literal de expresiones regulares.

`\Q` marca el comienzo del literal regex. `\E` marca el final de la expresión regular.

```
// the following throws a PatternSyntaxException because of the un-closed bracket
"[123".matches("[123");

// wrapping the bracket in \Q and \E allows the pattern to match as you would expect.
"[123".matches("\\Q[\\E123"); // returns true
```

Una forma más fácil de hacerlo sin tener que recordar las secuencias de escape `\Q` y `\E` es usar `Pattern.quote()`

```
"[123".matches(Pattern.quote("[") + "123"); // returns true
```

No coincide con una cadena dada

Para hacer coincidir algo que *no* contiene una cadena dada, uno puede usar lookahead negativo:

Sintaxis de Regex: `(?!string-to-not-match)`

Ejemplo:

```
//not matching "popcorn"
String regexString = "^(?!popcorn).*$";
System.out.println("[popcorn] " + ("popcorn".matches(regexString) ? "matched!" : "nope!"));
System.out.println("[unicorn] " + ("unicorn".matches(regexString) ? "matched!" : "nope!"));
```

Salida:

```
[popcorn] nope!
[unicorn] matched!
```

A juego con una barra invertida

Si quieres hacer coincidir una barra invertida en tu expresión regular, tendrás que escapar de ella.

La barra invertida es un carácter de escape en expresiones regulares. Puede usar '\\' para referirse a una sola barra invertida en una expresión regular.

Sin embargo, la barra invertida *también* es un carácter de escape en las cadenas literales de Java. Para hacer una expresión regular a partir de una cadena literal, debes escapar de cada una de *sus* barras invertidas. En una cadena, se puede usar el literal '\\\\' para crear una expresión regular con '\\', que a su vez puede coincidir con '\'.

Por ejemplo, considere cadenas coincidentes como "C: \ dir \ myfile.txt". Una expresión regular `([A-Za-z]):\\(.*)` Coincidirá y proporcionará la letra de la unidad como un grupo de captura. Tenga en cuenta la barra invertida doble.

Para expresar ese patrón en un literal de cadena de Java, cada una de las barras invertidas en la expresión regular debe escaparse.

```
    String path = "C:\\dir\\myfile.txt";
    System.out.println( "Local path: " + path ); // "C:\dir\myfile.txt"

    String regex = "([A-Za-z]):\\\\.*"; // Four to match one
    System.out.println("Regex:      " + regex ); // "([A-Za-z]):\\(.*)"

    Pattern pattern = Pattern.compile( regex );
    Matcher matcher = pattern.matcher( path );
    if ( matcher.matches()) {
```

```
        System.out.println( "This path is on drive " + matcher.group( 1 ) + ":.");
        // This path is on drive C:.
    }
```

Si desea hacer coincidir *dos* barras diagonales inversas, se encontrará usando ocho en una cadena literal, para representar cuatro en la expresión regular, para hacer coincidir dos.

```
    String path = "\\\\myhost\\share\\myfile.txt";
    System.out.println( "UNC path: " + path ); // \\myhost\share\myfile.txt"

    String regex = "\\\\\\\\(.+?)\\\\(.*)"; // Eight to match two
    System.out.println("Regex:    " + regex ); // \\\\(.+?)\\(.*)

    Pattern pattern = Pattern.compile( regex );
    Matcher matcher = pattern.matcher( path );

    if ( matcher.matches()) {
        System.out.println( "This path is on host '" + matcher.group( 1 ) + "'.");
        // This path is on host 'myhost'.
    }
```

Lea Expresiones regulares en línea: https://riptutorial.com/es/java/topic/135/expresiones-regulares

Capítulo 81: Fechas y hora (java.time. *)

Examples

Manipulaciones de fecha simple

Obtén la fecha actual.

```
LocalDate.now()
```

Consigue la fecha de ayer.

```
LocalDate y = LocalDate.now().minusDays(1);
```

Conseguir la fecha de mañana

```
LocalDate t = LocalDate.now().plusDays(1);
```

Consigue una fecha específica.

```
LocalDate t = LocalDate.of(1974, 6, 2, 8, 30, 0, 0);
```

Además de los métodos de `plus` y `minus` , hay un conjunto de métodos "con" que se pueden usar para establecer un campo particular en una instancia de `LocalDate` .

```
LocalDate.now().withMonth(6);
```

El ejemplo anterior devuelve una nueva instancia con el mes establecido en junio (esto difiere de `java.util.Date` donde `setMonth` fue indexado a 0 hasta el 5 de junio).

Debido a que las manipulaciones de LocalDate devuelven instancias inmutables de LocalDate, estos métodos también se pueden encadenar.

```
LocalDate ld = LocalDate.now().plusDays(1).plusYears(1);
```

Esto nos daría la fecha de mañana dentro de un año.

Fecha y hora

Fecha y hora sin información de zona horaria

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 27, 8, 0);
LocalDateTime now = LocalDateTime.now();
LocalDateTime parsed = LocalDateTime.parse("2016-07-27T07:00:00");
```

Fecha y hora con información de zona horaria.

```
ZoneId zoneId = ZoneId.of("UTC+2");
ZonedDateTime dateTime = ZonedDateTime.of(2016, Month.JULY, 27, 7, 0, 0, 235, zoneId);
ZonedDateTime composition = ZonedDateTime.of(localDate, localTime, zoneId);
ZonedDateTime now = ZonedDateTime.now(); // Default time zone
ZonedDateTime parsed = ZonedDateTime.parse("2016-07-27T07:00:00+01:00[Europe/Stockholm]");
```

Fecha y hora con información de compensación (es decir, no se tienen en cuenta los cambios de horario de verano)

```
ZoneOffset zoneOffset = ZoneOffset.ofHours(2);
OffsetDateTime dateTime = OffsetDateTime.of(2016, 7, 27, 7, 0, 0, 235, zoneOffset);
OffsetDateTime composition = OffsetDateTime.of(localDate, localTime, zoneOffset);
OffsetDateTime now = OffsetDateTime.now(); // Offset taken from the default ZoneId
OffsetDateTime parsed = OffsetDateTime.parse("2016-07-27T07:00:00+02:00");
```

Operaciones en fechas y horarios.

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
LocalDateTime anHourFromNow = LocalDateTime.now().plusHours(1);
Long daysBetween = java.time.temporal.ChronoUnit.DAYS.between(LocalDate.now(),
LocalDate.now().plusDays(3)); // 3
Duration duration = Duration.between(Instant.now(), ZonedDateTime.parse("2016-07-
27T07:00:00+01:00[Europe/Stockholm]"))
```

Instante

Representa un instante en el tiempo. Puede considerarse como una envoltura alrededor de una marca de tiempo Unix.

```
Instant now = Instant.now();
Instant epoch1 = Instant.ofEpochMilli(0);
Instant epoch2 = Instant.parse("1970-01-01T00:00:00Z");
java.time.temporal.ChronoUnit.MICROS.between(epoch1, epoch2); // 0
```

Uso de varias clases de Date Time API

El siguiente ejemplo también tiene una explicación requerida para comprender el ejemplo dentro de él.

```
import java.time.Clock;
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.TimeZone;
public class SomeMethodsExamples {
```

```
/**
 * Has the methods of the class {@link LocalDateTime}
 */
public static void checkLocalDateTime() {
    LocalDateTime localDateTime = LocalDateTime.now();
    System.out.println("Local Date time using static now() method ::: >>> "
            + localDateTime);

    LocalDateTime ldt1 = LocalDateTime.now(ZoneId.of(ZoneId.SHORT_IDS
            .get("AET")));
    System.out
            .println("LOCAL TIME USING now(ZoneId zoneId) method ::: >>>>"
                    + ldt1);

    LocalDateTime ldt2 = LocalDateTime.now(Clock.system(ZoneId
            .of(ZoneId.SHORT_IDS.get("PST"))));
    System.out
            .println("Local TIME USING now(Clock.system(ZoneId.of())) ::: >>>> "
                    + ldt2);

    System.out
            .println("Following is a static map in ZoneId class which has mapping of short
timezone names to their Actual timezone names");
    System.out.println(ZoneId.SHORT_IDS);

}

/**
 * This has the methods of the class {@link LocalDate}
 */
public static void checkLocalDate() {
    LocalDate localDate = LocalDate.now();
    System.out.println("Gives date without Time using now() method. >> "
            + localDate);
    LocalDate localDate2 = LocalDate.now(ZoneId.of(ZoneId.SHORT_IDS
            .get("ECT")));
    System.out
            .println("now() is overridden to take ZoneID as parametere using this we can get
the same date under different timezones. >> "
                    + localDate2);
}

/**
 * This has the methods of abstract class {@link Clock}. Clock can be used
 * for time which has time with {@link TimeZone}.
 */
public static void checkClock() {
    Clock clock = Clock.systemUTC();
    // Represents time according to ISO 8601
    System.out.println("Time using Clock class : " + clock.instant());
}

/**
 * This has the {@link Instant} class methods.
 */
public static void checkInstant() {
    Instant instant = Instant.now();

    System.out.println("Instant using now() method :: " + instant);

    Instant ins1 = Instant.now(Clock.systemUTC());
```

```
    System.out.println("Instants using now(Clock clock) :: " + ins1);

}

/**
 * This class checks the methods of the {@link Duration} class.
 */
public static void checkDuration() {
    // toString() converts the duration to PTnHnMnS format according to ISO
    // 8601 standard. If a field is zero its ignored.

    // P is the duration designator (historically called "period") placed at
    // the start of the duration representation.
    // Y is the year designator that follows the value for the number of
    // years.
    // M is the month designator that follows the value for the number of
    // months.
    // W is the week designator that follows the value for the number of
    // weeks.
    // D is the day designator that follows the value for the number of
    // days.
    // T is the time designator that precedes the time components of the
    // representation.
    // H is the hour designator that follows the value for the number of
    // hours.
    // M is the minute designator that follows the value for the number of
    // minutes.
    // S is the second designator that follows the value for the number of
    // seconds.

    System.out.println(Duration.ofDays(2));
}

/**
 * Shows Local time without date. It doesn't store or represenet a date and
 * time. Instead its a representation of Time like clock on the wall.
 */
public static void checkLocalTime() {
    LocalTime localTime = LocalTime.now();
    System.out.println("LocalTime :: " + localTime);
}

/**
 * A date time with Time zone details in ISO-8601 standards.
 */
public static void checkZonedDateTime() {
    ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId
            .of(ZoneId.SHORT_IDS.get("CST")));
    System.out.println(zonedDateTime);

}
}
```

Formato de fecha y hora

Antes de Java 8, existían las clases `DateFormat` y `SimpleDateFormat` en el paquete `java.text` y este código heredado se seguirá utilizando durante algún tiempo.

Pero, Java 8 ofrece un enfoque moderno para manejar el formateo y el análisis.

Al formatear y analizar primero, pasa un objeto `String` a `DateTimeFormatter` y, a su vez, lo utiliza para formatear o analizar.

```
import java.time.*;
import java.time.format.*;

class DateTimeFormat
{
    public static void main(String[] args) {

        //Parsing
        String pattern = "d-MM-yyyy HH:mm";
        DateTimeFormatter dtF1 = DateTimeFormatter.ofPattern(pattern);

        LocalDateTime ldp1 = LocalDateTime.parse("2014-03-25T01:30"), //Default format
                      ldp2 = LocalDateTime.parse("15-05-2016 13:55",dtF1); //Custom format

        System.out.println(ldp1 + "\n" + ldp2); //Will be printed in Default format

        //Formatting
        DateTimeFormatter dtF2 = DateTimeFormatter.ofPattern("EEE d, MMMM, yyyy HH:mm");

        DateTimeFormatter dtF3 = DateTimeFormatter.ISO_LOCAL_DATE_TIME;

        LocalDateTime ldtf1 = LocalDateTime.now();

        System.out.println(ldtf1.format(dtF2) +"\n"+ldtf1.format(dtF3));
    }
}
```

Un aviso importante, en lugar de usar patrones personalizados, es una buena práctica usar formateadores predefinidos. Su código se ve más claro y el uso de ISO8061 definitivamente lo ayudará a largo plazo.

Calcular la diferencia entre 2 fechas locales

Use `LocalDate` y `ChronoUnit` :

```
LocalDate d1 = LocalDate.of(2017, 5, 1);
LocalDate d2 = LocalDate.of(2017, 5, 18);
```

ahora, ya que el método `between` el enumerador `ChronoUnit` toma 2 `Temporal` s como parámetros para que pueda pasar sin problemas las instancias de `LocalDate`

```
long days = ChronoUnit.DAYS.between(d1, d2);
System.out.println( days );
```

Lea Fechas y hora (java.time. *) en línea: https://riptutorial.com/es/java/topic/4813/fechas-y-hora--java-time----

Capítulo 82: FileUpload a AWS

Introducción

Cargue el archivo a AWS s3 bucket utilizando la API spring rest.

Examples

Subir archivo a s3 bucket

> Aquí crearemos un APi de descanso que tomará el objeto de archivo como un parámetro de varias partes desde el front-end y lo subiremos a S3 bucket utilizando la API de Java Rest.

Requisito : - secreta la clave y la clave de acceso para el cubo s3 donde desea cargar su archivo.

código: - DocumentController.java

```
@RestController
@RequestMapping("/api/v2")
public class DocumentController {

    private static String bucketName = "pharmerz-chat";
    //   private static String keyName        = "Pharmerz"+ UUID.randomUUID();

    @RequestMapping(value = "/upload", method = RequestMethod.POST, consumes =
MediaType.MULTIPART_FORM_DATA)
    public URL uploadFileHandler(@RequestParam("name") String name,
                                 @RequestParam("file") MultipartFile file) throws IOException
{

/******* Printing all the possible parameter from @RequestParam *************/

        System.out.println("****************************");

        System.out.println("file.getOriginalFilename() " + file.getOriginalFilename());
        System.out.println("file.getContentType()" + file.getContentType());
        System.out.println("file.getInputStream() " + file.getInputStream());
        System.out.println("file.toString() " + file.toString());
        System.out.println("file.getSize() " + file.getSize());
        System.out.println("name " + name);
        System.out.println("file.getBytes() " + file.getBytes());
        System.out.println("file.hashCode() " + file.hashCode());
        System.out.println("file.getClass() " + file.getClass());
        System.out.println("file.isEmpty() " + file.isEmpty());

        /*************Parameters to b pass to s3 bucket put Object **************/
        InputStream is = file.getInputStream();
        String keyName = file.getOriginalFilename();


// Credentials for Aws
```

```
        AWSCredentials credentials = new BasicAWSCredentials("AKIA*************",
"zr***********************");

        /****************** DocumentController.uploadfile(credentials);
***************************/


        AmazonS3 s3client = new AmazonS3Client(credentials);
        try {
            System.out.println("Uploading a new object to S3 from a file\n");
            //File file = new File(awsuploadfile);
            s3client.putObject(new PutObjectRequest(
                    bucketName, keyName, is, new ObjectMetadata()));


            URL url = s3client.generatePresignedUrl(bucketName, keyName,
Date.from(Instant.now().plus(5, ChronoUnit.MINUTES)));
            // URL url=s3client.generatePresignedUrl(bucketName,keyName,
Date.from(Instant.now().plus(5, ChronoUnit.)));
            System.out.println("************************************");
            System.out.println(url);

            return url;

        } catch (AmazonServiceException ase) {
            System.out.println("Caught an AmazonServiceException, which " +
                    "means your request made it " +
                    "to Amazon S3, but was rejected with an error response" +
                    " for some reason.");
            System.out.println("Error Message:    " + ase.getMessage());
            System.out.println("HTTP Status Code: " + ase.getStatusCode());
            System.out.println("AWS Error Code:   " + ase.getErrorCode());
            System.out.println("Error Type:       " + ase.getErrorType());
            System.out.println("Request ID:       " + ase.getRequestId());
        } catch (AmazonClientException ace) {
            System.out.println("Caught an AmazonClientException, which " +
                    "means the client encountered " +
                    "an internal error while trying to " +
                    "communicate with S3, " +
                    "such as not being able to access the network.");
            System.out.println("Error Message: " + ace.getMessage());
        }

        return null;

    }


}
```

Función frontal

```
var form = new FormData();
form.append("file", "image.jpeg");

var settings = {
 "async": true,
 "crossDomain": true,
 "url": "http://url/",
 "method": "POST",
```

```
 "headers": {
   "cache-control": "no-cache"
 },
 "processData": false,
 "contentType": false,
 "mimeType": "multipart/form-data",
 "data": form
}

$.ajax(settings).done(function (response) {
 console.log(response);
});
```

Lea FileUpload a AWS en línea: https://riptutorial.com/es/java/topic/10589/fileupload-a-aws

Capítulo 83: Formato numérico

Examples

Formato numérico

Los diferentes países tienen diferentes formatos numéricos y, teniendo en cuenta esto, podemos tener diferentes formatos utilizando Locale of java. Usar locale puede ayudar en el formateo

```
Locale locale = new Locale("en", "IN");
NumberFormat numberFormat = NumberFormat.getInstance(locale);
```

Usando el formato anterior puede realizar varias tareas

1. Número de formato

   ```
   numberFormat.format(10000000.99);
   ```

2. Formato de moneda

   ```
   NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
   currencyFormat.format(10340.999);
   ```

3. Formato Porcentaje

   ```
   NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
   percentageFormat.format(10929.999);
   ```

4. Número de control de dígitos

```
numberFormat.setMinimumIntegerDigits(int digits)
numberFormat.setMaximumIntegerDigits(int digits)
numberFormat.setMinimumFractionDigits(int digits)
numberFormat.setMaximumFractionDigits(int digits)
```

Lea Formato numérico en línea: https://riptutorial.com/es/java/topic/7399/formato-numerico

Capítulo 84: FTP (Protocolo de transferencia de archivos)

Sintaxis

- FTPClient connect (host InetAddress, puerto int)
- Inicio de sesión de FTPClient (nombre de usuario de String, contraseña de String)
- FTPClient desconectar ()
- FTPReply getReplyStrings ()
- boolean storeFile (cadena remota, InputStream local)
- OutputStream storeFileStream (cadena remota)
- boolean setFileType (int fileType)
- boolean completePendingCommand ()

Parámetros

Parámetros	Detalles
anfitrión	El nombre de host o la dirección IP del servidor FTP
Puerto	El puerto del servidor FTP
nombre de usuario	El nombre de usuario del servidor FTP
contraseña	La contraseña del servidor FTP

Examples

Conexión e inicio de sesión en un servidor FTP

Para comenzar a usar FTP con Java, deberá crear un nuevo `FTPClient` y luego conectarse e iniciar sesión en el servidor utilizando `.connect(String server, int port)` y `.login(String username, String password)` .

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;
//Import all the required resource for this project.

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com"; //Server can be either host name or IP address.
        int port = 21;
        String user = "Username";
```

```
        String pass = "Password";

        FTPClient ftp = new FTPClient;
        ftp.connect(server, port);
        ftp.login(user, pass);
    }
}
```

Ahora tenemos lo básico hecho. ¿Pero qué pasa si tenemos un error de conexión al servidor? Queremos saber cuándo algo sale mal y aparece el mensaje de error. Agreguemos algo de código para detectar errores mientras se conecta.

```
try {
    ftp.connect(server, port);
    showServerReply(ftp);
    int replyCode = ftp.getReplyCode();
    if (!FTPReply.isPositiveCompletion(replyCode)) {
        System.out.printIn("Operation failed. Server reply code: " + replyCode)
        return;
    }
    ftp.login(user, pass);
} catch {

}
```

Vamos a desglosar lo que acabamos de hacer, paso a paso.

```
showServerReply(ftp);
```

Esto se refiere a una función que haremos en un paso posterior.

```
int replyCode = ftp.getReplyCode();
```

Esto toma el código de respuesta / error del servidor y lo almacena como un entero.

```
if (!FTPReply.isPositiveCompletion(replyCode)) {
    System.out.printIn("Operation failed. Server reply code: " + replyCode)
    return;
}
```

Esto verifica el código de respuesta para ver si hubo un error. Si hubo un error, simplemente imprimirá "Error en la operación. Código de respuesta del servidor:" seguido del código de error. También agregamos un bloque try / catch que agregaremos en el siguiente paso. A continuación, también creamos una función que verifique si `ftp.login()` errores en `ftp.login()` .

```
boolean success = ftp.login(user, pass);
showServerReply(ftp);
if (!success) {
    System.out.println("Failed to log into the server");
        return;
    } else {
        System.out.println("LOGGED IN SERVER");
    }
```

Vamos a romper este bloque también.

```
boolean success = ftp.login(user, pass);
```

Esto no solo intentará iniciar sesión en el servidor FTP, sino que también almacenará el resultado como un valor booleano.

```
showServerReply(ftp);
```

Esto comprobará si el servidor nos envió algún mensaje, pero primero tendremos que crear la función en el siguiente paso.

```
if (!success) {
System.out.println("Failed to log into the server");
    return;
} else {
    System.out.println("LOGGED IN SERVER");
}
```

Esta declaración comprobará si iniciamos sesión correctamente; si es así, se imprimirá "LOGGED IN SERVER", de lo contrario se imprimirá "Error al iniciar sesión en el servidor". Este es nuestro guión hasta ahora:

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {
                    System.out.println("Operation failed. Server reply code: " + replyCode);
                    return;
                }
            boolean success = ftp.login(user, pass);
            showServerReply(ftp);
            if (!success) {
                System.out.println("Failed to log into the server");
                return;
            } else {
                System.out.println("LOGGED IN SERVER");
            }
        } catch {

        }
```

```
    }
}
```

Ahora, a continuación, vamos a crear el bloque Catch completo en caso de que encontremos algún error en todo el proceso.

```
} catch (IOException ex) {
    System.out.println("Oops! Something went wrong.");
    ex.printStackTrace();
}
```

El bloque de captura completado ahora se imprimirá "¡Vaya! Algo salió mal". y el stacktrace si hay un error. Ahora nuestro último paso es crear el `showServerReply()` que hemos estado usando por un tiempo.

```
private static void showServerReply(FTPClient ftp) {
    String[] replies = ftp.getReplyStrings();
    if (replies != null && replies.length > 0) {
    for (String aReply : replies) {
        System.out.println("SERVER: " + aReply);
        }
    }
}
```

Esta función toma un `FTPClient` como una variable, y lo llama "ftp". Después de eso, almacena todas las respuestas del servidor desde una matriz de cadenas. A continuación, comprueba si los mensajes fueron almacenados. Si hay alguno, imprime cada uno de ellos como "SERVIDOR: [respuesta]". Ahora que hemos terminado esa función, este es el script completo:

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        String[] replies = ftp.getReplyStrings();
        if (replies != null && replies.length > 0) {
        for (String aReply : replies) {
            System.out.println("SERVER: " + aReply);
            }
        }
    }

    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
```

```
            if (!FTPReply.isPositiveCompletion(replyCode)) {
                System.out.println("Operation failed. Server reply code: " + replyCode);
                return;
            }
            boolean success = ftp.login(user, pass);
            showServerReply(ftp);
            if (!success) {
                System.out.println("Failed to log into the server");
                return;
            } else {
                System.out.println("LOGGED IN SERVER");
            }
        } catch (IOException ex) {
            System.out.println("Oops! Something went wrong.");
            ex.printStackTrace();
        }
    }
}
```

Primero debemos crear un nuevo `FTPClient` e intentar conectarlo al servidor e iniciar sesión usando `.connect(String server, int port)` y `.login(String username, String password)` . Es importante conectarse e iniciar sesión con un bloque try / catch en caso de que nuestro código no pueda conectarse con el servidor. También necesitaremos crear una función que verifique y muestre cualquier mensaje que recibamos del servidor cuando intentemos conectarnos e iniciar sesión. Llamaremos a esta función " `showServerReply(FTPClient ftp)` ".

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        if (replies != null && replies.length > 0) {
        for (String aReply : replies) {
            System.out.println("SERVER: " + aReply);
            }
        }
    }

public static void main(String[] args) {
    // SET THESE TO MATCH YOUR FTP SERVER //
    String server = "www.server.com";
    int port = 21;
    String user = "username"
    String pass = "password"

FTPClient ftp = new FTPClient
try {
    ftp.connect(server, port)
    showServerReply(ftp);
    int replyCode = ftpClient.getReplyCode();
    if (!FTPReply.isPositiveCompletion(replyCode)) {
            System.out.println("Operation failed. Server reply code: " + replyCode);
            return;
        }
    boolean success = ftp.login(user, pass);
    showServerReply(ftp);
    if (!success) {
```

```
        System.out.println("Failed to log into the server");
            return;
        } else {
            System.out.println("LOGGED IN SERVER");
        }
    } catch (IOException ex) {
        System.out.println("Oops! Something went wrong.");
        ex.printStackTrace();
        }
    }
}
```

Después de esto, ahora debería tener su servidor FTP conectado a su script Java.

Lea FTP (Protocolo de transferencia de archivos) en línea:
https://riptutorial.com/es/java/topic/5228/ftp--protocolo-de-transferencia-de-archivos-

Capítulo 85: Generación de números aleatorios

Observaciones

Nada es realmente aleatorio y, por lo tanto, el javadoc llama a esos números pseudoaleatorios. Esos números se crean con un generador de números pseudoaleatorios .

Examples

Números pseudoaleatorios

Java proporciona, como parte del paquete `utils` , un generador básico de números pseudoaleatorios, denominado apropiadamente `Random` . Este objeto se puede usar para generar un valor pseudoaleatorio como cualquiera de los tipos de datos numéricos incorporados (`int` , `float` , etc.). También puede usarlo para generar un valor booleano aleatorio, o una matriz aleatoria de bytes. Un ejemplo de uso es el siguiente:

```
import java.util.Random;

...

Random random = new Random();
int randInt = random.nextInt();
long randLong = random.nextLong();

double randDouble = random.nextDouble(); //This returns a value between 0.0 and 1.0
float randFloat = random.nextFloat(); //Same as nextDouble

byte[] randBytes = new byte[16];
random.nextBytes(randBytes); //nextBytes takes a user-supplied byte array, and fills it with
random bytes. It returns nothing.
```

NOTA: Esta clase solo produce números pseudoaleatorios de baja calidad, y nunca debe usarse para generar números aleatorios para operaciones criptográficas u otras situaciones en las que la aleatoriedad de mayor calidad es crítica (para eso, desearía usar la clase `SecureRandom` , como se indica a continuación). Una explicación para la distinción entre aleatoriedad "segura" e "insegura" está fuera del alcance de este ejemplo.

Números pseudoaleatorios en rango específico

El método `nextInt(int bound)` de `Random` acepta un límite exclusivo superior, es decir, un número que el valor aleatorio devuelto debe ser menor que. Sin embargo, solo el método `nextInt` acepta un límite; `nextLong` , `nextDouble` etc no lo hacen.

```
Random random = new Random();
random.nextInt(1000); // 0 - 999
```

```
int number = 10 + random.nextInt(100); // number is in the range of 10 to 109
```

A partir de Java 1.7, también puede utilizar `ThreadLocalRandom` (fuente). Esta clase proporciona un PRNG seguro para subprocesos (generador de números pseudoaleatorios). Tenga en cuenta que el método `nextInt` de esta clase acepta un límite superior e inferior.

```
import java.util.concurrent.ThreadLocalRandom;

// nextInt is normally exclusive of the top value,
// so add 1 to make it inclusive
ThreadLocalRandom.current().nextInt(min, max + 1);
```

Tenga en cuenta que la documentación oficial indica que `nextInt(int bound)` puede hacer cosas raras cuando el `bound` está cerca de $2^{30}+1$ (énfasis agregado):

> El algoritmo es un poco complicado. **Rechaza los valores que darían como resultado una distribución desigual** (debido al hecho de que 2 ^ 31 no es divisible por n). La probabilidad de que un valor sea rechazado depende de n. **El caso más desfavorable es n = 2 ^ 30 + 1, para el cual la probabilidad de un rechazo es 1/2, y el número esperado de iteraciones antes de que finalice el bucle es 2.**

En otras palabras, especificar un límite disminuirá (levemente) el rendimiento del método `nextInt` , y esta disminución del rendimiento será más pronunciada a medida que el `bound` aproxime a la mitad del valor int máximo.

Generación de números pseudoaleatorios criptográficamente seguros.

`Random` y `ThreadLocalRandom` son lo suficientemente buenos para el uso diario, pero tienen un gran problema: se basan en un generador lineal congruente , un algoritmo cuya salida se puede predecir con bastante facilidad. Por lo tanto, estas dos clases **no** son adecuadas para usos criptográficos (como la generación de claves).

Uno puede usar `java.security.SecureRandom` en situaciones donde se requiere un PRNG con una salida que sea muy difícil de predecir. Predecir los números aleatorios creados por las instancias de esta clase es lo suficientemente difícil como para etiquetar la clase como **criptográficamente segura** .

```
import java.security.SecureRandom;
import java.util.Arrays;

public class Foo {
    public static void main(String[] args) {
        SecureRandom rng = new SecureRandom();
        byte[] randomBytes = new byte[64];
        rng.nextBytes(randomBytes); // Fills randomBytes with random bytes (duh)
        System.out.println(Arrays.toString(randomBytes));
    }
}
```

Además de ser criptográficamente seguro, `SecureRandom` tiene un período gigantesco de 2^{160} , comparado con el período de `Random` de 2^{48} . Sin embargo, tiene un inconveniente de ser considerablemente más lento que el `Random` y otros PRNG lineales como Mersenne Twister y Xorshift .

Tenga en cuenta que la implementación de SecureRandom depende tanto de la plataforma como del proveedor. El `SecureRandom` predeterminado (dado por el proveedor de `SUN` en `sun.security.provider.SecureRandom`):

- en sistemas similares a Unix, sembrados con datos de `/dev/random` y / o `/dev/urandom` .
- en Windows, sembradas con llamadas a `CryptGenRandom()` en CryptoAPI .

Seleccionar números aleatorios sin duplicados

```
/**
 * returns a array of random numbers with no duplicates
 * @param range the range of possible numbers for ex. if 100 then it can be anywhere from 1-
100
 * @param length the length of the array of random numbers
 * @return array of random numbers with no duplicates.
 */
public static int[] getRandomNumbersWithNoDuplicates(int range, int length){
    if (length<range){
        // this is where all the random numbers
        int[] randomNumbers = new int[length];

        // loop through all the random numbers to set them
        for (int q = 0; q < randomNumbers.length; q++){

            // get the remaining possible numbers
            int remainingNumbers = range-q;

            // get a new random number from the remainingNumbers
            int newRandSpot = (int) (Math.random()*remainingNumbers);

            newRandSpot++;

            // loop through all the possible numbers
            for (int t = 1; t < range+1; t++){

                // check to see if this number has already been taken
                boolean taken = false;
                for (int number : randomNumbers){
                    if (t==number){
                        taken = true;
                        break;
                    }
                }

                // if it hasnt been taken then remove one from the spots
                if (!taken){
                    newRandSpot--;

                    // if we have gone though all the spots then set the value
                    if (newRandSpot==0){
                        randomNumbers[q] = t;
                    }
```

```
                }
            }
        }
        return randomNumbers;
    } else {
        // invalid can't have a length larger then the range of possible numbers
    }
    return null;
}
```

El método funciona en bucle a través de una matriz que tiene el tamaño de la longitud solicitada y encuentra la longitud restante de los números posibles. Establece un número aleatorio de esos posibles números `newRandSpot` y encuentra ese número dentro del número no tomado que queda. Lo hace recorriendo el rango y comprobando si ese número ya se ha tomado.

Por ejemplo, si el rango es 5 y la longitud es 3 y ya hemos elegido el número 2. Luego tenemos 4 números restantes, así que obtenemos un número aleatorio entre 1 y 4 y pasamos por el rango (5) saltando cualquier número que ya hemos utilizado (2).

Ahora digamos que el siguiente número elegido entre 1 y 4 es 3. En el primer bucle obtenemos 1 que aún no se ha tomado, por lo que podemos eliminar 1 de 3, lo que hace que sea 2. Ahora, en el segundo bucle obtenemos 2 que ha sido tomado así que no hacemos nada. Seguimos este patrón hasta que llegamos a 4, donde una vez que eliminamos 1, se convierte en 0, por lo que establecemos el nuevo número aleatorio en 4.

Generando números aleatorios con una semilla especificada

```
//Creates a Random instance with a seed of 12345.
Random random = new Random(12345L);

//Gets a ThreadLocalRandom instance
ThreadLocalRandom tlr = ThreadLocalRandom.current();

//Set the instance's seed.
tlr.setSeed(12345L);
```

Usar la misma semilla para generar números aleatorios devolverá los mismos números cada vez, por lo que establecer una semilla diferente para cada instancia `Random` es una buena idea si no quiere terminar con números duplicados.

Un buen método para obtener un `Long` que es diferente para cada llamada es `System.currentTimeMillis()` :

```
Random random = new Random(System.currentTimeMillis());
ThreadLocalRandom.current().setSeed(System.currentTimeMillis());
```

Generando números aleatorios usando apache-common lang3

Podemos usar `org.apache.commons.lang3.RandomUtils` para generar números aleatorios usando una

sola línea.

```
int x = RandomUtils.nextInt(1, 1000);
```

El método `nextInt(int startInclusive, int endExclusive)` toma un rango.

Aparte de int, podemos generar aleatorios `long` , `double` , `float` y `bytes` utilizando esta clase.

`RandomUtils` clase `RandomUtils` contiene los siguientes métodos:

```
static byte[] nextBytes(int count) //Creates an array of random bytes.
static double nextDouble() //Returns a random double within 0 - Double.MAX_VALUE
static double nextDouble(double startInclusive, double endInclusive) //Returns a random double
within the specified range.
static float nextFloat() //Returns a random float within 0 - Float.MAX_VALUE
static float nextFloat(float startInclusive, float endInclusive) //Returns a random float
within the specified range.
static int nextInt() //Returns a random int within 0 - Integer.MAX_VALUE
static int nextInt(int startInclusive, int endExclusive) //Returns a random integer within the
specified range.
static long nextLong() //Returns a random long within 0 - Long.MAX_VALUE
static long nextLong(long startInclusive, long endExclusive) //Returns a random long within
the specified range.
```

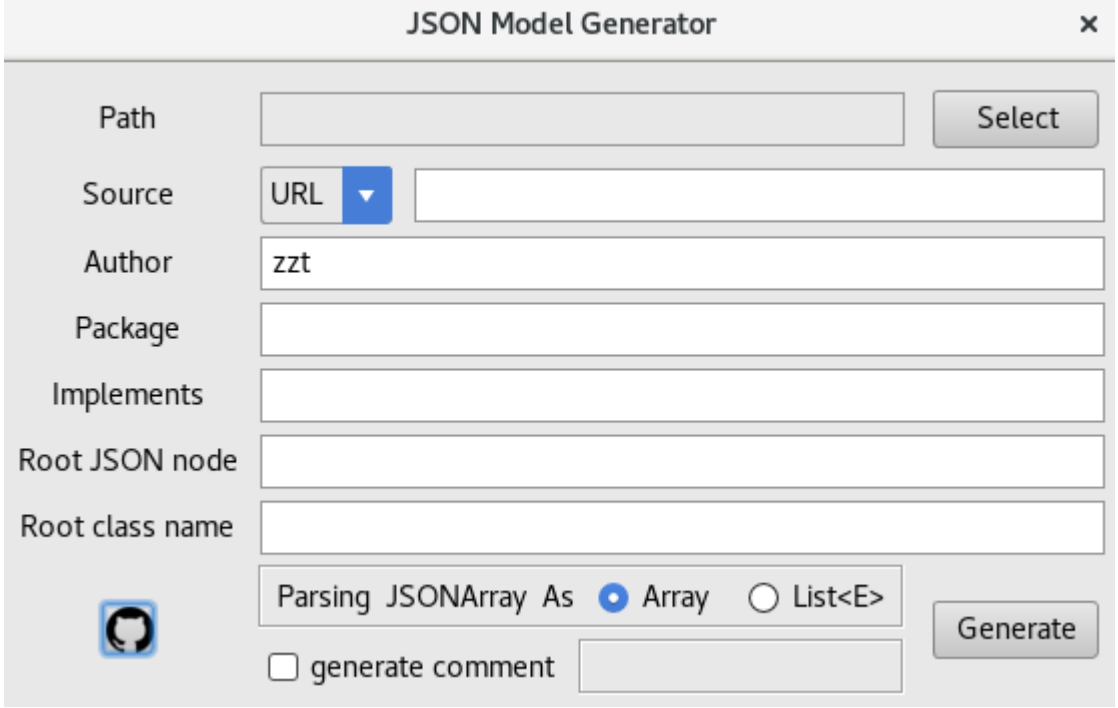
Lea Generación de números aleatorios en línea:
https://riptutorial.com/es/java/topic/890/generacion-de-numeros-aleatorios

Capítulo 86: Generando Código Java

Examples

Generar POJO desde JSON

- Instale el complemento JSON Model Genrator de Intellij buscando en la configuración de Intellij.
- Inicia el plugin desde 'Herramientas'
- Ingrese el campo de UI como se muestra a continuación (se requiere " Ruta " Fuente '、' Paquete '):



- Haga clic en el botón 'Generar' y listo.

Lea Generando Código Java en línea: https://riptutorial.com/es/java/topic/9400/generando-codigo-java

Capítulo 87: Genéricos

Introducción

Los genéricos son una instalación de programación genérica que extienden el sistema de tipos de Java para permitir que un tipo o método opere en objetos de varios tipos mientras proporciona seguridad de tipo de tiempo de compilación. En particular, el marco de colecciones Java es compatible con los genéricos para especificar el tipo de objetos almacenados en una instancia de colección.

Sintaxis

- class ArrayList <E> {} // una clase genérica con parámetro de tipo E
- clase HashMap <K, V> {} // una clase genérica con dos parámetros de tipo K y V
- <E> void print (elemento E) {} // un método genérico con el parámetro de tipo E
- ArrayList <String> nombres; // declaración de una clase genérica
- ArrayList <?> Objetos; // declaración de una clase genérica con un parámetro de tipo desconocido
- nueva ArrayList <String> () // creación de instancias de una clase genérica
- nueva ArrayList <> () // creación de instancias con inferencia de tipo "diamante" (Java 7 o posterior)

Observaciones

Los genéricos se implementan en Java a través del borrado de tipos, lo que significa que durante el tiempo de ejecución la información de tipo especificada en la creación de instancias de una clase genérica no está disponible. Por ejemplo, la declaración `List<String> names = new ArrayList<>();` produce un objeto de lista desde el cual el tipo de elemento `String` no se puede recuperar en tiempo de ejecución. Sin embargo, si la lista se almacena en un campo de tipo `List<String>` , o se pasa a un parámetro de método / constructor de este mismo tipo, o se devuelve desde un método de ese tipo de retorno, entonces la información de tipo completo *se* puede recuperar en tiempo de ejecución a través de la API de reflexión de Java.

Esto también significa que cuando se realiza la conversión a un tipo genérico (por ejemplo: `(List<String>) list`), la conversión es una conversión *sin verificar* . Debido a que el parámetro `<String>` se borra, la JVM no puede verificar si una conversión de una `List<?>` una `List<String>` es correcta; La JVM solo ve un reparto de `List` a `List` en tiempo de ejecución.

Examples

Creando una clase genérica

Los genéricos permiten que las clases, interfaces y métodos tomen otras clases e interfaces como parámetros de tipo.

Este ejemplo utiliza la clase genérica `Param` para tomar un solo **tipo de parámetro** `T` , delimitado por corchetes angulares (`<>`):

```
public class Param<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

Para crear una instancia de esta clase, proporcione un **argumento de tipo** en lugar de `T` Por ejemplo, `Integer` :

```
Param<Integer> integerParam = new Param<Integer>();
```

El argumento de tipo puede ser cualquier tipo de referencia, incluidas las matrices y otros tipos genéricos:

```
Param<String[]> stringArrayParam;
Param<int[][]> int2dArrayParam;
Param<Param<Object>> objectNestedParam;
```

En Java SE 7 y versiones posteriores, el argumento de tipo se puede reemplazar con un conjunto vacío de argumentos de tipo (`<>`) llamado *diamante* :

Java SE 7

```
Param<Integer> integerParam = new Param<>();
```

A diferencia de otros identificadores, los parámetros de tipo no tienen restricciones de denominación. Sin embargo, sus nombres suelen ser la primera letra de su propósito en mayúsculas. (Esto es cierto incluso en los JavaDocs oficiales).
Los ejemplos incluyen `T` para "tipo" , `E` para "elemento" y `K` / `V` para "clave" / "valor" .

Extendiendo una clase genérica

```
public abstract class AbstractParam<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
```

```
        this.value = value;
    }
}
```

`AbstractParam` es una clase abstracta declarada con un parámetro de tipo de `T` Al extender esta clase, ese parámetro de tipo puede reemplazarse por un argumento de tipo escrito dentro de `<>` , o el parámetro de tipo puede permanecer sin cambios. En el primer y segundo ejemplo a continuación, `String` y `Integer` reemplazan el parámetro de tipo. En el tercer ejemplo, el parámetro de tipo permanece sin cambios. El cuarto ejemplo no usa genéricos en absoluto, por lo que es similar a si la clase tuviera un parámetro `Object` . El compilador advertirá que `AbstractParam` es un tipo sin `ObjectParam` , pero compilará la clase `ObjectParam` . El quinto ejemplo tiene 2 parámetros de tipo (consulte "parámetros de tipo múltiple" a continuación), seleccionando el segundo parámetro como el parámetro de tipo pasado a la superclase.

```
public class Email extends AbstractParam<String> {
    // ...
}

public class Age extends AbstractParam<Integer> {
    // ...
}

public class Height<T> extends AbstractParam<T> {
    // ...
}

public class ObjectParam extends AbstractParam {
    // ...
}

public class MultiParam<T, E> extends AbstractParam<E> {
    // ...
}
```

El siguiente es el uso:

```
Email email = new Email();
email.setValue("test@example.com");
String retrievedEmail = email.getValue();

Age age = new Age();
age.setValue(25);
Integer retrievedAge = age.getValue();
int autounboxedAge = age.getValue();

Height<Integer> heightInInt = new Height<>();
heightInInt.setValue(125);

Height<Float> heightInFloat = new Height<>();
heightInFloat.setValue(120.3f);

MultiParam<String, Double> multiParam = new MultiParam<>();
multiParam.setValue(3.3);
```

Observe que en la clase de `Email` , el método `T getValue()` actúa como si tuviera una firma de

`String getValue()` , y el método `void setValue(T)` actúa como si fuera declarado `void setValue(String)` .

También es posible crear una instancia con una clase interna anónima con llaves vacías (`{}`):

```
AbstractParam<Double> height = new AbstractParam<Double>(){};
height.setValue(198.6);
```

Tenga en cuenta que no se permite usar el diamante con clases internas anónimas.

Parámetros de tipo múltiple

Java ofrece la posibilidad de utilizar más de un parámetro de tipo en una clase o interfaz genérica. Se pueden usar múltiples parámetros de tipo en una clase o interfaz colocando una **lista** de tipos **separados por comas** entre los corchetes angulares. Ejemplo:

```
public class MultiGenericParam<T, S> {
    private T firstParam;
    private S secondParam;

    public MultiGenericParam(T firstParam, S secondParam) {
        this.firstParam = firstParam;
        this.secondParam = secondParam;
    }

    public T getFirstParam() {
        return firstParam;
    }

    public void setFirstParam(T firstParam) {
        this.firstParam = firstParam;
    }

    public S getSecondParam() {
        return secondParam;
    }

    public void setSecondParam(S secondParam) {
        this.secondParam = secondParam;
    }
}
```

El uso se puede hacer de la siguiente manera:

```
MultiGenericParam<String, String> aParam = new MultiGenericParam<String, String>("value1",
"value2");
MultiGenericParam<Integer, Double> dayOfWeekDegrees = new MultiGenericParam<Integer,
Double>(1, 2.6);
```

Declarar un método genérico

Los métodos también pueden tener parámetros de tipo genérico .

```
public class Example {

    // The type parameter T is scoped to the method
    // and is independent of type parameters of other methods.
    public <T> List<T> makeList(T t1, T t2) {
        List<T> result = new ArrayList<T>();
        result.add(t1);
        result.add(t2);
        return result;
    }

    public void usage() {
        List<String> listString = makeList("Jeff", "Atwood");
        List<Integer> listInteger = makeList(1, 2);
    }
}
```

Tenga en cuenta que no tenemos que pasar un argumento de tipo real a un método genérico. El compilador infiere el argumento de tipo para nosotros, basado en el tipo de destino (por ejemplo, la variable a la que asignamos el resultado), o en los tipos de los argumentos reales. Por lo general, inferirá el argumento de tipo más específico que hará que la llamada sea correcta.

A veces, aunque rara vez, puede ser necesario anular esta inferencia de tipo con argumentos de tipo explícito:

```
void usage() {
    consumeObjects(this.<Object>makeList("Jeff", "Atwood").stream());
}

void consumeObjects(Stream<Object> stream) { ... }
```

Es necesario en este ejemplo porque el compilador no puede "mirar hacia adelante" para ver que el `Object` se desea para `T` después de llamar a `stream()` y, de lo contrario, `makeList String` función de los argumentos de la lista de elementos. Tenga en cuenta que el lenguaje Java no admite omitir la clase u objeto en el que se llama al método (`this` en el ejemplo anterior) cuando se proporcionan explícitamente argumentos de tipo.

El diamante

Java SE 7

Java 7 introdujo el *Diamond* [1] para eliminar algunas placas de calderas en torno a la creación de instancias de clase genérica. Con Java 7+ puedes escribir:

```
List<String> list = new LinkedList<>();
```

Donde tenías que escribir en versiones anteriores, esto:

```
List<String> list = new LinkedList<String>();
```

Una limitación es para las clases anónimas , donde aún debe proporcionar el parámetro de tipo en la creación de instancias:

```
// This will compile:

Comparator<String> caseInsensitiveComparator = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};

// But this will not:

Comparator<String> caseInsensitiveComparator = new Comparator<>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};
```

Java SE 8

Aunque el uso de Diamond con Anonymous Inner Classes no es compatible con Java 7 y 8, **se incluirá como una nueva función en Java 9** .

Nota:

1 - Algunas personas llaman al uso `<>` el " *operador de* diamante". Esto es incorrecto. El diamante no se comporta como un operador, y no se describe ni se enumera en ningún lugar en el JLS o en los Tutoriales de Java (oficiales) como operador. De hecho, `<>` ni siquiera es un token de Java distinto. Más bien, es un `<` token seguido de un `>` token, y es legal (aunque mal estilo) tener espacios en blanco o comentarios entre los dos. El JLS y los Tutoriales se refieren constantemente a `<>` como "el diamante", y ese es el término correcto para ello.

Requerir múltiples límites superiores ("extiende A y B")

Puede requerir un tipo genérico para extender varios límites superiores.

Ejemplo: queremos ordenar una lista de números, pero `Number` no implementa `Comparable` .

```
public <T extends Number & Comparable<T>> void sortNumbers( List<T> n ) {
  Collections.sort( n );
}
```

En este ejemplo, `T` debe extender el `Number` *e* implementar `Comparable<T>` que debe ajustarse a todas las implementaciones de números incorporados "normales" como `Integer` o `BigDecimal` pero no se ajusta a las más exóticas como `Striped64` .

Dado que no se permite la herencia múltiple, puede usar como máximo una clase como límite y debe ser la primera en la lista. Por ejemplo, `<T extends Comparable<T> & Number>` no está permitido porque Comparable es una interfaz y no una clase.

Creando una clase genérica limitada

Puede restringir los tipos válidos utilizados en una **clase genérica** al delimitar ese tipo en la definición de clase. Dada la siguiente jerarquía de tipos simple:

```
public abstract class Animal {
    public abstract String getSound();
}

public class Cat extends Animal {
    public String getSound() {
        return "Meow";
    }
}

public class Dog extends Animal {
    public String getSound() {
        return "Woof";
    }
}
```

Sin **genéricos limitados** , no podemos crear una clase de contenedor que sea genérica y que sepa que cada elemento es un animal:

```
public class AnimalContainer<T> {

    private Collection<T> col;

    public AnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // Illegal, type T doesn't have makeSound()
            // it is used as an java.lang.Object here
            System.out.println(t.makeSound());
        }
    }
}
```

Con el límite genérico en la definición de clase, esto ahora es posible.

```
public class BoundedAnimalContainer<T extends Animal> { // Note bound here.

    private Collection<T> col;

    public BoundedAnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
```

```
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // Now works because T is extending Animal
            System.out.println(t.makeSound());
        }
    }
}
```

Esto también restringe las instancias válidas del tipo genérico:

```
// Legal
AnimalContainer<Cat> a = new AnimalContainer<Cat>();

// Legal
AnimalContainer<String> a = new AnimalContainer<String>();
```

```
// Legal because Cat extends Animal
BoundedAnimalContainer<Cat> b = new BoundedAnimalContainer<Cat>();

// Illegal because String doesn't extends Animal
BoundedAnimalContainer<String> b = new BoundedAnimalContainer<String>();
```

Decidir entre 'T', `? super T`, y `? extiende T`

La sintaxis de los comodines delimitados de los genéricos de Java, que representa el tipo desconocido por `?` es:

- `? extends T` representa un comodín acotado superior. El tipo desconocido representa un tipo que debe ser un subtipo de T o el mismo tipo de T.
- `? super T` representa un comodín acotado más bajo. El tipo desconocido representa un tipo que debe ser un supertipo de T o el mismo tipo de T.

Como regla general, debes usar

- `? extends T` si solo necesita acceso de "lectura" ("entrada")
- `? super T` si necesita acceso de "escritura" ("salida")
- `T` si necesita ambos ("modificar")

El uso de `extends` o `super` es generalmente *mejor* porque hace que su código sea más flexible (como en: permite el uso de subtipos y supertipos), como verá a continuación.

```
class Shoe {}
class IPhone {}
interface Fruit {}
class Apple implements Fruit {}
class Banana implements Fruit {}
class GrannySmith extends Apple {}

   public class FruitHelper {
```

```
        public void eatAll(Collection<? extends Fruit> fruits) {}

        public void addApple(Collection<? super Apple> apples) {}
}
```

El compilador ahora podrá detectar cierto mal uso:

```
 public class GenericsTest {
      public static void main(String[] args){
  FruitHelper fruitHelper = new FruitHelper() ;
    List<Fruit> fruits = new ArrayList<Fruit>();
    fruits.add(new Apple()); // Allowed, as Apple is a Fruit
    fruits.add(new Banana()); // Allowed, as Banana is a Fruit
    fruitHelper.addApple(fruits); // Allowed, as "Fruit super Apple"
    fruitHelper.eatAll(fruits); // Allowed

    Collection<Banana> bananas = new ArrayList<>();
    bananas.add(new Banana()); // Allowed
    //fruitHelper.addApple(bananas); // Compile error: may only contain Bananas!
    fruitHelper.eatAll(bananas); // Allowed, as all Bananas are Fruits

    Collection<Apple> apples = new ArrayList<>();
    fruitHelper.addApple(apples); // Allowed
    apples.add(new GrannySmith()); // Allowed, as this is an Apple
    fruitHelper.eatAll(apples); // Allowed, as all Apples are Fruits.

    Collection<GrannySmith> grannySmithApples = new ArrayList<>();
    fruitHelper.addApple(grannySmithApples); //Compile error: Not allowed.
                                  // GrannySmith is not a supertype of Apple
    apples.add(new GrannySmith()); //Still allowed, GrannySmith is an Apple
    fruitHelper.eatAll(grannySmithApples);//Still allowed, GrannySmith is a Fruit

    Collection<Object> objects = new ArrayList<>();
    fruitHelper.addApple(objects); // Allowed, as Object super Apple
    objects.add(new Shoe()); // Not a fruit
    objects.add(new IPhone()); // Not a fruit
    //fruitHelper.eatAll(objects); // Compile error: may contain a Shoe, too!
}
```

La elección de la derecha `T` , `? super T` o `? extends T` es *necesario* para permitir el uso con subtipos. El compilador puede entonces garantizar la seguridad del tipo; no debería tener que emitir (que no es de tipo seguro y puede causar errores de programación) si los usa correctamente.

Si no es fácil de entender, recuerde la regla de **PECS** :

> **P** roducer usa " **E** xtends" y **C** onsumer usa " **S** uper".

(El productor solo tiene acceso de escritura y el consumidor solo tiene acceso de lectura)

Beneficios de la clase genérica y la interfaz

El código que utiliza genéricos tiene muchos beneficios sobre el código no genérico. A continuación se presentan los principales beneficios.

Controles de tipo más fuertes en tiempo de compilación

Un compilador de Java aplica una comprobación de tipo segura al código genérico y emite errores si el código viola la seguridad del tipo. Reparar errores de tiempo de compilación es más fácil que corregir errores de tiempo de ejecución, que pueden ser difíciles de encontrar.

Eliminación de moldes

El siguiente fragmento de código sin genéricos requiere el lanzamiento:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Cuando se vuelve a escribir para *usar genéricos* , el código no requiere conversión:

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0);   // no cast
```

Permitiendo a los programadores implementar algoritmos genéricos

Mediante el uso de genéricos, los programadores pueden implementar algoritmos genéricos que funcionan en colecciones de diferentes tipos, pueden personalizarse, son seguros y más fáciles de leer.

Vinculación de parámetro genérico a más de 1 tipo

Los parámetros genéricos también pueden vincularse a más de un tipo utilizando la sintaxis `T extends Type1 & Type2 & ...`

Digamos que desea crear una clase cuyo tipo debe implementar tanto Genérico `Flushable` y `Closeable` , se puede escribir

```
class ExampleClass<T extends Flushable & Closeable> {
}
```

Ahora, el `ExampleClass` sólo acepta como parámetros genéricos, tipos que implementan tanto `Flushable` **y** `Closeable` .

```
ExampleClass<BufferedWriter> arg1; // Works because BufferedWriter implements both Flushable
and Closeable

ExampleClass<Console> arg4; // Does NOT work because Console only implements Flushable
ExampleClass<ZipFile> arg5; // Does NOT work because ZipFile only implements Closeable

ExampleClass<Flushable> arg2; // Does NOT work because Closeable bound is not satisfied.
ExampleClass<Closeable> arg3; // Does NOT work because Flushable bound is not satisfied.
```

Los métodos de clase pueden optar por deducir argumentos de tipo genérico como tampoco `Closeable` **o** `Flushable` .

```
class ExampleClass<T extends Flushable & Closeable> {
    /* Assign it to a valid type as you want. */
    public void test (T param) {
        Flushable arg1 = param; // Works
        Closeable arg2 = param; // Works too.
    }

    /* You can even invoke the methods of any valid type directly. */
    public void test2 (T param) {
        param.flush(); // Method of Flushable called on T and works fine.
        param.close(); // Method of Closeable called on T and works fine too.
    }
}
```

Nota:

No puede enlazar el parámetro genérico a ninguno de los tipos usando la cláusula *OR* (`|`). Solo se admite la cláusula *AND* (`&`). El tipo genérico puede extender solo una clase y muchas interfaces. La clase debe colocarse al principio de la lista.

Creando un tipo genérico

Debido al borrado de tipo no funcionará lo siguiente:

```
public <T> void genericMethod() {
    T t = new T(); // Can not instantiate the type T.
}
```

El tipo `T` se borra. Dado que, en el tiempo de ejecución, la JVM no sabe qué era `T` originalmente, no sabe a qué constructor llamar.

Soluciones

1. Pasando la clase de `T` al llamar a `genericMethod` :

```
public <T> void genericMethod(Class<T> cls) {
    try {
        T t = cls.newInstance();
    } catch (InstantiationException | IllegalAccessException e) {
         System.err.println("Could not instantiate: " + cls.getName());
    }
}
```

```
genericMethod(String.class);
```

Lo que genera excepciones, ya que no hay forma de saber si la clase pasada tiene un constructor predeterminado accesible.

Java SE 8

2. Pasando una referencia al constructor de `T` :

```
public <T> void genericMethod(Supplier<T> cons) {
    T t = cons.get();
}
```

```
genericMethod(String::new);
```

Refiriéndose al tipo genérico declarado dentro de su propia declaración.

¿Cómo se utiliza una instancia de un tipo genérico heredado (posiblemente más) dentro de una declaración de método en el tipo genérico que se está declarando? Este es uno de los problemas a los que se enfrentará cuando profundice un poco en los genéricos, pero sigue siendo bastante común.

Supongamos que tenemos un tipo `DataSeries<T>` (interfaz aquí), que define una serie de datos genéricos que contienen valores de tipo `T` Es complicado trabajar con este tipo directamente cuando queremos realizar muchas operaciones con, por ejemplo, valores dobles, por lo que definimos `DoubleSeries extends DataSeries<Double>` . Ahora supongamos que el tipo original `DataSeries<T>` tiene un método `add(values)` que agrega otra serie de la misma longitud y devuelve una nueva. ¿Cómo `DoubleSeries` el tipo de `values` y el tipo de retorno para que sea `DoubleSeries` lugar de `DataSeries<Double>` en nuestra clase derivada?

El problema se puede resolver agregando un parámetro de tipo genérico que se refiera y extienda el tipo que se está declarando (aplicado aquí a una interfaz, pero lo mismo significa clases):

```
public interface DataSeries<T, DS extends DataSeries<T, DS>> {
    DS add(DS values);
    List<T> data();
}
```

Aquí `T` representa el tipo de datos que contiene la serie, por ejemplo, `Double` y `DS` la propia serie. Un tipo (o tipos) heredado ahora se puede implementar fácilmente sustituyendo el parámetro mencionado anteriormente por un tipo derivado correspondiente, dando así una definición

concreta de `Double` base de la forma:

```
public interface DoubleSeries extends DataSeries<Double, DoubleSeries> {
    static DoubleSeries instance(Collection<Double> data) {
        return new DoubleSeriesImpl(data);
    }
}
```

En este momento, incluso un IDE implementará la interfaz anterior con los tipos correctos en su lugar, que, después de un poco de llenado de contenido, pueden tener este aspecto:

```
class DoubleSeriesImpl implements DoubleSeries {
    private final List<Double> data;

    DoubleSeriesImpl(Collection<Double> data) {
        this.data = new ArrayList<>(data);
    }

    @Override
    public DoubleSeries add(DoubleSeries values) {
        List<Double> incoming = values != null ? values.data() : null;
        if (incoming == null || incoming.size() != data.size()) {
            throw new IllegalArgumentException("bad series");
        }
        List<Double> newdata = new ArrayList<>(data.size());
        for (int i = 0; i < data.size(); i++) {
            newdata.add(this.data.get(i) + incoming.get(i)); // beware autoboxing
        }
        return DoubleSeries.instance(newdata);
    }

    @Override
    public List<Double> data() {
        return Collections.unmodifiableList(data);
    }
}
```

Como puede ver, el método de `add` se declara como `DoubleSeries add(DoubleSeries values)` y el compilador está contento.

El patrón se puede anidar más si es necesario.

Uso de instanceof con Genéricos.

Usando genéricos para definir el tipo en instancia de

Considere el siguiente `Example` clase genérica declarado con el parámetro formal `<T>` :

```
class Example<T> {
    public boolean isTypeAString(String s) {
        return s instanceof T; // Compilation error, cannot use T as class type here
    }
}
```

Esto siempre dará un error de compilación porque en cuanto el compilador compila la *fuente Java*

en el *código de bytes de Java* , aplica un proceso conocido como *borrado de tipo* , que convierte todo el código genérico en código no genérico, lo que hace imposible distinguir entre los tipos T en tiempo de ejecución. El tipo utilizado con `instanceof` tiene que ser *verificable* , lo que significa que toda la información sobre el tipo debe estar disponible en tiempo de ejecución, y esto no suele ser el caso para los tipos genéricos.

La siguiente clase representa el aspecto de dos clases diferentes de `Example` , `Example<String>` y `Example<Number>` , después de que los genéricos se hayan eliminado por *borrado de tipo* :

```
class Example { // formal parameter is gone
    public boolean isTypeAString(String s) {
        return s instanceof Object; // Both <String> and <Number> are now Object
    }
}
```

Dado que los tipos han desaparecido, no es posible que la JVM sepa qué tipo es `T`

Excepción a la regla anterior.

Siempre puede usar *un comodín ilimitado* (?) Para especificar un tipo en el `instanceof` siguiente manera:

```
    public boolean isAList(Object obj) {
        return obj instanceof List<?>;
    }
```

Esto puede ser útil para evaluar si una instancia `obj` es una `List` o no:

```
System.out.println(isAList("foo")); // prints false
System.out.println(isAList(new ArrayList<String>()); // prints true
System.out.println(isAList(new ArrayList<Float>()); // prints true
```

De hecho, el comodín ilimitado se considera un tipo confiable.

Usando una instancia genérica con instanceof

El otro lado de la moneda es que usar una instancia `t` de `T` con `instanceof` es legal, como se muestra en el siguiente ejemplo:

```
class Example<T> {
    public boolean isTypeAString(T t) {
        return t instanceof String; // No compilation error this time
    }
}
```

porque después del borrado de tipo, la clase tendrá el siguiente aspecto:

```
class Example { // formal parameter is gone
    public boolean isTypeAString(Object t) {
```

```
        return t instanceof String; // No compilation error this time
    }
}
```

Dado que, incluso si el borrado de tipo ocurre de todos modos, ahora la JVM puede distinguir entre diferentes tipos en la memoria, incluso si usan el mismo tipo de referencia (`Object`), como muestra el siguiente fragmento de código:

```
Object obj1 = new String("foo"); // reference type Object, object type String
Object obj2 = new Integer(11); // reference type Object, object type Integer
System.out.println(obj1 instanceof String); // true
System.out.println(obj2 instanceof String); // false, it's an Integer, not a String
```

Diferentes formas de implementar una interfaz genérica (o extender una clase genérica)

Supongamos que la siguiente interfaz genérica ha sido declarada:

```
public interface MyGenericInterface<T> {
    public void foo(T t);
}
```

A continuación se enumeran las posibles formas de implementarlo.

Implementación de clase no genérica con un tipo específico

Elija un tipo específico para reemplazar el parámetro de tipo formal `<T>` de `MyGenericClass` e implementarlo, como lo hace el siguiente ejemplo:

```
public class NonGenericClass implements MyGenericInterface<String> {
    public void foo(String t) { } // type T has been replaced by String
}
```

Esta clase solo trata con `String` , y esto significa que el uso de `MyGenericInterface` con diferentes parámetros (por ejemplo, `Integer` , `Object` , etc.) no se compilará, como muestra el siguiente fragmento de código:

```
NonGenericClass myClass = new NonGenericClass();
myClass.foo("foo_string"); // OK, legal
myClass.foo(11); // NOT OK, does not compile
myClass.foo(new Object()); // NOT OK, does not compile
```

Implementación genérica de clase.

Declare otra interfaz genérica con el parámetro de tipo formal `<T>` que implementa `MyGenericInterface` , de la siguiente manera:

```
public class MyGenericSubclass<T> implements MyGenericInterface<T> {
```

```
    public void foo(T t) { } // type T is still the same
    // other methods...
}
```

Tenga en cuenta que se puede haber utilizado un parámetro de tipo formal diferente, de la siguiente manera:

```
public class MyGenericSubclass<U> implements MyGenericInterface<U> { // equivalent to the
previous declaration
    public void foo(U t) { }
    // other methods...
}
```

Implementación de clase de tipo raw

Declare una clase no genérica que implementa `MyGenericInteface` como un *tipo sin* `MyGenericInteface` (sin usar genérico en absoluto), como sigue:

```
public class MyGenericSubclass implements MyGenericInterface {
    public void foo(Object t) { } // type T has been replaced by Object
    // other possible methods
}
```

De esta forma **no se** recomienda, ya que no es 100% seguro en el tiempo de ejecución porque mezcla el *tipo sin formato* (de la subclase) con los *genéricos* (de la interfaz) y también es confuso. Los compiladores de Java modernos generarán una advertencia con este tipo de implementación; sin embargo, el código, por razones de compatibilidad con JVM más antiguos (1.4 o anteriores), se compilará.

También se permiten todas las formas enumeradas anteriormente cuando se utiliza una clase genérica como supertipo en lugar de una interfaz genérica.

Usando genéricos para auto-cast

Con los genéricos, es posible devolver lo que la persona que llama espera:

```
private Map<String, Object> data;
public <T> T get(String key) {
    return (T) data.get(key);
}
```

El método se compilará con una advertencia. El código es en realidad más seguro de lo que parece porque el tiempo de ejecución de Java se convertirá en una conversión cuando lo uses:

```
Bar bar = foo.get("bar");
```

Es menos seguro cuando usas tipos genéricos:

```
List<Bar> bars = foo.get("bars");
```

Aquí, la conversión funcionará cuando el tipo devuelto sea cualquier tipo de `List` (es decir, devolver la `List<String>` no `ClassCastException` una `ClassCastException` ; finalmente la obtendría al eliminar elementos de la lista).

Para solucionar este problema, puede crear una API que use claves escritas:

```
public final static Key<List<Bar>> BARS = new Key<>("BARS");
```

junto con este método `put()` :

```
public <T> T put(Key<T> key, T value);
```

Con este enfoque, no puede poner el tipo incorrecto en el mapa, por lo que el resultado siempre será correcto (a menos que cree accidentalmente dos claves con el mismo nombre pero tipos diferentes).

Relacionado:

- Mapa de tipo seguro

Obtenga una clase que satisfaga el parámetro genérico en tiempo de ejecución

Muchos parámetros genéricos no vinculados, como los que se utilizan en un método estático, no se pueden recuperar en tiempo de ejecución (consulte *Otros subprocesos* en *Erasure*). Sin embargo, hay una estrategia común empleada para acceder al tipo que satisface un parámetro genérico en una clase en tiempo de ejecución. Esto permite un código genérico que depende del acceso al tipo *sin* tener que pasar información de tipo a través de cada llamada.

Fondo

La parametrización genérica en una clase se puede inspeccionar creando una clase interna anónima. Esta clase capturará la información de tipo. En general, este mecanismo se conoce como **tokens de tipo super** , que se detallan en la publicación del blog de Neal Gafter .

Implementaciones

Tres implementaciones comunes en Java son:

- Tipo de guayaba hablado
- Spring's ParameterizedTypeReference
- Tipo de referencia de Jackson

Ejemplo de uso

```
public class DataService<MODEL_TYPE> {
     private final DataDao dataDao = new DataDao();
```

```
     private final Class<MODEL_TYPE> type = (Class<MODEL_TYPE>) new TypeToken<MODEL_TYPE>
                                                                 (getClass()){}.getRawType();
     public List<MODEL_TYPE> getAll() {
         return dataDao.getAllOfType(type);
    }
}

// the subclass definitively binds the parameterization to User
// for all instances of this class, so that information can be
// recovered at runtime
public class UserService extends DataService<User> {}

public class Main {
    public static void main(String[] args) {
          UserService service = new UserService();
          List<User> users = service.getAll();
    }
}
```

Lea Genéricos en línea: https://riptutorial.com/es/java/topic/92/genericos

Capítulo 88: Gerente de seguridad

Examples

Habilitando el SecurityManager

Las máquinas virtuales Java (JVM) se pueden ejecutar con un SecurityManager instalado. El SecurityManager gobierna lo que se permite que haga el código que se ejecuta en la JVM, en función de factores tales como desde dónde se cargó el código y qué certificados se usaron para firmar el código.

El SecurityManager se puede instalar configurando la propiedad del sistema java.security.manager en la línea de comandos al iniciar la JVM:

```
java -Djava.security.manager <main class name>
```

o programáticamente desde el código de Java:

```
System.setSecurityManager(new SecurityManager())
```

El Java SecurityManager estándar otorga permisos sobre la base de una Política, que se define en un archivo de políticas. Si no se especifica ningún archivo de políticas, se utilizará el archivo de políticas predeterminado en `$JAVA_HOME/lib/security/java.policy` .

Clases de sandboxing cargadas por un ClassLoader

El ClassLoader debe proporcionar un dominio de `ProtectionDomain` identifique la fuente del código:

```
public class PluginClassLoader extends ClassLoader {
    private final ClassProvider provider;

    private final ProtectionDomain pd;


    public PluginClassLoader(ClassProvider provider) {
        this.provider = provider;
        Permissions permissions = new Permissions();

        this.pd = new ProtectionDomain(provider.getCodeSource(), permissions, this, null);
    }


    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] classDef = provider.getClass(name);
        Class<?> clazz = defineClass(name, classDef, 0, classDef.length, pd);
        return clazz;
    }
}
```

Al reemplazar `findClass` lugar de `loadClass` el modelo delegacional se conserva, y el

PluginClassLoader primero consultará el sistema y el cargador de clases principal para las definiciones de clase.

Crear una política:

```
public class PluginSecurityPolicy extends Policy {
    private final Permissions appPermissions = new Permissions();
    private final Permissions pluginPermissions = new Permissions();

    public PluginSecurityPolicy() {
        // amend this as appropriate
        appPermissions.add(new AllPermission());
        // add any permissions plugins should have to pluginPermissions
    }

    @Override
    public Provider getProvider() {
        return super.getProvider();
    }

    @Override
    public String getType() {
        return super.getType();
    }

    @Override
    public Parameters getParameters() {
        return super.getParameters();
    }

    @Override
    public PermissionCollection getPermissions(CodeSource codesource) {
        return new Permissions();
    }

    @Override
    public PermissionCollection getPermissions(ProtectionDomain domain) {
        return isPlugin(domain)?pluginPermissions:appPermissions;
    }

    private boolean isPlugin(ProtectionDomain pd){
        return pd.getClassLoader() instanceof PluginClassLoader;
    }

}
```

Finalmente, configure la política y un SecurityManager (la implementación predeterminada está bien):

```
    Policy.setPolicy(new PluginSecurityPolicy());
    System.setSecurityManager(new SecurityManager());
```

Implementando reglas de negación de políticas.

Ocasionalmente es conveniente *negar* un cierto `Permission` a un dominio de `ProtectionDomain` , *independientemente* de cualquier otro permiso que acumule el dominio. Este ejemplo muestra

solo uno de todos los enfoques posibles para satisfacer este tipo de requisitos. Introduce una clase de permiso "negativa", junto con un envoltorio que permite que la `Policy` predeterminada se reutilice como un repositorio de dichos permisos.

Notas:

- La sintaxis y el mecanismo de archivo de política estándar para la asignación de permisos en general no se ven afectados. Esto significa que las reglas de *denegación* dentro de los archivos de políticas aún se expresan como *concesiones* .
- El contenedor de políticas está destinado a encapsular específicamente la `Policy` respaldada por archivos predeterminada (se supone que es `com.sun.security.provider.PolicyFile`).
- Los permisos denegados solo se procesan como tales en el nivel de la política. Si se asignan estáticamente a un dominio, de forma predeterminada serán tratados por ese dominio como permisos "positivos" ordinarios.

La clase de `DeniedPermission`

```
package com.example;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Modifier;
import java.security.BasicPermission;
import java.security.Permission;
import java.security.UnresolvedPermission;
import java.text.MessageFormat;

/**
 * A representation of a "negative" privilege.
 * <p>
 * A <code>DeniedPermission</code>, when "granted" (to some <code>ProtectionDomain</code>
and/or
 * <code>Principal</code>), represents a privilege which <em>cannot</em> be exercised,
regardless of
 * any positive permissions (<code>AllPermission</code> included) possessed. In other words,
if a
 * set of granted permissions, <em>P</em>, contains a permission of this class, <em>D</em>,
then the
 * set of effectively granted permissions is<br/>
 * <br/>
 *     <em>{ P<sub>implied</sub> - D<sub>implied</sub> }</em>.
 * </p>
 * <p>
 * Each instance of this class encapsulates a <em>target permission</em>, representing the
 * "positive" permission being denied.
 * </p>
 * Denied permissions employ the following naming scheme:<br/>
 * <br/>
 *
    <em>&lt;target_class_name&gt;:&lt;target_name&gt;(:&lt;target_actions&gt;)</em>

 * <br/>
 * where:
 * <ul>
 * <li><em>target_class_name</em> is the name of the target permission's class,</li>
```

```
 * <li><em>target_name</em> is the name of the target permission, and</li>
 * <li><em>target_actions</em> is, optionally, the actions string of the target
permission.</li>
 * </ul>
 * A denied permission, having a target permission <em>t</em>, is said to <em>imply</em>
another
 * permission <em>p</em>, if:
 * <ul>
 * <li>p <em>is not</em> itself a denied permission, and <code>(t.implies(p) == true)</code>,
 * or</li>
 * <li>p <em>is</em> a denied permission, with a target <em>t1</em>, and
 * <code>(t.implies(t1) == true)</code>.
 * </ul>
 * <p>
 * It is the responsibility of the policy decision point (e.g., the <code>Policy</code>
provider) to
 * take denied permission semantics into account when issuing authorization statements.
 * </p>
 */
public final class DeniedPermission extends BasicPermission {

    private final Permission target;
    private static final long serialVersionUID = 4736251638698006 79L;

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates a target permission of
the
     * indicated class, specified name and, optionally, actions.
     *
     * @throws IllegalArgumentException
     *             if:
     *             <ul>
     *             <li><code>targetClassName</code> is <code>null</code>, the empty string,
does not
     *             refer to a concrete <code>Permission</code> descendant, or refers to
     *             <code>DeniedPermission.class</code> or
<code>UnresolvedPermission.class</code>.</li>
     *             <li><code>targetName</code> is <code>null</code>.</li>
     *             <li><code>targetClassName</code> cannot be instantiated, and it's the
caller's fault;
     *             e.g., because <code>targetName</code> and/or <code>targetActions</code> do
not adhere
     *             to the naming constraints of the target class; or due to the target class
not
     *             exposing a <code>(String name)</code>, or <code>(String name, String
actions)</code>
     *             constructor, depending on whether <code>targetActions</code> is
<code>null</code> or
     *             not.</li>
     *             </ul>
     */
    public static DeniedPermission newDeniedPermission(String targetClassName, String
targetName,
            String targetActions) {
        if (targetClassName == null || targetClassName.trim().isEmpty() || targetName == null)
{
            throw new IllegalArgumentException(
                    "Null or empty [targetClassName], or null [targetName] argument was
supplied.");
        }
        StringBuilder sb = new StringBuilder(targetClassName).append(":").append(targetName);
```

```
        if (targetName != null) {
            sb.append(":").append(targetName);
        }
        return new DeniedPermission(sb.toString());
    }

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates a target permission of
the class,
     * name and, optionally, actions, collectively provided as the <code>name</code> argument.
     *
     * @throws IllegalArgumentException
     *             if:
     *             <ul>
     *             <li><code>name</code>'s target permission class name component is empty,
does not
     *             refer to a concrete <code>Permission</code> descendant, or refers to
     *             <code>DeniedPermission.class</code> or
<code>UnresolvedPermission.class</code>.</li>
     *             <li><code>name</code>'s target name component is <code>empty</code></li>
     *             <li>the target permission class cannot be instantiated, and it's the
caller's fault;
     *             e.g., because <code>name</code>'s target name and/or target actions
component(s) do
     *             not adhere to the naming constraints of the target class; or due to the
target class
     *             not exposing a <code>(String name)</code>, or
     *             <code>(String name, String actions)</code> constructor, depending on
whether the
     *             target actions component is empty or not.</li>
     *             </ul>
     */
    public DeniedPermission(String name) {
        super(name);
        String[] comps = name.split(":");
        if (comps.length < 2) {
            throw new IllegalArgumentException(MessageFormat.format("Malformed name [{0}]
argument.", name));
        }
        this.target = initTarget(comps[0], comps[1], ((comps.length < 3) ? null : comps[2]));
    }

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates the given target
permission.
     *
     * @throws IllegalArgumentException
     *             if <code>target</code> is <code>null</code>, a
<code>DeniedPermission</code>, or an
     *             <code>UnresolvedPermission</code>.
     */
    public static DeniedPermission newDeniedPermission(Permission target) {
        if (target == null) {
            throw new IllegalArgumentException("Null [target] argument.");
        }
        if (target instanceof DeniedPermission || target instanceof UnresolvedPermission) {
            throw new IllegalArgumentException("[target] must not be a DeniedPermission or an
UnresolvedPermission.");
        }
        StringBuilder sb = new
StringBuilder(target.getClass().getName()).append(":").append(target.getName());
```

```
        String targetActions = target.getActions();
        if (targetActions != null) {
            sb.append(":").append(targetActions);
        }
        return new DeniedPermission(sb.toString(), target);
    }

    private DeniedPermission(String name, Permission target) {
        super(name);
        this.target = target;
    }

    private Permission initTarget(String targetClassName, String targetName, String
targetActions) {
        Class<?> targetClass;
        try {
            targetClass = Class.forName(targetClassName);
        }
        catch (ClassNotFoundException cnfe) {
            if (targetClassName.trim().isEmpty()) {
                targetClassName = "<empty>";
            }
            throw new IllegalArgumentException(
                    MessageFormat.format("Target Permission class [{0}] not found.",
targetClassName));
        }
        if (!Permission.class.isAssignableFrom(targetClass) ||
Modifier.isAbstract(targetClass.getModifiers())) {
            throw new IllegalArgumentException(MessageFormat
                    .format("Target Permission class [{0}] is not a (concrete) Permission.",
targetClassName));
        }
        if (targetClass == DeniedPermission.class || targetClass ==
UnresolvedPermission.class) {
            throw new IllegalArgumentException("Target Permission class cannot be a
DeniedPermission itself.");
        }
        Constructor<?> targetCtor;
        try {
            if (targetActions == null) {
                targetCtor = targetClass.getConstructor(String.class);
            }
            else {
                targetCtor = targetClass.getConstructor(String.class, String.class);
            }
        }
        catch (NoSuchMethodException nsme) {
            throw new IllegalArgumentException(MessageFormat.format(
                    "Target Permission class [{0}] does not provide or expose a (String name)
or (String name, String actions) constructor.",
                    targetClassName));
        }
        try {
            return (Permission) targetCtor
                    .newInstance(((targetCtor.getParameterCount() == 1) ? new Object[] {
targetName }
                            : new Object[] { targetName, targetActions }));
        }
        catch (ReflectiveOperationException roe) {
            if (roe instanceof InvocationTargetException) {
                if (targetName == null) {
```

```
                    targetName = "<null>";
                }
                else if (targetName.trim().isEmpty()) {
                    targetName = "<empty>";
                }
                if (targetActions == null) {
                    targetActions = "<null>";
                }
                else if (targetActions.trim().isEmpty()) {
                    targetActions = "<empty>";
                }
                throw new IllegalArgumentException(MessageFormat.format(
                        "Could not instantiate target Permission class [{0}]; provided target
name [{1}] and/or target actions [{2}] potentially erroneous.",
                        targetClassName, targetName, targetActions), roe);
            }
            throw new RuntimeException(
                    "Could not instantiate target Permission class [{0}]; an unforeseen error
occurred - see attached cause for details",
                    roe);
        }
    }

    /**
     * Checks whether the given permission is implied by this one, as per the {@link
DeniedPermission
     * overview}.
     */
    @Override
    public boolean implies(Permission p) {
        if (p instanceof DeniedPermission) {
            return target.implies(((DeniedPermission) p).target);
        }
        return target.implies(p);
    }

    /**
     * Returns this denied permission's target permission (the actual positive permission
which is not
     * to be granted).
     */
    public Permission getTargetPermission() {
        return target;
    }

}
```

La clase `DenyingPolicy`

```
package com.example;

import java.security.CodeSource;
import java.security.NoSuchAlgorithmException;
import java.security.Permission;
import java.security.PermissionCollection;
import java.security.Policy;
import java.security.ProtectionDomain;
import java.security.UnresolvedPermission;
import java.util.Enumeration;
```

```
/**
 * Wrapper that adds rudimentary {@link DeniedPermission} processing capabilities to the
standard
 * file-backed <code>Policy</code>.
 */
public final class DenyingPolicy extends Policy {

    {
        try {
            defaultPolicy = Policy.getInstance("javaPolicy", null);
        }
        catch (NoSuchAlgorithmException nsae) {
            throw new RuntimeException("Could not acquire default Policy.", nsae);
        }
    }

    private final Policy defaultPolicy;

    @Override
    public PermissionCollection getPermissions(CodeSource codesource) {
        return defaultPolicy.getPermissions(codesource);
    }

    @Override
    public PermissionCollection getPermissions(ProtectionDomain domain) {
        return defaultPolicy.getPermissions(domain);
    }

    /**
     * @return
     *         <ul>
     *         <li><code>true</code> if:</li>
     *         <ul>
     *         <li><code>permission</code> <em>is not</em> an instance of
     *         <code>DeniedPermission</code>,</li>
     *         <li>an <code>implies(domain, permission)</code> invocation on the system-
default
     *         <code>Policy</code> yields <code>true</code>, and</li>
     *         <li><code>permission</code> <em>is not</em> implied by any
<code>DeniedPermission</code>s
     *         having potentially been assigned to <code>domain</code>.</li>
     *         </ul>
     *         <li><code>false</code>, otherwise.
     *         </ul>
     */
    @Override
    public boolean implies(ProtectionDomain domain, Permission permission) {
        if (permission instanceof DeniedPermission) {
            /*
             * At the policy decision level, DeniedPermissions can only themselves imply, not
be implied (as
             * they take away, rather than grant, privileges). Furthermore, clients aren't
supposed to use this
             * method for checking whether some domain _does not_ have a permission (which is
what
             * DeniedPermissions express after all).
             */
            return false;
        }
```

```
        if (!defaultPolicy.implies(domain, permission)) {
            // permission not granted, so no need to check whether denied
            return false;
        }

        /*
         * Permission granted--now check whether there's an overriding DeniedPermission. The
following
         * assumes that previousPolicy is a sun.security.provider.PolicyFile (different
implementations
         * might not support #getPermissions(ProtectionDomain) and/or handle
UnresolvedPermissions
         * differently).
         */

        Enumeration<Permission> perms = defaultPolicy.getPermissions(domain).elements();
        while (perms.hasMoreElements()) {
            Permission p = perms.nextElement();
            /*
             * DeniedPermissions will generally remain unresolved, as no code is expected to
check whether other
             * code has been "granted" such a permission.
             */
            if (p instanceof UnresolvedPermission) {
                UnresolvedPermission up = (UnresolvedPermission) p;
                if (up.getUnresolvedType().equals(DeniedPermission.class.getName())) {
                    // force resolution
                    defaultPolicy.implies(domain, up);
                    // evaluate right away, to avoid reiterating over the collection
                    p = new DeniedPermission(up.getUnresolvedName());
                }
            }
            if (p instanceof DeniedPermission && p.implies(permission)) {
                // permission denied
                return false;
            }
        }
        // permission granted
        return true;
    }

    @Override
    public void refresh() {
        defaultPolicy.refresh();
    }

}
```

Manifestación

```
package com.example;

import java.security.Policy;

public class Main {

    public static void main(String... args) {
        Policy.setPolicy(new DenyingPolicy());
        System.setSecurityManager(new SecurityManager());
```

```
        // should fail
        System.getProperty("foo.bar");
    }

}
```

Asignar algunos permisos:

```
grant codeBase "file:///path/to/classes/bin/-"
    permission java.util.PropertyPermission "*", "read,write";
    permission com.example.DeniedPermission "java.util.PropertyPermission:foo.bar:read";
};
```

Por último, ejecute el `Main` y observe cómo falla, debido a la regla de "denegar" (`DeniedPermission`) que `DeniedPermission` la `grant` (su `PropertyPermission`). Tenga en cuenta que una `setProperty("foo.baz", "xyz")` lugar habría tenido éxito, ya que el permiso denegado solo cubre la acción "leer", y solo para la propiedad "foo.bar".

Lea Gerente de seguridad en línea: https://riptutorial.com/es/java/topic/5712/gerente-de-seguridad

Capítulo 89: Gestión de memoria de Java

Observaciones

En Java, los objetos se asignan en el montón y la memoria del montón se reclama mediante la recolección automática de basura. Un programa de aplicación no puede eliminar explícitamente un objeto Java.

Los principios básicos de la recolección de basura Java se describen en el ejemplo de recolección de basura . Otros ejemplos describen la finalización, cómo activar el recolector de basura a mano y el problema de las fugas de almacenamiento.

Examples

Finalización

Un objeto Java puede declarar un método de `finalize` . Este método se llama justo antes de que Java libere la memoria para el objeto. Normalmente se verá así:

```
public class MyClass {

    //Methods for the class

    @Override
    protected void finalize() throws Throwable {
        // Cleanup code
    }
}
```

Sin embargo, hay algunas advertencias importantes sobre el comportamiento de la finalización de Java.

- Java no ofrece ninguna garantía sobre cuándo se llamará a un método `finalize()` .
- Java ni siquiera garantiza que se llame a un método `finalize()` algún momento durante el tiempo de vida de la aplicación en ejecución.
- Lo único que se garantiza es que se llamará al método antes de que se elimine el objeto ... si se elimina el objeto.

Las advertencias anteriores significan que es una mala idea confiar en el método de `finalize` para realizar acciones de limpieza (u otras) que deben realizarse de manera oportuna. La dependencia excesiva de la finalización puede provocar fugas de almacenamiento, fugas de memoria y otros problemas.

En resumen, hay muy pocas situaciones en las que la finalización es realmente una buena solución.

Los finalizadores solo se ejecutan una vez.

Normalmente, un objeto se elimina después de que se ha finalizado. Sin embargo, esto no sucede todo el tiempo. Considere el siguiente ejemplo [1] :

```
public class CaptainJack {
    public static CaptainJack notDeadYet = null;

    protected void finalize() {
        // Resurrection!
        notDeadYet = this;
    }
}
```

Cuando una instancia de `CaptainJack` vuelve inaccesible y el recolector de basura intenta reclamarla, el método `finalize()` asignará una referencia a la instancia a la variable `notDeadYet` . Eso hará que la instancia sea accesible una vez más, y el recolector de basura no la eliminará.

Pregunta: ¿Es el Capitán Jack inmortal?

Respuesta: No.

El problema es que la JVM solo ejecutará un finalizador en un objeto una vez en su vida. Si asigna `null` a `notDeadYet` hace que una instancia resurectada no sea accesible una vez más, el recolector de basura no llamará a `finalize()` en el objeto.

1 - Ver https://en.wikipedia.org/wiki/Jack_Harkness .

Activación manual de GC

Puede activar manualmente el recolector de basura llamando

```
System.gc();
```

Sin embargo, Java no garantiza que el recolector de basura se haya ejecutado cuando se devuelva la llamada. Este método simplemente "sugiere" a la JVM (Java Virtual Machine) que desea que ejecute el recolector de basura, pero no lo obliga a hacerlo.

En general, se considera una mala práctica tratar de activar manualmente la recolección de basura. La JVM se puede ejecutar con la `-XX:+DisableExplicitGC` para deshabilitar las llamadas a `System.gc()` . La activación de la recolección de basura mediante una llamada a `System.gc()` puede interrumpir las actividades normales de gestión de basura / promoción de objetos de la implementación específica del recolector de basura en uso por la JVM.

Recolección de basura

El enfoque de C ++ - nuevo y eliminar

En un lenguaje como C ++, el programa de aplicación es responsable de administrar la memoria utilizada por la memoria asignada dinámicamente. Cuando se crea un objeto en el ++ montón usando el C `new` operador, es necesario que haya un uso correspondiente de la `delete` operador para disponer del objeto:

- Si el programa se olvida de `delete` un objeto y simplemente se "olvida" de él, la memoria asociada se pierde en la aplicación. El término para esta situación es una *pérdida de memoria* , y demasiada *pérdida de* memoria de una aplicación puede usar más y más memoria, y eventualmente bloquearse.
- Por otro lado, si una aplicación intenta `delete` el mismo objeto dos veces, o usar un objeto después de que se haya eliminado, la aplicación puede fallar debido a problemas con la corrupción de la memoria.

En un programa de C ++ complicado, la implementación de la administración de memoria usando `new` y `delete` puede llevar mucho tiempo. De hecho, la gestión de la memoria es una fuente común de errores.

El enfoque de Java - recolección de basura

Java tiene un enfoque diferente. En lugar de un operador de `delete` explícita, Java proporciona un mecanismo automático conocido como recolección de basura para recuperar la memoria utilizada por objetos que ya no son necesarios. El sistema de tiempo de ejecución de Java asume la responsabilidad de encontrar los objetos que se deben eliminar. Esta tarea es realizada por un componente llamado *recolector de basura* , o GC para abreviar.

En cualquier momento durante la ejecución de un programa Java, podemos dividir el conjunto de todos los objetos existentes en dos subconjuntos distintos [1] :

- Los objetos accesibles son definidos por el JLS de la siguiente manera:

  > Un objeto accesible es cualquier objeto al que se pueda acceder en cualquier posible cálculo continuo desde cualquier hilo en vivo.

  En la práctica, esto significa que hay una cadena de referencias a partir de una variable local dentro del alcance o una variable `static` por la cual algún código podría alcanzar el objeto.

- Los objetos inalcanzables son objetos a los que *no se* puede acceder como se mencionó anteriormente.

Cualquier objeto que sea inalcanzable es *elegible* para la recolección de basura. Esto no significa que *serán* recolectados en la basura. De hecho:

- Un objeto inalcanzable *no* se recolecta inmediatamente al volverse inalcanzable [1] .
- Un objeto inalcanzable *nunca* puede ser recogido basura.

La especificación del lenguaje Java le da mucha latitud a una implementación de JVM para decidir cuándo recoger objetos inalcanzables. También (en la práctica) otorga permiso para que

una implementación de JVM sea conservadora en la forma en que detecta objetos inalcanzables.

Lo único que garantiza el JLS es que nunca se recogerá la basura en ningún objeto *accesible* .

¿Qué sucede cuando un objeto se vuelve inalcanzable?

En primer lugar, nada sucede específicamente cuando un objeto se *vuelve* inalcanzable. Las cosas solo suceden cuando el recolector de basura se ejecuta *y* detecta que el objeto es inalcanzable. Además, es común que una ejecución de GC no detecte todos los objetos inalcanzables.

Cuando el GC detecta un objeto inalcanzable, pueden ocurrir los siguientes eventos.

1. Si hay objetos de `Reference` que se refieren al objeto, esas referencias se borrarán antes de que se elimine el objeto.
2. Si el objeto es *finalizable* , entonces será finalizado. Esto sucede antes de que se elimine el objeto.
3. El objeto se puede eliminar y la memoria que ocupa se puede recuperar.

Tenga en cuenta que hay una secuencia clara en la que *pueden* ocurrir los eventos anteriores, pero nada requiere que el recolector de basura realice la eliminación final de cualquier objeto específico en un período de tiempo específico.

Ejemplos de objetos alcanzables e inalcanzables

Considere las siguientes clases de ejemplo:

```
// A node in simple "open" linked-list.
public class Node {
    private static int counter = 0;

    public int nodeNumber = ++counter;
    public Node next;
}

public class ListTest {
    public static void main(String[] args) {
        test();                     // M1
        System.out.prinln("Done"); // M2
    }

    private static void test() {
        Node n1 = new Node();      // T1
        Node n2 = new Node();      // T2
        Node n3 = new Node();      // T3
        n1.next = n2;              // T4
        n2 = null;                 // T5
        n3 = null;                 // T6
    }
}
```

Examinemos lo que sucede cuando se llama `test()` . Las declaraciones T1, T2 y T3 crean objetos de `Node` , y todos los objetos son accesibles a través de las variables `n1` , `n2` y `n3` , respectivamente. La declaración T4 asigna la referencia al objeto del segundo `Node` al `next` campo del primero. Cuando se hace eso, se puede `Node` al 2do `Node` a través de dos rutas:

```
n2 -> Node2
n1 -> Node1, Node1.next -> Node2
```

En la declaración T5, asignamos `null` a `n2` . Esto rompe la primera de las cadenas de accesibilidad para `Node2` , pero la segunda permanece intacta, por lo que `Node2` todavía es accesible.

En la declaración T6, asignamos `null` a `n3` . Esto rompe la única cadena de accesibilidad para `Node3` , lo que hace que `Node3` sea inalcanzable. Sin embargo, `Node1` y `Node2` aún son accesibles a través de la variable `n1` .

Finalmente, cuando el método `test()` regresa, sus variables locales `n1` , `n2` y `n3` quedan fuera del alcance y, por lo tanto, nadie puede acceder a ellas. Esto rompe las cadenas de accesibilidad restantes para `Node1` y `Node2` , y todos los objetos de `Node` son inalcanzables y son *elegibles* para la recolección de basura.

---

1 - Esta es una simplificación que ignora la finalización y las clases de `Reference` . 2 - Hipotéticamente, una implementación de Java podría hacer esto, pero el costo de rendimiento de hacerlo lo hace impráctico.

Configuración de los tamaños Heap, PermGen y Stack

Cuando se inicia una máquina virtual Java, necesita saber qué tan grande es el Heap y el tamaño predeterminado para las pilas de hilos. Estos se pueden especificar usando las opciones de línea de comando en el comando `java` . Para las versiones de Java anteriores a Java 8, también puede especificar el tamaño de la región PermGen del montón.

Tenga en cuenta que PermGen se eliminó en Java 8, y si intenta establecer el tamaño de PermGen, la opción se ignorará (con un mensaje de advertencia).

Si no especifica explícitamente los tamaños de pila y pila, la JVM utilizará los valores predeterminados que se calculan de una manera específica de la versión y la plataforma. Esto puede hacer que su aplicación use muy poca o demasiada memoria. Esto suele ser correcto para las pilas de hilos, pero puede ser problemático para un programa que utiliza mucha memoria.

Configuración de los tamaños de pila, PermGen y pila predeterminada:

Las siguientes opciones de JVM establecen el tamaño del montón:

- `-Xms<size>` : establece el tamaño del montón inicial
- `-Xmx<size>` : establece el tamaño máximo del montón
- `-XX:PermSize<size>` - establece el tamaño inicial de PermGen
- `-XX:MaxPermSize<size>` : establece el tamaño máximo de PermGen
- `-Xss<size>` : establece el tamaño de pila de subprocesos predeterminado

El parámetro `<size>` puede ser un número de bytes, o puede tener un sufijo de `k` , `m` o `g` . Los últimos especifican el tamaño en kilobytes, megabytes y gigabytes respectivamente.

Ejemplos:

```
$ java -Xms512m -Xmx1024m JavaApp
$ java -XX:PermSize=64m -XX:MaxPermSize=128m JavaApp
$ java -Xss512k JavaApp
```

Encontrar los tamaños por defecto:

La `-XX:+printFlagsFinal` se puede usar para imprimir los valores de todas las banderas antes de iniciar la JVM. Esto se puede usar para imprimir los valores predeterminados para las configuraciones de tamaño de pila y pila de la siguiente manera:

- Para Linux, Unix, Solaris y Mac OSX.

  $ java -XX: + PrintFlagsFinal -version | grep -iE 'HeapSize | PermSize | ThreadStackSize'

- Para ventanas:

  java -XX: + PrintFlagsFinal -version | findstr / i "HeapSize PermSize ThreadStackSize"

La salida de los comandos anteriores se asemejará a lo siguiente:

```
uintx InitialHeapSize                          := 20655360        {product}
uintx MaxHeapSize                              := 331350016       {product}
uintx PermSize                                  = 21757952        {pd product}
uintx MaxPermSize                               = 85983232        {pd product}
 intx ThreadStackSize                           = 1024            {pd product}
```

Los tamaños se dan en bytes.

Fugas de memoria en Java

En el ejemplo de recolección de basura , implicamos que Java resuelve el problema de las fugas de memoria. Esto no es realmente cierto. Un programa Java puede perder memoria, aunque las causas de las fugas son bastante diferentes.

Los objetos alcanzables pueden fugarse

Considere la siguiente implementación de pila ingenua.

```
public class NaiveStack {
    private Object[] stack = new Object[100];
    private int top = 0;

    public void push(Object obj) {
        if (top >= stack.length) {
            throw new StackException("stack overflow");
```

```
        }
        stack[top++] = obj;
    }

    public Object pop() {
        if (top <= 0) {
            throw new StackException("stack underflow");
        }
        return stack[--top];
    }

    public boolean isEmpty() {
        return top == 0;
    }
}
```

Cuando `push` un objeto y luego lo haces `pop` , todavía habrá una referencia al objeto en la matriz de la `stack` .

La lógica de la implementación de la pila significa que esa referencia no se puede devolver a un cliente de la API. Si se ha extraído un objeto, podemos probar que no se *puede acceder a él en ningún posible cálculo continuo desde cualquier secuencia activa* . El problema es que una JVM de generación actual no puede probar esto. Las JVM de generación actual no tienen en cuenta la lógica del programa para determinar si se puede acceder a las referencias. (Para empezar, no es práctico).

Pero dejando de lado el problema de lo que realmente significa *accesibilidad* , claramente tenemos una situación aquí donde la implementación de `NaiveStack` está " `NaiveStack` " a objetos que deberían ser reclamados. Eso es una pérdida de memoria.

En este caso, la solución es sencilla:

```
    public Object pop() {
        if (top <= 0) {
            throw new StackException("stack underflow");
        }
        Object popped = stack[--top];
        stack[top] = null;              // Overwrite popped reference with null.
        return popped;
    }
```

Los cachés pueden ser fugas de memoria.

Una estrategia común para mejorar el rendimiento del servicio es almacenar en caché los resultados. La idea es que mantenga un registro de las solicitudes comunes y sus resultados en una estructura de datos en memoria conocida como caché. Luego, cada vez que se realiza una solicitud, busca la solicitud en el caché. Si la búsqueda tiene éxito, devuelve los resultados guardados correspondientes.

Esta estrategia puede ser muy efectiva si se implementa adecuadamente. Sin embargo, si se implementa incorrectamente, un caché puede ser una pérdida de memoria. Considere el siguiente ejemplo:

```
public class RequestHandler {
    private Map<Task, Result> cache = new HashMap<>();

    public Result doRequest(Task task) {
        Result result = cache.get(task);
        if (result == null) {
            result == doRequestProcessing(task);
            cache.put(task, result);
        }
        return result;
    }
}
```

El problema con este código es que, si bien cualquier llamada a `doRequest` puede agregar una nueva entrada al caché, no hay nada para eliminarlos. Si el servicio obtiene continuamente diferentes tareas, la memoria caché consumirá toda la memoria disponible. Esta es una forma de pérdida de memoria.

Un método para resolver esto es utilizar un caché con un tamaño máximo y eliminar las entradas antiguas cuando el caché excede el máximo. (Lanzar la entrada menos utilizada recientemente es una buena estrategia). Otro método consiste en crear la memoria caché utilizando `WeakHashMap` para que JVM pueda expulsar las entradas de la memoria caché si el montón comienza a llenarse demasiado.

Lea Gestión de memoria de Java en línea: https://riptutorial.com/es/java/topic/2804/gestion-de-memoria-de-java

Capítulo 90: Gráficos 2D en Java

Introducción

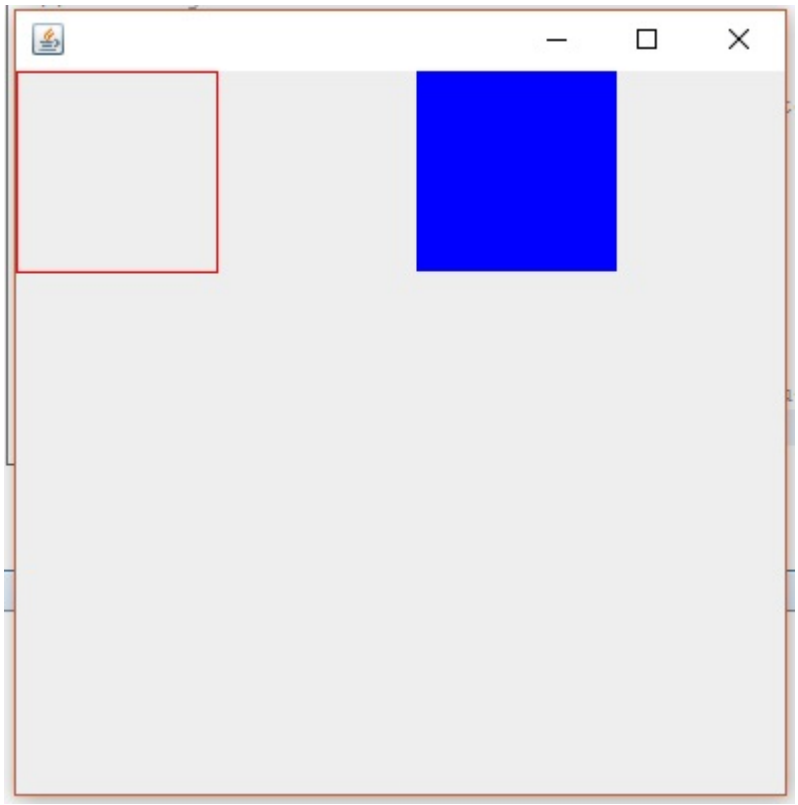
Los gráficos son imágenes visuales o diseños en alguna superficie, como una pared, lienzo, pantalla, papel o piedra para informar, ilustrar o entretener. Incluye: representación gráfica de datos, como en el diseño y fabricación asistidos por computadora, en la composición tipográfica y las artes gráficas, y en software educativo y recreativo. Las imágenes que son generadas por una computadora se llaman gráficos de computadora.

La API Java 2D es potente y compleja. Hay varias formas de hacer gráficos 2D en Java.

Examples

Ejemplo 1: dibujar y rellenar un rectángulo utilizando Java

Este es un ejemplo que imprime un rectángulo y color de relleno en el rectángulo.



https://i.stack.imgur.com/dlC5v.jpg

La mayoría de los métodos de la clase Gráficos se pueden dividir en dos grupos básicos:

1. Dibuje y complete métodos, que le permiten representar formas básicas, texto e imágenes.
2. Métodos de configuración de atributos, que afectan la forma en que aparecen los dibujos y rellenos.

Ejemplo de código: Comencemos con un pequeño ejemplo de dibujo de un rectángulo y color de relleno. Allí declaramos dos clases, una clase es MyPanel y otra clase es prueba. En la clase

MyPanel usamos mathods drawRect () & fillRect () para dibujar un rectángulo y colorear en él. Establecemos el color mediante el método setColor (Color.blue). En Segunda clase, probamos nuestro gráfico, que es la clase de prueba, creamos un cuadro y colocamos MyPanel con p = nuevo objeto MyPanel (). Al ejecutar la clase de prueba, vemos un rectángulo y un rectángulo de color azul.

Primera clase: MyPanel

```
import javax.swing.*;
import java.awt.*;
// MyPanel extends JPanel, which will eventually be placed in a JFrame
public class MyPanel extends JPanel {
    // custom painting is performed by the paintComponent method
    @Override
    public void paintComponent(Graphics g){
        // clear the previous painting
        super.paintComponent(g);
        // cast Graphics to Graphics2D
        Graphics2D g2 = (Graphics2D) g;
        g2.setColor(Color.red); // sets Graphics2D color
        // draw the rectangle
        g2.drawRect(0,0,100,100); // drawRect(x-position, y-position, width, height)
        g2.setColor(Color.blue);
        g2.fillRect(200,0,100,100); // fill new rectangle with color blue
    }
}
```

Segunda clase: prueba

```
import javax.swing.;
import java.awt.;
public class Test { //the Class by which we display our rectangle
    JFrame f;
    MyPanel p;
    public Test(){
        f = new JFrame();
        // get the content area of Panel.
        Container c = f.getContentPane();
        // set the LayoutManager
        c.setLayout(new BorderLayout());
        p = new MyPanel();
        // add MyPanel object into container
        c.add(p);
        // set the size of the JFrame
        f.setSize(400,400);
        // make the JFrame visible
        f.setVisible(true);
        // sets close behavior; EXIT_ON_CLOSE invokes System.exit(0) on closing the JFrame
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String args[ ]){
        Test t = new Test();
    }
}
```

Para obtener más información sobre el diseño de los bordes:

https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html

paintComponent ()

- Es un método principal para pintar.
- Por defecto, primero pinta el fondo.
- Después de eso, realiza pintura personalizada (dibujo de círculo, rectángulos, etc.)

Graphic2D se refiere a la clase Graphic2D

Nota: la API Java 2D le permite realizar fácilmente las siguientes tareas:

- Dibuja líneas, rectángulos y cualquier otra forma geométrica.
- Rellena esas formas con colores sólidos o degradados y texturas.
- Dibuje texto con opciones para un control preciso sobre la fuente y el proceso de renderizado.
- Dibujar imágenes, opcionalmente aplicando operaciones de filtrado.
- Aplique operaciones como la composición y la transformación durante cualquiera de las operaciones de representación anteriores.

Ejemplo 2: Dibujo y relleno oval

```
import javax.swing.*;
import java.awt.*;

public class MyPanel extends JPanel {
    @Override
    public void paintComponent(Graphics g){
        // clear the previous painting
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setColor(Color.blue);
        g2.drawOval(0, 0, 20,20);
        g2.fillOval(50,50,20,20);
    }
}
```

g2.drawOval (int x, int y, int height, int ancho);
Este método dibujará un óvalo en las posiciones x e y especificadas con la altura y el ancho dados.

g2.fillOval (int x, int y, int height, int ancho); Este método llenará un óvalo en las posiciones x e y especificadas con la altura y el ancho dados.

Lea Gráficos 2D en Java en línea: https://riptutorial.com/es/java/topic/10127/graficos-2d-en-java

Capítulo 91: Hechiceros y Setters

Introducción

Este artículo analiza los captadores y setters; La forma estándar de proporcionar acceso a los datos en las clases de Java.

Examples

Adición de Getters y Setters

La encapsulación es un concepto básico en la POO. Se trata de envolver los datos y el código como una sola unidad. En este caso, es una buena práctica declarar las variables como `private` y luego acceder a ellas a través de `Getters` y `Setters` para verlas y / o modificarlas.

```
public class Sample {
  private String  name;
  private int age;

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

No se puede acceder a estas variables privadas directamente desde fuera de la clase. Por lo tanto, están protegidos contra el acceso no autorizado. Pero si desea verlos o modificarlos, puede usar los Getters y Setters.

`getXxx()` método `getXxx()` devolverá el valor actual de la variable `xxx` , mientras que puede establecer el valor de la variable `xxx` utilizando `setXxx()` .

La convención de nomenclatura de los métodos es (en la variable de ejemplo se llama `variableName`):

- Todas las variables no `boolean`

  ```
  getVariableName()   //Getter, The variable name should start with uppercase
  setVariableName(..) //Setter, The variable name should start with uppercase
  ```

- variables `boolean`

```
isVariableName()     //Getter, The variable name should start with uppercase
setVariableName(...) //Setter, The variable name should start with uppercase
```

Los captadores y definidores públicos forman parte de la definición de propiedad de un Java Bean.

Usar un setter o getter para implementar una restricción

Setters y Getters permiten que un objeto contenga variables privadas a las que se puede acceder y cambiar con restricciones. Por ejemplo,

```
public class Person {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if(name!=null && name.length()>2)
           this.name = name;
    }
}
```

En esta clase de `Person` , hay una sola variable: `name` . Se puede acceder a esta variable usando el método `getName()` y cambiarla usando el `setName(String)` , sin embargo, establecer un nombre requiere que el nuevo nombre tenga una longitud mayor que 2 caracteres y no sea nulo. El uso de un método de establecimiento en lugar de hacer público el `name` la variable permite a otros establecer el valor del `name` con ciertas restricciones. Lo mismo se puede aplicar al método getter:

```
public String getName(){
   if(name.length()>16)
      return "Name is too large!";
   else
      return name;
}
```

En el método `getName()` modificado anterior, el `name` se devuelve solo si su longitud es menor o igual a 16. De lo contrario, se devuelve `"Name is too large"` . Esto permite al programador crear variables que son alcanzables y modificables como lo deseen, evitando que las clases de clientes editen las variables de forma no deseada.

¿Por qué usar Getters y Setters?

Considere una clase básica que contiene un objeto con captadores y definidores en Java:

```
public class CountHolder {
  private int count = 0;
```

```
  public int getCount() { return count; }
  public void setCount(int c) { count = c; }
}
```

No podemos acceder a la variable de `count` porque es privada. Pero podemos acceder a los `getCount()` y `setCount(int)` porque son públicos. Para algunos, esto podría plantear la cuestión; ¿Por qué introducir el intermediario? ¿Por qué no simplemente hacer que cuenten público?

```
public class CountHolder {
  public int count = 0;
}
```

Para todos los efectos, estos dos son exactamente iguales, en cuanto a funcionalidad. La diferencia entre ellos es la extensibilidad. Considera lo que dice cada clase:

- **Primero** : "Tengo un método que le dará un valor `int` , y un método que establecerá ese valor en otro `int` ".
- **Segundo** : "Tengo un `int` que puedes configurar y obtener como desees".

Estos pueden sonar similares, pero el primero es en realidad mucho más protegido en su naturaleza; solo te permite interactuar con su naturaleza interna según **lo** dicte. Esto deja el balón en su corte; Se llega a elegir cómo se producen las interacciones internas. El segundo ha expuesto su implementación interna externamente, y ahora no solo es propenso a usuarios externos, sino que, en el caso de una API, se **compromete** a mantener esa implementación (o, de lo contrario, a liberar una API no compatible con versiones anteriores).

Consideremos si queremos sincronizar el acceso para modificar y acceder al conteo. En el primero, esto es simple:

```
public class CountHolder {
  private int count = 0;

  public synchronized int getCount() { return count; }
  public synchronized void setCount(int c) { count = c; }
}
```

pero en el segundo ejemplo, esto es ahora casi imposible sin pasar por y modificar cada lugar donde se hace referencia a la variable de `count` . Peor aún, si este es un elemento que está proporcionando en una biblioteca para que otros lo consuman, **no** tiene una forma de realizar esa modificación y se ve obligado a tomar la decisión difícil mencionada anteriormente.

Así que plantea la pregunta; ¿Son las variables públicas algo bueno (o, al menos, no malvado)?

No estoy seguro Por un lado, puede ver ejemplos de variables públicas que han pasado la prueba del tiempo (IE: la variable de `out` referenciada en `System.out`). Por otro lado, proporcionar una variable pública no ofrece beneficios más allá de los gastos generales extremadamente mínimos y la posible reducción de la notoriedad. Mi pauta aquí sería que, si planea hacer pública una variable, debe juzgarla con estos criterios con **un** prejuicio **extremo** :

1. La variable debe tener ninguna razón concebible cambiar **nunca** en su aplicación. Esto es algo que es extremadamente fácil de arruinar (y, aunque lo hagas bien, los requisitos pueden cambiar), por lo que el método común es el método común. Si va a tener una variable pública, esto realmente debe ser pensado, especialmente si se publica en una biblioteca / framework / API.
2. La variable debe ser referenciada con la frecuencia suficiente para que las ganancias mínimas de reducir la verbosidad lo justifiquen. Ni siquiera creo que la sobrecarga para usar un método en lugar de una referencia directa deba considerarse aquí. Es demasiado insignificante para lo que estimaría de manera conservadora que es el 99.9% de las aplicaciones.

Probablemente hay más de lo que no he considerado de la cabeza. Si alguna vez tienes dudas, usa siempre getters / setters.

Lea Hechiceros y Setters en línea: https://riptutorial.com/es/java/topic/3560/hechiceros-y-setters

Capítulo 92: Herencia

Introducción

La herencia es una característica básica orientada a objetos en la que una clase adquiere y se extiende sobre las propiedades de otra clase, utilizando la palabra clave `extends` . Para Interfaces y los `implements` palabras clave, vea interfaces .

Sintaxis

- clase ClassB extiende ClassA {...}
- clase ClassB implementa InterfaceA {...}
- interface InterfaceB extiende InterfaceA {...}
- clase ClassB extiende ClassA implementa InterfaceC, InterfaceD {...}
- clase abstracta AbstractClassB extiende ClassA {...}
- clase abstracta AbstractClassB extiende AbstractClassA {...}
- clase abstracta AbstractClassB extiende ClassA implementa InterfaceC, InterfaceD {...}

Observaciones

La herencia a menudo se combina con los genéricos de modo que la clase base tenga uno o más parámetros de tipo. Consulte Creación de una clase genérica .

Examples

Clases abstractas

Una clase abstracta es una clase marcada con la palabra clave `abstract` . Contrariamente a la clase no abstracta, puede contener métodos abstractos sin implementación. Sin embargo, es válido crear una clase abstracta sin métodos abstractos.

Una clase abstracta no puede ser instanciada. Puede ser subclasificado (extendido) siempre que la subclase sea abstracta, o implemente todos los métodos marcados como abstractos por súper clases.

Un ejemplo de una clase abstracta:

```
public abstract class Component {
    private int x, y;

    public setPosition(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public abstract void render();
```

```
}
```

La clase debe estar marcada como abstracta, cuando tiene al menos un método abstracto. Un método abstracto es un método que no tiene implementación. Se pueden declarar otros métodos dentro de una clase abstracta que tiene implementación para proporcionar código común para cualquier subclase.

Intentar crear una instancia de esta clase proporcionará un error de compilación:

```
//error: Component is abstract; cannot be instantiated
Component myComponent = new Component();
```

Sin embargo, una clase que amplía el `Component` y proporciona una implementación para todos sus métodos abstractos y se puede instanciar.

```
public class Button extends Component {

    @Override
    public void render() {
        //render a button
    }
}

public class TextBox extends Component {

    @Override
    public void render() {
        //render a textbox
    }
}
```

Las instancias de clases heredadas también se pueden convertir como la clase principal (herencia normal) y proporcionan un efecto polimórfico cuando se llama al método abstracto.

```
Component myButton = new Button();
Component myTextBox = new TextBox();

myButton.render(); //renders a button
myTextBox.render(); //renders a text box
```

Clases abstractas vs interfaces

Tanto las clases abstractas como las interfaces proporcionan una forma de definir firmas de métodos al tiempo que requieren que la clase extendida / implementada proporcione la implementación.

Hay dos diferencias clave entre las clases abstractas y las interfaces:

- Una clase solo puede extender una sola clase, pero puede implementar muchas interfaces.
- Una clase abstracta puede contener campos de instancia (no `static`), pero las interfaces solo pueden contener campos `static` .

Java SE 8

Los métodos declarados en las interfaces no pueden contener implementaciones, por lo que se usaron clases abstractas cuando fue útil proporcionar métodos adicionales que las implementaciones llamaron métodos abstractos.

Java SE 8

Java 8 permite que las interfaces contengan métodos predeterminados , generalmente implementados usando los otros métodos de la interfaz , haciendo que las interfaces y las clases abstractas sean igualmente poderosas en este sentido.

Subclases anónimas de clases abstractas

Como conveniencia, java permite crear instancias anónimas de subclases de clases abstractas, que proporcionan implementaciones para los métodos abstractos al crear el nuevo objeto. Usando el ejemplo anterior, esto podría verse así:

```
Component myAnonymousComponent = new Component() {
    @Override
    public void render() {
        // render a quick 1-time use component
    }
}
```

Herencia estática

El método estático se puede heredar de forma similar a los métodos normales, sin embargo, a diferencia de los métodos normales, es imposible crear métodos " abstractos " para forzar la anulación del método estático. Escribir un método con la misma firma que un método estático en una súper clase parece ser una forma de anulación, pero en realidad esto simplemente crea una nueva función que oculta a la otra.

```
public class BaseClass {

    public static int num = 5;

    public static void sayHello() {
        System.out.println("Hello");
    }

    public static void main(String[] args) {
        BaseClass.sayHello();
        System.out.println("BaseClass's num: " + BaseClass.num);

        SubClass.sayHello();
        //This will be different than the above statement's output, since it runs
        //A different method
        SubClass.sayHello(true);

        StaticOverride.sayHello();
        System.out.println("StaticOverride's num: " + StaticOverride.num);
    }
}

public  class SubClass extends BaseClass {
```

```
    //Inherits the sayHello function, but does not override it
    public static void sayHello(boolean test) {
        System.out.println("Hey");
    }
}

public static class StaticOverride extends BaseClass {

    //Hides the num field from BaseClass
    //You can even change the type, since this doesn't affect the signature
    public static String num = "test";

    //Cannot use @Override annotation, since this is static
    //This overrides the sayHello method from BaseClass
    public static void sayHello() {
        System.out.println("Static says Hi");
    }

}
```

La ejecución de cualquiera de estas clases produce el resultado:

```
Hello
BaseClass's num: 5
Hello
Hey
Static says Hi
StaticOverride's num: test
```

Tenga en cuenta que, a diferencia de la herencia normal, en los métodos de herencia estática no están ocultos. Siempre puede llamar al método base `sayHello` usando `BaseClass.sayHello()` . Pero las clases heredan métodos estáticos *si* no se encuentran métodos con la misma firma en la subclase. Si las firmas de dos métodos varían, ambos métodos pueden ejecutarse desde la subclase, incluso si el nombre es el mismo.

Los campos estáticos se ocultan entre sí de forma similar.

Usando 'final' para restringir la herencia y anular

Clases finales

Cuando se usa en una declaración de `class` , el modificador `final` evita que se declaren otras clases que `extend` la clase. Una clase `final` es una clase "hoja" en la jerarquía de clases de herencia.

```
// This declares a final class
final class MyFinalClass {
    /* some code */
}

// Compilation error: cannot inherit from final MyFinalClass
class MySubClass extends MyFinalClass {
    /* more code */
```

```
}
```

Casos de uso para las clases finales.

Las clases finales se pueden combinar con un constructor `private` para controlar o evitar la creación de instancias de una clase. Esto se puede usar para crear una llamada "clase de utilidad" que solo define miembros estáticos; Es decir, constantes y métodos estáticos.

```
public final class UtilityClass {

    // Private constructor to replace the default visible constructor
    private UtilityClass() {}

    // Static members can still be used as usual
    public static int doSomethingCool() {
        return 123;
    }

}
```

Las clases inmutables también deben ser declaradas como `final` . (Una clase inmutable es aquella cuyas instancias no se pueden cambiar después de haber sido creadas; consulte el tema Imutables de objetos). Al hacer esto, hace imposible crear una subclase mutable de una clase inmutable. Eso violaría el principio de sustitución de Liskov, que exige que un subtipo obedezca el "contrato de comportamiento" de sus supertipos.

Desde una perspectiva práctica, declarar que una clase inmutable es `final` hace que sea más fácil razonar sobre el comportamiento del programa. También aborda los problemas de seguridad en el escenario donde el código no confiable se ejecuta en un entorno limitado de seguridad. (Por ejemplo, como `String` se declara como `final` , una clase confiable no tiene que preocuparse de que pueda ser engañada para que acepte una subclase mutable, que la persona que llama no de confianza podría cambiar de manera subrepticia).

Una desventaja de las clases `final` es que no funcionan con algunos marcos burlones como Mockito. Actualización: la versión 2 de Mockito ahora admite la burla de las clases finales.

Métodos finales

El modificador `final` también se puede aplicar a los métodos para evitar que se sobrescriban en las subclases:

```
public class MyClassWithFinalMethod {

    public final void someMethod() {
    }
}

public class MySubClass extends MyClassWithFinalMethod {

    @Override
```

```
    public void someMethod() { // Compiler error (overridden method is final)
    }
}
```

Los métodos finales se usan normalmente cuando se quiere restringir lo que una subclase puede cambiar en una clase sin prohibir por completo las subclases.

El modificador `final` también se puede aplicar a las variables, pero el significado de `final` para las variables no está relacionado con la herencia.

El principio de sustitución de Liskov

Posibilidad de sustitución es un principio en la programación orientada a objetos introducidos por Barbara Liskov en un discurso de apertura 1987 declarando que, si la clase `B` es una subclase de la clase `A` , a continuación, siempre que sea `A` se espera, `B` se puede utilizar en su lugar:

```
class A {...}
class B extends A {...}

public void method(A obj) {...}

A a = new B(); // Assignment OK
method(new B()); // Passing as parameter OK
```

Esto también se aplica cuando el tipo es una interfaz, donde no se necesita ninguna relación jerárquica entre los objetos:

```
interface Foo {
    void bar();
}

class A implements Foo {
    void bar() {...}
}

class B implements Foo {
    void bar() {...}
}

List<Foo> foos = new ArrayList<>();
foos.add(new A()); // OK
foos.add(new B()); // OK
```

Ahora la lista contiene objetos que no son de la misma jerarquía de clases.

Herencia

Con el uso de la `extends` palabra clave entre las clases, todas las propiedades de la superclase (también conocido como la *clase padre* o *Clase Base)* están presentes en la subclase (también conocida como la *clase hija* o *clase derivada)*

```
public class BaseClass {

    public void baseMethod(){
        System.out.println("Doing base class stuff");
    }
}

public class SubClass extends BaseClass {

}
```

Las instancias de `SubClass` han *heredado* el método `baseMethod()` :

```
SubClass s = new SubClass();
s.baseMethod();  //Valid, prints "Doing base class stuff"
```

Se puede agregar contenido adicional a una subclase. Si lo hace, permite la funcionalidad adicional en la subclase sin ningún cambio en la clase base o cualquier otra subclase de la misma clase base:

```
public class Subclass2 extends BaseClass {

    public void anotherMethod() {
        System.out.println("Doing subclass2 stuff");
    }
}

Subclass2 s2 = new Subclass2();
s2.baseMethod(); //Still valid , prints "Doing base class stuff"
s2.anotherMethod(); //Also valid, prints "Doing subclass2 stuff"
```

Los campos también se heredan:

```
public class BaseClassWithField {

    public int x;

}

public class SubClassWithField extends BaseClassWithField {

    public SubClassWithField(int x) {
        this.x = x; //Can access fields
    }
}
```

`private` campos y métodos `private` aún existen dentro de la subclase, pero no son accesibles:

```
public class BaseClassWithPrivateField {

    private int x = 5;

    public int getX() {
        return x;
    }
```

```
}

public class SubClassInheritsPrivateField extends BaseClassWithPrivateField {

    public void printX() {
        System.out.println(x); //Illegal, can't access private field x
        System.out.println(getX()); //Legal, prints 5
    }
}

SubClassInheritsPrivateField s = new SubClassInheritsPrivateField();
int x = s.getX(); //x will have a value of 5.
```

En Java, cada clase puede extenderse a lo sumo a otra clase.

```
public class A{}
public class B{}
public class ExtendsTwoClasses extends A, B {} //Illegal
```

Esto se conoce como herencia múltiple, y si bien es legal en algunos idiomas, Java no lo permite con clases.

Como resultado de esto, cada clase tiene una cadena de clases ancestrales no ramificadas que conducen a `Object` , de la que descienden todas las clases.

Herencia y métodos estáticos

En Java, las clases padre e hijo pueden tener métodos estáticos con el mismo nombre. Pero en tales casos, la implementación del método estático en hijo oculta la implementación de la clase padre, no se trata de una anulación del método. Por ejemplo:

```
class StaticMethodTest {

  // static method and inheritance
  public static void main(String[] args) {
    Parent p = new Child();
    p.staticMethod(); // prints Inside Parent
    ((Child) p).staticMethod(); // prints Inside Child
  }

  static class Parent {
    public static void staticMethod() {
      System.out.println("Inside Parent");
    }
  }

  static class Child extends Parent {
    public static void staticMethod() {
      System.out.println("Inside Child");
    }
  }
}
```

Los métodos estáticos se unen a una clase y no a una instancia y este enlace de método ocurre en tiempo de compilación. Dado que en la primera llamada a `staticMethod()` , matriz de referencia

de clase `p` se utilizó, `Parent` versión 's de `staticMethod()` se invoca. En el segundo caso, hicimos `p` en `Child` clase `Child` , se ejecutó `staticMethod() Child` .

Sombreado variable

Las variables están SOMBRADAS y los métodos son ANULADOS. La variable que se utilizará depende de la clase de la que se declara la variable. El método que se utilizará depende de la clase real del objeto al que hace referencia la variable.

```
class Car {
    public int gearRatio = 8;

    public String accelerate() {
        return "Accelerate : Car";
    }
}

class SportsCar extends Car {
    public int gearRatio = 9;

    public String accelerate() {
        return "Accelerate : SportsCar";
    }

    public void test() {

    }


    public static void main(String[] args) {

        Car car = new SportsCar();
        System.out.println(car.gearRatio + "  " + car.accelerate());
        // will print out 8  Accelerate : SportsCar
    }
}
```

Reducción y ampliación de referencias de objetos.

Convertir una instancia de una clase base en una subclase como en: `b = (B) a;` se denomina *estrechamiento* (cuando intenta restringir el objeto de clase base a un objeto de clase más específico) y necesita una conversión de tipos explícita.

Convertir una instancia de una subclase en una clase base como en: `A a = b;` Se llama *ampliación* y no necesita una conversión de tipo.

Para ilustrar, considere las siguientes declaraciones de clase y el código de prueba:

```
class Vehicle {
}

class Car extends Vehicle {
}

class Truck extends Vehicle {
```

```
}

class MotorCycle extends Vehicle {
}

class Test {

    public static void main(String[] args) {

        Vehicle vehicle = new Car();
        Car car = new Car();

        vehicle = car; // is valid, no cast needed

        Car c = vehicle // not valid
        Car c = (Car) vehicle; //valid
    }
}
```

La declaración `Vehicle vehicle = new Car();` Es una declaración válida de Java. Cada instancia de `Car` es también un `Vehicle` . Por lo tanto, la asignación es legal sin la necesidad de una conversión de tipos explícita.

Por otro lado, `Car c = vehicle;` no es válido. El tipo estático de la variable de `vehicle` es `Vehicle` que significa que podría referirse a una instancia de `Car` , Camión , Motocicleta , `or any other current or future subclass of` Vehículo . `(Or indeed, an instance of` Vehículo en `itself, since we did not declare it as an class.) The assignment cannot be allowed, since that might lead to` abstracta `class.) The assignment cannot be allowed, since that might lead to` automóvil se `referring to a` instancia de `referring to a` Camión.

Para evitar esta situación, necesitamos agregar una conversión de tipos explícita:

```
Car c = (Car) vehicle;
```

La conversión de tipos le dice al compilador que *esperamos que* el valor del `vehicle` sea un `Car` o una subclase de `Car` . Si es necesario, el compilador insertará el código para realizar una verificación de tipo en tiempo de ejecución. Si la comprobación falla, se emitirá una `ClassCastException` cuando se ejecute el código.

Tenga en cuenta que no todos los tipos de conversión son válidos. Por ejemplo:

```
String s = (String) vehicle;  // not valid
```

El compilador de Java sabe que una instancia que es compatible con Type con `Vehicle` *no puede ser* compatible con `String` . La conversión de tipos nunca podría tener éxito, y JLS exige que esto dé un error de compilación.

Programación a una interfaz

La idea detrás de la programación de una interfaz es basar el código principalmente en interfaces y solo usar clases concretas en el momento de la creación de instancias. En este contexto, el

buen código que se ocupa, por ejemplo, de las colecciones de Java se verá así:

```
public <T> Set<T> toSet(Collection<T> collection) {
  return Sets.newHashSet(collection);
}
```

mientras que el código malo podría tener este aspecto:

```
public <T> HashSet<T> toSet(ArrayList<T> collection) {
  return Sets.newHashSet(collection);
}
```

No solo los primeros se pueden aplicar a una variedad más amplia de argumentos, sino que sus resultados serán más compatibles con el código proporcionado por otros desarrolladores que generalmente se adhieren al concepto de programación para una interfaz. Sin embargo, las razones más importantes para usar el primero son:

- la mayoría de las veces, el contexto en el que se usa el resultado no necesita ni debería necesitar tantos detalles como lo proporciona la implementación concreta;
- adherirse a una interfaz obliga a un código más limpio y menos hacks, como otro método público que se agrega a una clase que atiende a un escenario específico;
- el código es más comprobable ya que las interfaces son fácilmente simulables;
- finalmente, el concepto ayuda incluso si solo se espera una implementación (al menos para la capacidad de prueba).

Entonces, ¿cómo se puede aplicar fácilmente el concepto de programación a una interfaz al escribir un nuevo código teniendo en cuenta una implementación en particular? Una opción que usamos comúnmente es una combinación de los siguientes patrones:

- programación a una interfaz
- fábrica
- constructor

El siguiente ejemplo basado en estos principios es una versión simplificada y truncada de una implementación RPC escrita para varios protocolos diferentes:

```
public interface RemoteInvoker {
  <RQ, RS> CompletableFuture<RS> invoke(RQ request, Class<RS> responseClass);
}
```

Se supone que la interfaz anterior no debe crearse una instancia directamente a través de una fábrica, en su lugar derivamos interfaces más concretas, una para la invocación HTTP y otra para AMQP, cada una con una fábrica y un constructor para construir instancias, que a su vez también son instancias de la interfaz anterior:

```
public interface AmqpInvoker extends RemoteInvoker {
  static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
    return new AmqpInvokerBuilder(instanceId, factory);
  }
}
```

Las instancias de `RemoteInvoker` para el uso con AMQP ahora pueden construirse tan fácilmente (o más involucradas según el constructor):

```
RemoteInvoker invoker = AmqpInvoker.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

Y una invocación de una solicitud es tan fácil como:

```
Response res = invoker.invoke(new Request(data), Response.class).get();
```

Debido a que Java 8 permite la colocación de métodos estáticos directamente en interfaces, la fábrica intermedia se ha convertido en implícita en el código anterior reemplazado con `AmqpInvoker.with()` . En Java anterior a la versión 8, se puede lograr el mismo efecto con una clase de `Factory` interna:

```
public interface AmqpInvoker extends RemoteInvoker {
  class Factory {
    public static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
      return new AmqpInvokerBuilder(instanceId, factory);
    }
  }
}
```

La instanciación correspondiente se convertiría entonces en:

```
RemoteInvoker invoker = AmqpInvoker.Factory.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

El generador utilizado anteriormente podría tener este aspecto (aunque esto es una simplificación, ya que el actual permite definir hasta 15 parámetros que se desvían de los valores predeterminados). Tenga en cuenta que la construcción no es pública, por lo que solo se puede utilizar específicamente desde la interfaz `AmqpInvoker` anterior:

```
public class AmqpInvokerBuilder {
  ...
  AmqpInvokerBuilder(String instanceId, ConnectionFactory factory) {
    this.instanceId = instanceId;
    this.factory = factory;
  }

  public AmqpInvokerBuilder requestRouter(RequestRouter requestRouter) {
    this.requestRouter = requestRouter;
    return this;
  }

  public AmqpInvoker build() throws TimeoutException, IOException {
    return new AmqpInvokerImpl(instanceId, factory, requestRouter);
  }
}
```

En general, un generador también se puede generar utilizando una herramienta como FreeBuilder.

Finalmente, la implementación estándar (y la única esperada) de esta interfaz se define como una clase de paquete local para imponer el uso de la interfaz, la fábrica y el constructor:

```
class AmqpInvokerImpl implements AmqpInvoker {
  AmqpInvokerImpl(String instanceId, ConnectionFactory factory, RequestRouter requestRouter) {
    ...
  }

  @Override
  public <RQ, RS> CompletableFuture<RS> invoke(final RQ request, final Class<RS> respClass) {
    ...
  }
}
```

Mientras tanto, este patrón demostró ser muy eficiente en el desarrollo de todo nuestro nuevo código, sin importar cuán simple o compleja sea la funcionalidad.

Clase abstracta y uso de la interfaz: "Is-a" relationship vs "Has-a" capacity

Cuándo usar clases abstractas: implementar el mismo comportamiento o diferente entre múltiples objetos relacionados

Cuándo usar interfaces: para implementar un contrato por múltiples objetos no relacionados

Las clases abstractas crean que "es una" relación mientras que las interfaces proporcionan "tiene una" capacidad.

Esto se puede ver en el siguiente código:

```
public class InterfaceAndAbstractClassDemo{
    public static void main(String args[]){

        Dog dog = new Dog("Jack",16);
        Cat cat = new Cat("Joe",20);

        System.out.println("Dog:"+dog);
        System.out.println("Cat:"+cat);

        dog.remember();
        dog.protectOwner();
        Learn dl = dog;
        dl.learn();

        cat.remember();
        cat.protectOwner();

        Climb c = cat;
        c.climb();

        Man man = new Man("Ravindra",40);
        System.out.println(man);
```

```
        Climb cm = man;
        cm.climb();
        Think t = man;
        t.think();
        Learn l = man;
        l.learn();
        Apply a = man;
        a.apply();
    }
}

abstract class Animal{
    String name;
    int lifeExpentency;
    public Animal(String name,int lifeExpentency ){
        this.name = name;
        this.lifeExpentency=lifeExpentency;
    }
    public abstract void remember();
    public abstract void protectOwner();

    public String toString(){
        return this.getClass().getSimpleName()+":"+name+":"+lifeExpentency;
    }
}
class Dog extends Animal implements Learn{

    public Dog(String name,int age){
        super(name,age);
    }
    public void remember(){
        System.out.println(this.getClass().getSimpleName()+" can remember for 5 minutes");
    }
    public void protectOwner(){
        System.out.println(this.getClass().getSimpleName()+ " will protect owner");
    }
    public void learn(){
        System.out.println(this.getClass().getSimpleName()+ " can learn:");
    }
}
class Cat extends Animal implements Climb {
    public Cat(String name,int age){
        super(name,age);
    }
    public void remember(){
        System.out.println(this.getClass().getSimpleName() + " can remember for 16 hours");
    }
    public void protectOwner(){
        System.out.println(this.getClass().getSimpleName()+ " won't protect owner");
    }
    public void climb(){
        System.out.println(this.getClass().getSimpleName()+ " can climb");
    }
}
interface Climb{
    void climb();
}
interface Think {
    void think();
}
```

```
interface Learn {
    void learn();
}
interface Apply{
    void apply();
}

class Man implements Think,Learn,Apply,Climb{
    String name;
    int age;

    public Man(String name,int age){
        this.name = name;
        this.age = age;
    }
    public void think(){
        System.out.println("I can think:"+this.getClass().getSimpleName());
    }
    public void learn(){
        System.out.println("I can learn:"+this.getClass().getSimpleName());
    }
    public void apply(){
        System.out.println("I can apply:"+this.getClass().getSimpleName());
    }
    public void climb(){
        System.out.println("I can climb:"+this.getClass().getSimpleName());
    }
    public String toString(){
        return "Man :"+name+":Age:"+age;
    }
}
```

salida:

```
Dog:Dog:Jack:16
Cat:Cat:Joe:20
Dog can remember for 5 minutes
Dog will protect owner
Dog can learn:
Cat can remember for 16 hours
Cat won't protect owner
Cat can climb
Man :Ravindra:Age:40
I can climb:Man
I can think:Man
I can learn:Man
I can apply:Man
```

Notas clave:

1. `Animal` es una clase abstracta con atributos compartidos: `name` y `lifeExpectancy` y métodos abstractos: `remember()` y `protectOwner()` . `Dog` y `Cat` son `Animals` que han implementado los métodos `remember()` y `protectOwner()` .

2. `Cat` puede `climb()` pero el `Dog` no puede. `Dog` puede `think()` pero el `Cat` no puede. Estas capacidades específicas se agregan a `Cat` y `Dog` por implementación.

3. `Man`

no es un `Animal` pero puede `Think` , `Learn` , `Apply` y `Climb` .

4. `Cat` no es un `Man` pero puede `Climb` .

5. `Dog` no es un `Man` pero puede `Learn`

6. `Man` no es ni un `Cat` ni un `Dog` pero puede tener algunas de las capacidades de los dos últimos sin extender `Animal` , `Cat` o `Dog` . Esto se hace con interfaces.

7. Aunque `Animal` es una clase abstracta, tiene un constructor, a diferencia de una interfaz.

TL; DR:

Las clases no relacionadas pueden tener capacidades a través de interfaces, pero las clases relacionadas cambian el comportamiento a través de la extensión de las clases base.

Consulte la página de documentación de Java para comprender cuál usar en un caso de uso específico.

Considera usar clases abstractas si ...

1. Desea compartir código entre varias clases estrechamente relacionadas.
2. Usted espera que las clases que extiendan su clase abstracta tengan muchos métodos o campos comunes, o que requieran modificadores de acceso que no sean públicos (como protegidos y privados).
3. Desea declarar campos no estáticos o no finales.

Considera usar interfaces si ...

1. Usted espera que las clases no relacionadas implementen su interfaz. Por ejemplo, muchos objetos no relacionados pueden implementar la interfaz `Serializable` .
2. Desea especificar el comportamiento de un tipo de datos en particular, pero no le preocupa quién implementa su comportamiento.
3. Desea aprovechar la herencia de tipo múltiple.

Anulando en herencia

La anulación de herencia se usa cuando se usa un método ya definido de una superclase en una subclase, pero de una manera diferente a como se diseñó originalmente el método en la superclase. La anulación permite al usuario reutilizar el código utilizando el material existente y modificándolo para que se adapte mejor a las necesidades del usuario.

El siguiente ejemplo muestra cómo `ClassB` anula la funcionalidad de `ClassA` al cambiar lo que se envía a través del método de impresión:

Ejemplo:

```
public static void main(String[] args) {
    ClassA a = new ClassA();
```

```
    ClassA b = new ClassB();
    a.printing();
    b.printing();
}

class ClassA {
    public void printing() {
        System.out.println("A");
    }
}

class ClassB extends ClassA {
    public void printing() {
         System.out.println("B");
    }
}
```

Salida:

UNA

segundo

Lea Herencia en línea: https://riptutorial.com/es/java/topic/87/herencia

Capítulo 93: Hora local

Sintaxis

- LocalTime time = LocalTime.now (); // Inicializa con el reloj del sistema actual
- LocalTime time = LocalTime.MIDNIGHT; // 00:00
- LocalTime time = LocalTime.NOON; // 12:00
- LocalTime time = LocalTime.of (12, 12, 45); // 12:12:45

Parámetros

Método	Salida
`LocalTime.of(13, 12, 11)`	13:12:11
`LocalTime.MIDNIGHT`	00:00
`LocalTime.NOON`	12:00
`LocalTime.now()`	Hora actual desde el reloj del sistema
`LocalTime.MAX`	La hora local máxima admitida 23: 59: 59.999999999
`LocalTime.MIN`	La hora local mínima admitida 00:00
`LocalTime.ofSecondOfDay(84399)`	23:59:59, obtiene el tiempo del segundo valor del día
`LocalTime.ofNanoOfDay(2000000000)`	00:00:02, Obtiene el tiempo del valor nanos del día

Observaciones

Como el nombre de la clase indica, `LocalTime` representa una hora sin una zona horaria. No representa una fecha. Es una etiqueta simple para un tiempo dado.

La clase está basada en valores y el método `equals` se debe usar al hacer comparaciones.

Esta clase es del paquete java.time.

Examples

Modificación de tiempo

Puedes agregar horas, minutos, segundos y nanosegundos:

```
LocalTime time = LocalTime.now();
```

```
LocalTime addHours = time.plusHours(5); // Add 5 hours
LocaLTime addMinutes = time.plusMinutes(15) // Add 15 minutes
LocalTime addSeconds = time.plusSeconds(30) // Add 30 seconds
LocalTime addNanoseconds = time.plusNanos(150_000_000) // Add 150.000.000ns (150ms)
```

Zonas horarias y su diferencia horaria.

```
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.temporal.ChronoUnit;

public class Test {
    public static void main(String[] args)
    {
            ZoneId zone1 = ZoneId.of("Europe/Berlin");
            ZoneId zone2 = ZoneId.of("Brazil/East");

            LocalTime now = LocalTime.now();
            LocalTime now1 = LocalTime.now(zone1);
            LocalTime now2 = LocalTime.now(zone2);

            System.out.println("Current Time : " + now);
            System.out.println("Berlin Time : " + now1);
            System.out.println("Brazil Time : " + now2);

            long minutesBetween = ChronoUnit.MINUTES.between(now2, now1);
            System.out.println("Minutes Between Berlin and Brazil : " + minutesBetween
+"mins");
    }
}
```

Cantidad de tiempo entre dos LocalTime

Hay dos formas equivalentes de calcular la cantidad de unidad de tiempo entre dos `LocalTime` : (1) `until(Temporal, TemporalUnit)` método `until(Temporal, TemporalUnit)` y (2) `TemporalUnit.between(Temporal, Temporal)` .

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;

public class AmountOfTime {

    public static void main(String[] args) {

        LocalTime start = LocalTime.of(1, 0, 0); // hour, minute, second
        LocalTime end = LocalTime.of(2, 10, 20); // hour, minute, second

        long halfDays1 = start.until(end, ChronoUnit.HALF_DAYS); // 0
        long halfDays2 = ChronoUnit.HALF_DAYS.between(start, end); // 0

        long hours1 = start.until(end, ChronoUnit.HOURS); // 1
        long hours2 = ChronoUnit.HOURS.between(start, end); // 1

        long minutes1 = start.until(end, ChronoUnit.MINUTES); // 70
        long minutes2 = ChronoUnit.MINUTES.between(start, end); // 70
```

```
        long seconds1 = start.until(end, ChronoUnit.SECONDS); // 4220
        long seconds2 = ChronoUnit.SECONDS.between(start, end); // 4220

        long millisecs1 = start.until(end, ChronoUnit.MILLIS); // 4220000
        long millisecs2 = ChronoUnit.MILLIS.between(start, end); // 4220000

        long microsecs1 = start.until(end, ChronoUnit.MICROS); // 4220000000
        long microsecs2 = ChronoUnit.MICROS.between(start, end); // 4220000000

        long nanosecs1 = start.until(end, ChronoUnit.NANOS); // 4220000000000
        long nanosecs2 = ChronoUnit.NANOS.between(start, end); // 4220000000000

        // Using others ChronoUnit will be thrown UnsupportedTemporalTypeException.
        // The following methods are examples thereof.
        long days1 = start.until(end, ChronoUnit.DAYS);
        long days2 = ChronoUnit.DAYS.between(start, end);
    }
}
```

Introducción

LocalTime es una clase inmutable y segura para subprocesos, utilizada para representar el tiempo, a menudo vista como hora-min-seg. El tiempo está representado en nanosegundos de precisión. Por ejemplo, el valor "13: 45.30.123456789" se puede almacenar en un LocalTime.

Esta clase no almacena ni representa una fecha o zona horaria. En cambio, es una descripción de la hora local como se ve en un reloj de pared. No puede representar un instante en la línea de tiempo sin información adicional, como un desplazamiento o una zona horaria. Esta es una clase basada en valores, el método igual debe usarse para comparaciones.

Campos

MAX: el tiempo local máximo admitido, '23: 59: 59.999999999 '. Medianoche, min, mediodía

Métodos estáticos importantes

now (), now (Clock clock), now (ZoneId zone), parse (CharSequence text)

Métodos de instancia importantes

isAfter (LocalTime other), isBefore (LocalTime other), minus (TemporalAmount amountToSubtract), minus (long amountToSubtract, TemporalUnit unit), más (TemporalAmount amountToAdd), plus (long listAdAdd, TemporalUnit unit)

```
ZoneId zone = ZoneId.of("Asia/Kolkata");
LocalTime now = LocalTime.now();
LocalTime now1 = LocalTime.now(zone);
LocalTime then = LocalTime.parse("04:16:40");
```

La diferencia en el tiempo se puede calcular de cualquiera de las siguientes maneras

```
long timeDiff = Duration.between(now, now1).toMinutes();
long timeDiff1 = java.time.temporal.ChronoUnit.MINUTES.between(now2, now1);
```

También puede agregar / restar horas, minutos o segundos de cualquier objeto de LocalTime.

minusHours (long hoursToSubtract), minusMinutes (long hoursToMinutes), minusNanos (long nanosToSubtract), minusSeconds (long secondsToSubtract), plusHours (long hoursToSubtract), plusSsopssperssopssopssperssopssopssperssopssopssonssperssonssperssonss-sos-s.jpg

```
now.plusHours(1L);
now1.minusMinutes(20L);
```

Lea Hora local en línea: https://riptutorial.com/es/java/topic/3065/hora-local

Capítulo 94: HttpURLConnection

Observaciones

- El uso de HttpUrlConnection en Android requiere que agregue el permiso de Internet a su aplicación (en `AndroidManifest.xml`).
- También hay otros clientes y bibliotecas HTTP de Java, como OkHttp de Square, que son más fáciles de usar y pueden ofrecer un mejor rendimiento o más funciones.

Examples

Obtener el cuerpo de respuesta de una URL como una cadena

```
String getText(String url) throws IOException {
    HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();
    //add headers to the connection, or check the status if desired..

    // handle error response code it occurs
    int responseCode = conn.getResponseCode();
    InputStream inputStream;
    if (200 <= responseCode && responseCode <= 299) {
        inputStream = connection.getInputStream();
    } else {
        inputStream = connection.getErrorStream();
    }

    BufferedReader in = new BufferedReader(
        new InputStreamReader(
            inputStream));

    StringBuilder response = new StringBuilder();
    String currentLine;

    while ((currentLine = in.readLine()) != null)
        response.append(currentLine);

    in.close();

    return response.toString();
}
```

Esto descargará los datos de texto de la URL especificada y los devolverá como una cadena.

Cómo funciona esto:

- Primero, creamos una `HttpUrlConnection` desde nuestra URL, con la `new URL(url).openConnection()` . Lanzamos la `UrlConnection` esto vuelve a una `HttpUrlConnection` , por lo que tenemos acceso a cosas como agregar encabezados (como User Agent) o verificar el código de respuesta. (Este ejemplo no lo hace, pero es fácil de agregar).

- Luego, cree `InputStream` basándose en el código de respuesta (para el manejo de errores)
- Luego, cree un `BufferedReader` que nos permita leer el texto de `InputStream` que obtenemos de la conexión.
- Ahora, agregamos el texto a un `StringBuilder` , línea por línea.
- Cierre el `InputStream` y devuelva la cadena que ahora tenemos.

Notas:

- Este método arrojará una `IoException` en caso de error (como un error de red o sin conexión a Internet), y también lanzará una *excepción* `MalformedUrlException` *no verificada* si la URL dada no es válida.
- Puede utilizarse para leer desde cualquier URL que devuelva texto, como páginas web (HTML), API REST que devuelvan JSON o XML, etc.
- Vea también: Lea la URL a Cadena en algunas líneas de código Java .

Uso:

Es muy simple:

```
String text = getText("http://example.com");
//Do something with the text from example.com, in this case the HTML.
```

Datos POST

```
public static void post(String url, byte [] data, String contentType) throws IOException {
    HttpURLConnection connection = null;
    OutputStream out = null;
    InputStream in = null;

    try {
        connection = (HttpURLConnection) new URL(url).openConnection();
        connection.setRequestProperty("Content-Type", contentType);
        connection.setDoOutput(true);

        out = connection.getOutputStream();
        out.write(data);
        out.close();

        in = connection.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        String line = null;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        in.close();

    } finally {
        if (connection != null) connection.disconnect();
        if (out != null) out.close();
```

```
        if (in != null) in.close();
    }
}
```

Esto enviará los datos a la URL especificada y luego leerá la respuesta línea por línea.

Cómo funciona

- Como de costumbre obtenemos el `HttpURLConnection` desde una `URL` .
- Establezca el tipo de contenido usando `setRequestProperty` , por defecto es `application/x-www-form-urlencoded`
- `setDoOutput(true)` le dice a la conexión que enviaremos datos.
- Luego obtenemos el `OutputStream` llamando a `getOutputStream()` y escribimos datos en él. No olvides cerrarla una vez que hayas terminado.
- Por fin leemos la respuesta del servidor.

Eliminar recurso

```
   public static void delete (String urlString, String contentType) throws IOException {
        HttpURLConnection connection = null;

        try {
            URL url = new URL(urlString);
            connection = (HttpURLConnection) url.openConnection();
            connection.setDoInput(true);
            connection.setRequestMethod("DELETE");
            connection.setRequestProperty("Content-Type", contentType);

            Map<String, List<String>> map = connection.getHeaderFields();
            StringBuilder sb = new StringBuilder();
            Iterator<Map.Entry<String, String>> iterator =
responseHeader.entrySet().iterator();
            while(iterator.hasNext())
            {
                Map.Entry<String, String> entry = iterator.next();
                sb.append(entry.getKey());
                sb.append('=').append('"');
                sb.append(entry.getValue());
                sb.append('"');
                if(iterator.hasNext())
                {
                    sb.append(',').append(' ');
                }
            }
            System.out.println(sb.toString());

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (connection != null) connection.disconnect();
        }
    }
```

Esto BORRARÁ el recurso en la URL especificada, luego imprimirá el encabezado de la respuesta.

Cómo funciona

- obtenemos la `HttpURLConnection` desde una `URL` .
- Establezca el tipo de contenido usando `setRequestProperty` , por defecto es `application/x-www-form-urlencoded`
- `setDoInput(true)` le dice a la conexión que tenemos la intención de usar la conexión URL para la entrada.
- `setRequestMethod("DELETE")` para realizar HTTP DELETE

Por fin imprimimos el encabezado de respuesta del servidor.

Compruebe si el recurso existe

```
/**
 * Checks if a resource exists by sending a HEAD-Request.
 * @param url The url of a resource which has to be checked.
 * @return true if the response code is 200 OK.
 */
public static final boolean checkIfResourceExists(URL url) throws IOException {
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("HEAD");
    int code = conn.getResponseCode();
    conn.disconnect();
    return code == 200;
}
```

Explicación:

Si solo está comprobando si existe un recurso, es mejor usar una solicitud HEAD que un GET. Esto evita la sobrecarga de transferir el recurso.

Tenga en cuenta que el método solo devuelve `true` si el código de respuesta es `200` . Si anticipa respuestas de redireccionamiento (es decir, 3XX), entonces es posible que el método deba mejorarse para cumplirlas.

Ejemplo:

```
checkIfResourceExists(new URL("http://images.google.com/")); // true
checkIfResourceExists(new URL("http://pictures.google.com/")); // false
```

Lea HttpURLConnection en línea: https://riptutorial.com/es/java/topic/156/httpurlconnection

Capítulo 95: Implementaciones del sistema de plugin Java

Observaciones

Si usa un IDE y / o un sistema de compilación, es mucho más fácil configurar este tipo de proyecto. Crea un módulo de aplicación principal, luego el módulo API, luego crea un módulo de complemento y lo hace dependiente del módulo API o de ambos. A continuación, configure dónde se colocarán los artefactos del proyecto; en nuestro caso, los archivos compilados de complementos se pueden enviar directamente al directorio de "complementos", evitando así realizar movimientos manuales.

Examples

Utilizando URLClassLoader

Hay varias formas de implementar un sistema de complementos para una aplicación Java. Una de las más sencillas es usar *URLClassLoader* . El siguiente ejemplo involucrará un poco de código JavaFX.

Supongamos que tenemos un módulo de una aplicación principal. Se supone que este módulo carga complementos en forma de archivos jar de la carpeta 'complementos'. Código inicial:

```
package main;

public class MainApplication extends Application
{
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins"); //arbitrary directory
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
        VBox loadedPlugins=new VBox(6); //a container to show the visual info later
        Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
        Scene scene=new
Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] a)
    {
            launch(a);
    }
}
```

Luego, creamos una interfaz que representará un complemento.

```
package main;
```

```
public interface Plugin
{
    default void initialize()
    {
        System.out.println("Initialized "+this.getClass().getName());
    }
    default String name(){return getClass().getSimpleName();}
}
```

Queremos cargar clases que implementen esta interfaz, así que primero necesitamos filtrar los archivos que tienen una extensión '.jar':

```
File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
```

Si hay algún archivo, necesitamos crear colecciones de URL y nombres de clase:

```
    if(files!=null && files.length>0)
    {
        ArrayList<String> classes=new ArrayList<>();
        ArrayList<URL> urls=new ArrayList<>(files.length);
        for(File file:files)
        {
            JarFile jar=new JarFile(file);
            jar.stream().forEach(jarEntry -> {
                if(jarEntry.getName().endsWith(".class"))
                {
                    classes.add(jarEntry.getName());
                }
            });
            URL url=file.toURI().toURL();
            urls.add(url);
        }

    }
```

Agreguemos un HashSet estático a *MainApplication* que contendrá los complementos cargados:

```
static HashSet<Plugin> plugins=new HashSet<>();
```

A continuación, creamos una instancia de *URLClassLoader* e *iteramos* sobre los nombres de las clases, creando instancias que implementan la interfaz del *complemento* :

```
URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new URL[urls.size()]));
classes.forEach(className->{
    try
    {
        Class
cls=urlClassLoader.loadClass(className.replaceAll("/",".").replace(".class",""));
//transforming to binary name
        Class[] interfaces=cls.getInterfaces();
        for(Class intface:interfaces)
        {
            if(intface.equals(Plugin.class)) //checking presence of Plugin interface
            {
```

```
                Plugin plugin=(Plugin) cls.newInstance(); //instantiating the Plugin
                plugins.add(plugin);
                break;
            }
        }
    }
    catch (Exception e){e.printStackTrace();}
});
```

Luego, podemos llamar métodos de plugin, por ejemplo, para inicializarlos:

```
if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded plugins:"));
    plugins.forEach(plugin -> {
        plugin.initialize();
        loadedPlugins.getChildren().add(new Label(plugin.name()));
});
```

El código final de *MainApplication* :

```
package main;
public class MainApplication extends Application
{
    static HashSet<Plugin> plugins=new HashSet<>();
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins");
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
        File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
        VBox loadedPlugins=new VBox(6);
        loadedPlugins.setAlignment(Pos.CENTER);
        if(files!=null && files.length>0)
        {
            ArrayList<String> classes=new ArrayList<>();
            ArrayList<URL> urls=new ArrayList<>(files.length);
            for(File file:files)
            {
                JarFile jar=new JarFile(file);
                jar.stream().forEach(jarEntry -> {
                    if(jarEntry.getName().endsWith(".class"))
                    {
                        classes.add(jarEntry.getName());
                    }
                });
                URL url=file.toURI().toURL();
                urls.add(url);
            }
            URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new
URL[urls.size()]));
            classes.forEach(className->{
                try
                {
                    Class
cls=urlClassLoader.loadClass(className.replaceAll("/",".").replace(".class",""));
                    Class[] interfaces=cls.getInterfaces();
                    for(Class intface:interfaces)
                    {
                        if(intface.equals(Plugin.class))
                        {
```

```
                            Plugin plugin=(Plugin) cls.newInstance();
                            plugins.add(plugin);
                            break;
                        }
                    }
                }
                catch (Exception e){e.printStackTrace();}
            });
            if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded
plugins:"));
            plugins.forEach(plugin -> {
                plugin.initialize();
                loadedPlugins.getChildren().add(new Label(plugin.name()));
            });
        }
        Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
        Scene scene=new
Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] a)
    {
            launch(a);
    }
}
```

Vamos a crear dos complementos. Obviamente, la fuente del complemento debe estar en un módulo separado.

```
package plugins;

import main.Plugin;

public class FirstPlugin implements Plugin
{
    //this plugin has default behaviour
}
```

Segundo complemento:

```
package plugins;

import main.Plugin;

public class AnotherPlugin implements Plugin
{
    @Override
    public void initialize() //overrided to show user's home directory
    {
        System.out.println("User home directory: "+System.getProperty("user.home"));
    }
}
```

Estos complementos deben estar empaquetados en tarros estándar: este proceso depende de su IDE u otras herramientas.

Cuando Jars se coloque en 'plugins' directamente, *MainApplication* los detectará y creará las clases apropiadas.

Lea Implementaciones del sistema de plugin Java en línea:
https://riptutorial.com/es/java/topic/7160/implementaciones-del-sistema-de-plugin-java

Capítulo 96: InputStreams y OutputStreams

Sintaxis

- int read (byte [] b) lanza IOException

Observaciones

Tenga en cuenta que la mayoría de las veces NO usa `InputStream` s directamente, sino que usa `BufferedStream` s, o similar. Esto se debe a que `InputStream` lee desde la fuente cada vez que se llama al método de lectura. Esto puede causar un uso significativo de la CPU en contextos que entran y salen del kernel.

Examples

Leyendo InputStream en una cadena

A veces es posible que desee leer la entrada de bytes en una cadena. Para hacer esto, tendrá que encontrar algo que convierta entre `byte` y los puntos de código UTF-16 "Java nativos" utilizados como caracteres `char` . Eso se hace con un `InputStreamReader` .

Para acelerar un poco el proceso, es "habitual" asignar un búfer, de modo que no tengamos demasiada sobrecarga al leer desde la entrada.

Java SE 7

```
public String inputStreamToString(InputStream inputStream) throws Exception {
     StringWriter writer = new StringWriter();

     char[] buffer = new char[1024];
     try (Reader reader = new BufferedReader(new InputStreamReader(inputStream, "UTF-8"))) {
           int n;
           while ((n = reader.read(buffer)) != -1) {
                // all this code does is redirect the output of `reader` to `writer` in
                // 1024 byte chunks
                writer.write(buffer, 0, n);
           }
     }
     return writer.toString();
}
```

Al transformar este ejemplo en Java SE 6 (y en una versión inferior), el código compatible se omite como un ejercicio para el lector.

Escritura de bytes en un OutputStream

Escribir bytes en un `OutputStream` un byte a la vez

```
OutputStream stream = object.getOutputStream();

byte b = 0x00;
stream.write( b );
```

Escribiendo una matriz de bytes

```
byte[] bytes = new byte[] { 0x00, 0x00 };

stream.write( bytes );
```

Escribir una sección de una matriz de bytes

```
int offset = 1;
int length = 2;
byte[] bytes = new byte[] { 0xFF, 0x00, 0x00, 0xFF };

stream.write( bytes, offset, length );
```

Cierre de arroyos

La mayoría de las transmisiones deben cerrarse cuando haya terminado con ellas, de lo contrario podría introducir una pérdida de memoria o dejar un archivo abierto. Es importante que las transmisiones estén cerradas incluso si se lanza una excepción.

Java SE 7

```
try(FileWriter fw = new FileWriter("outfilename");
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter out = new PrintWriter(bw))
{
    out.println("the text");
    //more code
    out.println("more text");
    //more code
} catch (IOException e) {
    //handle this however you
}
```

Recuerde: try-with-resources garantiza que los recursos se han cerrado cuando se sale del bloque, ya sea que ocurra con el flujo de control habitual o debido a una excepción.

Java SE 6

A veces, intentar con recursos no es una opción, o quizás esté soportando una versión anterior de Java 6 o anterior. En este caso, el manejo adecuado es utilizar un `finally` de bloque:

```
FileWriter fw = null;
BufferedWriter bw = null;
PrintWriter out = null;
try {
    fw = new FileWriter("myfile.txt");
    bw = new BufferedWriter(fw);
```

```
    out = new PrintWriter(bw);
    out.println("the text");
    out.close();
} catch (IOException e) {
    //handle this however you want
}
finally {
    try {
        if(out != null)
            out.close();
    } catch (IOException e) {
        //typically not much you can do here...
    }
}
```

Tenga en cuenta que cerrar una secuencia de envoltura también cerrará su secuencia subyacente. Esto significa que no puede envolver una secuencia, cerrar la envoltura y luego continuar usando la secuencia original.

Copiar el flujo de entrada al flujo de salida

Esta función copia datos entre dos flujos:

```
void copy(InputStream in, OutputStream out) throws IOException {
    byte[] buffer = new byte[8192];
    while ((bytesRead = in.read(buffer)) > 0) {
        out.write(buffer, 0, bytesRead);
    }
}
```

Ejemplo -

```
 // reading from System.in and writing to System.out
copy(System.in, System.out);
```

Envolviendo flujos de entrada / salida

`OutputStream` y `InputStream` tienen muchas clases diferentes, cada una de ellas con una funcionalidad única. Al envolver un flujo alrededor de otro, obtiene la funcionalidad de ambos flujos.

Puedes envolver un flujo cualquier cantidad de veces, solo toma nota del pedido.

Combinaciones utiles

Escribir caracteres en un archivo mientras se usa un búfer

```
File myFile = new File("targetFile.txt");
PrintWriter writer = new PrintWriter(new BufferedOutputStream(new FileOutputStream(myFile)));
```

Comprimir y cifrar datos antes de escribir en un archivo mientras se usa un búfer

```
Cipher cipher = ... // Initialize cipher
File myFile = new File("targetFile.enc");
BufferedOutputStream outputStream = new BufferedOutputStream(new DeflaterOutputStream(new
CipherOutputStream(new FileOutputStream(myFile), cipher)));
```

Lista de envoltorios de flujo de entrada / salida

Envoltura	Descripción
BufferedOutputStream / BufferedInputStream	Mientras que `OutputStream` escribe datos de un byte a la vez, `BufferedOutputStream` escribe datos en fragmentos. Esto reduce el número de llamadas al sistema, mejorando así el rendimiento.
DeflaterOutputStream / DeflaterInputStream	Realiza la compresión de datos.
InflaterOutputStream / InflaterInputStream	Realiza descompresión de datos.
CipherOutputStream / CipherInputStream	Cifra / descifra datos.
DigestOutputStream / DigestInputStream	Genera Message Digest para verificar la integridad de los datos.
CheckedOutputStream / CheckedInputStream	Genera un CheckSum. CheckSum es una versión más trivial de Message Digest.
DataOutputStream / DataInputStream	Permite la escritura de tipos de datos primitivos y cadenas. Significado para escribir bytes. Plataforma independiente.
PrintStream	Permite la escritura de tipos de datos primitivos y cadenas. Significado para escribir bytes. Dependiente de la plataforma.
OutputStreamWriter	Convierte un OutputStream en un escritor. Un OutputStream trata con bytes mientras que Writers trata con caracteres
PrintWriter	Llama automáticamente a OutputStreamWriter. Permite la escritura de tipos de datos primitivos y cadenas. Estrictamente para escribir personajes y mejor para escribir personajes.

Ejemplo de DataInputStream

```
package com.streams;
import java.io.*;
public class DataStreamDemo {
  public static void main(String[] args) throws IOException {
    InputStream input = new FileInputStream("D:\\datastreamdemo.txt");
    DataInputStream inst = new DataInputStream(input);
    int count = input.available();
    byte[] arr = new byte[count];
    inst.read(arr);
    for (byte byt : arr) {
      char ki = (char) byt;
      System.out.print(ki+"-");
    }
  }
}
```

Lea InputStreams y OutputStreams en línea: https://riptutorial.com/es/java/topic/110/inputstreams-y-outputstreams

Capítulo 97: Instalando Java (Edición Estándar)

Introducción

Esta página de documentación brinda acceso a las instrucciones para instalar `java standard edition` en `Windows` con `Windows` , `Linux` y `macOS` .

Examples

Configurando% PATH% y% JAVA_HOME% después de instalar en Windows

Suposiciones

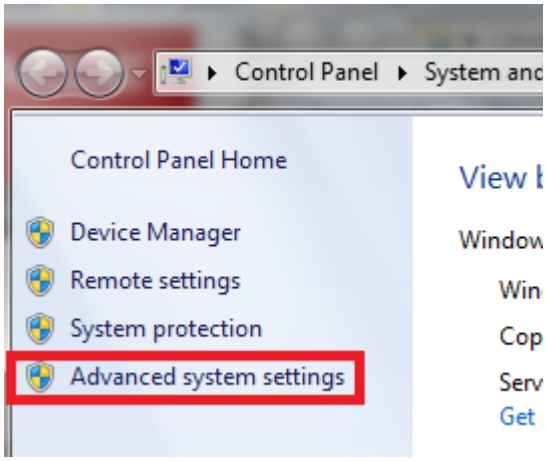
- Se ha instalado un JDK de Oracle.
- El JDK se instaló en el directorio predeterminado.

Pasos de configuración

1. Abra el Explorador de Windows.

2. En el panel de navegación de la izquierda, haga clic con el botón derecho en *Esta PC* (o *Computadora* para versiones anteriores de Windows). Hay una forma más corta sin usar el explorador en las versiones reales de Windows: simplemente presione `Win` + `Pausa`

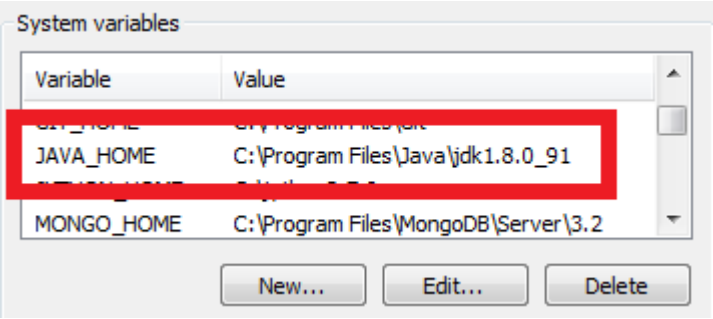
3. En la ventana recién abierta del Panel de control, haga clic izquierdo en *Configuración avanzada del sistema* que debería estar en la esquina superior izquierda. Esto abrirá la ventana *Propiedades* del *sistema* .



   Alternativamente, escriba `SystemPropertiesAdvanced` (no distingue mayúsculas y minúsculas) en *Ejecutar* (`Win` + `R`), y presione `Enter` .

4. En la pestaña *Avanzado* de *Propiedades* del *sistema,* seleccione el botón `Variables de entorno ...` en la esquina inferior derecha de la ventana.

5. Agregue una **Nueva Variable del Sistema** haciendo clic en el botón `Nuevo ...` en *Variables del Sistema* con el nombre `JAVA_HOME` y cuyo valor es la ruta al directorio donde se instaló el JDK. Después de ingresar estos valores, presione `OK` .



6. Desplácese hacia abajo en la lista de *Variables* del *sistema* y seleccione la variable `Path` .

7. **PRECAUCIÓN:** Windows se basa en la `Path` para encontrar programas importantes. Si se elimina parte o la totalidad, es posible que Windows no pueda funcionar correctamente. Debe modificarse para permitir que Windows ejecute el JDK. Con esto en mente, haga clic en el botón "Editar ..." con la variable `Path` seleccionada. Agregue `%JAVA_HOME%\bin;` al principio de la variable `Path` .

Es mejor agregarlo al principio de la línea porque el software de Oracle se utiliza para registrar su propia versión de Java en la `Path` . Esto hará que su versión sea ignorada si ocurre después de la declaración de Oracle.

Revisa tu trabajo

1. Abra el símbolo del sistema haciendo clic en Inicio, luego escriba `cmd` y presione `Enter` .
2. Introduzca `javac -version` en el indicador. Si tuvo éxito, la versión del JDK se imprimirá en la pantalla.

Nota: Si tiene que volver a intentarlo, cierre el indicador antes de verificar su trabajo. Esto obligará a las ventanas a obtener la nueva versión de `Path` .

Selección de una versión adecuada de Java SE

Ha habido muchas versiones de Java desde la versión original de Java 1.0 en 1995. (Consulte el historial de la versión de Java para obtener un resumen). Sin embargo, la mayoría de las versiones han pasado sus fechas oficiales de fin de vida. Esto significa que el proveedor (generalmente Oracle ahora) ha dejado de desarrollar un nuevo desarrollo para la versión y ya no proporciona parches públicos / gratuitos para cualquier error o problema de seguridad. (Las versiones de parches privados suelen estar disponibles para personas / organizaciones con un contrato de soporte; póngase en contacto con la oficina de ventas de su proveedor).

En general, la versión de Java SE recomendada para uso será la última actualización de la última versión pública. Actualmente, esto significa la última versión disponible de Java 8. El lanzamiento público de Java 9 está programado para 2017. (Java 7 ha pasado su fin de vida y el último lanzamiento público fue en abril de 2015. No se recomiendan los lanzamientos de Java 7 y anteriores).

Esta recomendación se aplica a todos los nuevos desarrollos de Java, y cualquiera que esté aprendiendo Java. También se aplica a las personas que solo desean ejecutar el software Java proporcionado por un tercero. En términos generales, el código Java bien escrito funcionará en una versión más nueva de Java. (Pero consulte las notas de la versión del software y póngase en contacto con el autor / proveedor / proveedor si tiene dudas).

Si está trabajando en una base de código Java más antigua, le recomendamos que se asegure de que su código se ejecute en la última versión de Java. Decidir cuándo comenzar a usar las características de las nuevas versiones de Java es más difícil, ya que esto afectará su capacidad para brindar asistencia a los clientes que no pueden o no quieren su instalación de Java.

Lanzamiento de Java y nombre de versión

El nombre de la versión de Java es un poco confuso. En realidad, hay dos sistemas de nombres y numeración, como se muestra en esta tabla:

Versión JDK	Nombre de marketing
jdk-1.0	JDK 1.0
jdk-1.1	JDK 1.1
jdk-1.2	J2SE 1.2
...	...
jdk-1.5	J2SE 1.5 renombrado Java SE 5
jdk-1.6	Java SE 6
jdk-1.7	Java SE 7
jdk-1.8	Java SE 8
jdk-9 [1]	Java SE 9 (no publicado aún)

1 - Parece que Oracle intenta romper su práctica anterior de usar un esquema de "número de versión semántica" en las cadenas de la versión de Java. Queda por ver si van a seguir con esto.

La "SE" en los nombres de marketing se refiere a la edición estándar. Esta es la versión base para ejecutar Java en la mayoría de las computadoras portátiles, PC y servidores (aparte de Android).

Hay otras dos ediciones oficiales de Java: "Java ME" es la Micro Edition, y "Java EE" es la Enterprise Edition. El Android de Java también es significativamente diferente de Java SE. Java ME, Java EE y Android Java están fuera del alcance de este tema.

El número de versión completo para una versión de Java se ve así:

```
1.8.0_101-b13
```

Esto dice JDK 1.8.0, Actualización 101, Compilación # 13. Oracle se refiere a esto en las notas de la versión como:

```
Java™ SE Development Kit 8, Update 101 (JDK 8u101)
```

El número de actualización es importante: Oracle emite regularmente actualizaciones a una versión principal con parches de seguridad, correcciones de errores y (en algunos casos) nuevas características. El número de compilación suele ser irrelevante. Tenga en cuenta que Java 8 y Java 1.8 se *refieren a lo mismo* ; Java 8 es solo el nombre de "marketing" para Java 1.8.

¿Qué necesito para el desarrollo de Java?

Una instalación de JDK y un editor de texto son lo mínimo para el desarrollo de Java. (Es bueno tener un editor de texto que pueda resaltar la sintaxis de Java, pero puede prescindir).

Sin embargo, para trabajos de desarrollo serio, se recomienda que también use lo siguiente:

- Un IDE de Java como Eclipse, Intellij IDEA o NetBeans
- Una herramienta de compilación de Java como Ant, Gradle o Maven
- Un sistema de control de versiones para administrar su base de código (con las copias de seguridad apropiadas y la replicación fuera del sitio)
- Herramientas de prueba y herramientas de CI (integración continua).

Instalación de un JDK de Java en Linux

Usando el Administrador de paquetes

Las versiones de JDK y / o JRE para OpenJDK u Oracle se pueden instalar utilizando el administrador de paquetes en la distribución de Linux más común. (Las opciones disponibles para usted dependerán de la distribución).

Como regla general, el procedimiento es abrir la ventana del terminal y ejecutar los comandos que se muestran a continuación. (Se supone que tiene acceso suficiente para ejecutar comandos como usuario "raíz" ... que es lo que hace el comando `sudo` . Si no lo hace, hable con los administradores de su sistema.)

Se recomienda usar el administrador de paquetes porque (generalmente) hace que sea más fácil mantener su instalación de Java actualizada.

`apt-get`

, distribuciones de Linux basadas en Debian (Ubuntu, etc.)

Las siguientes instrucciones instalarán Oracle Java 8:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Nota: para configurar automáticamente las variables de entorno de Java 8, puede instalar el siguiente paquete:

```
$ sudo apt-get install oracle-java8-set-default
```

Creando un archivo `.deb`

Si prefiere crear el archivo `.deb` desde el archivo `.tar.gz` descargado desde Oracle, haga lo siguiente (suponiendo que haya descargado el `./<jdk>.tar.gz` `.tar.gz` a `./<jdk>.tar.gz`):

```
$ sudo apt-get install java-package # might not be available in default repos
$ make-jpkg ./<jdk>.tar.gz          # should not be run as root
$ sudo dpkg -i *j2sdk*.deb
```

Nota : Esto espera que la entrada se proporcione como un archivo ".tar.gz".

`slackpkg` , distribuciones de Linux basadas en Slackware

```
sudo slapt-get install default-jdk
```

`yum` , RedHat, CentOS, etc.

```
sudo yum install java-1.8.0-openjdk-devel.x86_64
```

`dnf` , fedora

En los últimos lanzamientos de Fedora, `yum` ha sido reemplazado por `dnf` .

```
sudo dnf install java-1.8.0-openjdk-devel.x86_64
```

En las últimas versiones de Fedora, no hay paquetes para instalar Java 7 y versiones anteriores.

`pacman` , distribuciones de Linux basadas en arco.

```
sudo pacman -S jdk8-openjdk
```

No es necesario usar `sudo` si está ejecutando como usuario root.

Gentoo Linux

La guía de Gentoo Java es mantenida por el equipo de Gentoo Java y mantiene una página wiki

actualizada que incluye los paquetes de portage correctos y los indicadores USE necesarios.

Instalación de Oracle JDKs en Redhat, CentOS, Fedora

Instalación de JDK desde un archivo Oracle JDK o JRE `tar.gz`

1. Descargue el archivo de archivo de Oracle apropiado ("tar.gz") para la versión deseada del sitio de descargas de Oracle Java .

2. Cambie el directorio al lugar donde desea colocar la instalación;

3. Descomprima el archivo comprimido; p.ej

```
tar xzvf jdk-8u67-linux-x64.tar.gz
```

Instalación desde un archivo de Oracle Java RPM.

1. Recupere el archivo RPM requerido para el lanzamiento deseado del sitio de descargas de Oracle Java .

2. Instalar utilizando el comando `rpm` . Por ejemplo:

```
$ sudo rpm -ivh jdk-8u67-linux-x644.rpm
```

Instalación de un JDK o JRE de Java en Windows

Solo los JDK y JRE de Oracle están disponibles para las plataformas Windows. El procedimiento de instalación es sencillo:

1. Visite la página de descargas de Oracle Java:
2. Haga clic en el botón JDK, el botón JRE o el botón Servidor JRE. Tenga en cuenta que para desarrollar utilizando Java necesita JDK. Para conocer la diferencia entre JDK y JRE, ver aquí.
3. Desplácese hasta la versión que desea descargar. (En términos generales, se recomienda el más reciente.)
4. Seleccione el botón de opción "Aceptar acuerdo de licencia".
5. Descargue el instalador de Windows x86 (32 bits) o Windows x64 (64 bits).
6. Ejecute el instalador ... de la manera normal para su versión de Windows.

Una forma alternativa de instalar Java en Windows usando el símbolo del sistema es usar Chocolately:

1. Instala Chocolately desde https://chocolatey.org/

2. Abra una instancia de cmd, por ejemplo, presione `Win` + `R` y luego escriba "cmd" en la ventana "Ejecutar" seguido de una entrada.

3. En su instancia de cmd, ejecute el siguiente comando para descargar e instalar un JDK de Java 8:

```
C:\> choco install jdk8
```

Puesta en marcha y ejecución con versiones portátiles.

Hay casos en los que es posible que desee instalar JDK / JRE en un sistema con privilegios limitados, como una máquina virtual, o tal vez desee instalar y utilizar múltiples versiones o arquitecturas (x64 / x86) de JDK / JRE. Los pasos siguen siendo los mismos hasta el momento en que descarga el instalador (.EXE). Los pasos posteriores son los siguientes (los pasos son aplicables para JDK / JRE 7 y versiones posteriores, para versiones anteriores son ligeramente diferentes en los nombres de carpetas y archivos):

1. Mueva el archivo a una ubicación adecuada donde desee que sus archivos binarios de Java residan permanentemente.

2. Instale 7-Zip o su versión portátil si tiene privilegios limitados.

3. Con 7-Zip, extraiga los archivos del instalador de Java EXE a la ubicación.

4. Abra la línea de comandos allí manteniendo presionada la `Shift` y `Right-Click` en la carpeta en el explorador o navegue a esa ubicación desde cualquier lugar.

5. Navegue a la carpeta recién creada. Digamos que el nombre de la carpeta es `jdk-7u25-windows-x64` . Entonces escribe `cd jdk-7u25-windows-x64` . Luego escribe los siguientes comandos en orden:

   ```
   cd .rsrc\JAVA_CAB10

   extrac32 111
   ```

6. Esto creará un archivo `tools.zip` en esa ubicación. Extraiga el `tools.zip` con 7-Zip para que los archivos que contiene se creen ahora en las `tools` del mismo directorio.

7. Ahora ejecute estos comandos en el indicador de comandos ya abierto:

   ```
   cd tools

   for /r %x in (*.pack) do .\bin\unpack200 -r "%x" "%~dx%~px%~nx.jar"
   ```

8. Espere a que se complete el comando. Copie el contenido de las `tools` en la ubicación donde desea que estén sus binarios.

De esta manera, puede instalar cualquier versión de JDK / JRE que necesite instalar simultáneamente.

Publicación original: http://stackoverflow.com/a/6571736/1448252

Instalando un Java JDK en macOS

Oracle Java 7 y Java 8

Java 7 y Java 8 para macOS están disponibles en Oracle. Esta página de Oracle responde muchas preguntas sobre Java para Mac. Tenga en cuenta que Java 7 anterior a 7u25 ha sido deshabilitado por Apple por razones de seguridad.

En general, Oracle Java (versión 7 y posterior) requiere un Mac basado en Intel que ejecute macOS 10.7.3 o posterior.

Instalación de Oracle Java

Los instaladores Java 7 & 8 JDK y JRE para macOS se pueden descargar desde el sitio web de Oracle:

- Java 8 - Descargas de Java SE
- Java 7 - Oracle Java Archive.

Después de descargar el paquete correspondiente, haga doble clic en el paquete y siga el proceso normal de instalación. Un JDK debería instalarse aquí:

```
/Library/Java/JavaVirtualMachines/<version>.jdk/Contents/Home
```

donde corresponde a la versión instalada.

Cambio de línea de comando

Cuando se instala Java, la versión instalada se establece automáticamente como predeterminada. Para cambiar entre diferentes, usa:

```
export JAVA_HOME=/usr/libexec/java_home -v 1.6 #Or 1.7 or 1.8
```

Las siguientes funciones se pueden agregar al `~/.bash_profile` (si usa el shell Bash predeterminado) para facilitar su uso:

```
function java_version {
   echo 'java -version';
}

function java_set {
   if [[ $1 == "6" ]]
   then
       export JAVA_HOME='/usr/libexec/java_home -v 1.6';
       echo "Setting Java to version 6..."
       echo "$JAVA_HOME"
   elif [[ $1 == "7" ]]
   then
       export JAVA_HOME='/usr/libexec/java_home -v 1.7';
       echo "Setting Java to version 7..."
       echo "$JAVA_HOME"
   elif [[ $1 == "8" ]]
   then
       export JAVA_HOME='/usr/libexec/java_home -v 1.8';
       echo "Setting Java to version 8..."
```

```
        echo "$JAVA_HOME"
    fi
}
```

Apple Java 6 en macOS

En versiones anteriores de macOS (10.11 El Capitan y anteriores), el lanzamiento de Java 6 de Apple viene preinstalado. Si está instalado, se puede encontrar en esta ubicación:

```
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

Tenga en cuenta que Java 6 pasó su final de vida hace mucho tiempo, por lo que se recomienda actualizar a una versión más nueva. Hay más información sobre la reinstalación de Apple Java 6 en el sitio web de Oracle.

Configurando y cambiando versiones de Java en Linux usando alternativas

Usando Alternativas

Muchas distribuciones de Linux utilizan el comando de `alternatives` para cambiar entre diferentes versiones de un comando. Puede usar esto para cambiar entre diferentes versiones de Java instaladas en una máquina.

1. En un comando shell, establezca $ JDK a la ruta de un JDK recién instalado; p.ej

   ```
   $ JDK=/Data/jdk1.8.0_67
   ```

2. Use `alternatives --install` para agregar los comandos en el SDK de Java a las alternativas:

   ```
   $ sudo alternatives --install /usr/bin/java java $JDK/bin/java 2
   $ sudo alternatives --install /usr/bin/javac javac $JDK/bin/javac 2
   $ sudo alternatives --install /usr/bin/jar jar $JDK/bin/jar 2
   ```

   Y así.

Ahora puede cambiar entre diferentes versiones de un comando de Java de la siguiente manera:

```
$ sudo alternatives --config javac

There is 1 program that provides 'javac'.

  Selection    Command
-----------------------------------------------
*+ 1           /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.101-1.b14.fc23.x86_64/bin/javac
   2           /Data/jdk1.8.0_67/bin/javac

Enter to keep the current selection[+], or type selection number: 2
$
```

Para obtener más información sobre el uso de `alternatives` , consulte la entrada manual de alternativas (8) .

Instalaciones basadas en arco

Las instalaciones basadas en Arch Linux vienen con el comando `archlinux-java` para cambiar las versiones de java.

Listado de entornos instalados

```
$ archlinux-java status
Available Java environments:
  java-7-openjdk (default)
  java-8-openjdk/jre
```

Cambio de entorno actual

```
# archlinux-java set <JAVA_ENV_NAME>
```

P.ej:

```
# archlinux-java set java-8-openjdk/jre
```

Se puede encontrar más información en Arch Linux Wiki.

Comprobación y configuración posterior a la instalación en Linux

Después de instalar un SDK de Java, es recomendable verificar que esté listo para usar. Puede hacer esto ejecutando estos dos comandos, usando su cuenta de usuario normal:

```
$ java -version
$ javac -version
```

Estos comandos imprimen la información de la versión de JRE y JDK (respectivamente) que están en la ruta de búsqueda de comandos de su shell. Busque la cadena de versión JDK / JRE.

- Si cualquiera de los comandos anteriores falla, diciendo "comando no encontrado", entonces el JRE o el JDK no están en la ruta de búsqueda; Vaya a **Configuración de PATH directamente** debajo.
- Si alguno de los comandos anteriores muestra una cadena de versión diferente a la que estaba esperando, entonces su ruta de búsqueda o el sistema de "alternativas" deben ajustarse; ir a **Comprobando Alternativas**
- Si se muestran las cadenas de versión correctas, está casi terminado; saltar a la **comprobación de JAVA_HOME**

Configurando PATH directamente

Si no hay `java` o `javac` en la ruta de búsqueda en este momento, la solución simple es agregarla a la ruta de búsqueda.

Primero, encuentre dónde instaló Java; ver **¿Dónde se instaló Java?** Abajo si tienes dudas.

Luego, asumiendo que `bash` es su shell de comando, use un editor de texto para agregar las siguientes líneas al final de `~/.bash_profile` o `~/.bashrc` (si usa Bash como su shell).

```
JAVA_HOME=<installation directory>
PATH=$JAVA_HOME/bin:$PATH

export JAVA_HOME
export PATH
```

... reemplazando `<installation directory>` con la ruta para su directorio de instalación de Java. Tenga en cuenta que lo anterior supone que el directorio de instalación contiene un directorio `bin` y que el directorio `bin` contiene los comandos `java` y `javac` que está intentando usar.

A continuación, busque el archivo que acaba de editar para que se actualicen las variables de entorno para su shell actual.

```
$ source ~/.bash_profile
```

A continuación, repita las comprobaciones de las versiones `java` y `javac` . Si todavía hay problemas, utilice `which java` y `which javac` para verificar que ha actualizado correctamente las variables de entorno.

Finalmente, cierre la sesión y vuelva a iniciar sesión para que las variables de entorno actualizadas coincidan con todos sus shells. Ahora deberías haber terminado.

Comprobando Alternativas

Si `java -version` o `javac -version` funcionó pero dio un número de versión inesperado, debe verificar de dónde provienen los comandos. Usa `which` y `ls -l` para encontrar esto como sigue:

```
$ ls -l `which java`
```

Si la salida se ve así,

```
lrwxrwxrwx. 1 root root 22 Jul 30 22:18 /usr/bin/java -> /etc/alternatives/java
```

entonces se está utilizando la `alternatives` cambio de versión. Debe decidir si continuar usándolo o simplemente anularlo configurando la `PATH` directamente.

- Configurando y cambiando versiones de Java en Linux usando alternativas
- Vea "Configurando PATH directamente" arriba.

¿Dónde se instaló Java?

Java puede instalarse en una variedad de lugares, dependiendo del método de instalación.

- Los RPM de Oracle colocaron la instalación de Java en "/ usr / java".
- En Fedora, la ubicación predeterminada es "/ usr / lib / jvm".
- Si Java se instaló manualmente desde archivos ZIP o JAR, la instalación podría realizarse en cualquier lugar.

Si tiene dificultades para encontrar el directorio de instalación, le sugerimos que use `find` (o `slocate`) para encontrar el comando. Por ejemplo:

```
$ find / -name java -type f 2> /dev/null
```

Esto le proporciona las rutas para todos los archivos llamados `java` en su sistema. (La redirección del error estándar a "/ dev / null" suprime los mensajes sobre archivos y directorios a los que no puede acceder).

Instalando Oracle Java en Linux con el último archivo tar

Siga los pasos a continuación para instalar Oracle JDK desde el último archivo tar:

1. Descargue el último archivo tar desde aquí : la última versión actual es Java SE Development Kit 8u112.

2. Necesitas privilegios de sudo:

   ```
   sudo su
   ```

3. Crear un directorio para instalar jdk:

   ```
   mkdir /opt/jdk
   ```

4. Extraer tar descargado en él:

   ```
   tar -zxf jdk-8u5-linux-x64.tar.gz -C /opt/jdk
   ```

5. Verifique si los archivos son extraídos:

   ```
   ls /opt/jdk
   ```

6. Configuración de Oracle JDK como la JVM predeterminada:

   ```
   update-alternatives --install /usr/bin/java java /opt/jdk/jdk1.8.0_05/bin/java 100
   ```

   y

```
update-alternatives --install /usr/bin/javac javac /opt/jdk/jdk1.8.0_05/bin/javac 100
```

7. Compruebe la versión de Java:

   java -version

Rendimiento esperado:

```
java version "1.8.0_111"
Java(TM) SE Runtime Environment (build 1.8.0_111-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode)
```

Lea Instalando Java (Edición Estándar) en línea:
https://riptutorial.com/es/java/topic/4754/instalando-java--edicion-estandar-

Capítulo 98: Instrumentos de cuerda

Introducción

Las cadenas (`java.lang.String`) son fragmentos de texto almacenados en su programa. Las cadenas **no** son un tipo de datos primitivo en Java , sin embargo, son muy comunes en los programas Java.

En Java, las cadenas son inmutables, lo que significa que no se pueden cambiar. (Haga clic aquí para obtener una explicación más detallada de la inmutabilidad.)

Observaciones

Como las cadenas de Java son inmutables , todos los métodos que manipulen una `String` **devolverán un nuevo objeto de `String`** . No cambian la `String` original. Esto incluye los métodos de subcadena y reemplazo que los programadores de C y C ++ esperan que muten el objeto `String` destino.

Utilice un `StringBuilder` lugar de `String` si desea concatenar más de dos objetos `String` cuyos valores no se pueden determinar en tiempo de compilación. Esta técnica es más eficaz que crear nuevos objetos `String` y concatenarlos porque `StringBuilder` es mutable.

`StringBuffer` también se puede utilizar para concatenar objetos `String` . Sin embargo, esta clase tiene menos rendimiento porque está diseñada para ser segura para subprocesos y adquiere un mutex antes de cada operación. Dado que casi nunca necesita seguridad para subprocesos al concatenar cadenas, es mejor usar `StringBuilder` .

Si puede expresar una concatenación de cadenas como una sola expresión, entonces es mejor usar el operador `+` . El compilador de Java convertirá una expresión que contenga `+` concatenaciones en una secuencia eficiente de operaciones usando `String.concat(...)` o `StringBuilder` . El consejo para usar `StringBuilder` solo se aplica explícitamente cuando la concatenación involucra expresiones múltiples.

No almacene información confidencial en cadenas. Si alguien puede obtener un volcado de memoria de su aplicación en ejecución, entonces podrá encontrar todos los objetos `String` existentes y leer su contenido. Esto incluye los objetos de `String` que son inalcanzables y están en espera de recolección de basura. Si esto es un problema, deberá borrar los datos confidenciales de las cadenas tan pronto como termine. No puedes hacer esto con objetos `String` ya que son inmutables. Por lo tanto, es recomendable utilizar un objeto `char[]` para contener datos de caracteres confidenciales y borrarlos (por ejemplo, sobrescribirlos con caracteres `'\000'`) cuando haya terminado.

Todas las instancias de `String` se crean en el montón, incluso las instancias que corresponden a

literales de cadena. Lo especial de los literales de cadena es que la JVM garantiza que todos los literales que son iguales (es decir, que constan de los mismos caracteres) estén representados por un solo objeto de `String` (este comportamiento se especifica en JLS). Esto es implementado por los cargadores de clases JVM. Cuando un cargador de clases carga una clase, explora los literales de cadena que se utilizan en la definición de la clase, cada vez que ve una, comprueba si ya existe un registro en el conjunto de cadenas para este literal (utilizando el literal como una clave) . Si ya existe una entrada para el literal, se usa la referencia a una instancia de `String` almacenada como el par para ese literal. De lo contrario, se crea una nueva instancia de `String` y se almacena una referencia a la instancia para el literal (utilizado como clave) en el conjunto de cadenas. (Véase también internado de cadena).

El conjunto de cadenas se mantiene en el montón de Java y está sujeto a la recolección de elementos no utilizados normal.

Java SE 7

En las versiones de Java anteriores a Java 7, el grupo de cadenas se mantuvo en una parte especial del montón conocido como "PermGen". Esta parte solo fue recogida ocasionalmente.

Java SE 7

En Java 7, el grupo de cadenas se movió fuera de "PermGen".

Tenga en cuenta que los literales de cadena son implícitamente accesibles desde cualquier método que los use. Esto significa que los objetos `String` correspondientes solo se pueden recolectar si el código en sí es recolectado.

Hasta que Java 8, los objetos `String` se implementan como una matriz de caracteres UTF-16 (2 bytes por carácter). Hay una propuesta en Java 9 para implementar la `String` como una matriz de bytes con un campo de marca de codificación para observar si la cadena está codificada como bytes (LATIN-1) o caracteres (UTF-16).

Examples

Comparando cuerdas

Con el fin de comparar las cadenas para la igualdad, debe utilizar los métodos `equals` o `equals` `equalsIgnoreCase` .

Por ejemplo, el siguiente fragmento de código determinará si las dos instancias de `String` son iguales en todos los caracteres:

```
String firstString = "Test123";
String secondString = "Test" + 123;

if (firstString.equals(secondString)) {
   // Both Strings have the same content.
}
```

Demo en vivo

Este ejemplo los comparará, independientemente de su caso:

```
String firstString = "Test123";
String secondString = "TEST123";

if (firstString.equalsIgnoreCase(secondString)) {
    // Both Strings are equal, ignoring the case of the individual characters.
}
```

Demo en vivo

Tenga en cuenta que `equalsIgnoreCase` no le permite especificar una `equalsIgnoreCase` `Locale` . Por ejemplo, si comparas las dos palabras `"Taki"` y `"TAKI"` en inglés, son iguales; sin embargo, en turco son diferentes (en turco, la minúscula `I` es `ı`). Para casos como este, convertir ambas cadenas en minúsculas (o mayúsculas) con `Locale` y luego compararlas con `equals` es la solución.

```
String firstString = "Taki";
String secondString = "TAKI";

System.out.println(firstString.equalsIgnoreCase(secondString)); //prints true

Locale locale = Locale.forLanguageTag("tr-TR");

System.out.println(firstString.toLowerCase(locale).equals(
                   secondString.toLowerCase(locale))); //prints false
```

Demo en vivo

No use el operador == para comparar cadenas

A menos que pueda garantizar que todas las cadenas hayan sido internadas (ver más abajo), no **debe** usar los operadores `==` o `!=` Para comparar cadenas. Estos operadores realmente prueban las referencias, y dado que varios objetos `String` pueden representar el mismo String, esto puede dar una respuesta incorrecta.

En su lugar, use el `String.equals(Object)` , que comparará los objetos String en función de sus valores. Para obtener una explicación detallada, consulte Pitfall: usar == para comparar cadenas .

Comparando cadenas en una instrucción switch

Java SE 7

A partir de Java 1.7, es posible comparar una variable de cadena con literales en una instrucción de `switch` . Asegúrese de que la Cadena no sea nula, de lo contrario siempre lanzará una `NullPointerException` . Los valores se comparan utilizando `String.equals` , es decir, `String.equals` mayúsculas y minúsculas.

```
String stringToSwitch = "A";

switch (stringToSwitch) {
    case "a":
        System.out.println("a");
        break;
    case "A":
        System.out.println("A"); //the code goes here
        break;
    case "B":
        System.out.println("B");
        break;
    default:
        break;
}
```

Demo en vivo

Comparando cadenas con valores constantes

Al comparar una `String` con un valor constante, puede poner el valor constante en el lado izquierdo de `equals` para asegurarse de que no obtendrá una excepción `NullPointerException` si la otra cadena es `null` .

```
"baz".equals(foo)
```

Mientras que `foo.equals("baz")` lanzará una `NullPointerException` si `foo` es `null` , `"baz".equals(foo)` evaluará como `false` .

Java SE 7

Una alternativa más legible es usar `Objects.equals()` , que realiza una comprobación nula de ambos parámetros: `Objects.equals(foo, "baz")` .

(**Nota:** es discutible si es mejor evitar las `NullPointerExceptions` en general, o dejar que ocurran y luego corregir la causa raíz; ver aquí y aquí . Ciertamente, calificar de "mejor práctica" a la estrategia de evitación no es justificable).

Ordenaciones de cuerdas

La clase `String` implementa `Comparable<String>` con el método `String.compareTo` (como se describe

al comienzo de este ejemplo). Esto hace que el orden natural de los objetos `String` sea sensible a mayúsculas y minúsculas. La clase `String` proporciona una constante `Comparator<String>` denominada `CASE_INSENSITIVE_ORDER` adecuada para una clasificación que no distingue entre mayúsculas y minúsculas.

Comparando con cuerdas internadas

La especificación del lenguaje Java (JLS 3.10.6) establece lo siguiente:

> "Además, un literal de cadena siempre se refiere a la misma instancia de la clase `String` . Esto se debe a que los literales de cadena, o, más generalmente, las cadenas que son valores de expresiones constantes, se *internan* para compartir instancias únicas, utilizando el método `String.intern` ".

Esto significa que es seguro comparar referencias a dos *literales de* cadena utilizando `==` . Además, lo mismo es cierto para las referencias a objetos `String` que se han producido utilizando el método `String.intern()` .

Por ejemplo:

```
String strObj = new String("Hello!");
String str = "Hello!";

// The two string references point two strings that are equal
if (strObj.equals(str)) {
    System.out.println("The strings are equal");
}

// The two string references do not point to the same object
if (strObj != str) {
    System.out.println("The strings are not the same object");
}

// If we intern a string that is equal to a given literal, the result is
// a string that has the same reference as the literal.
String internedStr = strObj.intern();

if (internedStr == str) {
    System.out.println("The interned string and the literal are the same object");
}
```

Detrás de escena, el mecanismo de internado mantiene una tabla hash que contiene todas las cadenas internadas que aún son *accesibles* . Cuando llama a `intern()` en una `String` , el método busca el objeto en la tabla hash:

- Si se encuentra la cadena, ese valor se devuelve como la cadena internada.
- De lo contrario, se agrega una copia de la cadena a la tabla hash y esa cadena se devuelve como la cadena internada.

Es posible utilizar interning para permitir la comparación de cadenas utilizando `==` . Sin embargo, hay problemas significativos al hacer esto; Vea Pitfall - Interning Strings para que pueda usar ==

es una mala idea para obtener detalles. No se recomienda en la mayoría de los casos.

Cambiando el caso de los personajes dentro de una cadena

El tipo de `String` proporciona dos métodos para convertir cadenas entre mayúsculas y minúsculas:

- `toUpperCase` para convertir todos los caracteres a mayúsculas
- `toLowerCase` para convertir todos los caracteres a minúsculas

Estos dos métodos devuelven las cadenas convertidas como nuevas instancias de `String` : los objetos de `String` originales no se modifican porque la `String` es inmutable en Java. Vea esto para más información sobre la inmutabilidad: Inmutabilidad de las cadenas en Java

```
String string = "This is a Random String";
String upper = string.toUpperCase();
String lower = string.toLowerCase();

System.out.println(string);   // prints "This is a Random String"
System.out.println(lower);    // prints "this is a random string"
System.out.println(upper);    // prints "THIS IS A RANDOM STRING"
```

Los caracteres no alfabéticos, como los dígitos y los signos de puntuación, no se ven afectados por estos métodos. Tenga en cuenta que estos métodos también pueden tratar incorrectamente con ciertos caracteres Unicode bajo ciertas condiciones.

Nota : estos métodos son *sensibles a la configuración regional* y pueden producir resultados inesperados si se utilizan en cadenas que deben interpretarse independientemente de la configuración regional. Algunos ejemplos son los identificadores de lenguaje de programación, las claves de protocolo y las etiquetas `HTML` .

Por ejemplo, `"TITLE".toLowerCase()` en una configuración regional turca devuelve " `title` ", donde `ı` `(\u0131)` es el carácter LATIN SMALL LETTER DOTLESS I. Para obtener resultados correctos para las cadenas no sensibles al entorno local, pase `Locale.ROOT` como parámetro al método de conversión de casos correspondiente (por ejemplo, `toLowerCase(Locale.ROOT)` o `toUpperCase(Locale.ROOT)`).

Aunque el uso de `Locale.ENGLISH` también es correcto para la mayoría de los casos, el **idioma invariable** es `Locale.ROOT` .

Puede encontrar una lista detallada de los caracteres Unicode que requieren una carcasa especial en el sitio web de Unicode Consortium .

Cambio de caso de un carácter específico dentro de una cadena ASCII:

Para cambiar el caso de un carácter específico de una cadena ASCII, se puede utilizar el siguiente algoritmo:

Pasos:

1. Declara una cuerda.
2. Introduzca la cadena.
3. Convertir la cadena en una matriz de caracteres.
4. Introduzca el carácter que se va a buscar.
5. Busque el carácter en la matriz de caracteres.
6. Si lo encuentra, verifique si el carácter está en minúscula o en mayúscula.
    - Si está en mayúsculas, agregue 32 al código ASCII del carácter.
    - Si es minúscula, resta 32 del código ASCII del carácter.
7. Cambiar el carácter original de la matriz de caracteres.
8. Convertir la matriz de caracteres de nuevo en la cadena.

Voila, se cambia el caso del personaje.

Un ejemplo del código para el algoritmo es:

```
Scanner scanner = new Scanner(System.in);
System.out.println("Enter the String");
String s = scanner.next();
char[] a = s.toCharArray();
System.out.println("Enter the character you are looking for");
System.out.println(s);
String c = scanner.next();
char d = c.charAt(0);

for (int i = 0; i <= s.length(); i++) {
    if (a[i] == d) {
        if (d >= 'a' && d <= 'z') {
            d -= 32;
        } else if (d >= 'A' && d <= 'Z') {
            d += 32;
        }
        a[i] = d;
        break;
    }
}
s = String.valueOf(a);
System.out.println(s);
```

Encontrar una cadena dentro de otra cadena

Para verificar si una cadena en particular `a` está contenida en una cadena `b` o no, podemos usar el método `String.contains()` con la siguiente sintaxis:

```
b.contains(a); // Return true if a is contained in b, false otherwise
```

El método `String.contains()` se puede usar para verificar si se puede encontrar un `CharSequence` en el String. El método busca la Cadena `a` en la Cadena `b` de manera sensible a las mayúsculas y minúsculas.

```
String str1 = "Hello World";
String str2 = "Hello";
String str3 = "helLO";
```

```
System.out.println(str1.contains(str2)); //prints true
System.out.println(str1.contains(str3)); //prints false
```

Demo en vivo en Ideone

Para encontrar la posición exacta donde comienza una cadena dentro de otra cadena, use `String.indexOf()` :

```
String s = "this is a long sentence";
int i = s.indexOf('i');    // the first 'i' in String is at index 2
int j = s.indexOf("long"); // the index of the first occurrence of "long" in s is 10
int k = s.indexOf('z');    // k is -1 because 'z' was not found in String s
int h = s.indexOf("LoNg"); // h is -1 because "LoNg" was not found in String s
```

Demo en vivo en Ideone

El método `String.indexOf()` devuelve el primer índice de un `char` o `String` en otra `String` . El método devuelve `-1` si no se encuentra.

Nota : el método `String.indexOf()` mayúsculas y minúsculas.

Ejemplo de búsqueda ignorando el caso:

```
String str1 = "Hello World";
String str2 = "wOr";
str1.indexOf(str2);                                // -1
str1.toLowerCase().contains(str2.toLowerCase());  // true
str1.toLowerCase().indexOf(str2.toLowerCase());   // 6
```

Demo en vivo en Ideone

Obtener la longitud de una cadena

Para obtener la longitud de un objeto `String` , llame al método `length()` . La longitud es igual al número de unidades de código UTF-16 (caracteres) en la cadena.

```
String str = "Hello, World!";
System.out.println(str.length()); // Prints out 13
```

Demo en vivo en Ideone

Un `char` en una cadena es el valor UTF-16. Los puntos de código Unicode cuyos valores son ≥ 0x1000 (por ejemplo, la mayoría de los emojis) usan dos posiciones char. Para contar el número de puntos de código Unicode en una cadena, independientemente de si cada punto de código se ajusta a un valor de `char` UTF-16, puede usar el método `codePointCount` :

```
int length = str.codePointCount(0, str.length());
```

También puede usar un flujo de puntos de código, a partir de Java 8:

```
int length = str.codePoints().count();
```

Subcadenas

```
String s = "this is an example";
String a = s.substring(11); // a will hold the string starting at character 11 until the end
("example")
String b = s.substring(5, 10); // b will hold the string starting at character 5 and ending
right before character 10 ("is an")
String b = s.substring(5, b.length()-3); // b will hold the string starting at character 5
ending right before b' s lenght is out of 3  ("is an exam")
```

Las subcadenas también se pueden aplicar para dividir y agregar / reemplazar caracteres en su Cadena original. Por ejemplo, se enfrentó a una fecha china que contenía caracteres chinos, pero desea almacenarla como una cadena de fecha de formato correcto.

```
String datestring = "2015年11月17日"
datestring = datestring.substring(0, 4) + "-" + datestring.substring(5,7) + "-" +
datestring.substring(8,10);
//Result will be 2015-11-17
```

El método de subcadena extrae una pieza de un `String` . Cuando se proporciona un parámetro, el parámetro es el inicio y la pieza se extiende hasta el final de la `String` . Cuando se dan dos parámetros, el primer parámetro es el carácter inicial y el segundo parámetro es el índice del carácter justo después del final (el carácter en el índice no está incluido). Una forma fácil de verificar es la resta del primer parámetro del segundo debe producir la longitud esperada de la cadena.

Java SE 7

En las versiones JDK <7u6, el método de `substring` crea una instancia de una `String` que comparte el mismo `char[]` respaldo `char[]` que la `String` original y tiene los campos internos de `offset` y `count` establecidos en el inicio y la longitud del resultado. Dicha compartición puede provocar pérdidas de memoria, que pueden evitarse llamando a la `new String(s.substring(...))` para forzar la creación de una copia, después de lo cual el `char[]` puede ser recolectado como basura.

Java SE 7

Desde JDK 7u6, el método de la `substring` siempre copia la matriz `char[]` subyacente completa, haciendo que la complejidad sea lineal en comparación con la constante anterior pero garantizando la ausencia de pérdidas de memoria al mismo tiempo.

Obtención del personaje nth en una cadena

```
String str = "My String";
```

```
System.out.println(str.charAt(0)); // "M"
System.out.println(str.charAt(1)); // "y"
System.out.println(str.charAt(2)); // " "
System.out.println(str.charAt(str.length-1)); // Last character "g"
```

Para obtener el enésimo carácter de una cadena, simplemente llame a `charAt(n)` en una `String` , donde `n` es el índice del carácter que desea recuperar

NOTA: el índice `n` comienza en `0` , por lo que el primer elemento está en n = 0.

Plataforma independiente de nueva línea separadora.

Dado que el nuevo separador de línea varía de una plataforma a otra (por ejemplo, `\n` en sistemas similares a Unix o `\r\n` en Windows) a menudo es necesario tener una forma de acceso independiente de la plataforma. En Java se puede recuperar de una propiedad del sistema:

```
System.getProperty("line.separator")
```

Java SE 7

Debido a que el nuevo separador de línea es tan comúnmente necesario, desde Java 7 en un método de acceso directo que devuelve exactamente el mismo resultado que el código anterior, está disponible:

```
System.lineSeparator()
```

Nota : ya que es muy poco probable que el nuevo separador de línea cambie durante la ejecución del programa, es una buena idea almacenarlo en una variable final estática en lugar de recuperarlo de la propiedad del sistema cada vez que sea necesario.

Cuando use `String.format` , use `%n` lugar de `\n` o '\ r \ n' para generar un nuevo separador de línea independiente de la plataforma.

```
System.out.println(String.format('line 1: %s.%nline 2: %s%n', lines[0],lines[1]));
```

Añadiendo el método toString () para objetos personalizados

Supongamos que ha definido la siguiente clase de `Person` :

```
public class Person {

    String name;
    int age;

    public Person (int age, String name) {
        this.age = age;
        this.name = name;
    }
}
```

Si creas un nuevo objeto `Person` :

```
Person person = new Person(25, "John");
```

y más adelante en su código, use la siguiente declaración para imprimir el objeto:

```
System.out.println(person.toString());
```

Demo en vivo en Ideone

Obtendrá una salida similar a la siguiente:

```
Person@7ab89d
```

Este es el resultado de la implementación del método `toString()` definido en la clase `Object` , una superclase de `Person` . La documentación de `Object.toString()` establece:

> El método toString para la clase Object devuelve una cadena que consiste en el nombre de la clase de la que el objeto es una instancia, el carácter de signo at '@' y la representación hexadecimal sin signo del código hash del objeto. En otras palabras, este método devuelve una cadena igual al valor de:
>
> ```
> getClass().getName() + '@' + Integer.toHexString(hashCode())
> ```

Por lo tanto, para obtener resultados significativos, deberá **anular** el método `toString()` :

```
@Override
public String toString() {
    return "My name is " + this.name + " and my age is " + this.age;
}
```

Ahora la salida será:

```
My name is John and my age is 25
```

También puedes escribir

```
System.out.println(person);
```

Demo en vivo en Ideone

De hecho, `println()` invoca implícitamente el método `toString` en el objeto.

Dividir cuerdas

Puede dividir una `String` en un carácter delimitador particular o una expresión regular , puede usar el método `String.split()` que tiene la siguiente firma:

```
public String[] split(String regex)
```

Tenga en cuenta que el carácter delimitado o la expresión regular se eliminan de la matriz de cadenas resultante.

Ejemplo usando caracteres delimitadores:

```
String lineFromCsvFile = "Mickey;Bolton;12345;121216";
String[] dataCells = lineFromCsvFile.split(";");
// Result is dataCells = { "Mickey", "Bolton", "12345", "121216"};
```

Ejemplo usando expresión regular:

```
String lineFromInput = "What    do you need    from me?";
String[] words = lineFromInput.split("\\s+"); // one or more space chars
// Result is words = {"What", "do", "you", "need", "from", "me?"};
```

Incluso puedes dividir directamente un literal de `String` :

```
String[] firstNames = "Mickey, Frank, Alicia, Tom".split(", ");
// Result is firstNames = {"Mickey", "Frank", "Alicia", "Tom"};
```

Advertencia : No olvide que el parámetro siempre se trata como una expresión regular.

```
"aaa.bbb".split("."); // This returns an empty array
```

En el ejemplo anterior `.` se trata como el comodín de expresión regular que coincide con cualquier carácter, y como cada carácter es un delimitador, el resultado es una matriz vacía.

División basada en un delimitador que es un meta-carácter de expresiones regulares

Los siguientes caracteres se consideran especiales (también conocidos como meta-caracteres) en expresiones regulares

```
  < > - = ! ( ) [ ] { } \ ^ $ | ? * + .
```

Para dividir una cadena basada en uno de los delimitadores anteriores, necesita *escapar de* ellos usando `\\` o usar `Pattern.quote()` :

- Utilizando `Pattern.quote()` :

  ```
  String s = "a|b|c";
  String regex = Pattern.quote("|");
  String[] arr = s.split(regex);
  ```

- Escapar de los caracteres especiales:

  ```
  String s = "a|b|c";
  ```

```
String[] arr = s.split("\\|");
```

La división elimina los valores vacíos

`split(delimiter)` de forma predeterminada elimina las cadenas vacías finales de la matriz de resultados. Para desactivar este mecanismo, necesitamos usar una versión sobrecargada de `split(delimiter, limit)` con límite establecido en valor negativo como

```
String[] split = data.split("\\|", -1);
```

`split(regex)` devuelve internamente el resultado de `split(regex, 0)` .

El parámetro límite controla el número de veces que se aplica el patrón y, por lo tanto, afecta la longitud de la matriz resultante.
Si el límite *n* es mayor que cero, entonces el patrón se aplicará a lo sumo *n - 1* veces, la longitud de la matriz no será mayor que *n* , y la última entrada de la matriz contendrá toda la entrada más allá del último delimitador coincidente.
Si *n* es negativo, entonces el patrón se aplicará tantas veces como sea posible y la matriz puede tener cualquier longitud.
Si *n* es cero, el patrón se aplicará tantas veces como sea posible, la matriz puede tener cualquier longitud y las cadenas vacías finales se descartarán.

Dividir con un `StringTokenizer`

Además del método `split()` , las cadenas también se pueden dividir utilizando un `StringTokenizer` .

`StringTokenizer` es incluso más restrictivo que `String.split()` , y también un poco más difícil de usar. Básicamente, está diseñado para extraer tokens delimitados por un conjunto fijo de caracteres (dados como una `String`). Cada personaje actuará como separador. Debido a esta restricción, es aproximadamente el doble de rápido que `String.split()` .

El conjunto predeterminado de caracteres son espacios vacíos (`\t\n\r\f`). El siguiente ejemplo imprimirá cada palabra por separado.

```
String str = "the lazy fox jumped over the brown fence";
StringTokenizer tokenizer = new StringTokenizer(str);
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

Esto imprimirá:

```
the
lazy
fox
jumped
over
```

```
the
brown
fence
```

Puedes usar diferentes juegos de caracteres para la separación.

```
String str = "jumped over";
// In this case character `u` and `e` will be used as delimiters
StringTokenizer tokenizer = new StringTokenizer(str, "ue");
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

Esto imprimirá:

```
j
mp
d ov
r
```

Uniendo cuerdas con un delimitador

Java SE 8

Una matriz de cadenas se puede unir usando el método estático `String.join()` :

```
String[] elements = { "foo", "bar", "foobar" };
String singleString = String.join(" + ", elements);

System.out.println(singleString); // Prints "foo + bar + foobar"
```

De manera similar, hay un `String.join()` sobrecargado de `String.join()` para `Iterable` s.

Para tener un control preciso sobre las uniones , puede usar la clase StringJoiner :

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
    // The last two arguments are optional,
    // they define prefix and suffix for the result string

sj.add("foo");
sj.add("bar");
sj.add("foobar");

System.out.println(sj); // Prints "[foo, bar, foobar]"
```

Para unir un flujo de cadenas, puede utilizar el recopilador de unión :

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", "));
System.out.println(joined); // Prints "foo, bar, foobar"
```

Hay una opción para definir prefijo y sufijo aquí también:

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", ", "{", "}"));
System.out.println(joined); // Prints "{foo, bar, foobar}"
```

Cuerdas de inversión

Hay un par de maneras en que puedes revertir una cadena para hacerla al revés.

1. StringBuilder / StringBuffer:

```
 String code = "code";
 System.out.println(code);

 StringBuilder sb = new StringBuilder(code);
 code = sb.reverse().toString();

 System.out.println(code);
```

2. Matriz de caracteres:

```
String code = "code";
System.out.println(code);

char[] array = code.toCharArray();
for (int index = 0, mirroredIndex = array.length - 1; index < mirroredIndex; index++,
mirroredIndex--) {
    char temp = array[index];
    array[index] = array[mirroredIndex];
    array[mirroredIndex] = temp;
}

// print reversed
System.out.println(new String(array));
```

Contar las apariciones de una subcadena o carácter en una cadena

`countMatches` método `countMatches` de org.apache.commons.lang3.StringUtils se usa normalmente para contar las ocurrencias de una subcadena o un carácter en una `String` :

```
import org.apache.commons.lang3.StringUtils;

String text = "One fish, two fish, red fish, blue fish";

// count occurrences of a substring
String stringTarget = "fish";
int stringOccurrences = StringUtils.countMatches(text, stringTarget); // 4

// count occurrences of a char
char charTarget = ',';
int charOccurrences = StringUtils.countMatches(text, charTarget); // 3
```

De lo contrario, para lo mismo con las API de Java estándar, puede usar expresiones regulares:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

String text = "One fish, two fish, red fish, blue fish";
System.out.println(countStringInString("fish", text)); // prints 4
System.out.println(countStringInString(",", text)); // prints 3


public static int countStringInString(String search, String text) {
    Pattern pattern = Pattern.compile(search);
    Matcher matcher = pattern.matcher(text);

    int stringOccurrences = 0;
    while (matcher.find()) {
      stringOccurrences++;
    }
    return stringOccurrences;
}
```

Concatenación de cadenas y StringBuilders

La concatenación de cadenas se puede realizar con el operador `+` . Por ejemplo:

```
String s1 = "a";
String s2 = "b";
String s3 = "c";
String s = s1 + s2 + s3; // abc
```

Normalmente, una implementación del compilador realizará la concatenación anterior utilizando métodos que involucren un `StringBuilder` debajo del capó. Cuando se compila, el código sería similar al siguiente:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append("c").toString();
```

`StringBuilder` tiene varios métodos sobrecargados para agregar diferentes tipos, por ejemplo, para agregar un `int` lugar de una `String` . Por ejemplo, una implementación puede convertir:

```
String s1 = "a";
String s2 = "b";
String s = s1 + s2 + 2; // ab2
```

a lo siguiente:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append(2).toString();
```

Los ejemplos anteriores ilustran una operación de concatenación simple que se realiza efectivamente en un solo lugar en el código. La concatenación involucra una sola instancia del `StringBuilder` . En algunos casos, una concatenación se lleva a cabo de forma acumulativa, como en un bucle:

```
String result = "";
for(int i = 0; i < array.length; i++) {
    result += extractElement(array[i]);
}
return result;
```

En tales casos, la optimización del compilador generalmente no se aplica, y cada iteración creará un nuevo objeto `StringBuilder` . Esto se puede optimizar al transformar explícitamente el código para usar un único `StringBuilder` :

```
StringBuilder result = new StringBuilder();
for(int i = 0; i < array.length; i++) {
    result.append(extractElement(array[i]));
}
return result.toString();
```

Un `StringBuilder` se inicializará con un espacio vacío de solo 16 caracteres. Si sabe de antemano que construirá cadenas más grandes, puede ser beneficioso inicializarlo con suficiente tamaño por adelantado, de modo que no sea necesario cambiar el tamaño del búfer interno:

```
StringBuilder buf = new StringBuilder(30); // Default is 16 characters
buf.append("0123456789");
buf.append("0123456789"); // Would cause a reallocation of the internal buffer otherwise
String result = buf.toString(); // Produces a 20-chars copy of the string
```

Si está produciendo muchas cadenas, es recomendable reutilizar `StringBuilder` s:

```
StringBuilder buf = new StringBuilder(100);
for (int i = 0; i < 100; i++) {
    buf.setLength(0); // Empty buffer
    buf.append("This is line ").append(i).append('\n');
    outputfile.write(buf.toString());
}
```

Si (y solo si) múltiples hilos están escribiendo en el *mismo* búfer, use StringBuffer , que es una versión `synchronized` de `StringBuilder` . Pero debido a que normalmente solo un único hilo escribe en un búfer, usualmente es más rápido usar `StringBuilder` sin sincronización.

Utilizando el método concat ():

```
String string1 = "Hello ";
String string2 = "world";
String string3 = string1.concat(string2);  // "Hello world"
```

Esto devuelve una nueva cadena que es string1 con string2 agregada al final. También puedes usar el método concat () con cadenas literales, como en:

```
"My name is ".concat("Buyya");
```

Reemplazo de partes de cuerdas

Dos formas de reemplazar: por expresiones regulares o por coincidencia exacta.

Nota: el objeto String original no se modificará, el valor de retorno conserva la String modificada.

Coincidencia exacta

Reemplazar el carácter único con otro carácter único:

```
String replace(char oldChar, char newChar)
```

> Devuelve una nueva cadena resultante de reemplazar todas las apariciones de oldChar en esta cadena con newChar.

```
String s = "popcorn";
System.out.println(s.replace('p','W'));
```

Resultado:

```
WoWcorn
```

Reemplace la secuencia de caracteres con otra secuencia de caracteres:

```
String replace(CharSequence target, CharSequence replacement)
```

> Reemplaza cada subcadena de esta cadena que coincida con la secuencia de destino literal con la secuencia de reemplazo literal especificada.

```
String s = "metal petal et al.";
System.out.println(s.replace("etal","etallica"));
```

Resultado:

```
metallica petallica et al.
```

Regex

Nota : la agrupación utiliza el carácter `$` para hacer referencia a los grupos, como `$1` .

Reemplazar todos los partidos:

```
String replaceAll(String regex, String replacement)
```

> Reemplaza cada subcadena de esta cadena que coincide con la expresión regular dada con el reemplazo dado.

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Resultado:

```
spiral metallica petallica et al.
```

Reemplazar solo el primer partido:

```
String replaceFirst(String regex, String replacement)
```

> Reemplaza la primera subcadena de esta cadena que coincide con la expresión regular dada con el reemplazo dado

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Resultado:

```
spiral metallica petal et al.
```

Eliminar espacios en blanco desde el principio y el final de una cadena

El método `trim()` devuelve una nueva Cadena con los espacios en blanco iniciales y finales eliminados.

```
String s = new String("   Hello World!!  ");
String t = s.trim();  // t = "Hello World!!"
```

Si `trim` una cadena que no tiene espacios en blanco para eliminar, se le devolverá la misma instancia de cadena.

Tenga en cuenta que el método `trim()` tiene su propia noción de espacio en blanco , que difiere de la noción utilizada por el método `Character.isWhitespace()` :

- Todos los caracteres de control ASCII con los códigos `U+0000` a `U+0020` se consideran espacios en blanco y se eliminan mediante `trim()` . Esto incluye los caracteres `U+0020 'SPACE'` , `U+0009 'CHARACTER TABULATION'` , `U+000A 'LINE FEED'` y `U+000D 'CARRIAGE RETURN'` , pero también los caracteres como `U+0007 'BELL'` .
- Los espacios en blanco de Unicode, como `U+00A0 'NO-BREAK SPACE'` o `U+2003 'EM SPACE'` *no* son reconocidos por `trim()` .

Cadena de almacenamiento y almacenamiento en montón

Al igual que muchos objetos Java, **todas las** instancias de `String` se crean en el montón, incluso literales. Cuando la JVM encuentra un literal de `String` que no tiene una referencia equivalente en

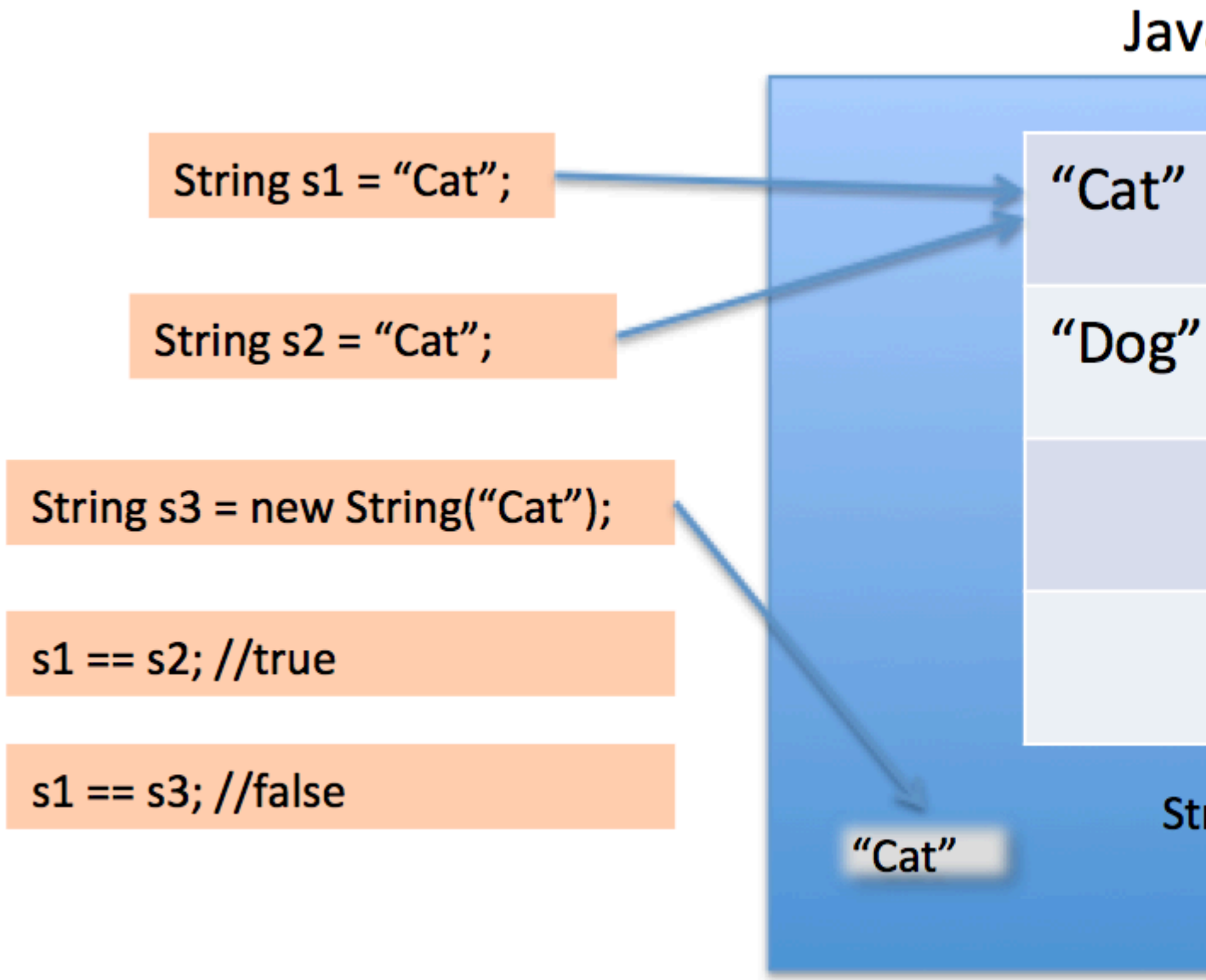
el montón, la JVM crea una instancia de `String` correspondiente en el montón **y** también almacena una referencia a la instancia de `String` recién creada en el conjunto de cadenas. Cualquier otra referencia al mismo literal de `String` se reemplaza con la instancia de `String` creada anteriormente en el montón.

Veamos el siguiente ejemplo:

```
class Strings
{
    public static void main (String[] args)
    {
        String a = "alpha";
        String b = "alpha";
        String c = new String("alpha");

        //All three strings are equivalent
        System.out.println(a.equals(b) && b.equals(c));

        //Although only a and b reference the same heap object
        System.out.println(a == b);
        System.out.println(a != c);
        System.out.println(b != c);
    }
}
```

La salida de lo anterior es:

```
true
true
true
true
```



Cuando usamos comillas dobles para crear una Cadena, primero busca Cadena con el mismo valor en el conjunto de Cadenas, si se encuentra, simplemente devuelve la referencia, sino que crea una nueva Cadena en el conjunto y luego devuelve la referencia.

Sin embargo, al utilizar un nuevo operador, obligamos a la clase String a crear un nuevo objeto String en el espacio de almacenamiento dinámico. Podemos usar el método intern () para ponerlo en el grupo o referirnos a otro objeto String del grupo de cadenas que tenga el mismo valor.

El conjunto de cadenas en sí también se crea en el montón.

Java SE 7

Antes de Java 7, los **literales de** `String` se almacenaban en el grupo de constantes de tiempo de ejecución en el área de método de `PermGen` , que tenía un tamaño fijo.

El grupo de cuerdas también residía en `PermGen` .

Java SE 7

RFC: 6962931

> En JDK 7, las cadenas internas ya no se asignan en la generación permanente del montón de Java, sino que se asignan en la parte principal del montón de Java (conocidas como generaciones jóvenes y viejas), junto con los otros objetos creados por la aplicación . Este cambio dará como resultado que más datos residan en el montón principal de Java y menos datos en la generación permanente, por lo que es posible que sea necesario ajustar los tamaños del montón. La mayoría de las aplicaciones solo verán diferencias relativamente pequeñas en el uso del montón debido a este cambio, pero las aplicaciones más grandes que cargan muchas clases o hacen un uso intensivo del método `String.intern()` verán diferencias más significativas.

Interruptor insensible a la caja

Java SE 7

`switch` sí no se puede parametrizar para que no distinga mayúsculas de minúsculas, pero si es absolutamente necesario, puede comportarse de manera insensible a la cadena de entrada utilizando `toLowerCase()` o `toUpperCase` :

```
switch (myString.toLowerCase()) {
     case "case1" :
        ...
     break;
     case "case2" :
        ...
     break;
}
```

Tener cuidado

- `Locale` puede afectar cómo suceden los casos cambiantes !
- Se debe tener cuidado de no tener caracteres en mayúscula en las etiquetas, ¡nunca se ejecutarán!

Lea Instrumentos de cuerda en línea: https://riptutorial.com/es/java/topic/109/instrumentos-de-cuerda

Capítulo 99: Interfaces

Introducción

Una *interfaz* es un tipo de referencia, similar a una clase, que se puede declarar utilizando la palabra clave de la `interface` . Las interfaces solo pueden contener constantes, firmas de métodos, métodos predeterminados, métodos estáticos y tipos anidados. Los cuerpos de los métodos existen solo para los métodos predeterminados y los métodos estáticos. Al igual que las clases abstractas, las interfaces no pueden ser instanciadas, solo pueden implementarse por clases o extenderse por otras interfaces. La interfaz es una forma común de lograr una abstracción completa en Java.

Sintaxis

- interfaz pública Foo {void foo (); / * cualquier otro método * /}
- interfaz pública Foo1 extiende Foo {void bar (); / * cualquier otro método * /}
- la clase pública Foo2 implementa Foo, Foo1 {/ * implementación de Foo y Foo1 * /}

Examples

Declarar e implementar una interfaz

Declaración de una interfaz usando la palabra clave de la `interface` :

```
public interface Animal {
    String getSound(); // Interface methods are public by default
}
```

Anular anotación

```
@Override
public String getSound() {
    // Code goes here...
}
```

Esto obliga al compilador a verificar que estamos anulando e impide que el programa defina un nuevo método o arruine la firma del método.

Las interfaces se implementan utilizando la palabra clave `implements` .

```
public class Cat implements Animal {

    @Override
    public String getSound() {
        return "meow";
    }
}
```

```
public class Dog implements Animal {

    @Override
    public String getSound() {
        return "woof";
    }
}
```

En el ejemplo, las clases `Cat` y `Dog` **deben** definir el método `getSound()` , ya que los métodos de una interfaz son inherentemente abstractos (con la excepción de los métodos predeterminados).

Usando las interfaces

```
Animal cat = new Cat();
Animal dog = new Dog();

System.out.println(cat.getSound()); // prints "meow"
System.out.println(dog.getSound()); // prints "woof"
```

Implementando multiples interfaces

Una clase de Java puede implementar múltiples interfaces.

```
public interface NoiseMaker {
    String noise = "Making Noise"; // interface variables are public static final by default

    String makeNoise(); //interface methods are public abstract by default
}

public interface FoodEater {
    void eat(Food food);
}

public class Cat implements NoiseMaker, FoodEater {
    @Override
    public String makeNoise() {
        return "meow";
    }

    @Override
    public void eat(Food food) {
        System.out.println("meows appreciatively");
    }
}
```

Observe cómo la clase `Cat` **debe** implementar los métodos `abstract` heredados en ambas interfaces. Además, observe cómo una clase puede implementar prácticamente tantas interfaces como sea necesario (hay un límite de **65,535** debido a la limitación de JVM).

```
NoiseMaker noiseMaker = new Cat(); // Valid
FoodEater foodEater = new Cat(); // Valid
Cat cat = new Cat(); // valid

Cat invalid1 = new NoiseMaker(); // Invalid
```

```
Cat invalid2 = new FoodEater(); // Invalid
```

Nota:

1. Todas las variables declaradas en una interfaz son `public static final`
2. Todos los métodos declarados en una interfaz son `public abstract` (esta declaración es válida solo a través de Java 7. Desde Java 8, se le permite tener métodos en una interfaz, que no tienen por qué ser abstractos; estos métodos se conocen como métodos predeterminados)
3. Las interfaces no pueden ser declaradas como `final`
4. Si más de una interfaz declara un método que tiene una firma idéntica, efectivamente se trata como un solo método y no se puede distinguir de qué método de interfaz se implementa
5. Se generará un archivo **InterfaceName.class** correspondiente para cada interfaz, luego de la compilación

Extendiendo una interfaz

Una interfaz puede extender otra interfaz a través de la `extends` palabra clave.

```
public interface BasicResourceService {
    Resource getResource();
}

public interface ExtendedResourceService extends BasicResourceService {
    void updateResource(Resource resource);
}
```

Ahora, una clase que implemente `ExtendedResourceService` deberá implementar tanto `getResource()` como `updateResource()` .

Extendiendo multiples interfaces

A diferencia de las clases, la `extends` palabra clave puede utilizarse para ampliar varias interfaces (separados por comas) permitiendo combinaciones de interfaces en una nueva interfaz

```
public interface BasicResourceService {
    Resource getResource();
}

public interface AlternateResourceService {
    Resource getAlternateResource();
}

public interface ExtendedResourceService extends BasicResourceService,
AlternateResourceService {
    Resource updateResource(Resource resource);
}
```

En este caso, una clase que implemente `ExtendedResourceService` deberá implementar `getResource()` , `getAlternateResource()` y `updateResource()` .

Usando interfaces con genéricos

Supongamos que desea definir una interfaz que permita publicar / consumir datos hacia y desde diferentes tipos de canales (por ejemplo, AMQP, JMS, etc.), pero desea poder cambiar los detalles de la implementación ...

Definamos una interfaz de IO básica que puede reutilizarse en múltiples implementaciones:

```
public interface IO<IncomingType, OutgoingType> {

    void publish(OutgoingType data);
    IncomingType consume();
    IncomingType RPCSubmit(OutgoingType data);

}
```

Ahora puedo crear una instancia de esa interfaz, pero como no tenemos implementaciones predeterminadas para esos métodos, necesitará una implementación cuando la creamos:

```
    IO<String, String> mockIO = new IO<String, String>() {

        private String channel = "somechannel";

        @Override
        public void publish(String data) {
            System.out.println("Publishing " + data + " to " + channel);
        }

        @Override
        public String consume() {
            System.out.println("Consuming from " + channel);
            return "some useful data";
        }

        @Override
        public String RPCSubmit(String data) {
            return "received " + data + " just now ";
        }

    };

    mockIO.consume(); // prints: Consuming from somechannel
    mockIO.publish("TestData"); // Publishing TestData to somechannel
    System.out.println(mockIO.RPCSubmit("TestData")); // received TestData just now
```

También podemos hacer algo más útil con esa interfaz, digamos que queremos usarla para envolver algunas funciones básicas de RabbitMQ:

```
public class RabbitMQ implements IO<String, String> {

    private String exchange;
    private String queue;

    public RabbitMQ(String exchange, String queue){
        this.exchange = exchange;
```

```
        this.queue = queue;
    }

    @Override
    public void publish(String data) {
        rabbit.basicPublish(exchange, queue, data.getBytes());
    }

    @Override
    public String consume() {
        return rabbit.basicConsume(exchange, queue);
    }

    @Override
    public String RPCSubmit(String data) {
        return rabbit.rpcPublish(exchange, queue, data);
    }

}
```

Digamos que quiero usar esta interfaz de E / S ahora como una forma de contar las visitas a mi sitio web desde mi último reinicio del sistema y luego poder mostrar el número total de visitas. Puede hacer algo como esto:

```
import java.util.concurrent.atomic.AtomicLong;

public class VisitCounter implements IO<Long, Integer> {

    private static AtomicLong websiteCounter = new AtomicLong(0);

    @Override
    public void publish(Integer count) {
        websiteCounter.addAndGet(count);
    }

    @Override
    public Long consume() {
        return websiteCounter.get();
    }

    @Override
    public Long RPCSubmit(Integer count) {
        return websiteCounter.addAndGet(count);
    }

}
```

Ahora vamos a usar el VisitCounter:

```
    VisitCounter counter = new VisitCounter();

    // just had 4 visits, yay
    counter.publish(4);
    // just had another visit, yay
    counter.publish(1);

    // get data for stats counter
    System.out.println(counter.consume()); // prints 5
```

```
    // show data for stats counter page, but include that as a page view
    System.out.println(counter.RPCSubmit(1)); // prints 6
```

Al implementar varias interfaces, no puede implementar la misma interfaz dos veces. Eso también se aplica a las interfaces genéricas. Por lo tanto, el siguiente código no es válido y generará un error de compilación:

```
interface Printer<T> {
    void print(T value);
}

// Invalid!
class SystemPrinter implements Printer<Double>, Printer<Integer> {
    @Override public void print(Double d){ System.out.println("Decimal: " + d); }
    @Override public void print(Integer i){ System.out.println("Discrete: " + i); }
}
```

Utilidad de las interfaces.

Las interfaces pueden ser extremadamente útiles en muchos casos. Por ejemplo, supongamos que tenía una lista de animales y quería recorrer la lista, cada uno imprimiendo el sonido que hacen.

```
{cat, dog, bird}
```

Una forma de hacer esto sería usar interfaces. Esto permitiría que se llame al mismo método en todas las clases

```
public interface Animal {
    public String getSound();
}
```

Cualquier clase que `implements Animal` también debe tener un método `getSound()` , pero todas pueden tener implementaciones diferentes

```
public class Dog implements Animal {
    public String getSound() {
        return "Woof";
    }
}

public class Cat implements Animal {
    public String getSound() {
        return "Meow";
    }
}

public class Bird implements Animal{
    public String getSound() {
        return "Chirp";
    }
```

```
}
```

Ahora tenemos tres clases diferentes, cada una de las cuales tiene un método `getSound()` . Debido a que todas estas clases `implement` la interfaz `Animal` , que declara el método `getSound()` , cualquier instancia de un `Animal` puede tener `getSound()` llamada.

```
Animal dog = new Dog();
Animal cat = new Cat();
Animal bird = new Bird();

dog.getSound(); // "Woof"
cat.getSound(); // "Meow"
bird.getSound(); // "Chirp"
```

Debido a que cada uno de estos es un `Animal` , incluso podríamos poner a los animales en una lista, recorrerlos e imprimir sus sonidos.

```
Animal[] animals = { new Dog(), new Cat(), new Bird() };
for (Animal animal : animals) {
    System.out.println(animal.getSound());
}
```

Debido a que el orden de la matriz es `Dog` , `Cat` y luego `Bird` , *"Woof Meow Chirp"* se imprimirá en la consola.

Las interfaces también se pueden utilizar como valor de retorno para las funciones. Por ejemplo, devolver un `Dog` si la entrada es *"perro"* , `Cat` si la entrada es *"gato"* y `Bird` si es *"pájaro"* , y luego imprimir el sonido de ese animal se podría hacer usando

```
public Animal getAnimalByName(String name) {
    switch(name.toLowerCase()) {
        case "dog":
            return new Dog();
        case "cat":
            return new Cat();
        case "bird":
            return new Bird();
        default:
            return null;
    }
}

public String getAnimalSoundByName(String name){
    Animal animal = getAnimalByName(name);
    if (animal == null) {
        return null;
    } else {
        return animal.getSound();
    }
}

String dogSound = getAnimalSoundByName("dog"); // "Woof"
String catSound = getAnimalSoundByName("cat"); // "Meow"
String birdSound = getAnimalSoundByName("bird"); // "Chirp"
String lightbulbSound = getAnimalSoundByName("lightbulb"); // null
```

Las interfaces también son útiles para la extensibilidad, porque si quieres agregar un nuevo tipo de `Animal` , no necesitarás cambiar nada con las operaciones que realices en ellos.

Implementando interfaces en una clase abstracta.

Un método definido en una `interface` es por defecto `public abstract` . Cuando una `abstract class` implementa una `interface` , los métodos que se definen en la `interface` no tienen que ser implementado por el `abstract class` . Esto se debe a que una `class` que se declara `abstract` puede contener declaraciones de métodos abstractos. Por lo tanto, es responsabilidad de la primera subclase concreta implementar cualquier método `abstract` heredado de cualquier interfaz y / o la `abstract class` .

```
public interface NoiseMaker {
    void makeNoise();
}

public abstract class Animal implements NoiseMaker {
    //Does not need to declare or implement makeNoise()
    public abstract void eat();
}

//Because Dog is concrete, it must define both makeNoise() and eat()
public class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Borf borf");
    }

    @Override
    public void eat() {
        System.out.println("Dog eats some kibble.");
    }
}
```

Desde Java 8 en adelante, es posible que una `interface` declare implementaciones `default` de métodos, lo que significa que el método no será `abstract` , por lo tanto, no se obligará a ninguna subclase concreta a implementar el método, sino que heredará la implementación `default` menos que se anule.

Métodos predeterminados

Introducido en Java 8, los métodos predeterminados son una forma de especificar una implementación dentro de una interfaz. Esto podría usarse para evitar la típica clase "Base" o "Resumen" al proporcionar una implementación parcial de una interfaz y restringir la jerarquía de subclases.

Implementación del patrón observador

Por ejemplo, es posible implementar el patrón Observer-Listener directamente en la interfaz, proporcionando más flexibilidad a las clases de implementación.

```
interface Observer {
    void onAction(String a);
}

interface Observable{
    public abstract List<Observer> getObservers();

    public default void addObserver(Observer o){
        getObservers().add(o);
    }

    public default void notify(String something ){
        for( Observer l : getObservers() ){
            l.onAction(something);
        }
    }
}
```

Ahora, cualquier clase puede convertirse en "Observable" solo con la implementación de la interfaz de Observable, y ser libre de ser parte de una jerarquía de clases diferente.

```
abstract class Worker{
    public abstract void work();
}

public class MyWorker extends Worker implements Observable {

    private List<Observer> myObservers = new ArrayList<Observer>();

    @Override
    public List<Observer> getObservers() {
        return myObservers;
    }

    @Override
    public void work(){
        notify("Started work");

        // Code goes here...

        notify("Completed work");
    }

    public static void main(String[] args) {
        MyWorker w = new MyWorker();

        w.addListener(new Observer() {
            @Override
            public void onAction(String a) {
                System.out.println(a + " (" + new Date() + ")");
            }
        });

        w.work();
    }
}
```

Problema del diamante

El compilador en Java 8 es consciente del problema de diamante que se produce cuando una clase implementa interfaces que contienen un método con la misma firma.

Para resolverlo, una clase de implementación debe anular el método compartido y proporcionar su propia implementación.

```
interface InterfaceA {
    public default String getName(){
        return "a";
    }
}

interface InterfaceB {
    public default String getName(){
        return "b";
    }
}

public class ImpClass implements InterfaceA, InterfaceB {

    @Override
    public String getName() {
        //Must provide its own implementation
        return InterfaceA.super.getName() + InterfaceB.super.getName();
    }

    public static void main(String[] args) {
        ImpClass c = new ImpClass();

        System.out.println( c.getName() );                    // Prints "ab"
        System.out.println( ((InterfaceA)c).getName() );      // Prints "ab"
        System.out.println( ((InterfaceB)c).getName() );      // Prints "ab"
    }
}
```

Todavía existe el problema de tener métodos con el mismo nombre y parámetros con diferentes tipos de retorno, que no se compilarán.

Utilice métodos predeterminados para resolver problemas de compatibilidad

Las implementaciones de los métodos predeterminados son muy útiles si se agrega un método a una interfaz en un sistema existente en el que varias clases usan las interfaces.

Para evitar dividir todo el sistema, puede proporcionar una implementación de método predeterminada cuando agrega un método a una interfaz. De esta manera, el sistema todavía se compilará y las implementaciones reales se pueden hacer paso a paso.

Para obtener más información, consulte el tema Métodos predeterminados .

Modificadores en interfaces

La Guía de estilo de Java de Oracle dice:

> Los modificadores no deben escribirse cuando están implícitos.

(Ver Modificadores en Oracle Official Code Standard para el contexto y un enlace al documento de Oracle real).

Esta guía de estilo se aplica particularmente a las interfaces. Consideremos el siguiente fragmento de código:

```
interface I {
    public static final int VARIABLE = 0;

    public abstract void method();

    public static void staticMethod() { ... }
    public default void defaultMethod() { ... }
}
```

Variables

Todas las variables de interfaz son *constantes* implícitamente con modificadores `public` (accesibles para todos), `static` (accesibles por nombre de interfaz) y `final` (deben inicializarse durante la declaración):

```
public static final int VARIABLE = 0;
```

Métodos

1. Todos los métodos que *no proporcionan implementación* son implícitamente `public` y `abstract` .

```
public abstract void method();
```

Java SE 8

2. Todos los métodos con modificador `static` o `default` *deben proporcionar implementación* y son implícitamente `public` .

```
public static void staticMethod() { ... }
```

Después de que se hayan aplicado todos los cambios anteriores, obtendremos lo siguiente:

```
interface I {
    int VARIABLE = 0;

    void method();

    static void staticMethod() { ... }
    default void defaultMethod() { ... }
}
```

Reforzar los parámetros de tipo acotado.

Los parámetros de tipo limitado le permiten establecer restricciones en los argumentos de tipo genérico:

```
class SomeClass {

}

class Demo<T extends SomeClass> {

}
```

Pero un parámetro de tipo solo puede vincularse a un solo tipo de clase.

Un tipo de interfaz se puede vincular a un tipo que ya tenía un enlace. Esto se logra usando el símbolo `&` :

```
interface SomeInterface {

}

class GenericClass<T extends SomeClass & SomeInterface> {

}
```

Esto fortalece el enlace, lo que potencialmente requiere que los argumentos de tipo se deriven de múltiples tipos.

Se pueden enlazar varios tipos de interfaz a un parámetro de tipo:

```
class Demo<T extends SomeClass & FirstInterface & SecondInterface> {

}
```

Pero debe utilizarse con precaución. Los enlaces de interfaz múltiples suelen ser un signo de olor de código , lo que sugiere que se debe crear un nuevo tipo que actúe como un adaptador para los otros tipos:

```
interface NewInterface extends FirstInterface, SecondInterface {

}

class Demo<T extends SomeClass & NewInterface> {
```

```
}
```

Lea Interfaces en línea: https://riptutorial.com/es/java/topic/102/interfaces

Capítulo 100: Interfaces funcionales

Introducción

En Java 8+, una *interfaz funcional* es una interfaz que tiene solo un método abstracto (aparte de los métodos de Object). Ver JLS §9.8. Interfaces funcionales .

Examples

Lista de interfaces funcionales estándar de Java Runtime Library por firma

Tipos de parámetros	Tipo de retorno	Interfaz
()	vacío	Ejecutable
()	T	Proveedor
()	booleano	BooleanSupplier
()	En t	IntSupplier
()	largo	LongSupplier
()	doble	DoubleSupplier
(T)	vacío	Consumidor <T>
(T)	T	UnaryOperator <T>
(T)	R	Función <T, R>
(T)	booleano	Predicado <T>
(T)	En t	ToIntFunction <T>
(T)	largo	ToLongFunction <T>
(T)	doble	ToDoubleFunction <T>
(T, T)	T	Operador Binario <T>
(T, U)	vacío	BiConsumer <T, U>
(T, U)	R	BiFunction <T, U, R>
(T, U)	booleano	BiPredicate <T, U>
(T, U)	En t	ToIntBiFunction <T, U>

Tipos de parámetros	Tipo de retorno	Interfaz
(T, U)	largo	ToLongBiFunction <T, U>
(T, U)	doble	ToDoubleBiFunction <T, U>
(T, int)	vacío	ObjIntConsumer <T>
(T, largo)	vacío	ObjLongConsumer <T>
(T, doble)	vacío	ObjDoubleConsumer <T>
(En t)	vacío	Consumidor
(En t)	R	IntFunction <R>
(En t)	booleano	IntPredicate
(En t)	En t	IntUnaryOperator
(En t)	largo	IntToLongFunction
(En t)	doble	IntToDoubleFunction
(int, int)	En t	IntBinaryOperator
(largo)	vacío	LongConsumer
(largo)	R	Función larga <R>
(largo)	booleano	LongPredicate
(largo)	En t	LongToIntFunction
(largo)	largo	LongUnaryOperator
(largo)	doble	LongToDoubleFunction
(largo largo)	largo	LongBinaryOperator
(doble)	vacío	DoubleConsumer
(doble)	R	DoubleFunction <R>
(doble)	booleano	DoublePredicate
(doble)	En t	DoubleToIntFunction
(doble)	largo	DoubleToLongFunction
(doble)	doble	DoubleUnaryOperator

Tipos de parámetros	Tipo de retorno	Interfaz
(doble doble)	doble	DoubleBinaryOperator

Lea Interfaces funcionales en línea: https://riptutorial.com/es/java/topic/10001/interfaces-funcionales

Capítulo 101: Interfaz de herramientas JVM

Observaciones

Interfaz de herramientas JVM TM

Versión 1.2

http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

Examples

Iterar sobre objetos accesibles desde objeto (Heap 1.0)

```
#include <vector>
#include <string>


#include "agent_util.hpp"
//this file can be found in Java SE Development Kit 8u101 Demos and Samples
//see http://download.oracle.com/otn-pub/java/jdk/8u101-b13-demos/jdk-8u101-windows-x64-
demos.zip
//jdk1.8.0_101.zip!\demo\jvmti\versionCheck\src\agent_util.h

/*
*   Struct used for jvmti->SetTag(object, <pointer to tag>);
*   http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#SetTag
*
*/
typedef struct Tag
{
    jlong referrer_tag;
    jlong size;
    char* classSignature;
    jint hashCode;
} Tag;

/*
* Utility function: jlong -> Tag*
*/
static Tag* pointerToTag(jlong tag_ptr)
{
    if (tag_ptr == 0)
    {
        return new Tag();
    }
    return (Tag*)(ptrdiff_t)(void*)tag_ptr;
}

/*
* Utility function: Tag* -> jlong
*/
static jlong tagToPointer(Tag* tag)
{
```

```
    return (jlong)(ptrdiff_t)(void*)tag;
}

/*
*  Heap 1.0 Callback
*  http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#jvmtiObjectReferenceCallback
*/
static jvmtiIterationControl JNICALL heabObjectReferencesCallback(
    jvmtiObjectReferenceKind reference_kind,
    jlong class_tag,
    jlong size,
    jlong* tag_ptr,
    jlong referrer_tag,
    jint referrer_index,
    void* user_data)
{
    //iterate only over reference field
    if (reference_kind != JVMTI_HEAP_REFERENCE_FIELD)
    {
        return JVMTI_ITERATION_IGNORE;
    }
    auto tag_ptr_list = (std::vector<jlong>*)(ptrdiff_t)(void*)user_data;
    //create and assign tag
    auto t = pointerToTag(*tag_ptr);
    t->referrer_tag = referrer_tag;
    t->size = size;
    *tag_ptr = tagToPointer(t);
    //collect tag
    (*tag_ptr_list).push_back(*tag_ptr);

    return JVMTI_ITERATION_CONTINUE;
}

/*
*  Main function for demonstration of Iterate Over Objects Reachable From Object
*
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject

*
*/
void iterateOverObjectHeapReferences(jvmtiEnv* jvmti, JNIEnv* env, jobject object)
{
    std::vector<jlong> tag_ptr_list;

    auto t = new Tag();
    jvmti->SetTag(object, tagToPointer(t));
    tag_ptr_list.push_back(tagToPointer(t));

    stdout_message("tag list size before call callback:  %d\n", tag_ptr_list.size());
    /*
    *    Call Callback for every reachable object reference
    *    see
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject

    */
    jvmti->IterateOverObjectsReachableFromObject(object, &heabObjectReferencesCallback,
(void*)&tag_ptr_list);
    stdout_message("tag list size after call callback:  %d\n", tag_ptr_list.size());

    if (tag_ptr_list.size() > 0)
    {
```

```
        jint found_count = 0;
        jlong* tags = &tag_ptr_list[0];
        jobject* found_objects;
        jlong* found_tags;

        /*
        *    collect all tagged object (via *tag_ptr = pointer to tag )
        *    see
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectsWithTags
        */
        jvmti->GetObjectsWithTags(tag_ptr_list.size(), tags, &found_count, &found_objects,
&found_tags);
        stdout_message("found %d objects\n", found_count);

        for (auto i = 0; i < found_count; ++i)
        {
            jobject found_object = found_objects[i];

            char* classSignature;
            jclass found_object_class = env->GetObjectClass(found_object);
            /*
            *    Get string representation of found_object_class
            *    see
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetClassSignature
            */
            jvmti->GetClassSignature(found_object_class, &classSignature, nullptr);

            jint hashCode;
            /*
            *    Getting hash code for found_object
            *    see
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectHashCode
            */
            jvmti->GetObjectHashCode(found_object, &hashCode);

            //save all it in Tag
            Tag* t = pointerToTag(found_tags[i]);
            t->classSignature = classSignature;
            t->hashCode = hashCode;
        }

        //print all saved information
        for (auto i = 0; i < found_count; ++i)
        {
            auto t = pointerToTag(found_tags[i]);
            auto rt = pointerToTag(t->referrer_tag);

            if (t->referrer_tag != 0)
            {
                stdout_message("referrer object %s#%d --> object %s#%d (size: %2d)\n",
                    rt->classSignature, rt->hashCode, t->classSignature, t->hashCode, t-
>size);
            }
        }
    }
}
```

Obtener entorno JVMTI

Dentro del método Agent_OnLoad:

```
jvmtiEnv* jvmti;
/* Get JVMTI environment */
vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
```

Ejemplo de inicialización dentro del método Agent_OnLoad

```
/* Callback for JVMTI_EVENT_VM_INIT */
static void JNICALL vm_init(jvmtiEnv* jvmti, JNIEnv* env, jthread thread)
{
    jint runtime_version;
    jvmti->GetVersionNumber(&runtime_version);
    stdout_message("JVMTI Version: %d\n", runtime_verision);
}

/* Agent_OnLoad() is called first, we prepare for a VM_INIT event here. */
JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM* vm, char* options, void* reserved)
{
    jint rc;
    jvmtiEventCallbacks callbacks;
    jvmtiCapabilities capabilities;
    jvmtiEnv* jvmti;

    /* Get JVMTI environment */
    rc = vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
    if (rc != JNI_OK)
    {
        return -1;
    }

    /* Immediately after getting the jvmtiEnv* we need to ask for the
    *  capabilities this agent will need.
    */
    jvmti->GetCapabilities(&capabilities);
    capabilities.can_tag_objects = 1;
    jvmti->AddCapabilities(&capabilities);

    /* Set callbacks and enable event notifications */
    memset(&callbacks, 0, sizeof(callbacks));
    callbacks.VMInit = &vm_init;

    jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));
    jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_VM_INIT, nullptr);

    return JNI_OK;
}
```

Lea Interfaz de herramientas JVM en línea: https://riptutorial.com/es/java/topic/3316/interfaz-de-herramientas-jvm

Capítulo 102: Interfaz de salida de cola

Introducción

Un Deque es una colección lineal que admite la inserción y eliminación de elementos en ambos extremos.

El nombre deque es corto para "doble cola de espera" y generalmente se pronuncia "deck".

La mayoría de las implementaciones de Deque no establecen límites fijos en el número de elementos que pueden contener, pero esta interfaz es compatible con los deques con capacidad limitada, así como aquellos sin límite de tamaño fijo.

La interfaz de Deque es un tipo de datos abstracto más rico que tanto la Pila como la Cola porque implementa pilas y colas al mismo tiempo

Observaciones

Los genéricos se pueden utilizar con Deque.

```
Deque<Object> deque = new LinkedList<Object>();
```

Cuando se utiliza un deque como una cola, se produce el comportamiento FIFO (primero en entrar, primero en salir).

Deques también se puede utilizar como pilas LIFO (Last-In-First-Out-Out).

Para obtener más información sobre los métodos, consulte esta documentación.

Examples

Añadiendo elementos a Deque

```
Deque deque = new LinkedList();

//Adding element at tail
deque.add("Item1");

//Adding element at head
deque.addFirst("Item2");

//Adding element at tail
deque.addLast("Item3");
```

Eliminando Elementos de Deque

```
//Retrieves and removes the head of the queue represented by this deque
Object headItem = deque.remove();

//Retrieves and removes the first element of this deque.
Object firstItem = deque.removeFirst();

//Retrieves and removes the last element of this deque.
Object lastItem = deque.removeLast();
```

Recuperando elemento sin quitar

```
//Retrieves, but does not remove, the head of the queue represented by this deque
Object headItem = deque.element();

//Retrieves, but does not remove, the first element of this deque.
Object firstItem = deque.getFirst();

//Retrieves, but does not remove, the last element of this deque.
Object lastItem  = deque.getLast();
```

Iterando a través de Deque

```
//Using Iterator
Iterator iterator = deque.iterator();
while(iterator.hasNext(){
  String Item = (String) iterator.next();
}

//Using For Loop
for(Object object : deque) {
    String Item = (String) object;
}
```

Lea Interfaz de salida de cola en línea: https://riptutorial.com/es/java/topic/10156/interfaz-de-salida-de-cola

Capítulo 103: Interfaz fluida

Observaciones

Metas

El objetivo principal de una interfaz fluida es una mayor legibilidad.

Cuando se usa para construir objetos, las opciones disponibles para el llamante se pueden hacer claramente y cumplir mediante controles en tiempo de compilación. Por ejemplo, considere el siguiente árbol de opciones que representan pasos a lo largo de la ruta para construir un objeto complejo:

```
A -> B
  -> C -> D -> Done
       -> E -> Done
       -> F -> Done.
       -> G -> H -> I -> Done.
```

Un constructor que utiliza una interfaz fluida permitiría a la persona que llama ver fácilmente qué opciones están disponibles en cada paso. Por ejemplo, **A -> B** es posible, pero **A -> C** no es y daría como resultado un error en tiempo de compilación.

Examples

Verdad - Marco de prueba de fluidez

De "Cómo utilizar la verdad" http://google.github.io/truth/

```
String string = "awesome";
assertThat(string).startsWith("awe");
assertWithMessage("Without me, it's just aweso").that(string).contains("me");

Iterable<Color> googleColors = googleLogo.getColors();
assertThat(googleColors)
    .containsExactly(BLUE, RED, YELLOW, BLUE, GREEN, RED)
    .inOrder();
```

Estilo de programación fluido

En el estilo de programación fluido, devuelve `this` desde métodos fluidos (configuradores) que no devolverían nada en un estilo de programación no fluido.

Esto le permite encadenar las diferentes llamadas de métodos, lo que hace que su código sea más corto y más fácil de manejar para los desarrolladores.

Considere este código no fluido:

```
public class Person {
  private  String firstName;
  private String lastName;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String whoAreYou() {
    return "I am " + firstName + " " + lastName;
  }

  public static void main(String[] args) {
    Person person = new Person();
    person.setFirstName("John");
    person.setLastName("Doe");
    System.out.println(person.whoAreYou());
  }
}
```

Como los métodos de establecimiento no devuelven nada, necesitamos 4 instrucciones en el método `main` para crear una instancia de una `Person` con algunos datos e imprimirla. Con un estilo fluido, este código se puede cambiar a:

```
public class Person {
  private  String firstName;
  private String lastName;

  public String getFirstName() {
    return firstName;
  }

  public Person withFirstName(String firstName) {
    this.firstName = firstName;
    return this;
  }

  public String getLastName() {
    return lastName;
  }

  public Person withLastName(String lastName) {
    this.lastName = lastName;
    return this;
  }

  public String whoAreYou() {
```

```
    return "I am " + firstName + " " + lastName;
  }

  public static void main(String[] args) {
    System.out.println(new Person().withFirstName("John")
      .withLastName("Doe").whoAreYou());
  }
}
```

La idea es devolver siempre algún objeto para permitir la construcción de una cadena de llamadas a métodos y usar nombres de métodos que reflejen el habla natural. Este estilo fluido hace que el código sea más legible.

Lea Interfaz fluida en línea: https://riptutorial.com/es/java/topic/5090/interfaz-fluida

Capítulo 104: Interfaz nativa de Java

Parámetros

Parámetro	Detalles
JNIEnv	Puntero al entorno JNI.
proyecto de trabajo	El objeto que invocó el método `native` no `static`
jclass	La clase que invocó el método `native static`

Observaciones

La configuración de JNI requiere tanto de Java como de un compilador nativo. Dependiendo del IDE y el sistema operativo, se requiere alguna configuración. Una guía para Eclipse se puede encontrar aquí . Un tutorial completo se puede encontrar aquí .

Estos son los pasos para configurar el enlace Java-C ++ en Windows:

- Compile los archivos fuente de Java (`.java`) en clases (`.class`) usando `javac` .
- Cree archivos de encabezado (`.h`) a partir de las clases de Java que contienen métodos `native` utilizando `javah` . Estos archivos "instruyen" al código nativo de qué métodos es responsable de implementar.
- Incluya los archivos de encabezado (`#include`) en los archivos fuente de C ++ (`.cpp`) que implementan los métodos `native` .
- Compile los archivos fuente de C ++ y cree una biblioteca (`.dll`). Esta biblioteca contiene la implementación del código nativo.
- Especifique la ruta de la biblioteca (`-Djava.library.path`) y `-Djava.library.path` en el archivo fuente de Java (`System.loadLibrary(...)`).

Las devoluciones de llamada (Llamar a métodos Java desde código nativo) requieren especificar un descriptor de método. Si el descriptor es incorrecto, se produce un error de tiempo de ejecución. Debido a esto, es útil tener los descriptores creados para nosotros, esto se puede hacer con `javap -s` .

Examples

Llamando a los métodos de C ++ desde Java

Los métodos estáticos y de miembro en Java se pueden marcar como *nativos* para indicar que su implementación se encuentra en un archivo de biblioteca compartida. Tras la ejecución de un método nativo, la JVM busca una función correspondiente en las bibliotecas cargadas (consulte Cómo cargar bibliotecas nativas), utilizando un esquema simple de identificación de nombres,

realiza la conversión de argumentos y la configuración de la pila, luego entrega el control al código nativo.

Código Java

```
/*** com/example/jni/JNIJava.java **/

package com.example.jni;

public class JNIJava {
    static {
        System.loadLibrary("libJNI_CPP");
    }

    // Obviously, native methods may not have a body defined in Java
    public native void printString(String name);
    public static native double average(int[] nums);

    public static void main(final String[] args) {
        JNIJava jniJava = new JNIJava();
        jniJava.printString("Invoked C++ 'printString' from Java");

        double d = average(new int[]{1, 2, 3, 4, 7});
        System.out.println("Got result from C++ 'average': " + d);
    }
}
```

Código C ++

Los archivos de encabezado que contienen declaraciones de funciones nativas deben generarse utilizando la herramienta `javah` en las clases de destino. Ejecutando el siguiente comando en el directorio de compilación:

```
javah -o com_example_jni_JNIJava.hpp com.example.jni.JNIJava
```

... produce el siguiente archivo de encabezado (*comentarios eliminados por brevedad*):

```
// com_example_jni_JNIJava.hpp

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h> // The JNI API declarations

#ifndef _Included_com_example_jni_JNIJava
#define _Included_com_example_jni_JNIJava
#ifdef __cplusplus
extern "C" { // This is absolutely required if using a C++ compiler
#endif

JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString
  (JNIEnv *, jobject, jstring);

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average
  (JNIEnv *, jclass, jintArray);
```

```
#ifdef __cplusplus
}
#endif
#endif
```

Aquí hay una implementación de ejemplo:

```
// com_example_jni_JNIJava.cpp

#include <iostream>
#include "com_example_jni_JNIJava.hpp"

using namespace std;

JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString(JNIEnv *env, jobject jthis,
jstring string) {
    const char *stringInC = env->GetStringUTFChars(string, NULL);
    if (NULL == stringInC)
        return;
    cout << stringInC << endl;
    env->ReleaseStringUTFChars(string, stringInC);
}

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average(JNIEnv *env, jclass jthis,
jintArray intArray) {
    jint *intArrayInC = env->GetIntArrayElements(intArray, NULL);
    if (NULL == intArrayInC)
        return -1;
    jsize length = env->GetArrayLength(intArray);
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += intArrayInC[i];
    }
    env->ReleaseIntArrayElements(intArray, intArrayInC, 0);
    return (double) sum / length;
}
```

Salida

Al ejecutar la clase de ejemplo anterior se obtiene el siguiente resultado:

> Invocó C ++ 'printString' desde Java
> Resultado obtenido de C ++ 'promedio': 3.4

Llamando a métodos Java desde C ++ (callback)

Llamar a un método Java desde un código nativo es un proceso de dos pasos:

1. obtenga un puntero de método con la función JNI de `GetMethodID` , utilizando el nombre y el descriptor del método;
2. llame a una de las funciones del `Call*Method` enumeradas aquí .

Código Java

```
/*** com.example.jni.JNIJavaCallback.java ***/

package com.example.jni;

public class JNIJavaCallback {
    static {
        System.loadLibrary("libJNI_CPP");
    }

    public static void main(String[] args) {
        new JNIJavaCallback().callback();
    }

    public native void callback();

    public static void printNum(int i) {
        System.out.println("Got int from C++: " + i);
    }

    public void printFloat(float i) {
        System.out.println("Got float from C++: " + i);
    }
}
```

Código C ++

```
// com_example_jni_JNICppCallback.cpp

#include <iostream>
#include "com_example_jni_JNIJavaCallback.h"

using namespace std;

JNIEXPORT void JNICALL Java_com_example_jni_JNIJavaCallback_callback(JNIEnv *env, jobject
jthis) {
    jclass thisClass = env->GetObjectClass(jthis);

    jmethodID printFloat = env->GetMethodID(thisClass, "printFloat", "(F)V");
    if (NULL == printFloat)
        return;
    env->CallVoidMethod(jthis, printFloat, 5.221);

    jmethodID staticPrintInt = env->GetStaticMethodID(thisClass, "printNum", "(I)V");
    if (NULL == staticPrintInt)
        return;
    env->CallVoidMethod(jthis, staticPrintInt, 17);
}
```

Salida

Consiguió flotar desde C ++: 5.221
Obtuvo int desde C ++: 17

Obteniendo el descriptor

Los descriptores (o *firmas de tipo interno*) se obtienen utilizando el programa **javap** en el archivo `.class` compilado. Aquí está la salida de `javap -p -s com.example.jni.JNIJavaCallback` :

```
Compiled from "JNIJavaCallback.java"
public class com.example.jni.JNIJavaCallback {
  static {};
    descriptor: ()V

  public com.example.jni.JNIJavaCallback();
    descriptor: ()V

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V

  public native void callback();
    descriptor: ()V

  public static void printNum(int);
    descriptor: (I)V  // <---- Needed

  public void printFloat(float);
    descriptor: (F)V  // <---- Needed
}
```

Cargando bibliotecas nativas

El idioma común para cargar archivos de biblioteca compartida en Java es el siguiente:

```
public class ClassWithNativeMethods {
    static {
        System.loadLibrary("Example");
    }

    public native void someNativeMethod(String arg);
    ...
```

Las llamadas a `System.loadLibrary` son casi siempre estáticas para que ocurran durante la carga de clases, asegurando que ningún método nativo pueda ejecutarse antes de que se haya cargado la biblioteca compartida. Sin embargo, lo siguiente es posible:

```
public class ClassWithNativeMethods {
    // Call this before using any native method
    public static void prepareNativeMethods() {
        System.loadLibrary("Example");
    }

    ...
```

Esto permite diferir la carga de la biblioteca compartida hasta que sea necesario, pero requiere un cuidado especial para evitar `java.lang.UnsatisfiedLinkError` s.

Búsqueda de archivos de destino

Los archivos de la biblioteca compartida se buscan en las rutas definidas por la propiedad del sistema `java.library.path` , que se pueden anular utilizando el argumento `-Djava.library.path=` JVM en el tiempo de ejecución:

```
java -Djava.library.path=path/to/lib/:path/to/other/lib MainClassWithNativeMethods
```

Tenga cuidado con los separadores de ruta del sistema: por ejemplo, Windows usa `;` en lugar de `:`

Tenga en cuenta que `System.loadLibrary` resuelve los nombres de archivo de la biblioteca de una manera dependiente de la plataforma: el fragmento de código anterior espera un archivo llamado `libExample.so` en Linux y `Example.dll` en Windows.

Una alternativa a `System.loadLibrary` es `System.load(String)` , que toma la ruta completa a un archivo de biblioteca compartida, evitando la búsqueda `java.library.path` :

```
public class ClassWithNativeMethods {
    static {
        System.load("/path/to/lib/libExample.so");
    }

    ...
```

Lea Interfaz nativa de Java en línea: https://riptutorial.com/es/java/topic/168/interfaz-nativa-de-java

Capítulo 105: Invocación de método remoto (RMI)

Observaciones

RMI requiere 3 componentes: cliente, servidor y una interfaz remota compartida. La interfaz remota compartida define el contrato cliente-servidor especificando los métodos que debe implementar un servidor. La interfaz debe ser visible para el servidor para que pueda implementar los métodos; la interfaz debe ser visible para el cliente para que sepa qué métodos ("servicios") proporciona el servidor.
Cualquier objeto que implemente una interfaz remota está destinado a asumir el rol de un servidor. Como tal, una relación cliente-servidor en la que el servidor también puede invocar métodos en el cliente es, de hecho, una relación servidor-servidor. Esto se denomina *devolución de llamada* ya que el servidor puede devolver la llamada al "cliente". Teniendo esto en cuenta, es aceptable utilizar el *cliente de* designación para los servidores que funcionan como tales.

La interfaz remota compartida es cualquier interfaz que se extiende a `Remote` . Un objeto que funciona como un servidor experimenta lo siguiente:

1. Implementa la interfaz remota compartida, ya sea explícita o implícitamente al extender `UnicastRemoteObject` que implementa `Remote` .
2. Se exporta, ya sea de forma implícita si extiende `UnicastRemoteObject` , o explícitamente se pasa a `UnicastRemoteObject#exportObject` .
3. Enlazado en un registro, ya sea directamente a través del `Registry` o indirectamente a través de `Naming` . Esto solo es necesario para establecer una comunicación inicial, ya que se pueden pasar más apéndices directamente a través de RMI.

En la configuración del proyecto, los proyectos de cliente y servidor no están relacionados, pero cada uno especifica un proyecto compartido en su ruta de compilación. El proyecto compartido contiene las interfaces remotas.

Examples

Cliente-Servidor: invocando métodos en una JVM desde otra

La interfaz remota compartida:

```
package remote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteServer extends Remote {

    int stringToInt(String string) throws RemoteException;
}
```

El servidor que implementa la interfaz remota compartida:

```
package server;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import remote.RemoteServer;

public class Server implements RemoteServer {

    @Override
    public int stringToInt(String string) throws RemoteException {

        System.out.println("Server received: \"" + string + "\"");
        return Integer.parseInt(string);
    }

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            Server server = new Server();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            reg.rebind("ServerName", server);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

El cliente que invoca un método en el servidor (de forma remota):

```
package client;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import remote.RemoteServer;

public class Client {

    static RemoteServer server;

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.getRegistry();
            server = (RemoteServer) reg.lookup("ServerName");
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }

        Client client = new Client();
        client.callServer();
```

```
    }

    void callServer() {

        try {
            int i = server.stringToInt("120");
            System.out.println("Client received: " + i);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Salida:

> Servidor recibido: "120"
> Cliente recibido: 120

Devolución de llamada: invocar métodos en un "cliente"

Visión general

En este ejemplo, 2 clientes se envían información a través de un servidor. Un cliente envía al servidor un número que se transmite al segundo cliente. El segundo cliente divide el número a la mitad y lo envía de vuelta al primer cliente a través del servidor. El primer cliente hace lo mismo. El servidor detiene la comunicación cuando el número devuelto por cualquiera de los clientes es inferior a 10. El valor devuelto por el servidor a los clientes (el número que se convirtió en representación de cadena) luego realiza un seguimiento del proceso.

1. Un servidor de inicio de sesión se une a un registro.
2. Un cliente busca el servidor de inicio de sesión y llama al método de `login` con su información. Entonces:
    - El servidor de inicio de sesión almacena la información del cliente. Incluye el talón del cliente con los métodos de devolución de llamada.
    - El servidor de inicio de sesión crea y devuelve un código auxiliar de servidor ("conexión" o "sesión") al cliente para que lo almacene. Incluye el código auxiliar del servidor con sus métodos, incluido un método de `logout` (no utilizado en este ejemplo).
3. Un cliente llama al `passInt` del servidor con el nombre del cliente destinatario y un `int` .
4. El servidor llama a la `half` en el cliente del destinatario con ese `int` . Esto inicia una comunicación de ida y vuelta (llamadas y devoluciones de llamada) hasta que el servidor la detiene.

Las interfaces remotas compartidas

El servidor de inicio de sesión:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;
```

```
public interface RemoteLogin extends Remote {

    RemoteConnection login(String name, RemoteClient client) throws RemoteException;
}
```

El servidor:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteConnection extends Remote {

    void logout() throws RemoteException;

    String passInt(String name, int i) throws RemoteException;
}
```

El cliente:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteClient extends Remote {

    void half(int i) throws RemoteException;
}
```

Las implementaciones

El servidor de inicio de sesión:

```
package callbackServer;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.util.Map;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class LoginServer implements RemoteLogin {

    static Map<String, RemoteClient> clients = new HashMap<>();

    @Override
    public RemoteConnection login(String name, RemoteClient client) {
```

```
        Connection connection = new Connection(name, client);
        clients.put(name, client);
        System.out.println(name + " logged in");
        return connection;
    }

    public static void main(String[] args) {

        try {
            Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            LoginServer server = new LoginServer();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            reg.rebind("LoginServerName", server);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

El servidor:

```
package callbackServer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.server.Unreferenced;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;

public class Connection implements RemoteConnection, Unreferenced {

    RemoteClient client;
    String name;

    public Connection(String name, RemoteClient client) {

        this.client = client;
        this.name = name;
        try {
            UnicastRemoteObject.exportObject(this, Registry.REGISTRY_PORT);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void unreferenced() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void logout() {
```

```
        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String passInt(String recipient, int i) {

        System.out.println("Server received from " + name + ":" + i);
        if (i < 10)
            return String.valueOf(i);
        RemoteClient client = LoginServer.clients.get(recipient);
        try {
            client.half(i);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
        return String.valueOf(i);
    }
}
```

El cliente:

```
package callbackClient;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class Client implements RemoteClient {

    RemoteConnection connection;
    String name, target;

    Client(String name, String target) {

        this.name = name;
        this.target = target;
    }

    public static void main(String[] args) {

        Client client = new Client(args[0], args[1]);
        try {
            Registry reg = LocateRegistry.getRegistry();
            RemoteLogin login = (RemoteLogin) reg.lookup("LoginServerName");
            UnicastRemoteObject.exportObject(client, Integer.parseInt(args[2]));
            client.connection = login.login(client.name, client);
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }
```

```
        if ("Client1".equals(client.name)) {
            try {
                client.connection.passInt(client.target, 120);
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    public void half(int i) throws RemoteException {

        String result = connection.passInt(target, i / 2);
        System.out.println(name + " received: \"" + result + "\"");
    }
}
```

Ejecutando el ejemplo:

1. Ejecutar el servidor de inicio de sesión.
2. Ejecuta un cliente con los argumentos `Client2 Client1 1097` .
3. Ejecutar un cliente con los argumentos `Client1 Client2 1098` .

Las salidas aparecerán en 3 consolas ya que hay 3 JVM. Aquí están agrupados:

Client2 ha iniciado sesión
Client1 ha iniciado sesión
Servidor recibido de Cliente1: 120
Servidor recibido de Cliente2: 60
Servidor recibido de Cliente1: 30
Servidor recibido de Cliente2: 15
Servidor recibido de Cliente1: 7
Cliente1 recibido: "7"
Cliente2 recibido: "15"
Cliente1 recibido: "30"
Cliente2 recibido: "60"

Ejemplo de RMI simple con implementación de cliente y servidor

Este es un ejemplo simple de RMI con cinco clases Java y dos paquetes, *servidor* y *cliente* .

Paquete de servidor

PersonListInterface.java

```
public interface PersonListInterface extends Remote
{
    /**
     * This interface is used by both client and server
     * @return List of Persons
     * @throws RemoteException
```

```
     */
    ArrayList<String> getPersonList() throws RemoteException;
}
```

PersonListImplementation.java

```
public class PersonListImplementation
extends UnicastRemoteObject
implements PersonListInterface
{

    private static final long serialVersionUID = 1L;

    // standard constructor needs to be available
    public PersonListImplementation() throws RemoteException
    {}

    /**
     * Implementation of "PersonListInterface"
     * @throws RemoteException
     */
    @Override
    public ArrayList<String> getPersonList() throws RemoteException
    {
        ArrayList<String> personList = new ArrayList<String>();

        personList.add("Peter Pan");
        personList.add("Pippi Langstrumpf");
        // add your name here :)

        return personList;
    }
}
```

Servidor.java

```
public class Server {

    /**
     * Register servicer to the known public methods
     */
    private static void createServer() {
        try {
            // Register registry with standard port 1099
            LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            System.out.println("Server : Registry created.");

            // Register PersonList to registry
            Naming.rebind("PersonList", new PersonListImplementation());
            System.out.println("Server : PersonList registered");

        } catch (final IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(final String[] args) {
        createServer();
    }
```

```
}
```

Paquete de cliente

PersonListLocal.java

```
public class PersonListLocal {
    private static PersonListLocal instance;
    private PersonListInterface personList;

    /**
     * Create a singleton instance
     */
    private PersonListLocal() {
        try {
            // Lookup to the local running server with port 1099
            final Registry registry = LocateRegistry.getRegistry("localhost",
                    Registry.REGISTRY_PORT);

            // Lookup to the registered "PersonList"
            personList = (PersonListInterface) registry.lookup("PersonList");
        } catch (final RemoteException e) {
            e.printStackTrace();
        } catch (final NotBoundException e) {
            e.printStackTrace();
        }
    }

    public static PersonListLocal getInstance() {
        if (instance == null) {
            instance = new PersonListLocal();
        }

        return instance;
    }

    /**
     * Returns the servers PersonList
     */
    public ArrayList<String> getPersonList() {
        if (instance != null) {
            try {
                return personList.getPersonList();
            } catch (final RemoteException e) {
                e.printStackTrace();
            }
        }

        return new ArrayList<>();
    }
       }
```

PersonTest.java

```
public class PersonTest
{
    public static void main(String[] args)
```

```
    {
        // get (local) PersonList
        ArrayList<String> personList = PersonListLocal.getInstance().getPersonList();

        // print all persons
        for(String person : personList)
        {
            System.out.println(person);
        }
    }
}
```

Prueba tu aplicación

- Iniciar el método principal de Server.java. Salida:

```
Server : Registry created.
Server : PersonList registered
```

- Iniciar el método principal de PersonTest.java. Salida:

```
Peter Pan
Pippi Langstrumpf
```

Lea Invocación de método remoto (RMI) en línea:
https://riptutorial.com/es/java/topic/171/invocacion-de-metodo-remoto--rmi-

Capítulo 106: Iterador e iterable

Introducción

El `java.util.Iterator` es la interfaz estándar de Java SE para el objeto que implementa el patrón de diseño del iterador. La interfaz `java.lang.Iterable` es para objetos que pueden *proporcionar* un iterador.

Observaciones

Es posible iterar sobre una matriz utilizando el bucle `for` -each, aunque las matrices java no implementan Iterable; la iteración se realiza mediante JVM utilizando un índice no accesible en segundo plano.

Examples

Usando Iterable en for loop

Las clases que implementan `Iterable<>` interfaz se pueden utilizar en `for` bucles. En realidad, esto es solo azúcar sintáctica para obtener un iterador del objeto y usarlo para obtener todos los elementos secuencialmente; hace que el código sea más claro, más rápido de escribir, y menos propenso a errores.

```
public class UsingIterable {

    public static void main(String[] args) {
        List<Integer> intList = Arrays.asList(1,2,3,4,5,6,7);

        // List extends Collection, Collection extends Iterable
        Iterable<Integer> iterable = intList;

        // foreach-like loop
        for (Integer i: iterable) {
            System.out.println(i);
        }

        // pre java 5 way of iterating loops
        for(Iterator<Integer> i = iterable.iterator(); i.hasNext(); ) {
            Integer item = i.next();
            System.out.println(item);
        }
    }
}
```

Usando el iterador crudo

Si bien el uso del bucle foreach (o "bucle extendido para") es simple, a veces es beneficioso usar el iterador directamente. Por ejemplo, si desea generar un conjunto de valores separados por

comas, pero no quiere que el último elemento tenga una coma:

```
List<String> yourData = //...
Iterator<String> iterator = yourData.iterator();
while (iterator.hasNext()){
    // next() "moves" the iterator to the next entry and returns it's value.
    String entry = iterator.next();
    System.out.print(entry);
    if (iterator.hasNext()){
        // If the iterator has another element after the current one:
        System.out.print(",");
    }
}
```

Esto es mucho más fácil y claro que tener una variable `isLastEntry` o hacer cálculos con el índice de bucle.

Creando tu propio iterable.

Para crear su propio Iterable como con cualquier interfaz, simplemente implemente los métodos abstractos en la interfaz. Para `Iterable` solo hay uno que se llama `iterator()` . Pero su tipo de retorno `Iterator` es en sí mismo una interfaz con tres métodos abstractos. Puede devolver un iterador asociado con alguna colección o crear su propia implementación personalizada:

```
public static class Alphabet implements Iterable<Character> {

    @Override
    public Iterator<Character> iterator() {
        return new Iterator<Character>() {
            char letter = 'a';

            @Override
            public boolean hasNext() {
                return letter <= 'z';
            }

            @Override
            public Character next() {
                return letter++;
            }

            @Override
            public void remove() {
                throw new UnsupportedOperationException("Doesn't make sense to remove a
letter");
            }
        };
    }
}
```

Usar:

```
public static void main(String[] args) {
    for(char c : new Alphabet()) {
        System.out.println("c = " + c);
```

```
    }
}
```

El nuevo `Iterator` debe venir con un estado que apunta al primer elemento, y cada llamada a la siguiente actualiza su estado para apuntar a la siguiente. El `hasNext()` comprueba si el iterador está al final. Si el iterador estuviera conectado a una colección modificable, entonces se podría implementar el método opcional `remove()` del iterador para eliminar el elemento al que se apunta actualmente desde la colección subyacente.

Eliminar elementos utilizando un iterador

El método `Iterator.remove()` es un método opcional que elimina el elemento devuelto por la llamada anterior a `Iterator.next()` . Por ejemplo, el siguiente código rellena una lista de cadenas y luego elimina todas las cadenas vacías.

```
List<String> names = new ArrayList<>();
names.add("name 1");
names.add("name 2");
names.add("");
names.add("name 3");
names.add("");
System.out.println("Old Size : " + names.size());
Iterator<String> it = names.iterator();
while (it.hasNext()) {
  String el = it.next();
  if (el.equals("")) {
    it.remove();
  }
}
System.out.println("New Size : " + names.size());
```

Salida:

```
Old Size : 5
New Size : 3
```

Tenga en cuenta que el código anterior es la forma segura de eliminar elementos al iterar una colección típica. Si, por el contrario, intenta eliminar elementos de una colección como esta:

```
for (String el: names) {
    if (el.equals("")) {
        names.remove(el); // WRONG!
    }
}
```

una colección típica (como `ArrayList`) que proporciona a *los* iteradores una semántica de iteradores de *error rápido* lanzará una `ConcurrentModificationException` .

El método `remove()` solo puede ser llamado (una vez) después de una llamada `next()` . Si se llama antes de llamar a `next()` o si se llama dos veces después de la llamada `next()` , entonces la llamada `remove()` lanzará una `IllegalStateException` .

La operación de `remove` se describe como una operación *opcional* ; Es decir, no todos los iteradores lo permitirán. Los ejemplos en los que no se admite incluyen iteradores para colecciones inmutables, vistas de solo lectura de colecciones o colecciones de tamaño fijo. Si se llama a `remove()` cuando el iterador no admite la eliminación, lanzará una `UnsupportedOperationException` .

Lea Iterador e iterable en línea: https://riptutorial.com/es/java/topic/172/iterador-e-iterable

Capítulo 107: JavaBean

Introducción

JavaBeans (TM) es un patrón para diseñar API de clase Java que permite que las instancias (beans) se utilicen en diversos contextos y utilicen varias herramientas *sin* escribir explícitamente el código Java. Los patrones consisten en convenciones para definir captadores y definidores para *propiedades* , para definir constructores y para definir API de escucha de eventos.

Sintaxis

- **Reglas de nombre de propiedad de JavaBean**
- Si la propiedad no es un valor booleano, se debe obtener el prefijo del método getter. Por ejemplo, getSize () es un nombre de obtención de JavaBeans válido para una propiedad llamada "tamaño". Tenga en cuenta que no es necesario tener una variable denominada tamaño. El nombre de la propiedad se deduce de los captadores y establecedores, no a través de ninguna variable en su clase. Lo que devuelvas de getSize () depende de ti.
- Si la propiedad es booleana, el prefijo del método getter es get o es. Por ejemplo, getStopped () o isStopped () son nombres válidos de JavaBeans para una propiedad booleana.
- El prefijo del método setter debe ser establecido. Por ejemplo, setSize () es el nombre válido de JavaBean para una propiedad denominada tamaño.
- Para completar el nombre de un método de obtención o establecimiento, cambie la primera letra del nombre de la propiedad a mayúsculas y luego agréguela al prefijo apropiado (obtener, es o establecer).
- Las firmas del método Setter deben marcarse como públicas, con un tipo de retorno nulo y un argumento que represente el tipo de propiedad.
- Las firmas de los métodos de captación deben marcarse como públicas, no aceptar argumentos y tener un tipo de retorno que coincida con el tipo de argumento del método de establecimiento para esa propiedad.
- **Reglas de nombre de escucha de JavaBean**
- Los nombres de los métodos de escucha utilizados para "registrar" una escucha con un origen de evento deben usar el prefijo agregar, seguido del tipo de escucha. Por ejemplo, addActionListener () es un nombre válido para un método que un origen de evento tendrá que permitir que otros se registren para eventos de acción.
- Los nombres de los métodos de escucha utilizados para eliminar ("anular el registro") un oyente deben usar el prefijo eliminar, seguido del tipo de escucha (usando las mismas reglas que el método de agregar registro).
- El tipo de escucha que se agregará o eliminará se debe pasar como el argumento al método.
- Los nombres de los métodos de escucha deben terminar con la palabra "Escucha".

Observaciones

Para que una clase sea Java Bean debe seguir este estándar , en resumen:

- Todas sus propiedades deben ser privadas y solo accesibles a través de captadores y configuradores.
- Debe tener un constructor público sin argumentos.
- Debe implementar la interfaz `java.io.Serializable` .

Examples

Bean Java Básico

```
public class BasicJavaBean implements java.io.Serializable{

    private int value1;
    private String value2;
    private boolean value3;

    public BasicJavaBean(){}

    public void setValue1(int value1){
        this.value1 = value1;
    }

    public int getValue1(){
        return value1;
    }

    public void setValue2(String value2){
        this.value2 = value2;
    }

    public String getValue2(){
        return value2;
    }

    public void setValue3(boolean value3){
        this.value3 = value3;
    }

    public boolean isValue3(){
        return value3;
    }
}
```

Lea JavaBean en línea: https://riptutorial.com/es/java/topic/8157/javabean

Capítulo 108: JAXB

Introducción

JAXB o Java Architecture for XML Binding (JAXB) es un marco de software que permite a los desarrolladores de Java asignar clases de Java a representaciones XML. Esta página presentará a los lectores a JAXB utilizando ejemplos detallados sobre sus funciones proporcionadas principalmente para calcular y desalmar los objetos de Java en formato xml y viceversa.

Sintaxis

- JAXB.marshall (objeto, fileObjOfXML);
- Object obj = JAXB.unmarshall (fileObjOfXML, className);

Parámetros

Parámetro	Detalles
fileObjOfXML	Objeto de `File` de un archivo XML
nombre de la clase	Nombre de una clase con extensión `.class`

Observaciones

Usando la herramienta XJC disponible en el JDK, se puede generar automáticamente el código Java para una estructura xml descrita en un esquema xml (archivo `.xsd`), vea el tema XJC .

Examples

Escribir un archivo XML (ordenando un objeto)

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

Al utilizar la anotación `XMLRootElement` , podemos marcar una clase como elemento raíz de un archivo XML.

```
import java.io.File;
import javax.xml.bind.JAXB;

public class XMLCreator {
    public static void main(String[] args) {
        User user = new User();
        user.setName("Jon Skeet");
        user.setUserID(8884321);

        try {
            JAXB.marshal(user, new File("UserDetails.xml"));
        } catch (Exception e) {
            System.err.println("Exception occurred while writing in XML!");
        } finally {
            System.out.println("XML created");
        }
    }
}
```

`marshal()` se utiliza para escribir el contenido del objeto en un archivo XML. Aquí, `user` objeto de `user` y un nuevo objeto de `File` se pasan como argumentos al `marshal()` .

En una ejecución exitosa, esto crea un archivo XML llamado `UserDetails.xml` en la ruta de clase con el contenido a continuación.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

Lectura de un archivo XML (no riguroso)

Para leer un archivo XML llamado `UserDetails.xml` con el contenido a continuación

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

Necesitamos una clase POJO llamada `User.java` como abajo

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

Aquí hemos creado las variables y el nombre de la clase de acuerdo con los nodos XML. Para

`XmlRootElement` , usamos la anotación `XmlRootElement` en la clase.

```
public class XMLReader {
    public static void main(String[] args) {
        try {
            User user = JAXB.unmarshal(new File("UserDetails.xml"), User.class);
            System.out.println(user.getName());   // prints Jon Skeet
            System.out.println(user.getUserID()); // prints 8884321
        } catch (Exception e) {
            System.err.println("Exception occurred while reading the XML!");
        }
    }
}
```

Aquí se utiliza el método `unmarshal()` para analizar el archivo XML. Toma el nombre del archivo XML y el tipo de clase como dos argumentos. Luego podemos usar los métodos de obtención del objeto para imprimir los datos.

Usando XmlAdapter para generar el formato xml deseado

Cuando el formato XML deseado difiere del modelo de objeto Java, se puede usar una implementación de XmlAdapter para transformar el objeto modelo en un objeto de formato xml y viceversa. Este ejemplo muestra cómo colocar el valor de un campo en un atributo de un elemento con el nombre del campo.

```
public class XmlAdapterExample {

    @XmlAccessorType(XmlAccessType.FIELD)
    public static class NodeValueElement {

        @XmlAttribute(name="attrValue")
        String value;

        public NodeValueElement() {
        }

        public NodeValueElement(String value) {
            super();
            this.value = value;
        }

        public String getValue() {
            return value;
        }

        public void setValue(String value) {
            this.value = value;
        }
    }

    public static class ValueAsAttrXmlAdapter extends XmlAdapter<NodeValueElement, String> {

        @Override
        public NodeValueElement marshal(String v) throws Exception {
            return new NodeValueElement(v);
        }
```

```
        @Override
        public String unmarshal(NodeValueElement v) throws Exception {
            if (v==null) return "";
            return v.getValue();
        }
    }

    @XmlRootElement(name="DataObject")
    @XmlAccessorType(XmlAccessType.FIELD)
    public static class DataObject {

        String elementWithValue;

        @XmlJavaTypeAdapter(value=ValueAsAttrXmlAdapter.class)
        String elementWithAttribute;
    }

    public static void main(String[] args) {
        DataObject data = new DataObject();
        data.elementWithValue="value1";
        data.elementWithAttribute ="value2";

        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        JAXB.marshal(data, baos);

        String xmlString = new String(baos.toByteArray(), StandardCharsets.UTF_8);

        System.out.println(xmlString);
    }
}
```

Configuración automática de asignación de campos / propiedades XML (@XmlAccessorType)

La anotación `@XmlAccessorType` determina si los campos / propiedades se serializarán automáticamente a XML. Tenga en cuenta que las anotaciones de campo y método `@XmlElement` , `@XmlAttribute` o `@XmlTransient` tienen prioridad sobre la configuración predeterminada.

```
public class XmlAccessTypeExample {

@XmlAccessorType(XmlAccessType.FIELD)
static class AccessorExampleField {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.NONE)
static class AccessorExampleNone {
    public String field="value1";

    public String getGetter() {
        return "getter";
```

```
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.PROPERTY)
static class AccessorExampleProperty {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)
static class AccessorExamplePublic {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

public static void main(String[] args) {
    try {
        System.out.println("\nField:");
        JAXB.marshal(new AccessorExampleField(), System.out);
        System.out.println("\nNone:");
        JAXB.marshal(new AccessorExampleNone(), System.out);
        System.out.println("\nProperty:");
        JAXB.marshal(new AccessorExampleProperty(), System.out);
        System.out.println("\nPublic:");
        JAXB.marshal(new AccessorExamplePublic(), System.out);
    } catch (Exception e) {
        System.err.println("Exception occurred while writing in XML!");
    }
}

} // outer class end
```

Salida

```
Field:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExampleField>
    <field>value1</field>
</accessorExampleField>

None:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExampleNone/>

Property:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExampleProperty>
    <getter>getter</getter>
```

```
</accessorExampleProperty>

Public:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<accessorExamplePublic>
    <field>value1</field>
    <getter>getter</getter>
</accessorExamplePublic>
```

Configuración manual de asignación de campos / propiedades XML

Las anotaciones `@XmlElement` , `@XmlAttribute` o `@XmlTransient` y otras en el paquete `javax.xml.bind.annotation` permiten al programador especificar qué campos y propiedades marcados deben ser serializados.

```
@XmlAccessorType(XmlAccessType.NONE) // we want no automatic field/property marshalling
public class ManualXmlElementsExample {

    @XmlElement
    private String field="field value";

    @XmlAttribute
    private String attribute="attr value";

    @XmlAttribute(name="differentAttribute")
    private String oneAttribute="other attr value";

    @XmlElement(name="different name")
    private String oneName="different name value";

    @XmlTransient
    private String transientField = "will not get serialized ever";


    @XmlElement
    public String getModifiedTransientValue() {
        return transientField.replace(" ever", ", unless in a getter");
    }

    public void setModifiedTransientValue(String val) {} // empty on purpose


    public static void main(String[] args) {
        try {
            JAXB.marshal(new ManualXmlElementsExample(), System.out);
        } catch (Exception e) {
            System.err.println("Exception occurred while writing in XML!");
        }
    }
}
```

Especificando una instancia de XmlAdapter para (re) usar datos existentes

A veces deben usarse instancias específicas de datos. La recreación no es deseada y la referenciación de datos `static` tendría un olor de código.

Es posible especificar un `XmlAdapter` ejemplo la `Unmarshaller` debe utilizar, que permite al usuario utilizar `XmlAdapter` s con ningún constructor cero-arg y / o pasar los datos al adaptador.

Ejemplo

Clase de usuario

La siguiente clase contiene un nombre y una imagen de usuario.

```
import java.awt.image.BufferedImage;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlRootElement
public class User {

    private String name;
    private BufferedImage image;

    @XmlAttribute
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @XmlJavaTypeAdapter(value=ImageCacheAdapter.class)
    @XmlAttribute
    public BufferedImage getImage() {
        return image;
    }

    public void setImage(BufferedImage image) {
        this.image = image;
    }

    public User(String name, BufferedImage image) {
        this.name = name;
        this.image = image;
    }

    public User() {
        this("", null);
    }

}
```

Adaptador

Para evitar crear la misma imagen en la memoria dos veces (además de volver a descargar los datos), el adaptador almacena las imágenes en un mapa.

Java SE 7

Para el código Java 7 válido, reemplace el método `getImage` con

```
public BufferedImage getImage(URL url) {
    BufferedImage image = imageCache.get(url);
    if (image == null) {
        try {
            image = ImageIO.read(url);
        } catch (IOException ex) {
            Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null, ex);
            return null;
        }
        imageCache.put(url, image);
        reverseIndex.put(image, url);
    }
    return image;
}
```

```
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.imageio.ImageIO;
import javax.xml.bind.annotation.adapters.XmlAdapter;

public class ImageCacheAdapter extends XmlAdapter<String, BufferedImage> {

    private final Map<URL, BufferedImage> imageCache = new HashMap<>();
    private final Map<BufferedImage, URL> reverseIndex = new HashMap<>();

    public BufferedImage getImage(URL url) {
        // using a single lookup using Java 8 methods
        return imageCache.computeIfAbsent(url, s -> {
            try {
                BufferedImage img = ImageIO.read(s);
                reverseIndex.put(img, s);
                return img;
            } catch (IOException ex) {
                Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null,
ex);
                return null;
            }
        });
    }

    @Override
    public BufferedImage unmarshal(String v) throws Exception {
        return getImage(new URL(v));
    }

    @Override
    public String marshal(BufferedImage v) throws Exception {
        return reverseIndex.get(v).toExternalForm();
    }

}
```

Ejemplos de XML

Los siguientes 2 xmls son para *Jon Skeet* y su homólogo de Earth 2, que se ven exactamente iguales y, por lo tanto, usan el mismo avatar.

```
<?xml version="1.0" encoding="UTF-8"?>

<user name="Jon Skeet"
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=328&amp;d=identicon&amp;r=PG
```

```
<?xml version="1.0" encoding="UTF-8"?>

<user name="Jon Skeet (Earth 2)"
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=328&amp;d=identicon&amp;r=PG
```

Usando el adaptador

```
ImageCacheAdapter adapter = new ImageCacheAdapter();

JAXBContext context = JAXBContext.newInstance(User.class);

Unmarshaller unmarshaller = context.createUnmarshaller();

// specifiy the adapter instance to use for every
// @XmlJavaTypeAdapter(value=ImageCacheAdapter.class)
unmarshaller.setAdapter(ImageCacheAdapter.class, adapter);

User result1 = (User) unmarshaller.unmarshal(Main.class.getResource("user.xml"));

// unmarshal second xml using the same adapter instance
Unmarshaller unmarshaller2 = context.createUnmarshaller();
unmarshaller2.setAdapter(ImageCacheAdapter.class, adapter);
User result2 = (User) unmarshaller2.unmarshal(Main.class.getResource("user2.xml"));

System.out.println(result1.getName());
System.out.println(result2.getName());

// yields true, since image is reused
System.out.println(result1.getImage() == result2.getImage());
```

Vincular un espacio de nombres XML a una clase Java serializable.

Este es un ejemplo de un archivo `package-info.java` que enlaza un espacio de nombres XML a una clase Java serializable. Esto debe colocarse en el mismo paquete que las clases de Java que deben ser serializadas usando el espacio de nombres.

```
/**
 * A package containing serializable classes.
 */
@XmlSchema
(
    xmlns =
```

```
    {
        @XmlNs(prefix = MySerializableClass.NAMESPACE_PREFIX, namespaceURI =
MySerializableClass.NAMESPACE)
    },
    namespace = MySerializableClass.NAMESPACE,
    elementFormDefault = XmlNsForm.QUALIFIED
)
package com.test.jaxb;

import javax.xml.bind.annotation.XmlNs;
import javax.xml.bind.annotation.XmlNsForm;
import javax.xml.bind.annotation.XmlSchema;
```

Usando XmlAdapter para recortar la cadena.

```
package com.example.xml.adapters;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class StringTrimAdapter extends XmlAdapter<String, String> {
    @Override
    public String unmarshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }

    @Override
    public String marshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }
}
```

Y en package-info.java agregue la siguiente declaración.

```
@XmlJavaTypeAdapter(value = com.example.xml.adapters.StringTrimAdapter.class, type =
String.class)
package com.example.xml.jaxb.bindings;// Packge where you intend to apply trimming filter

import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
```

Lea JAXB en línea: https://riptutorial.com/es/java/topic/147/jaxb

Capítulo 109: JAX-WS

Examples

Autenticación básica

La forma de realizar una llamada JAX-WS con autenticación básica es algo obvia.

Aquí hay un ejemplo donde `Service` es la representación de la clase de servicio y `Port` es el puerto de servicio al que desea acceder.

```
Service s = new Service();
Port port = s.getPort();

BindingProvider prov = (BindingProvider)port;
prov.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "myusername");
prov.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "mypassword");

port.call();
```

Lea JAX-WS en línea: https://riptutorial.com/es/java/topic/4105/jax-ws

Capítulo 110: JMX

Introducción

La tecnología JMX proporciona las herramientas para crear soluciones distribuidas, basadas en web, modulares y dinámicas para administrar y monitorear dispositivos, aplicaciones y redes impulsadas por servicios. Por diseño, este estándar es adecuado para adaptar sistemas heredados, implementar nuevas soluciones de administración y monitoreo, y conectarse a las del futuro.

Examples

Ejemplo simple con Platform MBean Server

Digamos que tenemos algún servidor que registra nuevos usuarios y los saluda con algún mensaje. Y queremos monitorear este servidor y cambiar algunos de sus parámetros.

Primero, necesitamos una interfaz con nuestros métodos de monitoreo y control.

```
public interface UserCounterMBean {
    long getSleepTime();

    void setSleepTime(long sleepTime);

    int getUserCount();

    void setUserCount(int userCount);

    String getGreetingString();

    void setGreetingString(String greetingString);

    void stop();
}
```

Y una implementación simple que nos permitirá ver cómo está funcionando y cómo lo afectamos.

```
public class UserCounter implements UserCounterMBean, Runnable {
    private AtomicLong sleepTime = new AtomicLong(10000);
    private AtomicInteger userCount = new AtomicInteger(0);
    private AtomicReference<String> greetingString = new AtomicReference<>("welcome");
    private AtomicBoolean interrupted = new AtomicBoolean(false);

    @Override
    public long getSleepTime() {
        return sleepTime.get();
    }

    @Override
    public void setSleepTime(long sleepTime) {
        this.sleepTime.set(sleepTime);
```

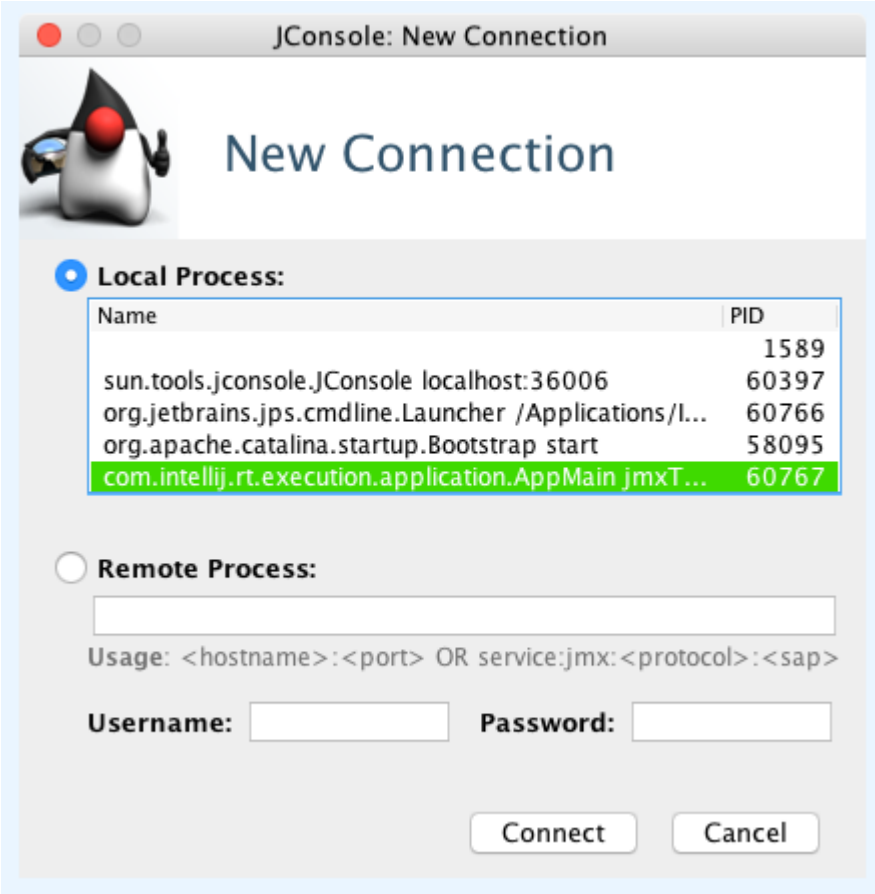
```
    }

    @Override
    public int getUserCount() {
        return userCount.get();
    }

    @Override
    public void setUserCount(int userCount) {
        this.userCount.set(userCount);
    }

    @Override
    public String getGreetingString() {
        return greetingString.get();
    }

    @Override
    public void setGreetingString(String greetingString) {
        this.greetingString.set(greetingString);
    }

    @Override
    public void stop() {
        this.interrupted.set(true);
    }

    @Override
    public void run() {
        while (!interrupted.get()) {
            try {
                System.out.printf("User %d, %s%n", userCount.incrementAndGet(),
greetingString.get());
                Thread.sleep(sleepTime.get());
            } catch (InterruptedException ignored) {
            }
        }
    }
}
```

Para un ejemplo simple con administración local o remota, necesitamos registrar nuestro MBean:

```
import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;

public class Main {
    public static void main(String[] args) throws MalformedObjectNameException,
NotCompliantMBeanException, InstanceAlreadyExistsException, MBeanRegistrationException,
InterruptedException {
        final UserCounter userCounter = new UserCounter();
        final MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
        final ObjectName objectName = new ObjectName("ServerManager:type=UserCounter");
        mBeanServer.registerMBean(userCounter, objectName);

        final Thread thread = new Thread(userCounter);
```

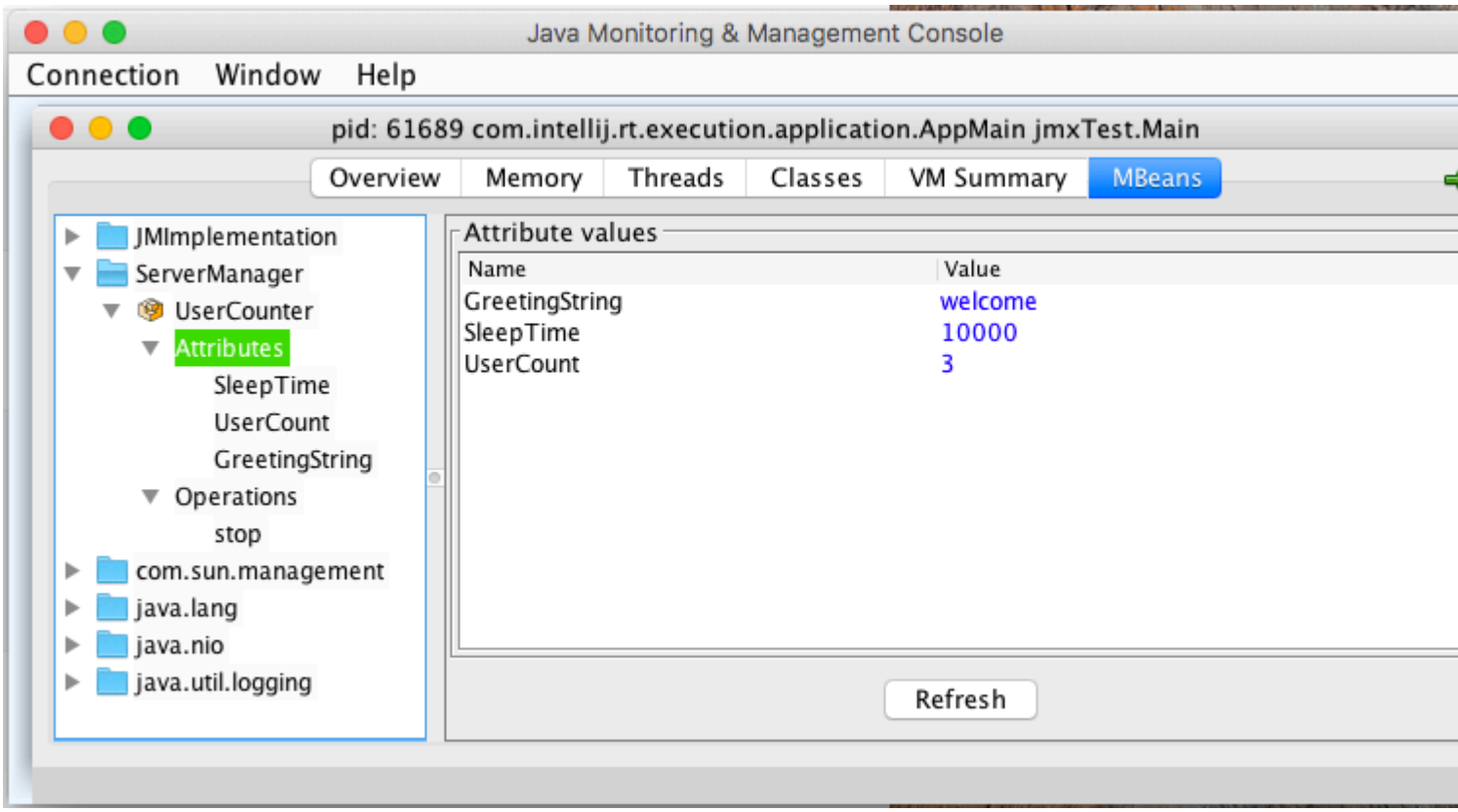
```
        thread.start();
        thread.join();
    }
}
```

Después de eso, podemos ejecutar nuestra aplicación y conectarla a través de jConsole, que se puede encontrar en su directorio `$JAVA_HOME/bin` . Primero, necesitamos encontrar nuestro proceso

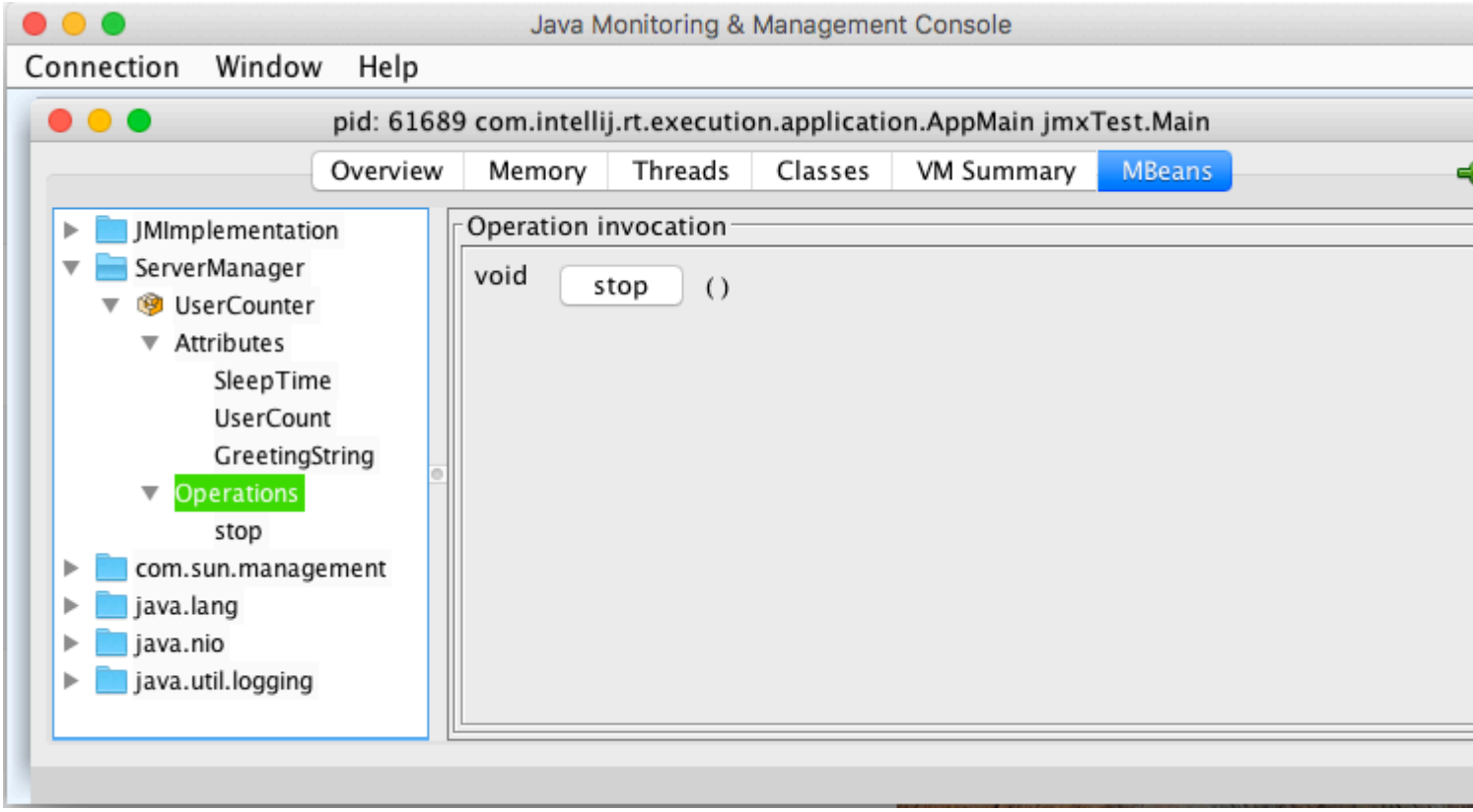


java local con nuestra aplicación

luego cambie a la pestaña MBeans y encuentre el MBean que usamos en nuestra clase Main como `ObjectName` (en el ejemplo anterior es `ServerManager`). En la sección `Attributes` podemos ver los atributos. Si ha especificado solo el método de obtención, el atributo será legible pero no grabable. Si especificara los métodos get y set, el atributo sería legible y grabable.



Los métodos especificados se pueden invocar en `Operations` sección `Operations` .



Si desea poder utilizar la administración remota, necesitará parámetros de JVM adicionales, como:

```
-Dcom.sun.management.jmxremote=true //true by default
-Dcom.sun.management.jmxremote.port=36006
-Dcom.sun.management.jmxremote.authenticate=false
```

```
-Dcom.sun.management.jmxremote.ssl=false
```

Estos parámetros se pueden encontrar en el Capítulo 2 de las guías JMX . Después de eso, podrá conectarse a su aplicación a través de jConsole de forma remota con `jconsole host:port` o con la especificación de `host:port` o `service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi` en jConsole GUI.

Enlaces útiles:

- Guias jmx
- Mejores practicas JMX

Lea JMX en línea: https://riptutorial.com/es/java/topic/9278/jmx

Capítulo 111: JNDI

Examples

RMI a través de JNDI

Este ejemplo muestra cómo funciona JNDI en RMI. Tiene dos roles:

- para proporcionar al servidor una API de vinculación / desvinculación / rebobinado al registro de RMI
- para proporcionar al cliente una API de búsqueda / lista para el registro de RMI.

El registro de RMI es parte de RMI, no JNDI.

Para hacer esto simple, usaremos `java.rmi.registry.CreateRegistry()` para crear el Registro RMI.

1. Server.java (el servidor JNDI)

```
package com.neohope.jndi.test;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.util.Hashtable;

/**
 * JNDI Server
 * 1.create a registry on port 1234
 * 2.bind JNDI
 * 3.wait for connection
 * 4.clean up and end
 */
public class Server {
    private static Registry registry;
    private static InitialContext ctx;

    public static void initJNDI() {
        try {
            registry = LocateRegistry.createRegistry(1234);
            final Hashtable jndiProperties = new Hashtable();
            jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
            jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");
            ctx = new InitialContext(jndiProperties);
        } catch (NamingException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
```

```
    public static void bindJNDI(String name, Object obj) throws NamingException {
        ctx.bind(name, obj);
    }

    public static void unbindJNDI(String name) throws NamingException {
        ctx.unbind(name);
    }

    public static void unInitJNDI() throws NamingException {
        ctx.close();
    }

    public static void main(String[] args) throws NamingException, IOException {
        initJNDI();
        NMessage msg = new NMessage("Just A Message");
        bindJNDI("/neohope/jndi/test01", msg);
        System.in.read();
        unbindJNDI("/neohope/jndi/test01");
        unInitJNDI();
    }
}
```

2. Client.java (el cliente JNDI)

```
package com.neohope.jndi.test;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Hashtable;

/**
 * 1.init context
 * 2.lookup registry for the service
 * 3.use the service
 * 4.end
 */
public class Client {
    public static void main(String[] args) throws NamingException {
        final Hashtable jndiProperties = new Hashtable();
        jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
        jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");

        InitialContext ctx = new InitialContext(jndiProperties);
        NMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
        System.out.println(msg.message);
        ctx.close();
    }
}
```

3. NMessage.java (clase de servidor RMI)

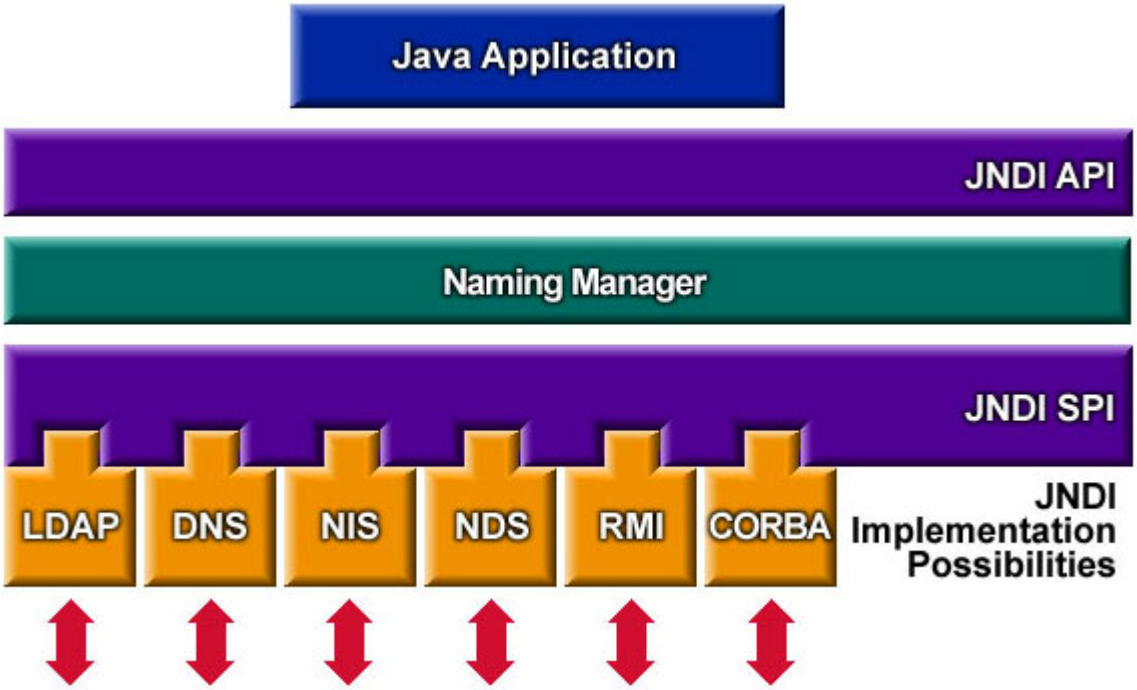
```
package com.neohope.jndi.test;

import java.io.Serializable;
import java.rmi.Remote;
```

```
/**
 * NMessage
 * RMI server class
 * must implements Remote and Serializable
 */
public class NMessage implements Remote, Serializable {
    public String message = "";

    public NMessage(String message)
    {
        this.message = message;
    }
}
```

Cómo ejecutar el eaxmple:

1. construir e iniciar el servidor
2. construir e iniciar el cliente

Introducir



La **interfaz de nombres y directorios de Java (JNDI)** es una API de Java para un servicio de directorio que permite a los clientes de software de Java descubrir y buscar datos y objetos a través de un nombre. Está diseñado para ser independiente de cualquier implementación específica de servicios de nombres o directorios.

La arquitectura JNDI consta de una **API** (interfaz de programación de aplicaciones) y una **SPI** (interfaz de proveedor de servicios). Las aplicaciones Java utilizan esta API para acceder a una variedad de servicios de nombres y directorios. El SPI permite la conexión transparente de una variedad de servicios de nombres y directorios, lo que permite que la aplicación Java que utiliza la API de la tecnología JNDI acceda a sus servicios.

Como puede ver en la imagen de arriba, JNDI admite LDAP, DNS, NIS, NDS, RMI y CORBA. Por

supuesto, puedes ampliarlo.

Cómo funciona

En este ejemplo, el RMI de Java utiliza la API JNDI para buscar objetos en una red. Si desea buscar un objeto, necesita al menos dos datos:

- Donde encontrar el objeto

El registro de RMI administra los enlaces de nombre, le indica dónde encontrar el objeto.

- El nombre del objeto

¿Cuál es el nombre de un objeto? Generalmente es una cadena, también puede ser un objeto que implementa la interfaz de nombre.

Paso a paso

1. Primero necesita un registro, que administre el enlace de nombre. En este ejemplo, usamos `java.rmi.registry.LocateRegistry` .

```
//This will start a registry on localhost, port 1234
registry = LocateRegistry.createRegistry(1234);
```

2. Tanto el cliente como el servidor necesitan un contexto. El servidor utiliza el contexto para enlazar el nombre y el objeto. El cliente usa el contexto para buscar el nombre y obtener el objeto.

```
//We use com.sun.jndi.rmi.registry.RegistryContextFactory as the InitialContextFactory
final Hashtable jndiProperties = new Hashtable();
jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
//the registry usrl is "rmi://localhost:1234"
jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");
InitialContext ctx = new InitialContext(jndiProperties);
```

3. El servidor enlaza el nombre y el objeto.

```
//The jndi name is "/neohope/jndi/test01"
bindJNDI("/neohope/jndi/test01", msg);
```

4. El cliente busca el objeto con el nombre "/ neohope / jndi / test01"

```
//look up the object by name "java:com/neohope/jndi/test01"
NeoMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
```

5. Ahora el cliente puede usar el objeto.

6. Cuando el servidor está terminando, necesita limpiar.

```
ctx.unbind("/neohope/jndi/test01");
ctx.close();
```

Lea JNDI en línea: https://riptutorial.com/es/java/topic/5720/jndi

Capítulo 112: JShell

Introducción

JShell es un REPL interactivo para Java agregado en JDK 9. Permite a los desarrolladores evaluar instantáneamente expresiones, probar clases y experimentar con el lenguaje Java. El acceso temprano para jdk 9 se puede obtener en: http://jdk.java.net/9/

Sintaxis

- $ jshell - Inicia el JShell REPL
- jshell> / <comando> - Ejecuta un comando JShell dado
- jshell> / exit - Salir de JShell
- jshell> / help - Ver una lista de comandos de JShell
- jshell> <java_expression> - Evalúa la expresión Java dada (punto y coma opcional)
- jshell> / vars O / métodos O / tipos: vea una lista de variables, métodos o clases, respectivamente.
- jshell> / open <archivo> - lee un archivo como entrada al shell
- jshell> / edit <identifier> - edita un fragmento en el editor de conjuntos
- jshell> / set editor <comando>: establece el comando que se usará para editar fragmentos de código usando / editar
- jshell> / drop <identifier> - borra un fragmento
- jshell> / reset - Restablece la JVM y borra todos los fragmentos

Observaciones

JShell requiere el Java 9 JDK, que actualmente se puede descargar (marzo de 2017) como instantáneas de acceso temprano desde jdk9.java.net . Si, cuando intenta ejecutar el comando `jshell` , obtiene un error que comienza con `Unable to locate an executable` , asegúrese de que `JAVA_HOME` esté configurado correctamente.

Importaciones por defecto

Los siguientes paquetes se importan automáticamente cuando se inicia JShell:

```
import java.io.*
import java.math.*
import java.net.*
import java.nio.file.*
import java.util.*
import java.util.concurrent.*
import java.util.function.*
import java.util.prefs.*
import java.util.regex.*
import java.util.stream.*
```

Examples

Entrando y saliendo de JShell

A partir de JShell

Antes de intentar iniciar JShell, asegúrese de que la variable de entorno `JAVA_HOME` apunta a una instalación JDK 9. Para iniciar JShell, ejecute el siguiente comando:

```
$ jshell
```

Si todo va bien, debería ver un indicador de `jshell>` .

Saliendo de JShell

Para salir de JShell, ejecute el siguiente comando desde el indicador de JShell:

```
jshell> /exit
```

Expresiones

Dentro de JShell, puede evaluar expresiones de Java, con o sin punto y coma. Estos pueden ir desde expresiones y declaraciones básicas hasta expresiones más complejas:

```
jshell> 4+2
jshell> System.out.printf("I am %d years old.\n", 421)
```

Los bucles y los condicionales también están bien:

```
jshell> for (int i = 0; i<3; i++) {
   ...> System.out.println(i);
   ...> }
```

Es importante tener en cuenta que las **expresiones dentro de los bloques deben tener punto y coma.**

Variables

Puedes declarar variables locales dentro de JShell:

```
jshell> String s = "hi"
jshell> int i = s.length
```

Tenga en cuenta que las variables se pueden redeclar con diferentes tipos; Esto es perfectamente válido en JShell:

```
jshell> String var = "hi"
jshell> int var = 3
```

Para ver una lista de variables, ingrese `/vars` en el indicador de JShell.

Métodos y Clases

Puedes definir métodos y clases dentro de JShell:

```
jshell> void speak() {
   ...> System.out.println("hello");
   ...> }

jshell> class MyClass {
   ...> void doNothing() {}
   ...> }
```

No se requieren modificadores de acceso. Al igual que con otros bloques, se requieren puntos y coma dentro de los cuerpos de los métodos. Tenga en cuenta que, al igual que con las variables, es posible redefinir los métodos y las clases. Para ver una lista de métodos o clases, ingrese `/methods` o `/types` en el indicador de JShell, respectivamente.

Editando Fragmentos

La unidad básica de código utilizada por JShell es el **fragmento** o la **entrada de origen** . Cada vez que declara una variable local o define un método o clase local, crea un fragmento de código cuyo nombre es el identificador de la variable / método / clase. En cualquier momento, puede editar un fragmento de código que haya creado con el comando `/edit` . Por ejemplo, digamos que he creado la clase `Foo` con una sola `bar` método:

```
jshell> class Foo {
   ...> void bar() {
   ...> }
   ...> }
```

Ahora, quiero rellenar el cuerpo de mi método. En lugar de reescribir toda la clase, puedo editarla:

```
jshell> /edit Foo
```

De forma predeterminada, aparecerá un editor de swing con las funciones más básicas posibles. Sin embargo, puedes cambiar el editor que JShell usa:

```
jshell> /set editor emacs
jshell> /set editor vi
jshell> /set editor nano
jshell> /set editor -default
```

Tenga en cuenta que si **la nueva versión del fragmento de código contiene errores de sintaxis, es posible que no se guarde.** Del mismo modo, un fragmento de código solo se crea si la declaración / definición original es sintácticamente correcta; lo siguiente no funciona:

```
jshell> String st = String 3
//error omitted
jshell> /edit st
|  No such snippet: st
```

Sin embargo, los fragmentos de código pueden compilarse y, por lo tanto, ser editables a pesar de ciertos errores de tiempo de compilación, como los tipos que no coinciden, los siguientes trabajos:

```
jshell> int i = "hello"
//error omitted
jshell> /edit i
```

Finalmente, los fragmentos de código se pueden eliminar con el comando `/drop` :

```
jshell> int i = 13
jshell> /drop i
jshell> System.out.println(i)
|  Error:
|  cannot find symbol
|    symbol:   variable i
|  System.out.println(i)
|
```

Para eliminar todos los fragmentos de código, restableciendo así el estado de la JVM, use `\reset` :

```
jshell> int i = 2

jshell> String s = "hi"

jshell> /reset
|  Resetting state.

jshell> i
|  Error:
|  cannot find symbol
|    symbol:   variable i
|  i
|  ^

jshell> s
|  Error:
|  cannot find symbol
|    symbol:   variable s
|  s
|  ^
```

Lea JShell en línea: https://riptutorial.com/es/java/topic/9511/jshell

Capítulo 113: JSON en Java

Introducción

JSON (JavaScript Object Notation) es un formato de intercambio de datos ligero, basado en texto e independiente del idioma que es fácil de leer y escribir para los humanos y las máquinas. JSON puede representar dos tipos estructurados: objetos y matrices. JSON se usa a menudo en aplicaciones Ajax, configuraciones, bases de datos y servicios web RESTful. La API de Java para el procesamiento de JSON proporciona API portátiles para analizar, generar, transformar y consultar JSON.

Observaciones

Este ejemplo se enfoca en analizar y crear JSON en Java usando varias bibliotecas, como la biblioteca de **Google Gson** , Jackson Object Mapper y otras.

Los ejemplos que utilizan otras bibliotecas se pueden encontrar aquí: Cómo analizar JSON en Java

Examples

Codificación de datos como JSON

Si necesita crear un objeto `JSONObject` y colocar datos en él, considere el siguiente ejemplo:

```
// Create a new javax.json.JSONObject instance.
JSONObject first = new JSONObject();

first.put("foo", "bar");
first.put("temperature", 21.5);
first.put("year", 2016);

// Add a second object.
JSONObject second = new JSONObject();
second.put("Hello", "world");
first.put("message", second);

// Create a new JSONArray with some values
JSONArray someMonths = new JSONArray(new String[] { "January", "February" });
someMonths.put("March");
// Add another month as the fifth element, leaving the 4th element unset.
someMonths.put(4, "May");

// Add the array to our object
object.put("months", someMonths);

// Encode
String json = object.toString();

// An exercise for the reader: Add pretty-printing!
```

```
/* {
       "foo":"bar",
       "temperature":21.5,
       "year":2016,
       "message":{"Hello":"world"},
       "months":["January","February","March",null,"May"]
   }
*/
```

Decodificación de datos JSON

Si necesita obtener datos de un `JSONObject` , considere el siguiente ejemplo:

```
String json =
"{\"foo\":\"bar\",\"temperature\":21.5,\"year\":2016,\"message\":{\"Hello\":\"world\"},\"months\":[\"Ja


// Decode the JSON-encoded string
JSONObject object = new JSONObject(json);

// Retrieve some values
String foo = object.getString("foo");
double temperature = object.getDouble("temperature");
int year = object.getInt("year");

// Retrieve another object
JSONObject secondary = object.getJSONObject("message");
String world = secondary.getString("Hello");

// Retrieve an array
JSONArray someMonths = object.getJSONArray("months");
// Get some values from the array
int nMonths = someMonths.length();
String february = someMonths.getString(1);
```

Métodos optXXX vs getXXX

`JSONObject` y `JSONArray` tienen algunos métodos que son muy útiles al tratar con la posibilidad de que un valor que está tratando de obtener no exista o sea de otro tipo.

```
JSONObject obj = new JSONObject();
obj.putString("foo", "bar");

// For existing properties of the correct type, there is no difference
obj.getString("foo");        // returns "bar"
obj.optString("foo");        // returns "bar"
obj.optString("foo", "tux"); // returns "bar"

// However, if a value cannot be coerced to the required type, the behavior differs
obj.getInt("foo");      // throws JSONException
obj.optInt("foo");      // returns 0
obj.optInt("foo", 123); // returns 123

// Same if a property does not exist
obj.getString("undefined");        // throws JSONException
obj.optString("undefined");        // returns ""
```

```
obj.optString("undefined", "tux"); // returns "tux"
```

Las mismas reglas se aplican a los métodos `getXXX` / `optXXX` de `JSONArray` .

Objeto a JSON (Gson Library)

Asumamos que tienes una clase llamada `Person` con solo `name`

```
private class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Código:

```
Gson g = new Gson();

Person person = new Person("John");
System.out.println(g.toJson(person)); // {"name":"John"}
```

Por supuesto, el tarro de Gson debe estar en el classpath.

JSON a Objeto (Biblioteca Gson)

Asumamos que tienes una clase llamada `Person` con solo `name`

```
private class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Código:

```
Gson gson = new Gson();
String json = "{\"name\": \"John\"}";

Person person = gson.fromJson(json, Person.class);
System.out.println(person.name); //John
```

Debes tener gson library en tu classpath.

Extraer un solo elemento de JSON

```
String json = "{\"name\": \"John\", \"age\":21}";
```

```
JsonObject jsonObject = new JsonParser().parse(json).getAsJsonObject();

System.out.println(jsonObject.get("name").getAsString()); //John
System.out.println(jsonObject.get("age").getAsInt()); //21
```

Usando Jackson Object Mapper

Modelo de pojo

```
public class Model {
    private String firstName;
    private String lastName;
    private int age;
    /* Getters and setters not shown for brevity */
}
```

Ejemplo: Cadena a objeto

```
Model outputObject = objectMapper.readValue(
     "{\"firstName\":\"John\",\"lastName\":\"Doe\",\"age\":23}",
     Model.class);
System.out.println(outputObject.getFirstName());
//result: John
```

Ejemplo: Objeto a Cadena

```
String jsonString = objectMapper.writeValueAsString(inputObject));
//result: {"firstName":"John","lastName":"Doe","age":23}
```

Detalles

Declaración de importación necesaria:

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

Dependencia de Maven: Jackson-databind

Instancia de `ObjectMapper`

```
//creating one
ObjectMapper objectMapper = new ObjectMapper();
```

- `ObjectMapper` es seguro para hilos
- recomendado: tener una instancia compartida, estática

Deserialización:

```
<T> T readValue(String content, Class<T> valueType)
```

- `valueType` necesita ser especificado - la devolución será de este tipo
- Arroja
  - `IOException` - en caso de un problema de E / S de bajo nivel
  - `JsonParseException` : si la entrada subyacente contiene contenido no válido
  - `JsonMappingException` : si la estructura JSON de entrada no coincide con la estructura del objeto

Ejemplo de uso (jsonString es la cadena de entrada):

```
Model fromJson = objectMapper.readValue(jsonString, Model.class);
```

Método para la serialización:

String writeValueAsString (valor de objeto)

- Arroja
  - `JsonProcessingException` en caso de error
  - Nota: antes de la versión 2.1, la cláusula throws incluía IOException; 2.1 lo eliminó.

Iteración json

`JSONObject` sobre propiedades `JSONObject`

```
JSONObject obj = new JSONObject("{\"isMarried\":\"true\", \"name\":\"Nikita\",
\"age\":\"30\"}");
Iterator<String> keys = obj.keys();//all keys: isMarried, name & age
while (keys.hasNext()) {                        //as long as there is another key
      String key = keys.next();                 //get next key
      Object value = obj.get(key);              //get next value by key
      System.out.println(key + " : " + value);//print key : value
}
```

`JSONArray` sobre los valores de `JSONArray`

```
JSONArray arr = new JSONArray();        //Initialize an empty array
//push (append) some values in:
arr.put("Stack");
arr.put("Over");
arr.put("Flow");
for (int i = 0; i < arr.length(); i++) {//iterate over all values
    Object value = arr.get(i);          //get value
    System.out.println(value);          //print each value
}
```

JSON Builder - métodos de encadenamiento

Puede utilizar el encadenamiento de métodos mientras trabaja con `JSONObject` y `JSONArray` .

Ejemplo de JSONObject

```
JSONObject obj = new JSONObject();//Initialize an empty JSON object
//Before: {}
obj.put("name","Nikita").put("age","30").put("isMarried","true");
//After: {"name":"Nikita","age":30,"isMarried":true}
```

JSONArray

```
JSONArray arr = new JSONArray();//Initialize an empty array
//Before: []
arr.put("Stack").put("Over").put("Flow");
//After: ["Stack","Over","Flow"]
```

JSONObject.NULL

Si necesita agregar una propiedad con un valor `null` , debe usar el `JSONObject.NULL` final estático predefinido y no la referencia `null` Java estándar.

`JSONObject.NULL` es un valor centinela utilizado para definir explícitamente una propiedad con un valor vacío.

```
JSONObject obj = new JSONObject();
obj.put("some", JSONObject.NULL);   //Creates: {"some":null}
System.out.println(obj.get("some"));//prints: null
```

Nota

```
JSONObject.NULL.equals(null); //returns true
```

Que es una **clara violación** del `Java.equals()` de `Java.equals()` :

> Para cualquier valor de referencia que no sea nulo x, x.equals (nulo) debe devolver falso

JsonArray a la lista de Java (Gson Library)

Aquí hay un JsonArray simple que le gustaría convertir a un `ArrayList` Java:

```
{
    "list": [
                "Test_String_1",
                "Test_String_2"
            ]
}
```

Ahora pase la 'lista' de `JsonArray` al siguiente método que devuelve una `ArrayList` Java correspondiente:

```
public ArrayList<String> getListString(String jsonList){
```

```
    Type listType = new TypeToken<List<String>>() {}.getType();
    //make sure the name 'list' matches the name of 'JsonArray' in your 'Json'.
    ArrayList<String> list = new Gson().fromJson(jsonList, listType);
    return list;
}
```

Debe agregar la siguiente dependencia de `POM.xml` a su archivo `POM.xml` :

```
<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.7</version>
</dependency>
```

O deberías tener el jar `com.google.code.gson:gson:jar:<version>` en tu classpath.

Deserializar la colección JSON a la colección de Objetos usando Jackson

Supongamos que tienes una `Person` clase pojo

```
public class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Y desea analizarlo en una matriz JSON o en un mapa de objetos Person. Debido al borrado de tipos, no puede construir clases de `List<Person>` y `Map<String, Person>` en tiempo de ejecución directamente *(y, por lo tanto, usarlas para deserializar JSON)* . Para superar esta limitación, jackson proporciona dos enfoques: `TypeFactory` y `TypeReference` .

TypeFactory

El enfoque adoptado aquí es utilizar una fábrica (y su función de utilidad estática) para crear su tipo para usted. Los parámetros que toma son la colección que desea utilizar (lista, conjunto, etc.) y la clase que desea almacenar en esa colección.

TypeReference

El enfoque de referencia de tipo parece más sencillo porque le ahorra un poco de escritura y se ve más limpio. TypeReference acepta un parámetro de tipo, donde pasa el tipo de `List<Person>` deseado `List<Person>` . Simplemente crea una instancia de este objeto TypeReference y lo utiliza como su contenedor de tipo.

Ahora veamos cómo deserializar su JSON en un objeto Java. Si su JSON está formateado como una matriz, puede deserializarlo como una Lista. Si hay una estructura anidada más compleja, deseará deserializar a un mapa. Veremos ejemplos de ambos.

Deserialización de matriz JSON

```
String jsonString = "[{\"name\": \"Alice\"}, {\"name\": \"Bob\"}]"
```

Enfoque de TypeFactory

```
CollectionType listType =
    factory.constructCollectionType(List.class, Person.class);
List<Preson> list = mapper.readValue(jsonString, listType);
```

Tipo de enfoque de referencia

```
TypeReference<Person> listType = new TypeReference<List<Person>>() {};
List<Person> list = mapper.readValue(jsonString, listType);
```

Deserialización del mapa JSON

```
String jsonString = "{\"0\": {\"name\": \"Alice\"}, \"1\": {\"name\": \"Bob\"}}"
```

Enfoque de TypeFactory

```
CollectionType mapType =
    factory.constructMapLikeType(Map.class, String.class, Person.class);
List<Person> list = mapper.readValue(jsonString, mapType);
```

Tipo de enfoque de referencia

```
TypeReference<Person> mapType = new TypeReference<Map<String, Person>>() {};
Map<String, Person> list = mapper.readValue(jsonString, mapType);
```

Detalles

Declaración de importación utilizada:

```
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.type.CollectionType;
```

Instancias utilizadas:

```
ObjectMapper mapper = new ObjectMapper();
TypeFactory factory = mapper.getTypeFactory();
```

Nota

Si `TypeReference` enfoque de `TypeReference` puede verse mejor, tiene varios inconvenientes:

1. `TypeReference` debe ser instanciada usando una clase anónima
2. Usted debe proporcionar una explicación genérica.

Si no lo hace, puede provocar la pérdida de un argumento de tipo genérico que llevará a un error de deserialización.

Lea JSON en Java en línea: https://riptutorial.com/es/java/topic/840/json-en-java

Capítulo 114: Just in Time (JIT) compilador

Observaciones

Historia

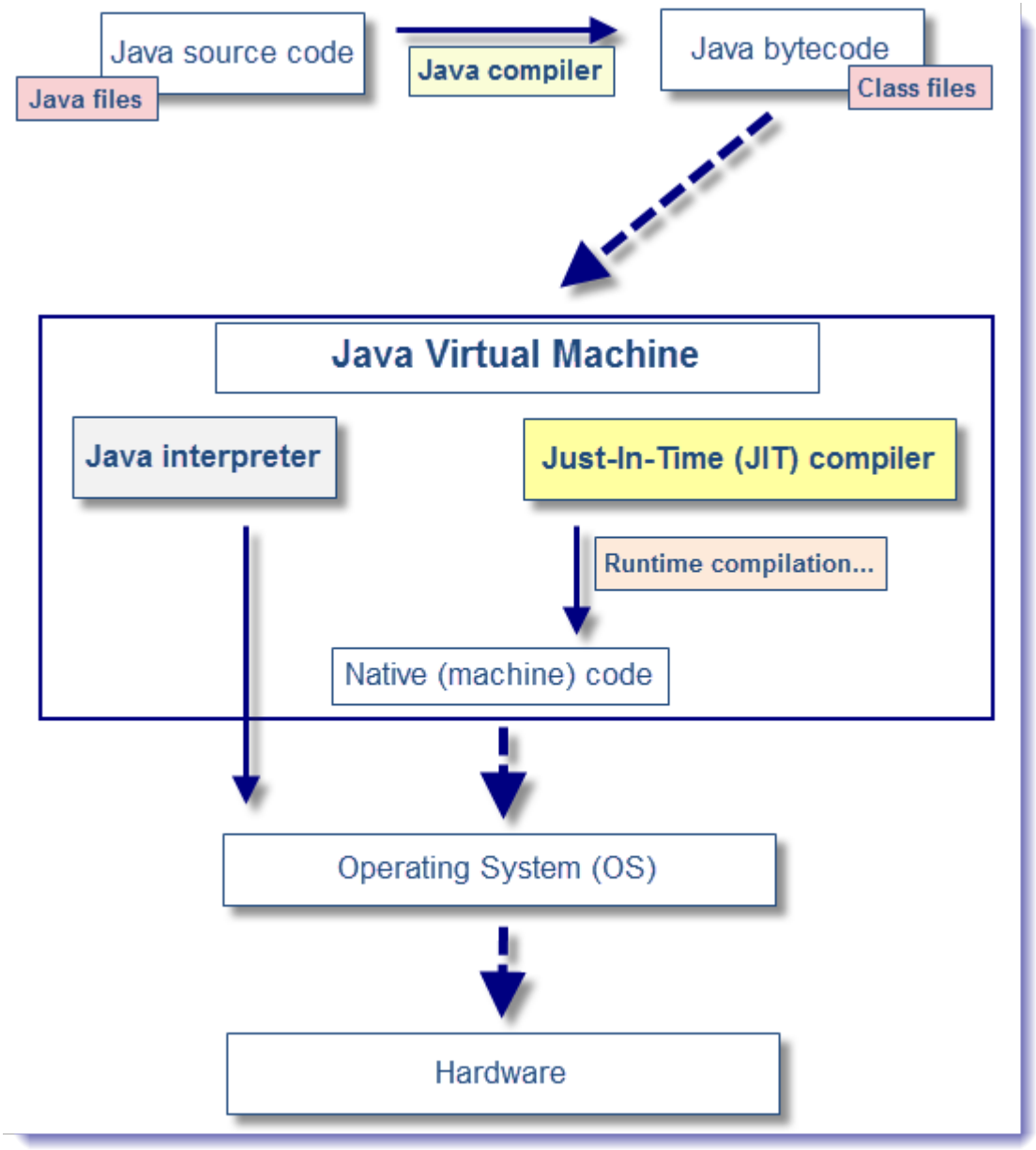
El compilador JIT de Symantec estaba disponible en Sun Java a partir de 1.1.5 en adelante, pero tenía problemas.

El compilador JIT de Hotspot se agregó a Sun Java en 1.2.2 como complemento. En Java 1.3, JIT estaba habilitado por defecto.

(Fuente: ¿ Cuándo Java obtuvo un compilador JIT?

Examples

Visión general



El compilador Just-In-Time (JIT) es un componente de Java ™ Runtime Environment que mejora el rendimiento de las aplicaciones Java en tiempo de ejecución.

- Los programas Java se componen de clases, que contienen códigos de byte independientes de la plataforma que pueden ser interpretados por una JVM en diferentes arquitecturas de computadoras.
- En tiempo de ejecución, la JVM carga los archivos de clase, determina la semántica de cada bytecode individual y realiza el cálculo apropiado.

  El uso adicional del procesador y la memoria durante la interpretación significa que una aplicación Java funciona más lentamente que una aplicación nativa.

  El compilador JIT ayuda a mejorar el rendimiento de los programas Java al compilar códigos de bytes en código de máquina nativo en tiempo de ejecución.

El compilador JIT está habilitado de forma predeterminada y se activa cuando se llama a un método Java. El compilador JIT compila los códigos de bytes de ese método en código de máquina nativo, compilándolos `"just in time"` para ejecutarse.

Cuando se ha compilado un método, la JVM llama al código compilado de ese método directamente en lugar de interpretarlo. Teóricamente, si la compilación no requería tiempo de procesador y uso de memoria, compilar todos los métodos podría permitir que la velocidad del programa Java se acerque a la de una aplicación nativa.

La compilación JIT requiere tiempo de procesador y uso de memoria. Cuando se inicia JVM por primera vez, se llaman miles de métodos. La compilación de todos estos métodos puede afectar significativamente el tiempo de inicio, incluso si el programa logra un rendimiento máximo muy bueno.

- En la práctica, los métodos no se compilan la primera vez que se llaman. Para cada método, la JVM mantiene un `call count` que se incrementa cada vez que se llama al método.
- La JVM interpreta un método hasta que su recuento de llamadas supera un umbral de compilación JIT.
- Por lo tanto, los métodos de uso frecuente se compilan poco después de que la JVM haya comenzado, y los métodos menos utilizados se compilan mucho más tarde, o no se hacen.
- El umbral de compilación JIT ayuda a que la JVM se inicie rápidamente y aún así tenga un rendimiento mejorado.
- El umbral se ha seleccionado cuidadosamente para obtener un equilibrio óptimo entre los tiempos de inicio y el rendimiento a largo plazo.
- Después de compilar un método, su cuenta de llamadas se restablece a cero y las llamadas subsiguientes al método continúan incrementando su cuenta.
- Cuando el recuento de llamadas de un método alcanza un umbral de recompilación JIT, el compilador JIT lo compila por segunda vez, aplicando una selección más grande de optimizaciones que en la compilación anterior.
- Este proceso se repite hasta que se alcanza el nivel máximo de optimización.

  Los métodos más ocupados de un programa Java siempre se optimizan de forma más agresiva, maximizando los beneficios de rendimiento del uso del compilador JIT.

El compilador JIT también puede medir `operational data at run time` , y usar esos datos para mejorar la calidad de las recompilaciones adicionales.

> El compilador JIT se puede desactivar, en cuyo caso se interpretará todo el programa Java. No se recomienda deshabilitar el compilador JIT, excepto para diagnosticar o solucionar los problemas de compilación JIT.

Lea Just in Time (JIT) compilador en línea: https://riptutorial.com/es/java/topic/5152/just-in-time--jit--compilador

Capítulo 115: La clase java.util.Objects

Examples

Uso básico para la comprobación del objeto nulo.

Para el método de verificación nulo

```
Object nullableObject = methodReturnObject();
if (Objects.isNull(nullableObject)) {
    return;
}
```

Para el método de check in no nulo

```
Object nullableObject = methodReturnObject();
if (Objects.nonNull(nullableObject)) {
    return;
}
```

Uso de referencia del método Objects.nonNull () en la API de flujo

De la manera antigua moda para colección null check

```
List<Object> someObjects = methodGetList();
for (Object obj : someObjects) {
    if (obj == null) {
        continue;
    }
    doSomething(obj);
}
```

Con el método `Objects.nonNull` y Java8 Stream API, podemos hacer lo anterior de esta manera:

```
List<Object> someObjects = methodGetList();
someObjects.stream()
            .filter(Objects::nonNull)
            .forEach(this::doSomething);
```

Lea La clase java.util.Objects en línea: https://riptutorial.com/es/java/topic/5768/la-clase-java-util-objects

Capítulo 116: Las trampas de Java - Hilos y concurrencia

Examples

Trampa: uso incorrecto de esperar () / notificar ()

Los métodos `object.wait()` , `object.notify()` y `object.notifyAll()` están destinados a ser utilizados de una manera muy específica. (consulte http://stackoverflow.com/documentation/java/5409/wait-notify#t=20160811161648303307)

El problema de "notificación perdida"

Un error común de principiante es llamar incondicionalmente a `object.wait()`

```
private final Object lock = new Object();

public void myConsumer() {
    synchronized (lock) {
        lock.wait();     // DON'T DO THIS!!
    }
    doSomething();
}
```

La razón por la que esto es incorrecto es que depende de algún otro subproceso para llamar a `lock.notify()` o `lock.notifyAll()` , pero nada garantiza que el otro subproceso no realizó esa llamada *antes* del subproceso del consumidor llamado `lock.wait()` .

`lock.notify()` y `lock.notifyAll()` no hacen nada en absoluto si algún otro hilo no *está* esperando la notificación. El hilo que llama a `myConsumer()` en este ejemplo se bloqueará para siempre si es demasiado tarde para detectar la notificación.

El error "Estado del monitor ilegal"

Si llama a `wait()` o `notify()` en un objeto sin mantener el bloqueo, la JVM lanzará la `IllegalMonitorStateException` .

```
public void myConsumer() {
    lock.wait();       // throws exception
    consume();
}

public void myProducer() {
    produce();
    lock.notify();     // throws exception
}
```

(El diseño para `wait()` / `notify()` requiere que el bloqueo se mantenga porque es necesario para evitar las condiciones de carrera sistémicas. Si fuera posible llamar a `wait()` o `notify()` sin bloqueo, sería imposible implementar el caso de uso principal de estas primitivas: esperar que ocurra una condición.)

Esperar / notificar es de muy bajo nivel

La *mejor* manera de evitar problemas con `wait()` y `notify()` es no usarlos. La mayoría de los problemas de sincronización se pueden resolver utilizando los objetos de sincronización de nivel superior (colas, barreras, semáforos, etc.) que están disponibles en el paquete `java.utils.concurrent` .

Pitfall - Extendiendo 'java.lang.Thread'

El javadoc para la clase `Thread` muestra dos formas de definir y usar un hilo:

Usando una clase de hilo personalizado:

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}

PrimeThread p = new PrimeThread(143);
p.start();
```

Usando un `Runnable` :

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}

PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

(Fuente: `java.lang.Thread` javadoc .)

El enfoque de clase de subproceso personalizado funciona, pero tiene algunos problemas:

1. Es incómodo usar `PrimeThread` en un contexto que usa un grupo de subprocesos clásico, un ejecutor o el marco de ForkJoin. (No es imposible, porque `PrimeThread` implementa indirectamente `Runnable` , pero usar una clase `Thread` personalizada como `Runnable` es ciertamente torpe y puede que no sea viable ... dependiendo de otros aspectos de la clase).

2. Hay más oportunidad de errores en otros métodos. Por ejemplo, si declara un `PrimeThread.start()` sin delegar a `Thread.start()` , terminará con un "hilo" que se ejecutó en el hilo actual.

El enfoque de poner la lógica del hilo en un `Runnable` evita estos problemas. De hecho, si usa una clase anónima (Java 1.1 en adelante) para implementar el `Runnable` el resultado es más sucinto y más legible que los ejemplos anteriores.

```
final long minPrime = ...
new Thread(new Runnable() {
    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}.start();
```

Con una expresión lambda (Java 8 en adelante), el ejemplo anterior sería aún más elegante:

```
final long minPrime = ...
new Thread(() -> {
   // compute primes larger than minPrime
    . . .
}).start();
```

Pitfall - Demasiados hilos hace que una aplicación sea más lenta.

Una gran cantidad de personas que son nuevas en los subprocesos múltiples piensan que el uso de subprocesos hace que una aplicación vaya más rápido. De hecho, es mucho más complicado que eso. Pero una cosa que podemos decir con certeza es que para cualquier computadora hay un límite en el número de subprocesos que se pueden ejecutar al mismo tiempo:

- Una computadora tiene un número fijo de *núcleos* (o *hipervínculos*).
- Se debe *programar* un subproceso de Java a un núcleo o hipervínculo para que se ejecute.
- Si hay más subprocesos Java ejecutables que núcleos / hipervínculos (disponibles), algunos de ellos deben esperar.

Esto nos dice que simplemente crear más y más hilos de Java *no puede* hacer que la aplicación vaya más rápido y más rápido. Pero hay otras consideraciones también:

- Cada subproceso requiere una región de memoria fuera del montón para su pila de subprocesos. El tamaño de pila de hilos típico (predeterminado) es 512Kbytes o 1Mbytes. Si tiene un número significativo de subprocesos, el uso de la memoria puede ser significativo.

- Cada subproceso activo hará referencia a una serie de objetos en el montón. Eso aumenta el conjunto de trabajo de objetos *alcanzables* , lo que afecta la recolección de basura y el

uso de la memoria física.

- La sobrecarga de cambiar entre hilos no es trivial. Por lo general, implica un cambio en el espacio del kernel del sistema operativo para tomar una decisión de programación de subprocesos.
- Los gastos generales de sincronización de subprocesos y señalización entre subprocesos (por ejemplo, esperar (), notificar () / notifyAll) *pueden ser* significativos.

Dependiendo de los detalles de su aplicación, estos factores generalmente significan que hay un "punto dulce" para el número de hilos. Más allá de eso, agregar más hilos proporciona una mejora mínima del rendimiento y puede empeorar el rendimiento.

Si su aplicación se crea para cada nueva tarea, un aumento inesperado en la carga de trabajo (por ejemplo, una alta tasa de solicitud) puede llevar a un comportamiento catastrófico.

Una mejor manera de lidiar con esto es usar un grupo de subprocesos limitados cuyo tamaño puede controlar (estática o dinámicamente). Cuando hay mucho trabajo por hacer, la aplicación necesita poner en cola las solicitudes. Si usa un `ExecutorService` , se encargará de la administración del grupo de subprocesos y la cola de tareas.

Pitfall - La creación de hilos es relativamente cara

Considere estos dos micro-puntos de referencia:

El primer punto de referencia simplemente crea, inicia y une hilos. El `Runnable` del hilo no funciona.

```
public class ThreadTest {
    public static void main(String[] args) throws Exception {
        while (true) {
            long start = System.nanoTime();
            for (int i = 0; i < 100_000; i++) {
                Thread t = new Thread(new Runnable() {
                        public void run() {
                }});
                t.start();
                t.join();
            }
            long end = System.nanoTime();
            System.out.println((end - start) / 100_000.0);
        }
    }
}

$ java ThreadTest
34627.91355
33596.66021
33661.19084
33699.44895
33603.097
33759.3928
33671.5719
33619.46809
33679.92508
```

```
33500.32862
33409.70188
33475.70541
33925.87848
33672.89529
^C
```

En una PC moderna típica que ejecuta Linux con 64bit Java 8 u101, este punto de referencia muestra un tiempo promedio que se tarda en crear, iniciar y unir hilos de entre 33.6 y 33.9 microsegundos.

El segundo punto de referencia hace el equivalente al primero, pero utiliza un `ExecutorService` para enviar tareas y un `Future` para encontrarse con el final de la tarea.

```
import java.util.concurrent.*;

public class ExecutorTest {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        while (true) {
            long start = System.nanoTime();
            for (int i = 0; i < 100_000; i++) {
                Future<?> future = exec.submit(new Runnable() {
                    public void run() {
                    }
                });
                future.get();
            }
            long end = System.nanoTime();
            System.out.println((end - start) / 100_000.0);
        }
    }
}

$ java ExecutorTest
6714.66053
5418.24901
5571.65213
5307.83651
5294.44132
5370.69978
5291.83493
5386.23932
5384.06842
5293.14126
5445.17405
5389.70685
^C
```

Como puedes ver, los promedios están entre 5.3 y 5.6 microsegundos.

Si bien los tiempos reales dependerán de una variedad de factores, la diferencia entre estos dos resultados es significativa. Claramente, es más rápido utilizar un grupo de subprocesos para reciclar subprocesos que crear nuevos subprocesos.

Pitfall: las variables compartidas requieren una sincronización adecuada

Considera este ejemplo:

```
public class ThreadTest implements Runnable {

    private boolean stop = false;

    public void run() {
        long counter = 0;
        while (!stop) {
            counter = counter + 1;
        }
        System.out.println("Counted " + counter);
    }

    public static void main(String[] args) {
        ThreadTest tt = new ThreadTest();
        new Thread(tt).start();    // Create and start child thread
        Thread.sleep(1000);
        tt.stop = true;            // Tell child thread to stop.
    }
}
```

La intención de este programa es iniciar un subproceso, dejar que se ejecute durante 1000 milisegundos y luego hacer que se detenga al establecer el indicador de `stop` .

¿Funcionará según lo previsto?

Tal vez sí tal vez no.

Una aplicación no necesariamente se detiene cuando el método `main` regresa. Si se ha creado otro hilo y ese hilo no se ha marcado como un hilo daemon, entonces la aplicación continuará ejecutándose después de que el hilo principal haya finalizado. En este ejemplo, eso significa que la aplicación continuará ejecutándose hasta que finalice el subproceso secundario. Eso debería suceder cuando `tt.stop` se establece en `true` .

Pero eso en realidad no es estrictamente cierto. De hecho, el subproceso secundario se detendrá después de que haya *observado una* `stop` con el valor `true` . ¿Eso sucederá? Tal vez sí tal vez no.

La especificación del lenguaje Java *garantiza* que las lecturas y escrituras de memoria hechas en un hilo sean visibles para ese hilo, según el orden de las declaraciones en el código fuente. Sin embargo, en general, esto NO está garantizado cuando un hilo escribe y otro hilo (posteriormente) lee. Para obtener una visibilidad garantizada, es necesario que haya una cadena de relaciones de *suceso antes de* una escritura y una lectura posterior. En el ejemplo anterior, no hay tal cadena para la actualización del indicador de `stop` , y por lo tanto, no se garantiza que el subproceso secundario verá el cambio de `stop` en `true` .

(Nota para los autores: Debe haber un tema separado en el Modelo de memoria de Java para profundizar en los detalles técnicos).

¿Cómo solucionamos el problema?

En este caso, hay dos formas sencillas de garantizar que la actualización de `stop` esté visible:

1. Declara `stop` de ser `volatile` ; es decir

   ```
   private volatile boolean stop = false;
   ```

   Para una variable `volatile` , el JLS especifica que hay una relación de *suceso antes* entre una escritura por un hilo y una lectura posterior por un segundo hilo.

2. Use un mutex para sincronizar de la siguiente manera:

```
public class ThreadTest implements Runnable {

    private boolean stop = false;

    public void run() {
        long counter = 0;
        while (true) {
            synchronize (this) {
                if (stop) {
                    break;
                }
            }
            counter = counter + 1;
        }
        System.out.println("Counted " + counter);
    }

    public static void main(String[] args) {
        ThreadTest tt = new ThreadTest();
        new Thread(tt).start();    // Create and start child thread
        Thread.sleep(1000);
        synchronize (tt) {
            tt.stop = true;        // Tell child thread to stop.
        }
    }
}
```

Además de garantizar que haya una exclusión mutua, el JLS especifica que hay una relación de suceso *antes* entre la liberación de un mutex en un hilo y obtener el mismo mutex en un segundo hilo.

¿Pero no es la asignación atómica?

¡Sí lo es!

Sin embargo, ese hecho no significa que los efectos de la actualización serán visibles simultáneamente a todos los subprocesos. Sólo una cadena adecuada de relaciones de *suceso-antes* lo garantizará.

¿Por qué hicieron esto?

Los programadores que realizan programación de subprocesos múltiples en Java por primera vez encuentran que el modelo de memoria es un desafío. Los programas se comportan de una manera no intuitiva porque la expectativa natural es que las escrituras son visibles de manera uniforme. Entonces, ¿por qué los diseñadores de Java diseñan el modelo de memoria de esta manera?

En realidad, se trata de un compromiso entre el rendimiento y la facilidad de uso (para el programador).

Una arquitectura de computadora moderna consiste en múltiples procesadores (núcleos) con conjuntos de registros individuales. La memoria principal es accesible para todos los procesadores o para grupos de procesadores. Otra propiedad del hardware moderno de las computadoras es que el acceso a los registros suele ser un orden de magnitud de acceso más rápido que el acceso a la memoria principal. A medida que aumenta la cantidad de núcleos, es fácil ver que leer y escribir en la memoria principal puede convertirse en el principal cuello de botella en el rendimiento del sistema.

Esta discrepancia se resuelve implementando uno o más niveles de almacenamiento en caché de memoria entre los núcleos del procesador y la memoria principal. Cada núcleo de acceso accede a las celdas a través de su caché. Normalmente, la lectura de la memoria principal solo ocurre cuando hay una falta de caché, y la escritura de la memoria principal solo ocurre cuando se debe vaciar una línea de caché. Para una aplicación donde el conjunto de trabajo de cada núcleo de las ubicaciones de la memoria se ajuste a su caché, la velocidad del núcleo ya no está limitada por la velocidad de la memoria principal / ancho de banda.

Pero eso nos da un nuevo problema cuando varios núcleos están leyendo y escribiendo variables compartidas. La última versión de una variable puede estar en el caché de un núcleo. A menos que ese núcleo descargue la línea de caché en la memoria principal, Y otros núcleos invaliden su copia en caché de versiones anteriores, algunos de ellos pueden ver versiones obsoletas de la variable. Pero si los cachés se vaciaran en la memoria cada vez que hay una escritura de caché ("por si acaso" hubo una lectura por parte de otro núcleo) que consumiría innecesariamente el ancho de banda de la memoria principal.

La solución estándar utilizada en el nivel de conjunto de instrucciones de hardware es proporcionar instrucciones para la invalidación de la memoria caché y la escritura de la memoria caché, y dejar que el compilador decida cuándo usarlas.

Volviendo a Java. El modelo de memoria está diseñado para que los compiladores de Java no tengan que emitir la invalidación de la memoria caché y las instrucciones de escritura directa donde no sean realmente necesarias. El supuesto es que el programador utilizará un mecanismo de sincronización apropiado (por ejemplo, mutexes primitivos, clases de concurrencia `volatile` , de alto nivel, etc.) para indicar que necesita visibilidad de memoria. En ausencia de una relación de *suceso antes* , los compiladores de Java son libres de *asumir* que no se requieren operaciones de caché (o similares).

Esto tiene importantes ventajas de rendimiento para aplicaciones de subprocesos múltiples, pero la desventaja es que escribir aplicaciones de subprocesos múltiples correctas no es una cuestión simple. El programador *tiene* que entender lo que él o ella está haciendo.

¿Por qué no puedo reproducir esto?

Hay varias razones por las que problemas como este son difíciles de reproducir:

1. Como se explicó anteriormente, la consecuencia de no hacer frente a la visibilidad de memoria emite problemas adecuadamente *suele* ser que su aplicación compilada no maneja los cachés de memoria correctamente. Sin embargo, como mencionamos anteriormente, los cachés de memoria a menudo se vacían de todos modos.

2. Cuando cambia la plataforma de hardware, las características de los cachés de memoria pueden cambiar. Esto puede llevar a un comportamiento diferente si su aplicación no se sincroniza correctamente.

3. Es posible que esté observando los efectos de la sincronización *fortuita* . Por ejemplo, si agrega traceprints, normalmente se produce una sincronización entre bambalinas en las secuencias de E / S que causan vacíos de caché. Por lo tanto, agregar traceprints a *menudo* hace que la aplicación se comporte de manera diferente.

4. La ejecución de una aplicación en un depurador hace que el compilador JIT compile de forma diferente. Los puntos de interrupción y el paso único exacerban esto. Estos efectos a menudo cambiarán el comportamiento de una aplicación.

Estas cosas hacen que los errores debidos a una sincronización inadecuada sean particularmente difíciles de resolver.

Lea Las trampas de Java - Hilos y concurrencia en línea:
https://riptutorial.com/es/java/topic/5567/las-trampas-de-java---hilos-y-concurrencia

Capítulo 117: Lectores y escritores

Introducción

Los lectores y escritores y sus respectivas subclases proporcionan I / O simple para datos basados en texto / caracteres.

Examples

BufferedReader

Introducción

La clase `BufferedReader` es una envoltura para otras clases de `Reader` que cumple dos propósitos principales:

1. Un `BufferedReader` proporciona almacenamiento en búfer para el `Reader` envuelto. Esto permite que una aplicación lea los caracteres uno a la vez sin sobrecargas de E / S indebidas.

2. Un `BufferedReader` proporciona funcionalidad para leer texto por línea a la vez.

Conceptos básicos sobre el uso de un BufferedReader

El patrón normal para usar un `BufferedReader` es el siguiente. Primero, obtienes el `Reader` que deseas leer los caracteres. A continuación, crea una instancia de un `BufferedReader` que envuelve el `Reader` . Luego lees los datos de los personajes. Finalmente, cierra `BufferedReader` que cierra el `Reader envuelto. Por ejemplo:

```
File someFile = new File(...);
int aCount = 0;
try (FileReader fr = new FileReader(someFile);
     BufferedReader br = new BufferedReader(fr)) {
    // Count the number of 'a' characters.
    int ch;
    while ((ch = br.read()) != -1) {
        if (ch == 'a') {
            aCount++;
        }
    }
    System.out.println("There are " + aCount + " 'a' characters in " + someFile);
}
```

Puedes aplicar este patrón a cualquier `Reader`

Notas:

1. Utilizamos Java 7 (o posterior) *try-with-resources* para garantizar que el lector subyacente esté siempre cerrado. Esto evita una posible fuga de recursos. En versiones anteriores de Java, cerraría explícitamente el `BufferedReader` en un bloque `finally` .

2. El código dentro del bloque `try` es virtualmente idéntico al que `FileReader` si leemos directamente desde el `FileReader` . De hecho, un `BufferedReader` funciona exactamente igual que el `Reader` que envuelve. La diferencia es que *esta* versión es mucho más eficiente.

El tamaño del búfer BufferedReader

El método BufferedReader.readLine ()

Ejemplo: leer todas las líneas de un archivo en una lista

Esto se hace al obtener cada línea en un archivo y agregarla a una `List<String>` . Luego se devuelve la lista:

```
public List<String> getAllLines(String filename) throws IOException {
    List<String> lines = new ArrayList<String>();
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        String line = null;
        while ((line = reader.readLine) != null) {
            lines.add(line);
        }
    }
    return lines;
}
```

Java 8 proporciona una forma más concisa de hacerlo utilizando el método `lines()` :

```
public List<String> getAllLines(String filename) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        return br.lines().collect(Collectors.toList());
    }
    return Collections.empty();
}
```

Ejemplo de StringWriter

La clase StringWriter de Java es un flujo de caracteres que recopila los resultados del búfer de cadena, que se puede usar para construir una cadena.

La clase StringWriter extiende la clase Writer.

En la clase StringWriter, los recursos del sistema como los sockets de red y los archivos no se utilizan, por lo que no es necesario cerrar el StringWriter.

```
import java.io.*;
public class StringWriterDemo {
    public static void main(String[] args) throws IOException {
        char[] ary = new char[1024];
        StringWriter writer = new StringWriter();
        FileInputStream input = null;
        BufferedReader buffer = null;
        input = new FileInputStream("c://stringwriter.txt");
        buffer = new BufferedReader(new InputStreamReader(input, "UTF-8"));
        int x;
        while ((x = buffer.read(ary)) != -1) {
                   writer.write(ary, 0, x);
        }
        System.out.println(writer.toString());
        writer.close();
        buffer.close();
    }
}
```

El ejemplo anterior nos ayuda a conocer un ejemplo simple de StringWriter que utiliza BufferedReader para leer datos de archivos de la secuencia.

Lea Lectores y escritores en línea: https://riptutorial.com/es/java/topic/10618/lectores-y-escritores

Capítulo 118: LinkedHashMap

Introducción

La clase LinkedHashMap es la implementación de la tabla Hash y la lista vinculada de la interfaz Map, con un orden de iteración predecible. Hereda la clase HashMap e implementa la interfaz Map.

Los puntos importantes sobre la clase LinkedHashMap de Java son: Un LinkedHashMap contiene valores basados en la clave. Sólo contiene elementos únicos. Puede tener una clave nula y varios valores nulos. Es igual que HashMap en cambio mantiene el orden de inserción.

Examples

Clase LinkedHashMap de Java

Puntos clave:-

- Es la implementación de la tabla Hash y la lista enlazada de la interfaz del mapa, con un orden de iteración predecible.
- hereda la clase HashMap e implementa la interfaz Map.
- Contiene valores basados en la clave.
- Sólo elementos únicos.
- puede tener una clave nula y múltiples valores nulos.
- igual que HashMap en cambio mantiene el orden de inserción.

Métodos: -

- vacío claro ().
- booleantainsKey (clave de objeto).
- Objeto obtener (clave de objeto).
- removeEldestEntry booleano protegido (Map.Entry eldest)

Ejemplo: -

```
public static void main(String arg[])
    {
        LinkedHashMap<String, String> lhm = new LinkedHashMap<String, String>();
        lhm.put("Ramesh", "Intermediate");
        lhm.put("Shiva", "B-Tech");
        lhm.put("Santosh", "B-Com");
        lhm.put("Asha", "Msc");
        lhm.put("Raghu", "M-Tech");
```

```
        Set set = lhm.entrySet();
        Iterator i = set.iterator();
        while (i.hasNext()) {
            Map.Entry me = (Map.Entry) i.next();
            System.out.println(me.getKey() + " : " + me.getValue());
        }

        System.out.println("The Key Contains : " + lhm.containsKey("Shiva"));
        System.out.println("The value to the corresponding to key : " + lhm.get("Asha"));
    }
```

Lea LinkedHashMap en línea: https://riptutorial.com/es/java/topic/10750/linkedhashmap

Capítulo 119: Lista vs SET

Introducción

¿Cuáles son las diferencias entre la colección List y Set en el nivel superior y cómo elegir cuándo usar List en java y cuándo usar Set en Java?

Examples

Lista vs conjunto

```
import java.util.ArrayList;
```

import java.util.HashSet; import java.util.List; import java.util.Set;

clase pública SetAndListExample {public static void main (String [] args) {System.out.println ("List list ....."); List list = new ArrayList (); list.add ("1"); list.add ("2"); list.add ("3"); list.add ("4"); list.add ("1");

```
    for (String temp : list){
        System.out.println(temp);
    }

    System.out.println("Set example .....");
    Set<String> set = new HashSet<String>();
    set.add("1");
    set.add("2");
    set.add("3");
    set.add("4");
    set.add("1");
    set.add("2");
    set.add("5");

    for (String temp : set){
        System.out.println(temp);
    }
}
```

}

Ejemplo de lista de salida ..... 1 2 3 4 1 Ejemplo de conjunto 3 2 10 5 4

Lea Lista vs SET en línea: https://riptutorial.com/es/java/topic/10125/lista-vs-set

Capítulo 120: Literales

Introducción

Un literal de Java es un elemento sintáctico (es decir, algo que se encuentra en el *código fuente* de un programa Java) que representa un valor. Los ejemplos son `1` , `0.333F` , `false` , `'X'` y `"Hello world\n"` .

Examples

Literales hexadecimales, octales y binarios.

Un número `hexadecimal` es un valor en base-16. Hay 16 dígitos, `0-9` y las letras `AF` (el caso no importa). `AF` representan `10-16` .

Un número `octal` es un valor en base-8, y usa los dígitos `0-7` .

Un número `binary` es un valor en base-2, y usa los dígitos `0` y `1` .

Todos estos números resultan en el mismo valor, `110` :

```
int dec = 110;            // no prefix   --> decimal literal
int bin = 0b1101110;      // '0b' prefix --> binary literal
int oct = 0156;           // '0' prefix  --> octal literal
int hex = 0x6E;           // '0x' prefix --> hexadecimal literal
```

Tenga en cuenta que la sintaxis literal binaria se introdujo en Java 7.

El literal octal puede ser fácilmente una trampa para los errores semánticos. Si define un `'0'` inicial en sus literales decimales, obtendrá el valor incorrecto:

```
int a = 0100;        // Instead of 100, a == 64
```

Usando el guión bajo para mejorar la legibilidad

Desde Java 7 ha sido posible usar uno o más guiones bajos (_) para separar grupos de dígitos en un número literal primitivo para mejorar su legibilidad.

Por ejemplo, estas dos declaraciones son equivalentes:

Java SE 7

```
int i1 = 123456;
int i2 = 123_456;
System.out.println(i1 == i2); // true
```

Esto se puede aplicar a todos los literales de números primitivos como se muestra a continuación:

Java SE 7

```
byte color = 1_2_3;
short yearsAnnoDomini= 2_016;
int socialSecurtyNumber = 999_99_9999;
long creditCardNumber = 1234_5678_9012_3456L;
float piFourDecimals =  3.14_15F;
double piTenDecimals =  3.14_15_92_65_35;
```

Esto también funciona usando prefijos para bases binarias, octales y hexadecimales:

Java SE 7

```
short binary= 0b0_1_0_1;
int octal = 07_7_7_7_7_7_7_7_0;
long hexBytes = 0xFF_EC_DE_5E;
```

Hay algunas reglas sobre los guiones bajos que **prohíben** su colocación en los siguientes lugares:

- Al principio o al final de un número (por ejemplo, `_123` o `123_` *no* son válidos)
- Adyacente a un punto decimal en un literal de punto flotante (por ejemplo, `1._23` o `1_.23` *no* son válidos)
- Antes de un sufijo F o L (por ejemplo, `1.23_F` o `9999999_L` *no* son válidos)
- En posiciones donde se espera una cadena de dígitos (por ejemplo, `0_xFFFF` *no* es válido)

Secuencias de escape en literales.

Los literales de cadenas y caracteres proporcionan un mecanismo de escape que permite expresar códigos de caracteres que de otro modo no se permitirían en el literal. Una secuencia de escape consiste en un carácter de barra invertida (`\`) seguido de uno o más caracteres adicionales. Las mismas secuencias son válidas tanto en caracteres como en literales de cadena.

El conjunto completo de secuencias de escape es el siguiente:

Secuencia de escape	Sentido
`\\`	Denota un carácter de barra invertida (`\`)
`\'`	Indica un carácter de comilla simple (`'`)
`\"`	Indica un carácter de comillas dobles (`"`)
`\n`	Indica un carácter de salto de línea (`LF`)
`\r`	Denota un carácter de retorno de carro (`CR`)
`\t`	Indica un carácter de tabulación horizontal (`HT`)
`\f`	Indica un carácter de alimentación de formulario (`FF`)
`\b`	Denota un carácter de retroceso (`BS`)

Secuencia de escape	Sentido
`\<octal>`	Indica un código de carácter en el rango de 0 a 255.

El `<octal>` de lo anterior consta de uno, dos o tres dígitos octales ('0' a '7') que representan un número entre 0 y 255 (decimal).

Tenga en cuenta que una barra invertida seguida de cualquier otro carácter es una secuencia de escape no válida. Las secuencias de escape no válidas se tratan como errores de compilación por el JLS.

Referencia:

- JLS 3.10.6. Secuencias de Escape para Literales de Caracteres y Cuerdas

Unicode se escapa

Además de las secuencias de escape de caracteres y cadenas descritas anteriormente, Java tiene un mecanismo de escape Unicode más general, como se define en JLS 3.3. Unicode Escapes . Un escape Unicode tiene la siguiente sintaxis:

```
'\' 'u' <hex-digit> <hex-digit> <hex-digit> <hex-digit>
```

donde `<hex-digit>` es uno de `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, `'9'`, `'a'`, `'b'`, `'c'`, `'d'`, `'e'`, `'f'`, `'A'`, `'B'`, `'C'`, `'D'`, `'E'`, `'F'`.

El compilador de Java asigna un escape de Unicode a un carácter (estrictamente hablando, una *unidad de código* Unicode de 16 bits), y se puede utilizar en cualquier parte del código fuente donde el carácter asignado sea válido. Se usa comúnmente en literales de caracteres y cadenas cuando necesita representar un carácter no ASCII en un literal.

Escapar en expresiones regulares

TBD

Decimales enteros literales

Los literales enteros proporcionan valores que se pueden usar donde se necesita una instancia de `byte`, `short`, `int`, `long` o `char`. (Este ejemplo se centra en las formas decimales simples. Otros ejemplos explican cómo los literales en octal, hexadecimal y binario, y el uso de guiones bajos para mejorar la legibilidad).

Literales enteros ordinarios

La forma más simple y común de literal entero es un literal entero decimal. Por ejemplo:

```
0       // The decimal number zero       (type 'int')
```

```
1       // The decimal number one        (type 'int')
42      // The decimal number forty two  (type 'int')
```

Tienes que tener cuidado con los ceros iniciales. Un cero inicial hace que un literal entero se interprete como *octal* no decimal.

```
077    // This literal actually means 7 x 8 + 7 ... or 63 decimal!
```

Los literales enteros están sin firmar. Si ve algo como `-10` o `+10` , estas son en realidad *expresiones que* usan los operadores unarios `-` y unarios `+` .

El rango de literales enteros de esta forma tiene un tipo intrínseco de `int` , y debe estar en el rango de cero a 2^{31} o 2,147,483,648.

Tenga en cuenta que 2^{31} es `1` mayor que `Integer.MAX_VALUE` . Los literales desde 0 hasta `2147483647` se pueden utilizar en cualquier lugar, pero es un error de compilación de usar `2147483648` sin precedentes unario `-` operador. (En otras palabras, está reservado para expresar el valor de `Integer.MIN_VALUE`).

```
int max = 2147483647;     // OK
int min = -2147483648;    // OK
int tooBig = 2147483648;  // ERROR
```

Literales enteros largos

Los literales de tipo `long` se expresan agregando un sufijo `L` Por ejemplo:

```
0L          // The decimal number zero       (type 'long')
1L          // The decimal number one        (type 'long')
2147483648L // The value of Integer.MAX_VALUE + 1

long big = 2147483648;    // ERROR
long big2 = 2147483648L;  // OK
```

Tenga en cuenta que la distinción entre literales `int` y `long` es significativa en otros lugares. Por ejemplo

```
int i = 2147483647;
long l = i + 1;           // Produces a negative value because the operation is
                          // performed using 32 bit arithmetic, and the
                          // addition overflows
long l2 = i + 1L;         // Produces the (intuitively) correct value.
```

Referencia: JLS 3.10.1 - Literales enteros

Literales booleanos

Los literales booleanos son los más simples de los literales en el lenguaje de programación Java.

Los dos valores `boolean` posibles están representados por los literales `true` y `false` . Estos son sensibles a las mayúsculas y minúsculas. Por ejemplo:

```
boolean flag = true;    // using the 'true' literal
flag = false;           // using the 'false' literal
```

Literales de cuerda

Los literales de cadena proporcionan la forma más conveniente de representar valores de cadena en el código fuente de Java. Un literal de cadena consiste en:

- Un carácter de comillas dobles de apertura (`"`).
- Cero o más caracteres que no sean ni comillas dobles ni caracteres de salto de línea. (Un carácter de barra diagonal inversa (`\`) altera el significado de los caracteres subsiguientes; consulte Secuencias de escape en literales).
- Un personaje de cierre de doble cita.

Por ejemplo:

```
"Hello world"   // A literal denoting an 11 character String
""              // A literal denoting an empty (zero length) String
"\""            // A literal denoting a String consisting of one
                //     double quote character
"1\t2\t3\n"     // Another literal with escape sequences
```

Tenga en cuenta que un único literal de cadena no puede abarcar varias líneas de código fuente. Es un error de compilación para que se produzca un salto de línea (o el final del archivo de origen) antes de la comilla doble de cierre de un literal. Por ejemplo:

```
"Jello world    // Compilation error (at the end of the line!)
```

Cuerdas largas

Si necesita una cadena que sea demasiado larga para que quepa en una línea, la forma convencional de expresarla es dividirla en varios literales y usar el operador de concatenación (`+`) para unir las piezas. Por ejemplo

```
String typingPractice = "The quick brown fox " +
                        "jumped over " +
                        "the lazy dog"
```

Una expresión como la anterior que consta de cadenas literales y `+` satisface los requisitos para ser una expresión constante . Eso significa que la expresión será evaluada por el compilador y representada en tiempo de ejecución por un solo objeto `String` .

Interning de literales de cuerda.

Cuando la JVM carga el archivo de clase que contiene literales de cadena, el sistema de tiempo de ejecución *introduce internamente los* objetos de `String` correspondientes. Esto significa que un literal de cadena utilizado en varias clases no ocupa más espacio que si se usara en una clase.

Para obtener más información sobre interning y el conjunto de cadenas, consulte el ejemplo de String pool y heap storage en el tema Strings.

El literal nulo

El literal nulo (escrito como `null`) representa el único valor del tipo nulo. Aquí hay unos ejemplos

```
MyClass object = null;
MyClass[] objects = new MyClass[]{new MyClass(), null, new MyClass()};

myMethod(null);

if (objects != null) {
     // Do something
}
```

El tipo nulo es bastante inusual. No tiene nombre, por lo que no puede expresarlo en el código fuente de Java. (Y tampoco tiene representación en tiempo de ejecución.)

El único propósito del tipo nulo es ser el tipo de `null` . Es compatible con la asignación de todos los tipos de referencia, y se puede convertir a cualquier tipo de referencia. (En este último caso, la conversión no implica una verificación de tipo de tiempo de ejecución).

Finalmente, `null` tiene la propiedad de que `null instanceof <SomeReferenceType>` evaluará como `false` , sin importar cuál sea el tipo.

Literales de punto flotante

Los literales de punto flotante proporcionan valores que se pueden usar cuando se necesita una instancia `float` o `double` . Hay tres tipos de literal de punto flotante.

- Formas decimales simples
- Formas decimales escaladas
- Formas hexadecimales

(Las reglas de sintaxis de JLS combinan las dos formas decimales en una sola forma. Las tratamos por separado para facilitar la explicación).

Hay tipos literales distintos para literales `float` y `double` , expresados con sufijos. Las diversas formas usan letras para expresar diferentes cosas. Estas letras son mayúsculas y minúsculas.

Formas decimales simples

La forma más simple de literal de punto flotante consiste en uno o más dígitos decimales y un punto decimal (`.`) Y un sufijo opcional (`f` , `F` , `d` o `D`). El sufijo opcional le permite especificar que

el literal es un valor `float` (`f` o `F`) o `double` (`d` o `D`). El valor predeterminado (cuando no se especifica ningún sufijo) es `double` .

Por ejemplo

```
0.0     // this denotes zero
.0      // this also denotes zero
0.      // this also denotes zero
3.14159 // this denotes Pi, accurate to (approximately!) 5 decimal places.
1.0F    // a `float` literal
1.0D    // a `double` literal.  (`double` is the default if no suffix is given)
```

De hecho, los dígitos decimales seguidos de un sufijo también son un literal de coma flotante.

```
1F      // means the same thing as 1.0F
```

El significado de un literal decimal es el número de punto flotante de IEEE *más cercano* al número real matemático de precisión infinita indicado por la forma de punto flotante decimal. Este valor conceptual se convierte a la representación de punto flotante binario IEEE utilizando la *ronda al más cercano* . (La semántica precisa de la conversión decimal se especifica en los javadocs para `Double.valueOf(String)` y `Float.valueOf(String)` , teniendo en cuenta que existen diferencias en las sintaxis numéricas).

Formas decimales escaladas

Las formas decimales escaladas consisten en un decimal simple con una parte de exponente introducida por una `E` o `e` , y seguidas de un entero con signo. La parte del exponente es una mano corta para multiplicar la forma decimal por una potencia de diez, como se muestra en los ejemplos a continuación. También hay un sufijo opcional para distinguir los literales `float` y `double` . Aquí hay unos ejemplos:

```
1.0E1    // this means 1.0 x 10^1 ... or 10.0 (double)
1E-1D    // this means 1.0 x 10^(-1) ... or 0.1 (double)
1.0e10f  // this means 1.0 x 10^(10) ... or 10000000000.0 (float)
```

El tamaño de un literal está limitado por la representación (`float` o `double`). Es un error de compilación si el factor de escala da como resultado un valor demasiado grande o demasiado pequeño.

Formas hexadecimales

A partir de Java 6, es posible expresar literales de punto flotante en hexadecimal. La forma hexadecimal tiene una sintaxis análoga a las formas decimales simples y escaladas con las siguientes diferencias:

1. Cada hexadecimal comienza literales de punto flotante con un cero (`0`) y luego un `x` o `X` .
2. Los dígitos del número (pero *no* la parte del exponente!) También incluyen los dígitos hexadecimales `a` medio de `f` y sus equivalentes en mayúsculas.

3. El exponente es *obligatorio* y se introduce mediante la letra `p` (o `P`) en lugar de una `e` o una `E` El exponente representa un factor de escala que es una potencia de 2 en lugar de una potencia de 10.

Aquí hay unos ejemplos:

```
0x0.0p0f    // this is zero expressed in hexadecimal form (`float`)
0xff.0p19   // this is 255.0 x 2^19 (`double`)
```

Consejo: dado que las formas hexadecimales de punto flotante no son familiares para la mayoría de los programadores de Java, es aconsejable usarlas con moderación.

Guiones bajos

Comenzando con Java 7, se permiten guiones bajos dentro de las cadenas de dígitos en las tres formas de literal de coma flotante. Esto se aplica también a las partes del "exponente". Consulte Uso de guiones bajos para mejorar la legibilidad .

Casos especiales

Es un error de compilación si un literal de punto flotante denota un número que es demasiado grande o demasiado pequeño para representarlo en la representación seleccionada; es decir, si el número se desbordaría a + INF o -INF, o subbordaría a 0.0. Sin embargo, es legal que un literal represente un número desnormalizado distinto de cero.

La sintaxis literal de punto flotante no proporciona representaciones literales para los valores especiales IEEE 754, como los valores INF y NaN. Si necesita expresarlos en código fuente, la forma recomendada es usar las constantes definidas por `java.lang.Float` y `java.lang.Double` ; por ejemplo, `Float.NaN` , `Float.NEGATIVE_INFINITY` y `Float.POSITIVE_INFINITY` .

Literales de personajes

Literales de caracteres proporcionan la forma más conveniente de expresar `char` valores en el código fuente de Java. Un carácter literal consiste en:

- Un carácter de comillas simples (`'`) de apertura.
- Una representación de un personaje. Esta representación no puede ser una comilla simple o un carácter de salto de línea, pero puede ser una secuencia de escape introducida por un carácter de barra invertida (`\`); Ver secuencias de escape en literales .
- Un carácter de comillas simples (`'`) de cierre.

Por ejemplo:

```
char a = 'a';
char doubleQuote = '"';
char singleQuote = '\'';
```

Un salto de línea en un literal de carácter es un error de compilación:

```
char newline = '
// Compilation error in previous line
char newLine = '\n';  // Correct
```

Lea Literales en línea: https://riptutorial.com/es/java/topic/8250/literales

Capítulo 121: Liza

Introducción

Una *lista* es una colección *ordenada* de valores. En Java, las listas son parte de Java Collections Framework . Las listas implementan la interfaz `java.util.List` , que extiende `java.util.Collection` .

Sintaxis

- ls.add (elemento E); // Agrega un elemento
- ls.remove (elemento E); // Elimina un elemento
- para (elemento E: ls) {} // itera sobre cada elemento
- ls.toArray (nueva cadena [ls.length]); // Convierte una lista de cadenas a una matriz de cadenas
- ls.get (índice int); // Devuelve el elemento en el índice especificado.
- ls.set (índice int, elemento E); // Reemplaza el elemento en una posición especificada.
- ls.isEmpty (); // Devuelve verdadero si la matriz no contiene elementos, de lo contrario, devuelve falso.
- ls.indexOf (Objeto o); // Devuelve el índice de la primera ubicación del elemento especificado o, o, si no está presente, devuelve -1.
- ls.lastIndexOf (Objeto o); // Devuelve el índice de la última ubicación del elemento especificado o, o, si no está presente, devuelve -1.
- ls.size (); // Devuelve el número de elementos en la Lista.

Observaciones

Una lista es un objeto que almacena una colección ordenada de valores. "Ordenado" significa que los valores se almacenan en un orden particular: un elemento aparece primero, otro ocupa el segundo lugar, etc. Los valores individuales son comúnmente llamados "elementos". Las listas de Java suelen proporcionar estas características:

- Las listas pueden contener cero o más elementos.
- Las listas pueden contener valores duplicados. En otras palabras, un elemento se puede insertar en una lista más de una vez.
- Las listas almacenan sus elementos en un orden particular, lo que significa que un elemento viene primero, uno viene a continuación, y así sucesivamente.
- Cada elemento tiene un *índice que* indica su posición dentro de la lista. El primer elemento tiene índice 0, el siguiente tiene índice 1 y así sucesivamente.
- Las listas permiten la inserción de elementos al principio, al final o en cualquier índice dentro de la lista.
- Comprobar si una lista contiene un valor particular generalmente significa examinar cada elemento de la lista. Esto significa que el tiempo para realizar esta comprobación es O (n) , proporcional al tamaño de la lista.

Agregar un valor a una lista en algún punto que no sea el final moverá todos los siguientes elementos "hacia abajo" o "hacia la derecha". En otras palabras, agregar un elemento en el índice *n* mueve el elemento que solía estar en el índice *n* al índice *n + 1* , y así sucesivamente. Por ejemplo:

```
List<String> list = new ArrayList<>();
list.add("world");
System.out.println(list.indexOf("world"));      // Prints "0"
// Inserting a new value at index 0 moves "world" to index 1
list.add(0, "Hello");
System.out.println(list.indexOf("world"));      // Prints "1"
System.out.println(list.indexOf("Hello"));      // Prints "0"
```

Examples

Ordenar una lista genérica

La clase `Collections` ofrece dos métodos estáticos estándar para ordenar una lista:

- `sort(List<T> list)` aplicable a listas donde `T extends Comparable<? super T>` , y
- `sort(List<T> list, Comparator<? super T> c)` aplicable a listas de cualquier tipo.

Aplicar lo anterior requiere modificar la clase de elementos de lista que se están ordenando, lo que no siempre es posible. También puede ser indeseable ya que, si bien proporciona la clasificación predeterminada, es posible que se requieran otras órdenes de clasificación en diferentes circunstancias, o la clasificación es una tarea única.

Considere que tenemos la tarea de ordenar los objetos que son instancias de la siguiente clase:

```
public class User {
    public final Long id;
    public final String username;

    public User(Long id, String username) {
        this.id = id;
        this.username = username;
    }

    @Override
    public String toString() {
        return String.format("%s:%d", username, id);
    }
}
```

Para utilizar `Collections.sort(List<User> list)` necesitamos modificar la clase de `User` para implementar la interfaz `Comparable` . Por ejemplo

```
public class User implements Comparable<User> {
    public final Long id;
    public final String username;

    public User(Long id, String username) {
```

```
        this.id = id;
        this.username = username;
    }

    @Override
    public String toString() {
        return String.format("%s:%d", username, id);
    }

    @Override
    /** The natural ordering for 'User' objects is by the 'id' field. */
    public int compareTo(User o) {
        return id.compareTo(o.id);
    }
}
```

(Aparte: muchas clases Java estándar como `String` , `Long` , `Integer` implementan la interfaz `Comparable` . Esto hace que las listas de esos elementos se puedan ordenar de forma predeterminada y simplifica la implementación de `compare` o `compareTo` en otras clases).

Con la modificación anterior, podemos ordenar fácilmente una lista de objetos de `User` según el *orden natural de* las clases. (En este caso, hemos definido que se ordenen según los valores de `id`). Por ejemplo:

```
List<User> users = Lists.newArrayList(
    new User(33L, "A"),
    new User(25L, "B"),
    new User(28L, ""));
Collections.sort(users);

System.out.print(users);
// [B:25, C:28, A:33]
```

Sin embargo, supongamos que queremos ordenar los objetos de `User` por `name` lugar de por `id` . Alternativamente, supongamos que no pudimos cambiar la clase para que sea implementable como `Comparable` .

Aquí es donde el método de `sort` con el argumento del `Comparator` es útil:

```
Collections.sort(users, new Comparator<User>() {
    @Override
    /* Order two 'User' objects based on their names. */
    public int compare(User left, User right) {
        return left.username.compareTo(right.username);
    }
});
System.out.print(users);
// [A:33, B:25, C:28]
```

Java SE 8

En Java 8 puedes usar un *lambda en* lugar de una clase anónima. Este último se reduce a una sola línea:

```
Collections.sort(users, (l, r) -> l.username.compareTo(r.username));
```

Además, Java 8 agrega un método de `sort` predeterminado en la interfaz de `List` , lo que simplifica la clasificación aún más.

```
users.sort((l, r) -> l.username.compareTo(r.username))
```

Creando una lista

Dando a tu lista un tipo

Para crear una lista necesita un tipo (cualquier clase, por ejemplo, `String`). Este es el tipo de su `List` . La `List` solo almacenará objetos del tipo especificado. Por ejemplo:

```
List<String> strings;
```

Puede almacenar `"string1"` , `"hello world!"` , `"goodbye"` , etc., pero no puede almacenar `9.2` , sin embargo:

```
List<Double> doubles;
```

Puede almacenar `9.2` , pero no `"hello world!"` .

Inicializando tu lista

Si intentas agregar algo a las listas anteriores, obtendrás una NullPointerException, ¡porque las `strings` y los `doubles` **nulos** iguales!

Hay dos formas de inicializar una lista:

Opción 1: usar una clase que implemente lista

`List` es una interfaz, lo que significa que no tiene un constructor, sino métodos que una clase debe reemplazar. `ArrayList` es el más comúnmente utilizado `List` , aunque `LinkedList` también es común. Así que inicializamos nuestra lista de esta manera:

```
List<String> strings = new ArrayList<String>();
```

o

```
List<String> strings = new LinkedList<String>();
```

Java SE 7

A partir de Java SE 7, puede utilizar un *operador de diamante* :

```
List<String> strings = new ArrayList<>();
```

o

```
List<String> strings = new LinkedList<>();
```

Opción 2: usar la clase Colecciones

La clase `Collections` proporciona dos métodos útiles para crear listas sin una variable de `List` :

- `emptyList()` : devuelve una lista vacía.
- `singletonList(T)` : crea una lista de tipo T y agrega el elemento especificado.

Y un método que utiliza una `List` existente para completar los datos:

- `addAll(L, T...)` : agrega todos los elementos especificados a la lista pasada como primer parámetro.

Ejemplos:

```
import java.util.List;
import java.util.Collections;

List<Integer> l = Collections.emptyList();
List<Integer> l1 = Collections.singletonList(42);
Collections.addAll(l1, 1, 2, 3);
```

Operaciones de acceso posicional

La API de lista tiene ocho métodos para las operaciones de acceso posicional:

- `add(T type)`
- `add(int index, T type)`
- `remove(Object o)`
- `remove(int index)`
- `get(int index)`
- `set(int index, E element)`
- `int indexOf(Object o)`
- `int lastIndexOf(Object o)`

Entonces, si tenemos una lista:

```
List<String> strings = new ArrayList<String>();
```

Y queríamos añadir las cuerdas "¡Hola mundo!" y "adiós mundo!" Para ello, lo haríamos como tal:

```
strings.add("Hello world!");
strings.add("Goodbye world!");
```

Y nuestra lista contendría los dos elementos. Ahora digamos que queríamos agregar "¡Programa empezando!" al **frente** de la lista. Haríamos esto así:

```
strings.add(0, "Program starting!");
```

NOTA: El primer elemento es 0.

Ahora, si quisiéramos eliminar el "¡Adiós mundo!" Línea, podríamos hacerlo así:

```
strings.remove("Goodbye world!");
```

Y si quisiéramos eliminar la primera línea (que en este caso sería "¡Programa empezando!", Podríamos hacerlo así:

```
strings.remove(0);
```

Nota:

1. Agregar y eliminar elementos de la lista modifica la lista, y esto puede llevar a una `ConcurrentModificationException` si la lista se está iterando simultáneamente.

2. La adición y eliminación de elementos puede ser `O(1)` u `O(N)` según la clase de lista, el método utilizado y si está agregando / eliminando un elemento al inicio, al final o en medio de la lista.

Para recuperar un elemento de la lista en una posición específica, puede utilizar el `E get(int index);` Método de la lista API. Por ejemplo:

```
strings.get(0);
```

Volverá el primer elemento de la lista.

Puede reemplazar cualquier elemento en una posición específica usando el `set(int index, E element);` . Por ejemplo:

```
strings.set(0,"This is a replacement");
```

Esto establecerá la Cadena "Esto es un reemplazo" como el primer elemento de la lista.

Nota: El método de configuración sobrescribirá el elemento en la posición 0. No agregará la nueva Cadena en la posición 0 y empujará la antigua a la posición 1.

El `int indexOf(Object o);` devuelve la posición de la primera aparición del objeto pasado como argumento. Si no hay apariciones del objeto en la lista, se devuelve el valor -1. Continuando con el ejemplo anterior si llama:

```
strings.indexOf("This is a replacement")
```

se espera que se devuelva el 0 cuando configuramos la Cadena "Esto es un reemplazo" en la posición 0 de nuestra lista. En caso de que haya más de una aparición en la lista cuando `int indexOf(Object o);` Se llama entonces como se mencionó se devolverá el índice de la primera aparición. Al llamar a `int lastIndexOf(Object o)` , puede recuperar el índice de la última aparición en la lista. Así que si agregamos otro "Este es un reemplazo":

```
strings.add("This is a replacement");
strings.lastIndexOf("This is a replacement");
```

Esta vez se devolverá el 1 y no el 0;

Iterando sobre elementos en una lista

Para el ejemplo, digamos que tenemos una Lista de tipo String que contiene cuatro elementos: "hola", "cómo", "son", "usted?"

La mejor manera de iterar sobre cada elemento es usando un bucle para cada uno:

```
public void printEachElement(List<String> list){
    for(String s : list){
        System.out.println(s);
    }
}
```

Que imprimiría:

```
hello,
how
are
you?
```

Para imprimirlos todos en la misma línea, puede usar un StringBuilder:

```
public void printAsLine(List<String> list){
    StringBuilder builder = new StringBuilder();
    for(String s : list){
        builder.append(s);
    }
    System.out.println(builder.toString());
}
```

Se imprimirá:

```
hello, how are you?
```

Alternativamente, puede usar la indexación de elementos (como se describe en Acceso a un elemento en el índice de ArrayList) para iterar una lista. Advertencia: este enfoque es ineficiente para las listas enlazadas.

Eliminar elementos de la lista B que están presentes en la lista A

Supongamos que tiene 2 listas A y B, y desea eliminar de **B** todos los elementos que tiene en **A,** el método en este caso es

```
 List.removeAll(Collection c);
```

#Ejemplo:

```
public static void main(String[] args) {
    List<Integer> numbersA = new ArrayList<>();
    List<Integer> numbersB = new ArrayList<>();
    numbersA.addAll(Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 }));
    numbersB.addAll(Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 }));
    System.out.println("A: " + numbersA);
    System.out.println("B: " + numbersB);

    numbersB.removeAll(numbersA);
    System.out.println("B cleared: " + numbersB);
    }
```

esto imprimirá

> A: [1, 3, 4, 7, 5, 2]
>
> B: [13, 32, 533, 3, 4, 2]
>
> B despejado: [13, 32, 533]

Encontrando elementos comunes entre 2 listas.

Supongamos que tiene dos listas: A y B, y necesita encontrar los elementos que existen en ambas listas.

Puede hacerlo simplemente invocando el método `List.retainAll()` .

Ejemplo:

```
public static void main(String[] args) {
    List<Integer> numbersA = new ArrayList<>();
    List<Integer> numbersB = new ArrayList<>();
    numbersA.addAll(Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 }));
    numbersB.addAll(Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 }));

    System.out.println("A: " + numbersA);
    System.out.println("B: " + numbersB);
    List<Integer> numbersC = new ArrayList<>();
    numbersC.addAll(numbersA);
    numbersC.retainAll(numbersB);

    System.out.println("List A : " + numbersA);
    System.out.println("List B : " + numbersB);
    System.out.println("Common elements between A and B: " + numbersC);

}
```

Convertir una lista de enteros en una lista de cadenas

```
List<Integer> nums = Arrays.asList(1, 2, 3);
List<String> strings = nums.stream()
    .map(Object::toString)
```

```
    .collect(Collectors.toList());
```

Es decir:

1. Crear un flujo de la lista
2. Mapea cada elemento usando `Object::toString`
3. Recopile los valores de `String` en una `List` usando `Collectors.toList()`

Creación, adición y eliminación de elementos de un ArrayList

`ArrayList` es una de las estructuras de datos incorporadas en Java. Es una matriz dinámica (donde el tamaño de la estructura de datos no necesita declararse primero) para almacenar elementos (Objetos).

Extiende la clase `AbstractList` e implementa la interfaz de `List` . Un `ArrayList` puede contener elementos duplicados donde mantiene el orden de inserción. Se debe tener en cuenta que la clase `ArrayList` no está sincronizada, por lo que se debe tener cuidado al manejar la concurrencia con `ArrayList` . `ArrayList` permite el acceso aleatorio porque la matriz funciona en la base del índice. La manipulación es lenta en `ArrayList` debido a los cambios que ocurren a menudo cuando se elimina un elemento de la lista de arreglos.

Un `ArrayList` se puede crear de la siguiente manera:

```
List<T> myArrayList = new ArrayList<>();
```

Donde `T` (Genéricos) es el tipo que se almacenará dentro de `ArrayList` .

El tipo de `ArrayList` puede ser cualquier Objeto. El tipo no puede ser un tipo primitivo (en su lugar, use sus *clases de envoltorio*).

Para agregar un elemento al `ArrayList` , use el método `add()` :

```
myArrayList.add(element);
```

O para agregar un artículo a un determinado índice:

```
myArrayList.add(index, element); //index of the element should be an int (starting from 0)
```

Para eliminar un elemento de `ArrayList` , use el método `remove()` :

```
myArrayList.remove(element);
```

O bien, para eliminar un elemento de un determinado índice:

```
myArrayList.remove(index); //index of the element should be an int (starting from 0)
```

Sustitución in situ de un elemento de lista

Este ejemplo se trata de reemplazar un elemento de la `List` mientras se asegura que el elemento de reemplazo esté en la misma posición que el elemento que se reemplaza.

Esto se puede hacer usando estos métodos:

- conjunto (índice int, tipo T)
- int indexOf (tipo T)

Considere una `ArrayList` contenga los elementos "¡Programa empezando!", "¡Hola mundo!" y "adiós mundo!"

```
List<String> strings = new ArrayList<String>();
strings.add("Program starting!");
strings.add("Hello world!");
strings.add("Goodbye world!");
```

Si conocemos el índice del elemento que queremos reemplazar, simplemente podemos usar `set` siguiente manera:

```
strings.set(1, "Hi world");
```

Si no conocemos el índice, podemos buscarlo primero. Por ejemplo:

```
int pos = strings.indexOf("Goodbye world!");
if (pos >= 0) {
    strings.set(pos, "Goodbye cruel world!");
}
```

Notas:

1. La operación `set` no generará una `ConcurrentModificationException` .
2. La operación `set` es rápida (`O(1)`) para `ArrayList` pero lenta (`O(N)`) para una lista `LinkedList` .
3. Una búsqueda `indexOf` en un `ArrayList` o `LinkedList` es lenta (`O(N)`).

Haciendo una lista no modificable

La clase Colecciones proporciona una manera de hacer que una lista no sea modificable:

```
List<String> ls = new ArrayList<String>();
List<String> unmodifiableList = Collections.unmodifiableList(ls);
```

Si desea una lista no modificable con un elemento, puede utilizar:

```
List<String> unmodifiableList = Collections.singletonList("Only string in the list");
```

Mover objetos alrededor de la lista

La clase Colecciones le permite mover objetos en la lista usando varios métodos (ls es la Lista):

Invertir una lista:

```
Collections.reverse(ls);
```

Posiciones rotativas de elementos en una lista.

El método de rotación requiere un argumento entero. Este es el número de puntos para moverlo a lo largo de la línea. Un ejemplo de esto es a continuación:

```
List<String> ls = new ArrayList<String>();
ls.add(" how");
ls.add(" are");
ls.add(" you?");
ls.add("hello,");
Collections.rotate(ls, 1);

for(String line : ls) System.out.print(line);
System.out.println();
```

Esto imprimirá "hola, ¿cómo estás?"

Mezclando elementos alrededor de una lista

Usando la misma lista anterior, podemos barajar los elementos en una lista:

```
Collections.shuffle(ls);
```

También podemos darle un objeto java.util.Random que utiliza para colocar objetos aleatoriamente en puntos:

```
Random random = new Random(12);
Collections.shuffle(ls, random);
```

Clases implementando Lista - Pros y Contras

La interfaz de `List` está implementada por diferentes clases. Cada uno de ellos tiene su propio camino para implementarlo con diferentes estrategias y proporcionar diferentes pros y contras.

Clases implementando la lista

Estas son todas las clases `public` en Java SE 8 que implementan la interfaz `java.util.List` :

1. **Clases abstractas:**
    - Lista abstracta
    - AbstractSequentialList
2. **Clases de concreto:**
    - Lista de arreglo
    - Lista de atributos

- CopyOnWriteArrayList
- Lista enlazada
- Lista de roles
- RoleUnresolvedList
- Apilar
- Vector

Pros y contras de cada implementación en términos de complejidad de tiempo

Lista de arreglo

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

ArrayList es una implementación de matriz de tamaño variable de la interfaz de lista. Al almacenar la lista en una matriz, *ArrayList* proporciona métodos (además de los métodos que implementan la interfaz de la *Lista*) para manipular el tamaño de la matriz.

Inicializar ArrayList de entero con tamaño 100

```
List<Integer> myList = new ArrayList<Integer>(100); // Constructs an empty list with the
specified initial capacity.
```

- PROS:

Las operaciones size, isEmpty, ***get*** , ***set*** , iterator y listIterator se ejecutan en tiempo constante. Por lo tanto, obtener y configurar cada elemento de la Lista tiene el mismo *costo de tiempo* :

```
int e1 = myList.get(0);  //   \
int e2 = myList.get(10); //    | => All the same constant cost => O(1)
myList.set(2,10);        //   /
```

- CONTRAS:

La implementación con una matriz (estructura estática) que agrega elementos sobre el tamaño de la matriz tiene un gran costo debido al hecho de que es necesario realizar una nueva asignación para toda la matriz. Sin embargo, a partir de la documentación :

> La operación de adición se ejecuta en tiempo constante amortizado, es decir, agregar n elementos requiere tiempo O (n)

La eliminación de un elemento requiere O (n) tiempo.

Lista de atributos

En venir

CopyOnWriteArrayList

En venir

Lista enlazada

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

LinkedList se implementa mediante una lista doblemente enlazada, una estructura de datos vinculados que consiste en un conjunto de registros vinculados secuencialmente llamados nodos.

Iitialize LinkedList of Integer

```
List<Integer> myList = new LinkedList<Integer>(); // Constructs an empty list.
```

- PROS:

Agregar o eliminar un elemento al principio de la lista o al final tiene un tiempo constante.

```
myList.add(10);  // \
myList.add(0,2); //  | => constant time => O(1)
myList.remove(); // /
```

- CONTRA: De la documentación :

> Las operaciones que indizan en la lista recorrerán la lista desde el principio o el final, lo que esté más cerca del índice especificado.

Operaciones como:

```
myList.get(10);    // \
myList.add(11,25); //  | => worst case done in O(n/2)
myList.set(15,35); // /
```

Lista de roles

En venir

RoleUnresolvedList

En venir

Apilar

En venir

Vector

En venir

Lea Liza en línea: https://riptutorial.com/es/java/topic/2989/liza

Capítulo 122: Localización e internacionalización.

Observaciones

Java viene con un mecanismo poderoso y flexible para localizar sus aplicaciones, pero también es fácil de usar mal y terminar con un programa que ignora o maneja la configuración regional del usuario, y por lo tanto cómo esperan que se comporte su programa.

Sus usuarios esperarán ver datos localizados en los formatos a los que están acostumbrados, y tratar de admitir esto manualmente es una tarea estúpida. Este es solo un pequeño ejemplo de las diferentes formas en que los usuarios esperan ver el contenido que usted podría asumir que se muestra "siempre" de cierta manera:

	fechas	Números	Moneda local	Moneda extranjera	Distancias
Brasil					
China					
Egipto					
Méjico	20/3/16	1.234.56	$ 1,000.50	1,000.50 USD	
Reino Unido	20/3/16	1,234.56	£ 1,000.50		100 km
Estados Unidos	20/20/16	1,234.56	$ 1,000.50	1,000.50 MXN	60 mi

Recursos Generales

- Wikipedia: Internacionalización y localización.

Recursos de Java

- Tutorial de Java: Internacionalización
- Oracle: Internacionalización: Entendiendo la configuración regional en la plataforma Java
- JavaDoc: `Locale`

Examples

Fechas formateadas automáticamente usando "locale"

`SimpleDateFormatter` es genial en caso de apuro, pero como su nombre indica, no se escala bien.

Si codificas `"MM/dd/yyyy"` toda tu aplicación, tus usuarios internacionales no estarán contentos.

Deja que Java haga el trabajo por ti.

Utilice los métodos `static` en `DateFormat` para recuperar el formato correcto para su usuario. Para una aplicación de escritorio (donde confiará en la configuración regional predeterminada), simplemente llame:

```
String localizedDate = DateFormat.getDateInstance(style).format(date);
```

Donde `style` es una de las constantes de formato (`FULL` , `LONG` , `MEDIUM` , `SHORT` , etc.) especificadas en `DateFormat` .

Para una aplicación del lado del servidor donde el usuario especifica su configuración regional como parte de la solicitud, debe pasarla explícitamente a `getDateInstance()` lugar:

```
String localizedDate =
    DateFormat.getDateInstance(style, request.getLocale()).format(date);
```

Comparación de cuerdas

Compara dos cuerdas ignorando el caso:

```
"School".equalsIgnoreCase("school"); // true
```

No usar

```
text1.toLowerCase().equals(text2.toLowerCase());
```

Los idiomas tienen diferentes reglas para convertir mayúsculas y minúsculas. Un 'I' se convertiría a 'i' en inglés. Pero en turco un 'yo' se convierte en un 'ı'. Si tiene que usar `toLowerCase()` use la sobrecarga que espera una `Locale` : `String.toLowerCase(Locale)` .

Comparando dos cadenas ignorando diferencias menores:

```
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.PRIMARY);
collator.equals("Gärten", "gaerten"); // returns true
```

Ordene las cadenas respetando el orden del lenguaje natural, ignorando mayúsculas y minúsculas (use la clave de intercalación para:

```
String[] texts = new String[] {"Birne", "äther", "Apfel"};
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.SECONDARY); // ignore case
Arrays.sort(texts, collator::compare); // will return {"Apfel", "äther", "Birne"}
```

Lugar

La clase `java.util.Locale` se utiliza para representar una región "geográfica, política o cultural" para localizar un texto, número, fecha u operación dado. Por lo tanto, un objeto Locale puede contener un país, región, idioma y también una variante de un idioma, por ejemplo, un dialecto hablado en una determinada región de un país, o hablado en un país diferente al país desde el cual se origina el idioma.

La instancia de Locale se entrega a los componentes que necesitan localizar sus acciones, ya sea para convertir la entrada, la salida o solo para las operaciones internas. La clase Locale no puede hacer ninguna internacionalización o localización por sí misma.

Idioma

El idioma debe ser un código de idioma ISO 639 de 2 o 3 caracteres, o una subetiqueta de idioma registrado de hasta 8 caracteres. En caso de que un idioma tenga un código de idioma de 2 y 3 caracteres, use el código de 2 caracteres. Puede encontrar una lista completa de códigos de idiomas en el Registro de subetiquetas de idiomas de la IANA.

Los códigos de idioma no distinguen entre mayúsculas y minúsculas, pero la clase Locale siempre usa versiones en minúsculas de los códigos de idioma

Creando una Localidad

La creación de una instancia de `java.util.Locale` se puede hacer de cuatro maneras diferentes:

```
Locale constants
Locale constructors
Locale.Builder class
Locale.forLanguageTag factory method
```

Java ResourceBundle

Usted crea una instancia de ResourceBundle como esta:

```
Locale locale = new Locale("en", "US");
ResourceBundle labels = ResourceBundle.getBundle("i18n.properties");
System.out.println(labels.getString("message"));
```

Considere que tengo un archivo de propiedades `i18n.properties` :

```
message=This is locale
```

Salida:

```
This is locale
```

Configuración regional

Si desea reproducir el estado utilizando otros idiomas, puede usar el método `setDefault()` . Su uso:

```
setDefault(Locale.JAPANESE); //Set Japanese
```

Lea Localización e internacionalización. en línea:
https://riptutorial.com/es/java/topic/4086/localizacion-e-internacionalizacion-

Capítulo 123: log4j / log4j2

Introducción

Apache Log4j es una utilidad de registro basada en Java, es uno de varios marcos de trabajo de registro de Java. Este tema muestra cómo configurar y configurar Log4j en Java con ejemplos detallados de todos sus posibles aspectos de uso.

Sintaxis

- Logger.debug ("texto para registrar"); // Registro de información de depuración
- Logger.info ("texto para registrar"); // Registro de información común
- Logger.error ("texto para registrar"); // Información de error de registro
- Logger.warn ("texto para registrar"); // Advertencias de registro
- Logger.trace ("texto para registrar"); // Registro de información de rastreo
- Logger.fatal ("texto para registrar"); // Registro de errores fatales
- Uso de Log4j2 con registro de parámetros:
- Logger.debug ("Debug params {} {} {}", param1, param2, param3); // Registro de depuración con parámetros
- Logger.info ("Información params {} {} {}", param1, param2, param3); // Registro de información con parámetros.
- Logger.error ("Parámetros de error {} {} {}", param1, param2, param3); // Error de registro con parámetros
- Logger.warn ("Advertir params {} {} {}", param1, param2, param3); // Registro de advertencias con parámetros.
- Logger.trace ("Trace params {} {} {}", param1, param2, param3); // Registro de seguimiento con parámetros
- Logger.fatal ("Fatal params {} {} {}", param1, param2, param3); // Registro fatal con parámetros
- Logger.error ("Excepción capturada:", ex); // Excepción de registro con message y stacktrace (se agregará automáticamente)

Observaciones

Final de vida para Log4j 1 alcanzado

> El 5 de agosto de 2015, el Comité de gestión de proyectos de servicios de explotación forestal anunció que Log4j 1.x había llegado al final de su vida útil. Para ver el texto completo del anuncio, consulte el Blog de Apache. **Se recomienda a los usuarios de Log4j 1 que actualicen a Apache Log4j 2** .

De: http://logging.apache.org/log4j/1.2/

Examples

Como obtener Log4j

Versión actual (log4j2)

Utilizando Maven:
Agregue la siguiente dependencia a su archivo `POM.xml` :

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.6.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Usando Ivy:

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-api" rev="2.6.2" />
  <dependency org="org.apache.logging.log4j" name="log4j-core" rev="2.6.2" />
</dependencies>
```

Utilizando Gradle:

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-api', version: '2.6.2'
  compile group: 'org.apache.logging.log4j', name: 'log4j-core', version: '2.6.2'
}
```

Obteniendo log4j 1.x

Nota: Log4j 1.x ha alcanzado el final de la vida útil (EOL) (ver Comentarios).

Utilizando Maven:

Declare esta dependencia en el archivo `POM.xml` :

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
```

```
</dependency>
```

Usando Ivy:

```
<dependency org="log4j" name="log4j" rev="1.2.17"/>
```

Usle Gradle:

```
compile group: 'log4j', name: 'log4j', version: '1.2.17'
```

Utilizando Buildr:

```
'log4j:log4j:jar:1.2.17'
```

Añadiendo manualmente en la construcción de la ruta:

Descargar desde el proyecto del sitio web Log4j

Cómo usar Log4j en código Java

Primero necesita crear un objeto `final static logger` :

```
final static Logger logger = Logger.getLogger(classname.class);
```

Luego, llame a los métodos de registro:

```
//logs an error message
logger.info("Information about some param: " + parameter); // Note that this line could throw
a NullPointerException!

//in order to improve performance, it is advised to use the `isXXXEnabled()` Methods
if( logger.isInfoEnabled() ){
    logger.info("Information about some param: " + parameter);
}

// In log4j2 parameter substitution is preferable due to readability and performance
// The parameter substitution only takes place if info level is active which obsoletes the use
of isXXXEnabled().
logger.info("Information about some param: {}" , parameter);

//logs an exception
logger.error("Information about some error: ", exception);
```

Configuración de archivo de propiedades

Log4j le da la posibilidad de registrar datos en la consola y el archivo al mismo tiempo. Cree un archivo `log4j.properties` y `log4j.properties` dentro de esta configuración básica:

```
# Root logger option
log4j.rootLogger=DEBUG, stdout, file
```

```
# Redirect log messages to console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# Redirect log messages to a log file, support file rolling.
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=C:\\log4j-application.log
log4j.appender.file.MaxFileSize=5MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n
```

Si está utilizando maven, ponga este archivo de propiedades en la ruta:

```
/ProjectFolder/src/java/resources
```

Archivo de configuración log4j2.xml básico

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Appenders>
    <Console name="STDOUT" target="SYSTEM_OUT">
      <PatternLayout pattern="%d %-5p [%t] %C{2} %m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>
```

Esta es una configuración básica de log4j2.xml que tiene una aplicación de consola y un registrador de raíz. El diseño del patrón especifica qué patrón se debe utilizar para registrar las declaraciones.
Para depurar la carga de log4j2.xml puede agregar el atributo `status = <WARN | DEBUG | ERROR | FATAL | TRACE | INFO>` en la etiqueta de configuración de su log4j2.xml.
También puede agregar un intervalo de supervisión para que vuelva a cargar la configuración después del período de intervalo especificado. El intervalo del monitor se puede agregar a la etiqueta de configuración de la siguiente manera: `monitorInterval = 30` . Esto significa que la configuración se cargará cada 30 segundos.

Migración de log4j 1.x a 2.x

Si desea migrar de log4j 1.x existente en su proyecto a log4j 2.x, elimine todas las dependencias de log4j 1.x existentes y agregue la siguiente dependencia:

Log4j 1.x API Bridge

Maven Build

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-1.2-api</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Construir hiedra

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-1.2-api" rev="2.6.2" />
</dependencies>
```

Construcción de Gradle

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-1.2-api', version: '2.6.2'
}
```

Puente de registro de Apache Commons Si su proyecto está utilizando el registro de Apache Commons que usa log4j 1.x y desea migrarlo a log4j 2.x, agregue las siguientes dependencias:

Maven Build

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-jcl</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Construir hiedra

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-jcl" rev="2.6.2" />
</dependencies>
```

Construcción de Gradle

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-jcl', version: '2.6.2'
}
```

Nota: No elimine las dependencias existentes del registro de Apache commons

Referencia: https://logging.apache.org/log4j/2.x/maven-artifacts.html

Propiedades-Archivo para iniciar sesión en DB

Para que este ejemplo funcione, necesitará un controlador JDBC compatible con el sistema en el

que se ejecuta la base de datos. Puede encontrar una fuente abierta que le permite conectarse a bases de datos DB2 en un IBM System i aquí: JT400

Aunque este ejemplo es específico de DB2, funciona para casi todos los demás sistemas si intercambia el controlador y adapta la URL de JDBC.

```
# Root logger option
log4j.rootLogger= ERROR, DB

# Redirect log messages to a DB2
# Define the DB appender
log4j.appender.DB=org.apache.log4j.jdbc.JDBCAppender

# Set JDBC URL (!!! adapt to your target system !!!)
log4j.appender.DB.URL=jdbc:as400://10.10.10.1:446/DATABASENAME;naming=system;errors=full;

# Set Database Driver (!!! adapt to your target system !!!)
log4j.appender.DB.driver=com.ibm.as400.access.AS400JDBCDriver

# Set database user name and password
log4j.appender.DB.user=USER
log4j.appender.DB.password=PASSWORD

# Set the SQL statement to be executed.
log4j.appender.DB.sql=INSERT INTO DB.TABLENAME VALUES('%d{yyyy-MM-
dd}','%d{HH:mm:ss}','%C','%p','%m')

# Define the layout for file appender
log4j.appender.DB.layout=org.apache.log4j.PatternLayout
```

Filtrar Logoutput por nivel (log4j 1.x)

Puede usar un filtro para registrar solo los mensajes "más bajos" que, por ejemplo, el nivel de `ERROR` . **Pero el filtro no es compatible con PropertyConfigurator. Así que debes cambiar a la configuración XML para usarlo** . Ver log4j-Wiki sobre filtros .

Ejemplo "nivel específico"

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
            <param name="File" value="info.log"/>
            <layout class="org.apache.log4j.PatternLayout">
                    <param name="ConversionPattern" value="%m%n"/>
            </layout>
            <filter class="org.apache.log4j.varia.LevelMatchFilter">
                    <param name="LevelToMatch" value="info" />
                    <param name="AcceptOnMatch" value="true"/>
            </filter>
            <filter class="org.apache.log4j.varia.DenyAllFilter" />
</appender>
```

O "rango de nivel"

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
            <param name="File" value="info.log"/>
            <layout class="org.apache.log4j.PatternLayout">
```

```
                <param name="ConversionPattern" value="%m%n"/>
            </layout>
            <filter class="org.apache.log4j.varia.LevelRangeFilter">
                <param name="LevelMax" value="info"/>
                <param name="LevelMin" value="info"/>
                <param name="AcceptOnMatch" value="true"/>
            </filter>
</appender>
```

Lea log4j / log4j2 en línea: https://riptutorial.com/es/java/topic/2472/log4j---log4j2

Capítulo 124: Los operadores

Introducción

Los operadores en el lenguaje de programación Java son símbolos especiales que realizan operaciones específicas en uno, dos o tres operandos y luego devuelven un resultado.

Observaciones

Un *operador* es un símbolo (o símbolos) que le dice a un programa Java que realice una *operación* en uno, dos o tres *operandos* . Un operador y sus operandos forman una *expresión* (consulte el tema Expresiones). Los operandos de un operador son en sí mismos expresiones.

Este tema describe los 40 o más operadores distintos definidos por Java. El tema de las expresiones separadas explica:

- cómo los operadores, los operandos y otras cosas se combinan en expresiones,
- cómo se evalúan las expresiones, y
- cómo funcionan los tipos de expresión, las conversiones y la evaluación de expresiones.

Examples

El operador de concatenación de cuerdas (+)

El símbolo `+` puede significar tres operadores distintos en Java:

- Si no hay ningún operando antes del `+` , entonces es el operador Unario Plus.
- Si hay dos operandos, y ambos son numéricos. entonces es el operador binario de adición.
- Si hay dos operandos, y al menos uno de ellos es un `String` , entonces es el operador binario de concatenación.

En el caso simple, el operador de concatenación une dos cadenas para dar una tercera cadena. Por ejemplo:

```
String s1 = "a String";
String s2 = "This is " + s1;    // s2 contains "This is a String"
```

Cuando uno de los dos operandos no es una cadena, se convierte en una `String` siguiente manera:

- Un operando cuyo tipo es un tipo primitivo se convierte *como si* llamando `toString()` en el valor en caja.

- Un operando cuyo tipo es un tipo de referencia se convierte llamando al método `toString()` del operando. Si el operando es `null` , o si el método `toString()` devuelve un `null` , en su

lugar se usa el literal de cadena `"null"` .

Por ejemplo:

```
int one = 1;
String s3 = "One is "  + one;         // s3 contains "One is 1"
String s4 = null + " is null";        // s4 contains "null is null"
String s5 = "{1} is " + new int[]{1}; // s5 contains something like
                                      // "{} is [I@xxxxxxxx"
```

La explicación para el ejemplo de `s5` es que el método `toString()` en tipos de matriz se hereda de `java.lang.Object` , y el comportamiento es producir una cadena que consiste en el nombre del tipo y el código de identidad del objeto.

El operador de concatenación se especifica para crear un nuevo objeto de `String` , excepto en el caso de que la expresión sea una expresión constante. En el último caso, la expresión se evalúa en el tipo de compilación, y su valor de tiempo de ejecución es equivalente a una cadena literal. Esto significa que no hay una sobrecarga de tiempo de ejecución al dividir un literal de cadena larga como este:

```
String typing = "The quick brown fox " +
                "jumped over the " +
                "lazy dog";           // constant expression
```

Optimización y eficiencia.

Como se señaló anteriormente, con la excepción de las expresiones constantes, cada expresión de concatenación de cadena crea un nuevo objeto de `String` . Considere este código:

```
public String stars(int count) {
    String res = "";
    for (int i = 0; i < count; i++) {
        res = res + "*";
    }
    return res;
}
```

En el método anterior, cada iteración del bucle creará una nueva `String` que es un carácter más largo que la iteración anterior. Cada concatenación copia todos los caracteres en las cadenas de operandos para formar la nueva `String` . Así, las `stars(N)` :

- crea `N` nuevos objetos `String` , y tira todos menos el último,
- copiar `N * (N + 1) / 2` caracteres, y
- generar `O(N^2)` bytes de basura.

Esto es muy caro para `N` grande. De hecho, cualquier código que concatene cadenas en un bucle puede tener este problema. Una mejor manera de escribir esto sería como sigue:

```
public String stars(int count) {
    // Create a string builder with capacity 'count'
```

```
    StringBuilder sb = new StringBuilder(count);
    for (int i = 0; i < count; i++) {
        sb.append("*");
    }
    return sb.toString();
}
```

Idealmente, debería ajustar la capacidad de la `StringBuilder` , pero si esto no es práctico, la clase va a *crecer* de forma automática la matriz respaldo que el constructor utiliza para contener caracteres. (Nota: la implementación expande la matriz de respaldo de manera exponencial. Esta estrategia mantiene esa cantidad de caracteres que se copian en `O(N)` lugar de `O(N^2)` .

Algunas personas aplican este patrón a todas las concatenaciones de cuerdas. Sin embargo, esto no es necesario porque JLS *permite que* un compilador de Java optimice las concatenaciones de cadenas dentro de una sola expresión. Por ejemplo:

```
String s1 = ...;
String s2 = ...;
String test = "Hello " + s1 + ". Welcome to " + s2 + "\n";
```

será *típicamente* optimizado por el compilador bytecode a algo como esto;

```
StringBuilder tmp = new StringBuilder();
tmp.append("Hello ")
tmp.append(s1 == null ? "null" + s1);
tmp.append("Welcome to ");
tmp.append(s2 == null ? "null" + s2);
tmp.append("\n");
String test = tmp.toString();
```

(El compilador JIT puede optimizar eso aún más si puede deducir que `s1` o `s2` no puede ser `null`). Pero tenga en cuenta que esta optimización solo está permitida dentro de una sola expresión.

En resumen, si le preocupa la eficacia de las concatenaciones de cadenas:

- Optimice a mano si está haciendo concatenación repetida en un bucle (o similar).
- No optimice manualmente una sola expresión de concatenación.

Los operadores aritméticos (+, -, *, /,%)

El lenguaje Java proporciona 7 operadores que realizan aritmética en valores enteros y de punto flotante.

- Hay dos operadores `+`
  - El operador de suma binaria agrega un número a otro. (También hay un operador binario `+` que realiza la concatenación de cadenas. Esto se describe en un ejemplo separado).
  - El operador Unary Plus no hace nada más que activar la promoción numérica (ver más abajo)
- Hay dos `-` operadores:

    - El operador de sustracción binaria resta un número de otro.
    - El operador menos unario es equivalente a restar su operando de cero.
- El operador de multiplicación binaria (*) multiplica un número por otro.
- El operador de división binaria (/) divide un número por otro.
- El operador del resto binario [1] (%) calcula el resto cuando un número se divide por otro.

1. A menudo, esto se conoce incorrectamente como operador de "módulo". "Resto" es el término utilizado por el JLS. "Módulo" y "resto" no son lo mismo.

Operandos y tipos de resultados, y promoción numérica.

Los operadores requieren operandos numéricos y producen resultados numéricos. Los tipos de operandos pueden ser cualquier tipo numérico primitivo (es decir, `byte` , `short` , `char` , `int` , `long` , `float` o `double`) o cualquier tipo de contenedor numérico definido en `java.lang` ; ie (`Byte` , `Character` , `Short` , `Integer` , `Long` , `Float` o `Double` .

El tipo de resultado se determina en base a los tipos de operandos u operandos, de la siguiente manera:

- Si cualquiera de los operandos es `double` o `Double` , entonces el tipo de resultado es `double` .
- De lo contrario, si alguno de los operandos es `float` o `Float` , el tipo de resultado es `float` .
- De lo contrario, si alguno de los operandos es `long` o `Long` , entonces el tipo de resultado es `long` .
- De lo contrario, el tipo de resultado es `int` . Esto cubre `byte` , `short` y `char` operandos, así como `int.

El tipo de resultado de la operación determina cómo se realiza la operación aritmética y cómo se manejan los operandos

- Si el tipo de resultado es `double` , los operandos se promueven a `double` , y la operación se realiza utilizando aritmética de punto flotante IEE 754 de 64 bits (binario de doble precisión).
- Si el tipo de resultado es `float` , los operandos se promueven a `float` , y la operación se realiza utilizando aritmética de punto flotante IEE 754 de 32 bits (binario de precisión simple).
- Si el tipo de resultado es `long` , los operandos se promueven a `long` , y la operación se realiza utilizando aritmética de enteros binarios de dos bits con signo de 64 bits.
- Si el tipo de resultado es `int` , los operandos se promueven a `int` , y la operación se realiza utilizando aritmética de enteros binarios de dos bits con signo de 32 bits.

La promoción se realiza en dos etapas:

- Si el tipo de operando es un tipo de envoltura, el valor del operando se *desajusta* a un valor del tipo primitivo correspondiente.
- Si es necesario, el tipo primitivo se promueve al tipo requerido:
    - La promoción de enteros a `int` o `long` tiene pérdidas.
    - La promoción de `float` al `double` tiene pérdidas.
    - La promoción de un número entero a un valor de punto flotante puede llevar a la pérdida de precisión. La conversión se realiza utilizando la semántica "redondeo a

más cercano" IEE 768.

El significado de división

El operador / divide el operando izquierdo `n` (el *dividendo*) y el operando derecho `d` (el *divisor*) y produce el resultado `q` (el *cociente*).

La división entera de Java se redondea hacia cero. La Sección 15.17.2 de JLS especifica el comportamiento de la división entera de Java de la siguiente manera:

> El cociente producido para los operandos `n` y `d` es un valor entero `q` cuya magnitud es lo más grande posible al tiempo que satisface `|d · q| ≤ |n|` . Además, `q` es positivo cuando `|n| ≥ |d| n` y `d` tienen el mismo signo, pero `q` es negativo cuando `|n| ≥ |d| n` y `d` tienen signos opuestos.

Hay un par de casos especiales:

- Si la `n` es `MIN_VALUE` y el divisor es -1, entonces se produce un desbordamiento de enteros y el resultado es `MIN_VALUE` . No se lanza ninguna excepción en este caso.
- Si `d` es 0, entonces se lanza `ArithmeticException.

Java división de punto flotante tiene más casos de borde a considerar. Sin embargo, la idea básica es que el resultado `q` es el valor más cercano a la satisfacción de `d . q = n` .

La división de punto flotante nunca dará lugar a una excepción. En cambio, las operaciones que se dividen por cero dan como resultado valores de INF y NaN; vea abajo.

El significado del resto

A diferencia de C y C ++, el operador de resto en Java funciona con operaciones de punto flotante y enteros.

Para casos de enteros, el resultado de `a % b` se define como el número `r` tal que `(a / b) * b + r` es igual a `a` , donde `/` , `*` y `+` son los operadores enteros de Java apropiados. Esto se aplica en todos los casos, excepto cuando `b` es cero. En ese caso, el resto da como resultado una `ArithmeticException` .

De la definición anterior se desprende que `a % b` puede ser negativo solo si `a` es negativo, y solo positivo si `a` es positivo. Además, la magnitud de `a % b` es siempre menor que la magnitud de `b` .

La operación de resto de punto flotante es una generalización del caso entero. El resultado de `a % b` es el resto `r` se define por la relación matemática `r = a - (b · q)` donde:

- `q` es un número entero,
- es negativo solo si `a / b` es negativo y positivo solo si `a / b` es positivo, y
- su magnitud es tan grande como sea posible sin exceder la magnitud del verdadero cociente matemático de `a` y `b` .

El resto de punto flotante puede producir valores de `INF` y `NaN` en casos de borde, como cuando `b` es cero; vea abajo. No lanzará una excepción.

Nota IMPORTANTE:

> El resultado de una operación de resto de punto flotante calculada por `%` **no es la misma** que la producida por la operación de resto definida por IEEE 754. El resto de IEEE 754 se puede calcular utilizando el método de biblioteca `Math.IEEEremainder` .

Desbordamiento de enteros

Los valores enteros de Java de 32 y 64 bits están firmados y utilizan una representación binaria de dos complementos. Por ejemplo, el rango de números representables como (32 bit) `int` -2^{31} a $+2^{31} - 1$.

Cuando sumas, restas o múltiples enteros de dos bits N (N == 32 o 64), el resultado de la operación puede ser demasiado grande para representarlo como un entero de N bits. En este caso, la operación conduce a *un desbordamiento de enteros* , y el resultado se puede calcular de la siguiente manera:

- La operación matemática se realiza para dar una representación intermedia de complemento de dos de todo el número. Esta representación será mayor que N bits.
- Los 32 o 64 bits inferiores de la representación intermedia se utilizan como resultado.

Cabe señalar que el desbordamiento de enteros no da lugar a excepciones en ningún caso.

Valores de punto flotante INF y NAN

Java utiliza representaciones de punto flotante IEE 754 para `float` y `double` . Estas representaciones tienen algunos valores especiales para representar valores que están fuera del dominio de los números reales:

- Los valores "infinitos" o INF denotan números que son demasiado grandes. El valor `+INF` denota números que son demasiado grandes y positivos. El valor `-INF` denota números que son demasiado grandes y negativos.
- El "indefinido" / "no un número" o NaN denota valores resultantes de operaciones sin sentido.

Los valores INF son producidos por operaciones flotantes que causan desbordamiento, o por división por cero.

Los valores de NaN se producen dividiendo cero por cero, o calculando cero resto cero.

Sorprendentemente, es posible realizar aritmética utilizando operandos INF y NaN sin desencadenar excepciones. Por ejemplo:

- Sumando + INF y un valor finito da + INF.
- Sumando + INF y + INF da + INF.

- Añadiendo + INF y -INF da NaN.
- La división por INF da +0.0 o -0.0.
- Todas las operaciones con uno o más operandos de NaN dan NaN.

Para obtener más detalles, consulte las subsecciones relevantes de JLS 15 . Tenga en cuenta que esto es en gran parte "académico". Para cálculos típicos, un `INF` o `NaN` significa que algo ha salido mal; por ejemplo, tiene datos de entrada incompletos o incorrectos, o el cálculo se ha programado incorrectamente.

Los operadores de igualdad (==,! =)

Los operadores `==` y `!=` Son operadores binarios que se evalúan como `true` o `false` dependiendo de si los operandos son iguales. El operador `==` da `true` si los operandos son iguales y `false` contrario. El operador `!=` Da `false` si los operandos son iguales y `true` contrario.

Estos operadores pueden usarse operandos con tipos primitivos y de referencia, pero el comportamiento es significativamente diferente. Según el JLS, en realidad hay tres conjuntos distintos de estos operadores:

- Los operadores booleanos `==` y `!=` .
- Los operadores numéricos `==` y `!=` .
- Los operadores Reference `==` y `!=` .

Sin embargo, en todos los casos, el tipo de resultado de los operadores `==` y `!=` Es `boolean` .

Los operadores numéricos `==` y `!=`

Cuando uno (o ambos) de los operandos de un operador `==` o `!=` Es un tipo numérico primitivo (`byte` , `short` , `char` , `int`, `long` , `float` o `double`), el operador es una comparación numérica. El segundo operando debe ser un tipo numérico primitivo o un tipo numérico en caja.

El comportamiento de otros operadores numéricos es el siguiente:

1. Si uno de los operandos es un tipo de caja, es sin caja.
2. Si cualquiera de los operandos ahora un `byte` , `short` o `char` , se promueve a un `int` .
3. Si los tipos de los operandos no son los mismos, entonces el operando con el tipo "más pequeño" se promueve al tipo "más grande".
4. La comparación se realiza de la siguiente manera:
    - Si los operandos promovidos son `int` o `long` , los valores se prueban para ver si son idénticos.
    - Si los operandos promovidos son `float` o `double` entonces:
        - las dos versiones de cero (`+0.0` y `-0.0`) se tratan como iguales
        - un valor `NaN` se trata como no igual a nada, y
        - otros valores son iguales si sus representaciones IEEE 754 son idénticas.

Nota: debe tener cuidado al usar `==` y `!=` Para comparar valores de punto flotante.

Los operadores booleanos `==` y `!=`

Si ambos operandos son `boolean` , o uno es `boolean` y el otro es `Boolean` , estos operadores son los operadores booleano `==` y `!=` . El comportamiento es el siguiente:

1. Si uno de los operandos es un `Boolean` , es unboxed.
2. Los operandos sin caja se prueban y el resultado booleano se calcula de acuerdo con la siguiente tabla de verdad

UNA	segundo	A == B	A! = B
falso	falso	cierto	falso
falso	cierto	falso	cierto
cierto	falso	falso	cierto
cierto	cierto	cierto	falso

Hay dos "trampas" que hacen que sea aconsejable usar `==` y `!=` moderación con los valores de verdad:

- Si usa `==` o `!=` Para comparar dos objetos `Boolean` , entonces se usan los operadores de referencia. Esto puede dar un resultado inesperado; ver Escaparate: usar == para comparar objetos de envoltorios primitivos, como Integer

- El operador `==` puede ser mal escrito como `=` . Para la mayoría de los tipos de operandos, este error conduce a un error de compilación. Sin embargo, para `Boolean` operandos `boolean` y `Boolean` el error conduce a un comportamiento de tiempo de ejecución incorrecto; ver Pitfall - Usar '==' para probar un booleano

Los operadores Reference `==` y `!=`

Si ambos operandos son referencias a objetos, los operadores `==` y `!=` Comprueban si los dos operandos se **refieren al mismo objeto** . Esto a menudo no es lo que quieres. Para probar si dos objetos son iguales *por valor* , se debe usar el método `.equals()` su lugar.

```
String s1 = "We are equal";
String s2 = new String("We are equal");

s1.equals(s2); // true

// WARNING - don't use == or != with String values
s1 == s2;      // false
```

Advertencia: el uso de `==` y `!=` Para comparar valores de `String` es **incorrecto** en la mayoría de los casos; vea http://www.riptutorial.com/java/example/16290/pitfall--using----to-compare-strings . Un problema similar se aplica a los tipos de envoltorios primitivos; vea

http://www.riptutorial.com/java/example/8996/pitfall--using----to-compare-primitive-wrappers-objects-such-as-integer .

Acerca de los casos de borde de NaN

JLS 15.21.1 establece lo siguiente:

> Si cualquiera de los operandos es `NaN` , entonces el resultado de `==` es `false` pero el resultado de `!=` Es `true` . De hecho, la prueba `x != x` es `true` si y solo si el valor de `x` es `NaN` .

Este comportamiento es (para la mayoría de los programadores) inesperado. Si prueba si un valor de `NaN` es igual a sí mismo, la respuesta es "¡No, no lo es!". En otras palabras, `==` no es *reflexivo* para los valores de `NaN` .

Sin embargo, esto no es una "rareza" de Java, este comportamiento se especifica en los estándares de punto flotante IEEE 754 y encontrará que está implementado por la mayoría de los lenguajes de programación modernos. (Para obtener más información, consulte http://stackoverflow.com/a/1573715/139985 ... ¡note que esto está escrito por alguien que estaba "en la sala cuando se tomaron las decisiones"!)

Los operadores de incremento / decremento (++ / -)

Las variables pueden incrementarse o disminuirse en 1 usando los operadores `++` y `--` , respectivamente.

Cuando los operadores `++` y `--` siguen las variables, se denominan **post-incremento** y **post-decremento** respectivamente.

```
int a = 10;
a++; // a now equals 11
a--; // a now equals 10 again
```

Cuando los operadores `++` y `--` preceden a las variables, las operaciones se llaman **pre-incremento** y **pre-decremento** respectivamente.

```
int x = 10;
--x; // x now equals 9
++x; // x now equals 10
```

Si el operador precede a la variable, el valor de la expresión es el valor de la variable después de ser incrementado o disminuido. Si el operador sigue la variable, el valor de la expresión es el valor de la variable antes de ser incrementada o disminuida.

```
int x=10;

System.out.println("x=" + x + " x=" + x++ + " x=" + x); // outputs x=10 x=10 x=11
System.out.println("x=" + x + " x=" + ++x + " x=" + x); // outputs x=11 x=12 x=12
System.out.println("x=" + x + " x=" + x-- + " x=" + x); // outputs x=12 x=12 x=11
```

```
System.out.println("x=" + x + " x=" + --x + " x=" + x); // outputs x=11 x=10 x=10
```

Tenga cuidado de no sobrescribir los incrementos o decrementos posteriores. Esto sucede si utiliza un operador de post-in / decremento al final de una expresión que se reasigna a la variable in / decremented. La in / decremento no tendrá efecto. Aunque la variable en el lado izquierdo se incrementa correctamente, su valor se sobrescribirá inmediatamente con el resultado evaluado previamente en el lado derecho de la expresión:

```
int x = 0;
x = x++ + 1 + x++;       // x = 0 + 1 + 1
                         // do not do this - the last increment has no effect (bug!)
System.out.println(x);   // prints 2 (not 3!)
```

Correcto:

```
int x = 0;
x = x++ + 1 + x;         // evaluates to x = 0 + 1 + 1
x++;                     // adds 1
System.out.println(x);   // prints 3
```

El Operador Condicional (? :)

Sintaxis

{condición para evaluar} **?** *{instrucción-ejecutada-en-verdadero}* **:** *{declaración-ejecutada-en-falso}*

Como se muestra en la sintaxis, el Operador condicional (también conocido como Operador Ternario [1]) usa el `?` (signo de interrogación) y `:` (dos puntos) caracteres para habilitar una expresión condicional de dos resultados posibles. Se puede usar para reemplazar bloques `if-else` largos para devolver uno de los dos valores según la condición.

```
result = testCondition ? value1 : value2
```

Es equivalente a

```
if (testCondition) {
    result = value1;
} else {
    result = value2;
}
```

Se puede leer como ***"Si testCondition es verdadero, establezca el resultado en value1; de lo contrario, establezca el resultado en valor2 ".***

Por ejemplo:

```
// get absolute value using conditional operator
a = -10;
```

```
int absValue = a < 0 ? -a : a;
System.out.println("abs = " + absValue); // prints "abs = 10"
```

Es equivalente a

```
// get absolute value using if/else loop
a = -10;
int absValue;
if (a < 0) {
    absValue = -a;
} else {
    absValue = a;
}
System.out.println("abs = " + absValue); // prints "abs = 10"
```

Uso común

Puede usar el operador condicional para asignaciones condicionales (como la comprobación de nulos).

```
String x = y != null ? y.toString() : ""; //where y is an object
```

Este ejemplo es equivalente a:

```
String x = "";

if (y != null) {
    x = y.toString();
}
```

Dado que el Operador Condicional tiene la segunda prioridad más baja, por encima de los Operadores de Asignación , rara vez es necesario usar paréntesis alrededor de la *condición* , pero se requiere paréntesis alrededor de la construcción completa del Operador Condicional cuando se combina con otros operadores:

```
// no parenthesis needed for expressions in the 3 parts
10 <= a && a < 19 ? b * 5 : b * 7

// parenthesis required
7 * (a > 0 ? 2 : 5)
```

El anidamiento de operadores condicionales también se puede hacer en la tercera parte, donde funciona más como un encadenamiento o como una instrucción switch.

```
a ? "a is true" :
b ? "a is false, b is true" :
c ? "a and b are false, c is true" :
    "a, b, and c are false"

//Operator precedence can be illustrated with parenthesis:
```

```
a ? x : (b ? y : (c ? z : w))
```

Nota:

1 - Tanto la especificación del lenguaje Java como el tutorial de Java llaman al operador (`? :` :) El operador *condicional* . El Tutorial dice que es "también conocido como el Operador Ternario" ya que es (actualmente) el único operador ternario definido por Java. La terminología de "Operador condicional" es consistente con C y C ++ y otros idiomas con un operador equivalente.

Los operadores bitwise y lógicos (~, &, |, ^)

El lenguaje Java proporciona 4 operadores que realizan operaciones en modo bit o lógicas en operandos enteros o booleanos.

- El operador de complemento (`~`) es un operador unario que realiza una inversión lógica o bit a bit de los bits de un operando; ver JLS 15.15.5. .
- El operador AND (`&`) es un operador binario que realiza un bit y lógico "y" de dos operandos; ver JLS 15.22.2. .
- El operador OR (`|`) es un operador binario que realiza un bit o lógico "inclusivo o" de dos operandos; ver JLS 15.22.2. .
- El operador XOR (`^`) es un operador binario que realiza una "exclusiva o" bitwise o lógica de dos operandos; ver JLS 15.22.2. .

Las operaciones lógicas realizadas por estos operadores cuando los operandos son booleanos se pueden resumir de la siguiente manera:

UNA	segundo	~ A	A y B	Un | segundo	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Tenga en cuenta que para los operandos enteros, la tabla anterior describe lo que sucede para los bits individuales. Los operadores realmente operan en todos los 32 o 64 bits del operando u operandos en paralelo.

Tipos de operandos y tipos de resultados.

Las conversiones aritméticas habituales se aplican cuando los operandos son enteros. Casos de uso comunes para los operadores bitwise

---

El operador `~` se usa para invertir un valor booleano, o cambiar todos los bits en un operando

entero.

El operador `&` se usa para "enmascarar" algunos de los bits en un operando de enteros. Por ejemplo:

```
int word = 0b00101010;
int mask = 0b00000011;   // Mask for masking out all but the bottom
                         // two bits of a word
int lowBits = word & mask;            // -> 0b00000010
int highBits = word & ~mask;          // -> 0b00101000
```

El `|` operador se utiliza para combinar los valores de verdad de dos operandos. Por ejemplo:

```
int word2 = 0b01011111;
// Combine the bottom 2 bits of word1 with the top 30 bits of word2
int combined = (word & mask) | (word2 & ~mask);   // -> 0b01011110
```

El operador `^` se usa para alternar o "voltear" los bits:

```
int word3 = 0b00101010;
int word4 = word3 ^ mask;             // -> 0b00101001
```

Para obtener más ejemplos del uso de los operadores bitwise, vea Manipulación de bits.

La instancia del operador

Este operador verifica si el objeto es de una clase particular / tipo de interfaz. **El** operador **instanceof** se escribe como:

```
( Object reference variable ) instanceof  (class/interface type)
```

Ejemplo:

```
public class Test {

   public static void main(String args[]){
      String name = "Buyya";
      // following will return true since name is type of String
      boolean result = name instanceof String;
      System.out.println( result );
   }
}
```

Esto produciría el siguiente resultado:

```
true
```

Este operador aún devolverá verdadero si el objeto que se compara es la asignación compatible con el tipo de la derecha.

Ejemplo:

```
class Vehicle {}

public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result );
   }
}
```

Esto produciría el siguiente resultado:

```
true
```

Los operadores de asignación (=, + =, - =, * =, / =,% =, << =, >> =, >>> =, & =, | = y ^ =)

El operando de la izquierda para estos operadores debe ser una variable no final o un elemento de una matriz. El operando de la derecha debe ser *compatible* con la *asignación* del operando de la izquierda. Esto significa que o los tipos deben ser iguales, o el tipo de operando derecho debe ser convertible al tipo de operandos izquierdo mediante una combinación de boxeo, desempaquetado o ampliación. (Para detalles completos refiérase a JLS 5.2).

El significado preciso de los operadores de "operación y asignación" se especifica en JLS 15.26.2 como:

> Una expresión de asignación compuesta de la forma `E1 op= E2` es equivalente a `E1 = (T) ((E1) op (E2))` , donde `T` es el tipo de `E1` , excepto que `E1` se evalúa solo una vez.

Tenga en cuenta que hay una conversión de tipos implícita antes de la asignación final.

1. `=`

El operador de asignación simple: asigna el valor del operando de la derecha al operando de la izquierda.

> Ejemplo: `c = a + b` agregará el valor de `a + b` al valor de `c` y lo asignará a `c`

2. `+=`

El operador "agregar y asignar": agrega el valor del operando de la mano derecha al valor del operando de la mano izquierda y asigna el resultado al operando de la mano izquierda. Si el operando de la izquierda tiene el tipo `String` , entonces este es un operador de "concatenar y asignar".

> Ejemplo: `c += a` es aproximadamente lo mismo que `c = c + a`

3. `-=`

El operador "restar y asignar": resta el valor del operando derecho del valor del operando de la izquierda y asigna el resultado al operando de la izquierda.

Ejemplo: `c -= a` es aproximadamente lo mismo que `c = c - a`

4. `*=`

El operador "multiplicar y asignar": multiplica el valor del operando de la mano derecha por el valor del operando de la mano izquierda y asigna el resultado al operando de la mano izquierda. .

Ejemplo: `c *= a` es aproximadamente lo mismo que `c = c * a`

5. `/=`

El operador "dividir y asignar": divide el valor del operando de la mano derecha por el valor del operando de la mano izquierda y asigna el resultado al operando de la mano izquierda.

Ejemplo: `c /*= a` es aproximadamente lo mismo que `c = c / a`

6. `%=`

El operador "módulo y asignación": calcula el módulo del valor del operando de la derecha por el valor del operando de la izquierda y asigna el resultado al operando de la izquierda.

Ejemplo: `c %*= a` es aproximadamente lo mismo que `c = c % a`

7. `<<=`

El operador "desplazar a la izquierda y asignar".

Ejemplo: `c <<= 2` es aproximadamente lo mismo que `c = c << 2`

8. `>>=`

El "operador aritmético de desplazamiento y asignación".

Ejemplo: `c >>= 2` es aproximadamente lo mismo que `c = c >> 2`

9. `>>>=`

El "operador lógico de desplazamiento y asignación".

Ejemplo: `c >>>= 2` es aproximadamente lo mismo que `c = c >>> 2`

10. `&=`

El operador "bitwise y y asignar".

Ejemplo: `c &= 2` es aproximadamente lo mismo que `c = c & 2`

11. `|=`

El operador "bitwise o y asignar".

Ejemplo: `c |= 2` es aproximadamente lo mismo que `c = c | 2`

12. `^=`

El operador "bitwise exclusivo o y asignar".

> Ejemplo: `c ^= 2` es aproximadamente lo mismo que `c = c ^ 2`

Los operadores condicionales y condicionales u (&& y ||)

Java proporciona un operador condicional y un operador condicional, que toman uno o dos operandos de tipo `boolean` y producen un resultado `boolean` . Estos son:

- `&&` - el operador condicional-AND,
- `||` - Los operadores condicionales-OR. La evaluación de `<left-expr> && <right-expr>` es equivalente al siguiente pseudocódigo:

```
{
   boolean L = evaluate(<left-expr>);
   if (L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return false;
   }
}
```

La evaluación de `<left-expr> || <right-expr>` es equivalente al siguiente pseudocódigo:

```
{
   boolean L = evaluate(<left-expr>);
   if (!L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return true;
   }
}
```

Como lo ilustra el pseudocódigo anterior, el comportamiento de los operadores de cortocircuitos es equivalente a usar las sentencias `if` / `else` .

Ejemplo: usar && como guardia en una expresión

El siguiente ejemplo muestra el patrón de uso más común para el operador `&&` . Compare estas dos versiones de un método para probar si un `Integer` suministrado es cero.

```
public boolean isZero(Integer value) {
    return value == 0;
}

public boolean isZero(Integer value) {
    return value != null && value == 0;
```

```
}
```

La primera versión funciona en la mayoría de los casos, pero si el argumento de `value` es `null` , se lanzará una `NullPointerException` .

En la segunda versión hemos añadido una prueba de "guarda". El `value != null && value == 0` expresión se evalúa realizando primero el `value != null` Prueba `value != null` . Si la prueba `null` tiene éxito (es decir, se evalúa como `true`), se evalúa el `value == 0` expresión. Si la prueba `null` falla, entonces la evaluación del `value == 0` se omite (en cortocircuito) y *no* obtenemos una `NullPointerException` .

Ejemplo: usar && para evitar un cálculo costoso

El siguiente ejemplo muestra cómo se puede usar `&&` para evitar un cálculo relativamente costoso:

```
public boolean verify(int value, boolean needPrime) {
    return !needPrime | isPrime(value);
}

public boolean verify(int value, boolean needPrime) {
    return !needPrime || isPrime(value);
}
```

En la primera versión, ambos operandos de la `|` siempre se evaluará, por lo que el método (costoso) `isPrime` se llamará innecesariamente. La segunda versión evita la llamada innecesaria usando `||` en lugar de `|` .

Los operadores de turno (<<, >> y >>>)

El lenguaje Java proporciona tres operadores para realizar cambios a nivel de bits en valores enteros de 32 y 64 bits. Todos estos son operadores binarios con el primer operando que es el valor que se va a cambiar, y el segundo operando que dice qué tan lejos desplazarse.

- El operador de desplazamiento `<<` o a la *izquierda* desplaza el valor dado por el primer operando a la *izquierda* por el número de posiciones de bit dado por el segundo operando. Las posiciones vacías en el extremo derecho están llenas de ceros.

- El operador ">>" o de *cambio aritmético* desplaza el valor dado por el primer operando *hacia* la *derecha* por el número de posiciones de bit dado por el segundo operando. Las posiciones vacías en el extremo izquierdo se llenan copiando el bit más a la izquierda. Este proceso se conoce como *extensión de signo* .

- El ">>>" o *el* operador *lógico de desplazamiento a la derecha* desplaza el valor dado por el primer operando *hacia* la *derecha* por el número de posiciones de bit dado por el segundo operando. Las posiciones vacías en el extremo izquierdo están llenas de ceros.

Notas:

1. Estos operadores requieren un valor `int` o `long` como el primer operando, y producen un

valor con el mismo tipo que el primer operando. (Necesitará usar un tipo explícito de conversión al asignar el resultado de un cambio a una variable de `byte` , `short` o `char`).

2. Si utiliza un operador de cambio con un primer operando que es un `byte` , `char` o `short` , se promueve a un `int` y la operación produce un `int` .)

3. El segundo operando se reduce en *módulo el número de bits de la operación* para dar la cantidad del desplazamiento. Para más información sobre el **concepto matemático mod** , ver ejemplos de módulo .

4. Los bits que se desvían del extremo izquierdo o derecho por la operación se descartan. (Java no proporciona un operador de "rotación" primitivo).

5. El operador de cambio aritmético es equivalente dividiendo un número (el complemento de dos) por una potencia de 2.

6. El operador de desplazamiento a la izquierda es equivalente al multiplicar un número (el complemento de dos) por una potencia de 2.

La siguiente tabla le ayudará a ver los efectos de los tres operadores de turno. (Los números se han expresado en notación binaria para ayudar a la visualización).

Operando1	Operando2	<<	>>	>>>
0b0000000000001011	0	0b0000000000001011	0b0000000000001011	0b0000000000001011
0b0000000000001011	1	0b0000000000010110	0b0000000000000101	0b0000000000000101
0b0000000000001011	2	0b0000000000101100	0b0000000000000010	0b0000000000000010
0b0000000000001011	28	0b1011000000000000	0b0000000000000000	0b0000000000000000
0b0000000000001011	31	0b1000000000000000	0b0000000000000000	0b0000000000000000
0b0000000000001011	32	0b0000000000001011	0b0000000000001011	0b0000000000001011
...	...	...	...	...
0b1000000000001011	0	0b1000000000001011	0b1000000000001011	0b1000000000001011
0b1000000000001011	1	0b0000000000010110	0b1100000000000101	0b0100000000000101
0b1000000000001011	2	0b0000000000101100	0b1110000000000010	0b00100000000000100
0b1000000000001011	31	0b1000000000000000	0b1111111111111111	0b0000000000000001

Hay ejemplos del usuario de los operadores de cambio en la manipulación de bits.

El operador Lambda (->)

Desde Java 8 en adelante, el operador Lambda (`->`) es el operador utilizado para introducir una expresión Lambda. Hay dos sintaxis comunes, como se ilustra en estos ejemplos:

```
  a -> a + 1              // a lambda that adds one to its argument
  a -> { return a + 1; }  // an equivalent lambda using a block.
```

Una expresión lambda define una función anónima o, más correctamente, una instancia de una clase anónima que implementa una *interfaz funcional* .

(Este ejemplo se incluye aquí para estar completo. Consulte el tema de Expresiones de Lambda para el tratamiento completo).

Los operadores relacionales (<, <=,>,> =)

Los operadores `<` , `<=` , `>` y `>=` son operadores binarios para comparar tipos numéricos. El significado de los operadores es como usted esperaría. Por ejemplo, si `a` y `b` se declaran como cualquiera de los tipos de `byte` , `short` , `char` , `int` , `long` , `float` , `double` o los cuadros correspondientes:

```
- `a < b` tests if the value of `a` is less than the value of `b`.
- `a <= b` tests if the value of `a` is less than or equal to the value of `b`.
- `a > b` tests if the value of `a` is greater than the value of `b`.
- `a >= b` tests if the value of `a` is greater than or equal to the value of `b`.
```

El tipo de resultado para estos operadores es `boolean` en todos los casos.

Los operadores relacionales se pueden utilizar para comparar números con diferentes tipos. Por ejemplo:

```
int i = 1;
long l = 2;
if (i < l) {
    System.out.println("i is smaller");
}
```

Los operadores relacionales pueden usarse cuando uno o ambos números son instancias de tipos numéricos en caja. Por ejemplo:

```
Integer i = 1;   // 1 is autoboxed to an Integer
Integer j = 2;   // 2 is autoboxed to an Integer
if (i < j) {
    System.out.println("i is smaller");
}
```

El comportamiento preciso se resume de la siguiente manera:

1. Si uno de los operandos es un tipo de caja, es sin caja.
2. Si cualquiera de los operandos ahora un `byte` , `short` o `char` , se promueve a un `int` .
3. Si los tipos de los operandos no son los mismos, entonces el operando con el tipo "más pequeño" se promueve al tipo "más grande".
4. La comparación se realiza en los valores resultantes `int` , `long` , `float` o `double` .

Debe tener cuidado con las comparaciones relacionales que involucran números de punto flotante:

- Las expresiones que computan números de punto flotante a menudo incurren en errores de redondeo debido al hecho de que las representaciones de punto flotante de la computadora tienen una precisión limitada.
- Al comparar un tipo de entero y un tipo de punto flotante, la conversión del entero a punto flotante también puede llevar a errores de redondeo.

Finalmente, Java sí admite el uso de operadores relacionales con cualquier tipo distinto a los enumerados anteriormente. Por ejemplo, no *puede* usar estos operadores para comparar cadenas, matrices de números, etc.

Lea Los operadores en línea: https://riptutorial.com/es/java/topic/176/los-operadores

Capítulo 125: Manipulación de bits

Observaciones

- A diferencia de C / C ++, Java es completamente neutral con respecto al hardware subyacente de la máquina. No obtienes un comportamiento endiano grande o pequeño por defecto; Tienes que especificar explícitamente qué comportamiento quieres.
- El tipo de `byte` está firmado, con el rango de -128 a +127. Para convertir un valor de byte a su equivalente sin signo, enmascararlo con 0xFF de esta manera: `(b & 0xFF)` .

Examples

Empaquetar / desempaquetar valores como fragmentos de bits

Es común que el rendimiento de la memoria comprima múltiples valores en un solo valor primitivo. Esto puede ser útil para pasar información diversa a una sola variable.

Por ejemplo, uno puede empaquetar 3 bytes, como el código de color en RGB , en un solo int.

Empaquetando los valores

```
// Raw bytes as input
byte[] b = {(byte)0x65, (byte)0xFF, (byte)0x31};

// Packed in big endian: x == 0x65FF31
int x = (b[0] & 0xFF) << 16  // Red
      | (b[1] & 0xFF) <<  8  // Green
      | (b[2] & 0xFF) <<  0; // Blue

// Packed in little endian: y == 0x31FF65
int y = (b[0] & 0xFF) <<  0
      | (b[1] & 0xFF) <<  8
      | (b[2] & 0xFF) << 16;
```

Desempaquetando los valores

```
// Raw int32 as input
int x = 0x31FF65;

// Unpacked in big endian: {0x65, 0xFF, 0x31}
byte[] c = {
    (byte)(x >> 16),
    (byte)(x >>  8),
    (byte)(x & 0xFF)
};

// Unpacked in little endian: {0x31, 0xFF, 0x65}
byte[] d = {
    (byte)(x & 0xFF),
    (byte)(x >>  8),
```

```
    (byte)(x >> 16)
};
```

Comprobación, configuración, borrado y conmutación de bits individuales. Utilizando mascarilla larga como bit

Suponiendo que deseamos modificar el bit `n` de una primitiva entera, `i` (byte, short, char, int o long):

```
(i & 1 << n) != 0 // checks bit 'n'
i |= 1 << n;      // sets bit 'n' to 1
i &= ~(1 << n);   // sets bit 'n' to 0
i ^= 1 << n;      // toggles the value of bit 'n'
```

Usando long / int / short / byte como una máscara de bits:

```
public class BitMaskExample {
    private static final long FIRST_BIT = 1L << 0;
    private static final long SECOND_BIT = 1L << 1;
    private static final long THIRD_BIT = 1L << 2;
    private static final long FOURTH_BIT = 1L << 3;
    private static final long FIFTH_BIT = 1L << 4;
    private static final long BIT_55 = 1L << 54;

    public static void main(String[] args) {
        checkBitMask(FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55);
    }

    private static void checkBitMask(long bitmask) {
        System.out.println("FIRST_BIT: " + ((bitmask & FIRST_BIT) != 0));
        System.out.println("SECOND_BIT: " + ((bitmask & SECOND_BIT) != 0));
        System.out.println("THIRD_BIT: " + ((bitmask & THIRD_BIT) != 0));
        System.out.println("FOURTh_BIT: " + ((bitmask & FOURTH_BIT) != 0));
        System.out.println("FIFTH_BIT: " + ((bitmask & FIFTH_BIT) != 0));
        System.out.println("BIT_55: " + ((bitmask & BIT_55) != 0));
    }
}
```

Huellas dactilares

```
FIRST_BIT: true
SECOND_BIT: false
THIRD_BIT: true
FOURTh_BIT: false
FIFTH_BIT: true
BIT_55: true
```

que coincide con la máscara que nos pasa como `checkBitMask` parámetro: `FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55` .

Expresando el poder de 2

Para expresar la potencia de 2 (2 ^ n) de enteros, se puede usar una operación de cambio de bits

que permita especificar explícitamente la `n` .

La sintaxis es básicamente:

```
int pow2 = 1<<n;
```

Ejemplos:

```
int twoExp4 = 1<<4; //2^4
int twoExp5 = 1<<5; //2^5
int twoExp6 = 1<<6; //2^6
...
int twoExp31 = 1<<31; //2^31
```

Esto es especialmente útil cuando se definen valores constantes que deberían hacer evidente que se usa una potencia de 2, en lugar de usar valores hexadecimales o decimales.

```
int twoExp4 = 0x10; //hexadecimal
int twoExp5 = 0x20; //hexadecimal
int twoExp6 = 64; //decimal
...
int twoExp31 = -2147483648; //is that a power of 2?
```

Un método simple para calcular la potencia int de 2 sería

```
int pow2(int exp){
    return 1<<exp;
}
```

Comprobando si un número es una potencia de 2

Si un entero `x` es una potencia de 2, solo se establece un bit, mientras que `x-1` tiene todos los bits establecidos después de eso. Por ejemplo: `4` es `100` y `3` es `011` como número binario, que satisface la condición mencionada anteriormente. El cero no es una potencia de 2 y debe comprobarse explícitamente.

```
boolean isPowerOfTwo(int x)
{
    return (x != 0) && ((x & (x - 1)) == 0);
}
```

Uso para turno izquierdo y derecho

Supongamos que tenemos tres tipos de permisos, **LEER** , **ESCRIBIR** y **EJECUTAR** . Cada permiso puede variar de 0 a 7. (Supongamos un sistema de números de 4 bits)

> RECURSO = LEER ESCRIBIR EJECUTAR (número de 12 bits)
>
> RECURSO = 0100 0110 0101 = 4 6 5 (número de 12 bits)

¿Cómo podemos obtener los permisos (número de 12 bits) establecidos anteriormente (número

de 12 bits)?

> 0100 0110 0101
>
> 0000 0000 0111 (&)
>
> 0000 0000 0101 = 5

Entonces, así es como podemos obtener los permisos de **EJECUTAR** del **RECURSO** . Ahora, ¿qué pasa si queremos obtener los permisos de **LEER** del **RECURSO** ?

> 0100 0110 0101
>
> 0111 0000 0000 (&)
>
> 0100 0000 0000 = 1024

¿Derecha? Probablemente estás asumiendo esto? Sin embargo, los permisos se obtienen en 1024. Queremos obtener solo los permisos de LEER para el recurso. No te preocupes, por eso tuvimos los operadores de turno. Si vemos, los permisos de LECTURA están 8 bits detrás del resultado real, por lo tanto, si se aplica algún operador de cambio, ¿cuál llevará los permisos de LECTURA a la derecha del resultado? ¿Qué pasa si hacemos:

> 0100 0000 0000 >> 8 => 0000 0000 0100 (debido a que es un número positivo reemplazado por 0, si no te importa la señal, solo usa el operador de turno derecho sin signo)

Ahora realmente tenemos los permisos de **lectura** que es 4.

Ahora, por ejemplo, tenemos **permisos de LEER** , **ESCRIBIR** , **EJECUTAR** para un **RECURSO** , ¿qué podemos hacer para hacer los permisos para este **RECURSO** ?

Tomemos primero el ejemplo de los permisos binarios. (Todavía asumiendo sistema de número de 4 bits)

> LEER = 0001
>
> ESCRIBIR = 0100
>
> EJECUTAR = 0110

Si estás pensando que simplemente haremos:

`READ | WRITE | EXECUTE` , tienes algo de razón, pero no exactamente. Mira, ¿qué pasará si vamos a realizar LEER | ESCRIBIR | EJECUTAR

0001 | 0100 | 0110 => 0111

Pero los permisos se representan en realidad (en nuestro ejemplo) como 0001 0100 0110

Entonces, para hacer esto, sabemos que **LEER** se coloca 8 bits atrás, **WRITE** se coloca 4 bits

atrás y **PERMISOS** se coloca en el último. El sistema numérico que se usa para los permisos de **RECURSOS** es en realidad 12 bits (en nuestro ejemplo). Puede (será) diferente en diferentes sistemas.

(LEER << 8) | (ESCRIBIR << 4) | (EJECUTAR)

0000 0000 0001 << 8 (LEER)

0001 0000 0000 (desplazamiento a la izquierda por 8 bits)

0000 0000 0100 << 4 (ESCRIBIR)

0000 0100 0000 (desplazamiento a la izquierda por 4 bits)

0000 0000 0001 (EJECUTAR)

Ahora, si agregamos los resultados del cambio anterior, será algo así como;

0001 0000 0000 (LEER)

0000 0100 0000 (ESCRIBIR)

0000 0000 0001 (EJECUTAR)

0001 0100 0001 (PERMISOS)

clase java.util.BitSet

Desde la versión 1.7 hay una clase java.util.BitSet que proporciona una interfaz de manipulación y almacenamiento de bits simple y fácil de usar:

```
final BitSet bitSet = new BitSet(8); // by default all bits are unset

IntStream.range(0, 8).filter(i -> i % 2 == 0).forEach(bitSet::set); // {0, 2, 4, 6}

bitSet.set(3); // {0, 2, 3, 4, 6}

bitSet.set(3, false); // {0, 2, 4, 6}

final boolean b = bitSet.get(3); // b = false

bitSet.flip(6); // {0, 2, 4}

bitSet.set(100); // {0, 2, 4, 100} - expands automatically
```

`BitSet` implementa `Clonable` y `Serializable` , y bajo el capó, todos los valores de bit se almacenan en el campo de `long[] words` , que se expande automáticamente.

También admite operaciones lógicas de conjunto completo `and` , `or` , `xor` , `andNot` :

```
bitSet.and(new BitSet(8));
bitSet.or(new BitSet(8));
bitSet.xor(new BitSet(8));
```

```
bitSet.andNot(new BitSet(8));
```

Cambio firmado / no firmado

En Java, todos los primitivos numéricos están firmados. Por ejemplo, un int siempre representa valores de [-2 ^ 31 - 1, 2 ^ 31], manteniendo el primer bit para firmar el valor: 1 para el valor negativo, 0 para el positivo.

Los operadores de turno básicos `>>` y `<<` son operadores firmados. Conservarán el signo del valor.

Pero es común que los programadores utilicen números para almacenar *valores sin firmar* . Para un int, significa cambiar el rango a [0, 2 ^ 32 - 1], para tener el doble de valor que con un int firmado.

Para aquellos usuarios avanzados, el bit para firmar no tiene significado. Es por eso que Java agregó `>>>` , un operador de cambio a la izquierda, sin tener en cuenta ese bit de signo.

```
                initial value:               4 (                                 100)
     signed left-shift: 4 << 1               8 (                                1000)
    signed right-shift: 4 >> 1               2 (                                  10)
 unsigned right-shift: 4 >>> 1               2 (                                  10)
                initial value:              -4 (   11111111111111111111111111111100)
    signed left-shift: -4 << 1              -8 (   11111111111111111111111111111000)
   signed right-shift: -4 >> 1              -2 (   11111111111111111111111111111110)
unsigned right-shift: -4 >>> 1      2147483646 (    1111111111111111111111111111110)
```

¿Por qué no hay `<<<` ?

Esto viene de la definición pretendida de cambio a la derecha. Como llena los lugares vacíos de la izquierda, no hay ninguna decisión que tomar con respecto al bit de señal. Como consecuencia, no hay necesidad de 2 operadores diferentes.

Vea esta pregunta para una respuesta más detallada.

Lea Manipulación de bits en línea: https://riptutorial.com/es/java/topic/1177/manipulacion-de-bits

Capítulo 126: Mapa débil

Introducción

Conceptos de Hashmap débil

Examples

Conceptos de HashHashmap

Puntos clave:-

- Implementación del Mapa.
- Solo almacena referencias débiles a sus claves.

Referencias débiles : los objetos a los que solo se hace referencia por referencias débiles se recolectan con entusiasmo; El GC no esperará hasta que necesite memoria en ese caso.

Diferencia entre Hashmap y WeakHashMap: -

Si el administrador de memoria Java ya no tiene una referencia segura al objeto especificado como clave, entonces la entrada en el mapa se eliminará en WeakHashMap.

Ejemplo: -

```
public class WeakHashMapTest {
    public static void main(String[] args) {
        Map hashMap= new HashMap();

        Map weakHashMap = new WeakHashMap();

        String keyHashMap = new String("keyHashMap");
        String keyWeakHashMap = new String("keyWeakHashMap");

        hashMap.put(keyHashMap, "Ankita");
        weakHashMap.put(keyWeakHashMap, "Atul");
        System.gc();
        System.out.println("Before: hash map value:"+hashMap.get("keyHashMap")+" and weak hash
map value:"+weakHashMap.get("keyWeakHashMap"));

        keyHashMap = null;
        keyWeakHashMap = null;

        System.gc();

        System.out.println("After: hash map value:"+hashMap.get("keyHashMap")+" and weak hash
map value:"+weakHashMap.get("keyWeakHashMap"));
    }
```

Diferencias de tamaño (HashMap vs WeakHashMap):

El método de tamaño de llamada () en el objeto HashMap devolverá el mismo número de pares clave-valor. el tamaño disminuirá solo si el método remove () se llama explícitamente en el objeto HashMap.

Debido a que el recolector de basura puede descartar claves en cualquier momento, un WeakHashMap puede comportarse como si un hilo desconocido estuviera eliminando entradas silenciosamente. Por lo tanto, es posible que el método de tamaño devuelva valores más pequeños a lo largo del tiempo. Por lo tanto, en **WeakHashMap, la reducción de tamaño ocurre automáticamente** .

Lea Mapa débil en línea: https://riptutorial.com/es/java/topic/10749/mapa-debil

Capítulo 127: Mapa Enum

Introducción

La clase Java EnumMap es la implementación especializada de mapas para claves de enumeración. Hereda las clases Enum y AbstractMap.

los parámetros para la clase java.util.EnumMap.

K: Es el tipo de claves mantenidas por este mapa. V: Es el tipo de valores mapeados.

Examples

Ejemplo de libro de mapas Enum

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}
public class EnumMapExample {
// Creating enum
    public enum Key{
           One, Two, Three
           };
public static void main(String[] args) {
    EnumMap<Key, Book> map = new EnumMap<Key, Book>(Key.class);
    // Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    // Adding Books to Map
       map.put(Key.One, b1);
       map.put(Key.Two, b2);
       map.put(Key.Three, b3);
    // Traversing EnumMap
       for(Map.Entry<Key, Book> entry:map.entrySet()){
            Book b=entry.getValue();
            System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

        }
}
}
```

Lea Mapa Enum en línea: https://riptutorial.com/es/java/topic/10158/mapa-enum

Capítulo 128: Mapas

Introducción

La interfaz java.util.Map representa una asignación entre las claves y sus valores. Un mapa no puede contener claves duplicadas; y cada clave puede asignarse a un máximo de un valor.

Como `Map` es una interfaz, entonces necesita crear una instancia de una implementación concreta de esa interfaz para poder usarla; hay varias implementaciones de `Map` , y en su mayoría se utilizan `java.util.HashMap` y `java.util.TreeMap`

Observaciones

Un *mapa* es un objeto que almacena *claves* con un *valor* asociado para cada clave. Una clave y su valor a veces se llaman un *par clave / valor* o una *entrada* . Los mapas suelen proporcionar estas características:

- Los datos se almacenan en el mapa en pares clave / valor.
- El mapa puede contener solo una entrada para una clave en particular. Si un mapa contiene una entrada con una clave particular e intenta almacenar una segunda entrada con la misma clave, la segunda entrada reemplazará a la primera. En otras palabras, esto cambiará el valor asociado con la clave.
- Los mapas proporcionan operaciones rápidas para probar si existe una clave en el mapa, para recuperar el valor asociado con una clave y para eliminar un par de clave / valor.

La implementación de mapa más utilizada es HashMap . Funciona bien con teclas que son cadenas o números.

Los mapas simples como HashMap no están ordenados. Iterar a través de pares clave / valor puede devolver entradas individuales en cualquier orden. Si necesita recorrer de forma controlada las entradas del mapa, debe mirar lo siguiente:

- Los mapas ordenados , como TreeMap , recorrerán las claves en su orden natural (o en un orden que puede especificar, al proporcionar un Comparador). Por ejemplo, se esperaría que un mapa ordenado que usa números como claves itere a través de sus entradas en orden numérico.

- LinkedHashMap permite iterar a través de entradas en el mismo orden en que se insertaron en el mapa, o por el orden de acceso más reciente.

Examples

Agregar un elemento

1. Adición

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
System.out.println(map.get(1));
```

Salida: `First element.`

2. Anular

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
map.put(1, "New element.");
System.out.println(map.get(1));
```

Salida: `New element.`

`HashMap` se utiliza como ejemplo. También se pueden usar otras implementaciones que implementan la interfaz de `Map` .

Añadir varios elementos

Podemos usar `V put(K key,V value)` :

> Asocia el valor especificado con la clave especificada en este mapa (operación opcional). Si el mapa contenía previamente una asignación para la clave, el valor anterior se reemplaza por el valor especificado.

```
String currentVal;
Map<Integer, String> map = new TreeMap<>();
currentVal = map.put(1, "First element.");
System.out.println(currentVal);// Will print null
currentVal = map.put(2, "Second element.");
System.out.println(currentVal); // Will print null yet again
currentVal = map.put(2, "This will replace 'Second element'");
System.out.println(currentVal); // will print Second element.
System.out.println(map.size()); // Will print 2 as key having
// value 2 was replaced.

Map<Integer, String> map2 = new HashMap<>();
map2.put(2, "Element 2");
map2.put(3, "Element 3");

map.putAll(map2);

System.out.println(map.size());
```

Salida:

```
3
```

Para agregar muchos artículos puedes usar una clase interna como esta:

```
Map<Integer, String> map = new HashMap<>() {{
    // This is now an anonymous inner class with an unnamed instance constructor
    put(5, "high");
```

```
    put(4, "low");
    put(1, "too slow");
}};
```

Tenga en cuenta que crear una clase interna anónima no siempre es eficiente y puede provocar pérdidas de memoria, por lo que, cuando sea posible, use un bloque de inicialización en su lugar:

```
static Map<Integer, String> map = new HashMap<>();

static {
    // Now no inner classes are created so we can avoid memory leaks
    put(5, "high");
    put(4, "low");
    put(1, "too slow");
}
```

El ejemplo anterior hace que el mapa sea estático. También se puede utilizar en un contexto no estático eliminando todas las apariciones de `static` .

Además de que la mayoría de las implementaciones son compatibles con `putAll` , que puede agregar todas las entradas de un mapa a otro de esta manera:

```
another.putAll(one);
```

Usando métodos de mapa predeterminados desde Java 8

Ejemplos de uso de métodos predeterminados introducidos en Java 8 en la interfaz del mapa

1. Utilizando **getOrDefault**

Devuelve el valor asignado a la clave, o si la clave no está presente, devuelve el valor predeterminado

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element");
map.get(1);                                  // => First element
map.get(2);                                  // => null
map.getOrDefault(2, "Default element");      // => Default element
```

2. Usando **forEach**

Permite realizar la operación especificada en la 'acción' en cada entrada de mapa

```
  Map<Integer, String> map = new HashMap<Integer, String>();
  map.put(1, "one");
  map.put(2, "two");
  map.put(3, "three");
  map.forEach((key, value) -> System.out.println("Key: "+key+ " :: Value: "+value));

   // Key: 1 :: Value: one
   // Key: 2 :: Value: two
   // Key: 3 :: Value: three
```

3. Usando **replaceAll**

Reemplazará con nuevo valor solo si la clave está presente

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.replaceAll((key,value)->value+10);   //{john=30, paul=40, peter=50}
```

4. Usando **putIfAbsent**

El par clave-valor se agrega al mapa, si la clave no está presente o asignada a nulo

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);
    map.putIfAbsent("kelly", 50);     //{john=20, paul=30, peter=40, kelly=50}
```

5. Utilizando **eliminar**

   Elimina la clave solo si está asociada con el valor dado

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.remove("peter",40); //{john=30, paul=40}
```

6. Usando **reemplazar**

   Si la clave está presente, el valor se reemplaza por un nuevo valor. Si la llave no está presente, no hace nada.

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.replace("peter",50); //{john=20, paul=30, peter=50}
map.replace("jack",60); //{john=20, paul=30, peter=50}
```

7. Usando **computeIfAbsent**

Este método agrega una entrada en el mapa. la clave se especifica en la función y el valor es el resultado de la aplicación de la función de mapeo

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);
    map.computeIfAbsent("kelly", k->map.get("john")+10); //{john=20, paul=30, peter=40,
kelly=30}
```

```
    map.computeIfAbsent("peter", k->map.get("john")+10); //{john=20, paul=30, peter=40,
kelly=30} //peter already present
```

8. Usando **computeIfPresent**

Este método agrega una entrada o modifica una entrada existente en el Mapa. No hace nada si una entrada con esa clave no está presente.

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);
    map.computeIfPresent("kelly", (k,v)->v+10); //{john=20, paul=30, peter=40} //kelly not
present
    map.computeIfPresent("peter", (k,v)->v+10); //{john=20, paul=30, peter=50} // peter
present, so increase the value
```

9. Utilizando **compute**

Este método reemplaza el valor de una clave por el valor recién calculado

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);
    map.compute("peter", (k,v)->v+50); //{john=20, paul=30, peter=90} //Increase the value
```

10. Usando **fusionar**

Agrega el par clave-valor al mapa, si la clave no está presente o el valor de la clave es nulo Reemplaza el valor con el valor recién calculado, si la clave está presente La clave se elimina del mapa, si el nuevo valor calculado es nulo

```
    Map<String, Integer> map = new HashMap<String, Integer>();
    map.put("john", 20);
    map.put("paul", 30);
    map.put("peter", 40);

    //Adds the key-value pair to the map, if key is not present or value for the key is null
    map.merge("kelly", 50 , (k,v)->map.get("john")+10); // {john=20, paul=30, peter=40,
kelly=50}

    //Replaces the value with the newly computed value, if the key is present
    map.merge("peter", 50 , (k,v)->map.get("john")+10); //{john=20, paul=30, peter=30,
kelly=50}

    //Key is removed from the map , if new value computed is null
    map.merge("peter", 30 , (k,v)->map.get("nancy")); //{john=20, paul=30, kelly=50}
```

Borrar el mapa

```
Map<Integer, String> map = new HashMap<>();
```

```
map.put(1, "First element.");
map.put(2, "Second element.");
map.put(3, "Third element.");

map.clear();

System.out.println(map.size());   // => 0
```

Iterando a través de los contenidos de un Mapa.

Los mapas proporcionan métodos que le permiten acceder a las claves, valores o pares clave-valor del mapa como colecciones. Puedes recorrer estas colecciones. Dado el siguiente mapa por ejemplo:

```
Map<String, Integer> repMap = new HashMap<>();
repMap.put("Jon Skeet", 927_654);
repMap.put("BalusC", 708_826);
repMap.put("Darin Dimitrov", 715_567);
```

Iterando a través de las teclas del mapa:

```
for (String key : repMap.keySet()) {
    System.out.println(key);
}
```

Huellas dactilares:

> Darin Dimitrov
> Jon Skeet
> Balus

`keySet()` proporciona las claves del mapa como un `Set` . `Set` se utiliza porque las claves no pueden contener valores duplicados. Iterando a través del conjunto produce cada tecla a su vez. Los HashMaps no se ordenan, por lo que en este ejemplo, las claves se pueden devolver en cualquier orden.

Iterando a través de los valores del mapa:

```
for (Integer value : repMap.values()) {
    System.out.println(value);
}
```

Huellas dactilares:

> 715567
> 927654
> 708826

`values()` devuelve los valores del mapa como una `Collection` . Iterando a través de la colección produce cada valor a su vez. Nuevamente, los valores pueden ser devueltos en cualquier orden.

Iterando a través de claves y valores juntos.

```
for (Map.Entry<String, Integer> entry : repMap.entrySet()) {
    System.out.printf("%s = %d\n", entry.getKey(), entry.getValue());
}
```

Huellas dactilares:

> Darin Dimitrov = 715567
> Jon Skeet = 927654
> BalusC = 708826

`entrySet()` devuelve una colección de objetos `Map.Entry` . Map.Entry da acceso a la clave y al valor de cada entrada.

Fusionar, combinar y componer mapas.

Use `putAll` para poner a cada miembro de un mapa en otro. Las claves ya presentes en el mapa tendrán sus valores correspondientes sobrescritos.

```
Map<String, Integer> numbers = new HashMap<>();
numbers.put("One", 1)
numbers.put("Three", 3)
Map<String, Integer> other_numbers = new HashMap<>();
other_numbers.put("Two", 2)
other_numbers.put("Three", 4)

numbers.putAll(other_numbers)
```

Esto produce el siguiente mapeo en `numbers` :

```
"One" -> 1
"Two" -> 2
"Three" -> 4 //old value 3 was overwritten by new value 4
```

Si desea combinar valores en lugar de sobrescribirlos, puede usar `Map.merge` , agregado en Java 8, que utiliza un `BiFunction` proporcionado por el `BiFunction` para combinar valores para claves duplicadas. `merge` funciona con claves y valores individuales, por lo que deberá usar un bucle o `Map.forEach` . Aquí concatenamos cadenas para claves duplicadas:

```
for (Map.Entry<String, Integer> e : other_numbers.entrySet())
    numbers.merge(e.getKey(), e.getValue(), Integer::sum);
//or instead of the above loop
other_numbers.forEach((k, v) -> numbers.merge(k, v, Integer::sum));
```

Si desea aplicar la restricción, no hay claves duplicadas, puede usar una función de combinación que lanza un `AssertionError` :

```
mapA.forEach((k, v) ->
    mapB.merge(k, v, (v1, v2) ->
```

```
        {throw new AssertionError("duplicate values for key: "+k);}));
```

Componiendo el Mapa <X, Y> y el Mapa <Y, Z> para obtener el Mapa <X, Z>

Si desea componer dos asignaciones, puede hacerlo de la siguiente manera

```
    Map<String, Integer> map1 = new HashMap<String, Integer>();
    map1.put("key1", 1);
    map1.put("key2", 2);
    map1.put("key3", 3);

    Map<Integer, Double> map2 = new HashMap<Integer, Double>();
    map2.put(1, 1.0);
    map2.put(2, 2.0);
    map2.put(3, 3.0);

    Map<String, Double> map3 = new new HashMap<String, Double>();
    map1.forEach((key,value)->map3.put(key,map2.get(value)));
```

Esto produce el siguiente mapeo

```
    "key1" -> 1.0
    "key2" -> 2.0
    "key3" -> 3.0
```

Comprobar si existe la clave

```
Map<String, String> num = new HashMap<>();
num.put("one", "first");

if (num.containsKey("one")) {
    System.out.println(num.get("one")); // => first
}
```

Los mapas pueden contener valores nulos

Para los mapas, hay que tener cuidado de no confundir "que contiene una clave" con "tener un valor". Por ejemplo, `HashMap` s puede contener null, lo que significa que el siguiente es un comportamiento perfectamente normal:

```
Map<String, String> map = new HashMap<>();
map.put("one", null);
if (map.containsKey("one")) {
    System.out.println("This prints !"); // This line is reached
}
if (map.get("one") != null) {
    System.out.println("This is never reached !"); // This line is never reached
}
```

Más formalmente, no hay garantía de que `map.contains(key) <=> map.get(key)!=null`

Iterando entradas de mapa de manera eficiente

Esta sección proporciona código y puntos de referencia para diez implementaciones de ejemplo únicas que repiten las entradas de un `Map<Integer, Integer>` y generan la suma de los valores de `Integer` . Todos los ejemplos tienen una complejidad algorítmica de `Θ(n)` , sin embargo, los puntos de referencia siguen siendo útiles para proporcionar información sobre qué implementaciones son más eficientes en un entorno del "mundo real".

1. Implementación usando `Iterator` con `Map.Entry`

```
Iterator<Map.Entry<Integer, Integer>> it = map.entrySet().iterator();
while (it.hasNext()) {
    Map.Entry<Integer, Integer> pair = it.next();
    sum += pair.getKey() + pair.getValue();
}
```

2. Implementación utilizando `for` con `Map.Entry`

```
for (Map.Entry<Integer, Integer> pair : map.entrySet()) {
    sum += pair.getKey() + pair.getValue();
}
```

3. Implementación usando `Map.forEach` (Java 8+)

```
map.forEach((k, v) -> sum[0] += k + v);
```

4. Implementación usando `Map.keySet` con `for`

```
for (Integer key : map.keySet()) {
    sum += key + map.get(key);
}
```

5. Implementación usando `Map.keySet` con `Iterator`

```
Iterator<Integer> it = map.keySet().iterator();
while (it.hasNext()) {
    Integer key = it.next();
    sum += key + map.get(key);
}
```

6. Implementación usando `for` con `Iterator` y `Map.Entry`

```
for (Iterator<Map.Entry<Integer, Integer>> entries =
         map.entrySet().iterator(); entries.hasNext(); ) {
    Map.Entry<Integer, Integer> entry = entries.next();
    sum += entry.getKey() + entry.getValue();
}
```

7. Implementación usando `Stream.forEach` (Java 8+)

```
map.entrySet().stream().forEach(e -> sum += e.getKey() + e.getValue());
```

8. Implementación usando `Stream.forEach` con `Stream.parallel` (Java 8+)

```
map.entrySet()
   .stream()
   .parallel()
   .forEach(e -> sum += e.getKey() + e.getValue());
```

9. Implementación utilizando `IterableMap` desde Apache Collections

```
MapIterator<Integer, Integer> mit = iterableMap.mapIterator();
while (mit.hasNext()) {
    sum += mit.next() + it.getValue();
}
```

10. Implementación usando `MutableMap` de Eclipse Collections

```
 mutableMap.forEachKeyValue((key, value) -> {
     sum += key + value;
 });
```

Pruebas de rendimiento (*Código disponible en Github*)
Entorno de prueba: Windows 8.1 de 64 bits, Intel i7-4790 3.60GHz, 16 GB

1. Rendimiento promedio de 10 ensayos (100 elementos) Mejor: 308 ± 21 ns / op

```
Benchmark                           Score     Error  Units
test3_UsingForEachAndJava8            308  ±     21  ns/op
test10_UsingEclipseMutableMap         309  ±      9  ns/op
test1_UsingWhileAndMapEntry           380  ±     14  ns/op
test6_UsingForAndIterator             387  ±     16  ns/op
test2_UsingForEachAndMapEntry         391  ±     23  ns/op
test7_UsingJava8StreamAPI             510  ±     14  ns/op
test9_UsingApacheIterableMap          524  ±      8  ns/op
test4_UsingKeySetAndForEach           816  ±     26  ns/op
test5_UsingKeySetAndIterator          863  ±     25  ns/op
test8_UsingJava8StreamAPIParallel    5552  ±    185  ns/op
```

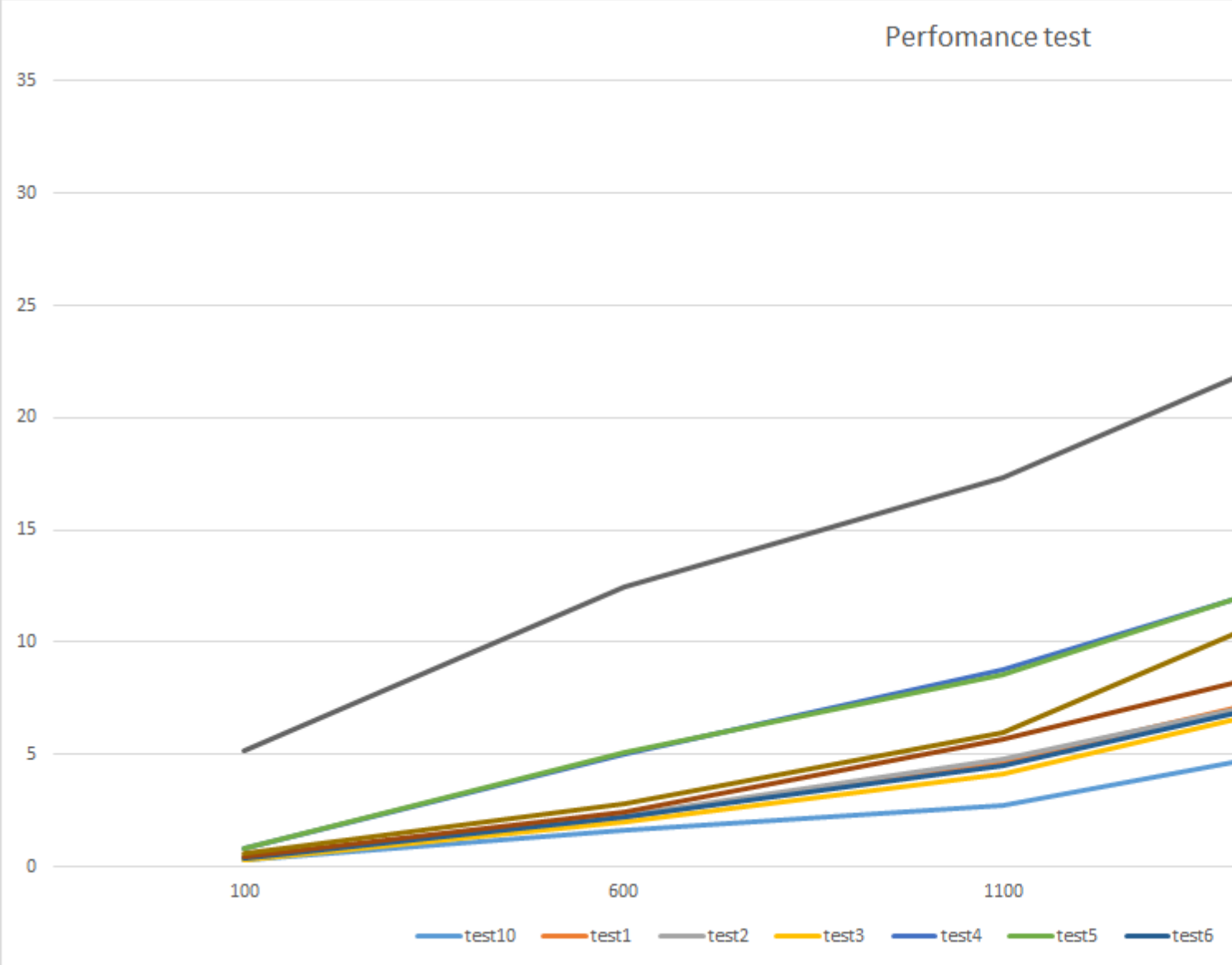
2. Rendimiento promedio de 10 ensayos (10000 elementos) Lo mejor: 37.606 ± 0.790 µs / op

```
Benchmark                           Score     Error  Units
test10_UsingEclipseMutableMap       37606  ±    790  ns/op
test3_UsingForEachAndJava8          50368  ±    887  ns/op
test6_UsingForAndIterator           50332  ±    507  ns/op
test2_UsingForEachAndMapEntry       51406  ±   1032  ns/op
test1_UsingWhileAndMapEntry         52538  ±   2431  ns/op
test7_UsingJava8StreamAPI           54464  ±    712  ns/op
test4_UsingKeySetAndForEach         79016  ±  25345  ns/op
test5_UsingKeySetAndIterator        91105  ±  10220  ns/op
test8_UsingJava8StreamAPIParallel  112511  ±    365  ns/op
test9_UsingApacheIterableMap       125714  ±   1935  ns/op
```

3. Rendimiento promedio de 10 ensayos (100000 elementos) Mejor: 1184.767 ± 332.968 µs / op

```
Benchmark                            Score       Error  Units
test1_UsingWhileAndMapEntry       1184.767  ±  332.968  µs/op
test10_UsingEclipseMutableMap     1191.735  ±  304.273  µs/op
test2_UsingForEachAndMapEntry     1205.815  ±  366.043  µs/op
test6_UsingForAndIterator         1206.873  ±  367.272  µs/op
test8_UsingJava8StreamAPIParallel 1485.895  ±  233.143  µs/op
test5_UsingKeySetAndIterator      1540.281  ±  357.497  µs/op
test4_UsingKeySetAndForEach       1593.342  ±  294.417  µs/op
test3_UsingForEachAndJava8        1666.296  ±  126.443  µs/op
test7_UsingJava8StreamAPI         1706.676  ±  436.867  µs/op
test9_UsingApacheIterableMap      3289.866  ± 1445.564  µs/op
```

4. Una comparación de las variaciones de rendimiento en relación con el tamaño del mapa




```
   x: Size of Map
f(x): Benchmark Score (µs/op)

              100      600     1100     1600     2100
```

```
        ---------------------------------------------------
        10  |   0.333    1.631    2.752    5.937    8.024
         3  |   0.309    1.971    4.147    8.147   10.473
         6  |   0.372    2.190    4.470    8.322   10.531
         1  |   0.405    2.237    4.616    8.645   10.707
Tests    2  |   0.376    2.267    4.809    8.403   10.910
 f(x)    7  |   0.473    2.448    5.668    9.790   12.125
         9  |   0.565    2.830    5.952    13.22   16.965
         4  |   0.808    5.012    8.813    13.939  17.407
         5  |   0.81     5.104    8.533    14.064  17.422
         8  |   5.173   12.499   17.351    24.671  30.403
```

Usar objeto personalizado como clave

Antes de usar su propio objeto como clave, debe anular los métodos hashCode () y equals () de su objeto.

En caso simple tendrías algo como:

```
class MyKey {
    private String name;
    MyKey(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof MyKey) {
            return this.name.equals(((MyKey)obj).name);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.name.hashCode();
    }
}
```

`hashCode` decidirá a qué hash bucket pertenecerá la clave e `equals` determinará qué objeto dentro de ese hash bucket.

Sin estos métodos, la referencia de su objeto se usará para la comparación anterior que no funcionará a menos que use la misma referencia de objeto cada vez.

Uso de HashMap

HashMap es una implementación de la interfaz de mapa que proporciona una estructura de datos para almacenar datos en pares clave-valor.

1. Declarar HashMap

```
Map<KeyType, ValueType> myMap = new HashMap<KeyType, ValueType>();
```

KeyType y ValueType deben ser tipos válidos en Java, como - Cadena, Entero, Flotante o cualquier clase personalizada como Empleado, Estudiante, etc.

Por ejemplo: `Map<String,Integer> myMap = new HashMap<String,Integer>();`

2. Poner valores en HashMap.

Para poner un valor en HashMap, tenemos que llamar al método `put` en el objeto HashMap pasando la clave y el valor como parámetros.

```
myMap.put("key1", 1);
myMap.put("key2", 2);
```

Si llama al método put con la clave que ya existe en el mapa, el método anulará su valor y devolverá el valor anterior.

3. Obtención de valores de HashMap.

Para obtener el valor de un HashMap, debe llamar al método `get` , pasando la clave como parámetro.

```
myMap.get("key1");    //return 1 (class Integer)
```

Si pasa una clave que no existe en el HashMap, este método devolverá `null`

4. Compruebe si la clave está en el mapa o no.

```
myMap.containsKey(varKey);
```

5. Compruebe si el valor está en el mapa o no.

```
myMap.containsValue(varValue);
```

Los métodos anteriores devolverán un valor `boolean` verdadero o falso si la clave, el valor existe en el Mapa o no.

Creación e inicialización de mapas

Introducción

`Maps` almacenan pares clave / valor, donde cada clave tiene un valor asociado. Dada una clave particular, el mapa puede buscar el valor asociado muy rápidamente.

`Maps` , también conocidos como matriz asociada, son un objeto que almacena los datos en forma de claves y valores. En Java, los mapas se representan mediante la interfaz de mapas, que no es una extensión de la interfaz de colección.

- Camino 1: -

```
/*J2SE < 5.0*/
Map map = new HashMap();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- Camino 2: -

```
/*J2SE 5.0+ style (use of generics):*/
Map<String, Object> map = new HashMap<>();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- Camino 3: -

```
Map<String, Object> map = new HashMap<String, Object>(){{
    put("name", "A");
    put("address", "Malviya-Nagar");
    put("city", "Jaipur");
}};
System.out.println(map);
```

- Camino 4: -

```
Map<String, Object> map = new TreeMap<String, Object>();
    map.put("name", "A");
    map.put("address", "Malviya-Nagar");
    map.put("city", "Jaipur");
System.out.println(map);
```

- Camino 5: -

```
//Java 8
final Map<String, String> map =
    Arrays.stream(new String[][] {
        { "name", "A" },
        { "address", "Malviya-Nagar" },
        { "city", "jaipur" },
    }).collect(Collectors.toMap(m -> m[0], m -> m[1]));
System.out.println(map);
```

- Camino 6: -

```
//This way for initial a map in outside the function
final static Map<String, String> map;
static
{
    map = new HashMap<String, String>();
    map.put("a", "b");
```

```
        map.put("c", "d");
    }
```

- Forma 7: - Creando un único mapa de valor-clave inmutable.

```
    //Immutable single key-value map
    Map<String, String> singletonMap = Collections.singletonMap("key", "value");
```

Tenga en cuenta que **es imposible modificar dicho mapa** .

Cualquier intento de modificar el mapa resultará en lanzar la excepción UnsupportedOperationException.

```
    //Immutable single key-value pair
    Map<String, String> singletonMap = Collections.singletonMap("key", "value");
    singletonMap.put("newKey", "newValue"); //will throw UnsupportedOperationException
    singletonMap.putAll(new HashMap<>()); //will throw UnsupportedOperationException
    singletonMap.remove("key"); //will throw UnsupportedOperationException
    singletonMap.replace("key", "value", "newValue"); //will throw
UnsupportedOperationException
    //and etc
```

Lea Mapas en línea: https://riptutorial.com/es/java/topic/105/mapas

Capítulo 129: Máquina virtual de Java (JVM)

Examples

Estos son los fundamentos.

JVM es una **máquina de computación abstracta** o **máquina virtual** que reside en su RAM. Tiene un entorno de ejecución independiente de la plataforma que interpreta el código de bytes de Java en un código de máquina nativo. (Javac es un compilador de Java que compila su código Java en Bytecode)

El programa Java se ejecutará dentro de la JVM que luego se asignará a la máquina física subyacente. Es una de las herramientas de programación en JDK.

(El `Byte code` es un código independiente de la plataforma que se ejecuta en todas las plataformas y el `Machine code` es un código específico de la plataforma que se ejecuta solo en plataformas específicas como Windows o Linux; depende de la ejecución).

Algunos de los componentes: -

- Class Loder - carga el archivo .class en la memoria RAM.
- Verificador de bytecode: verifique si hay violaciones de restricción de acceso en su código.
- Motor de ejecución: convierte el código de byte en código de máquina ejecutable.
- JIT (justo a tiempo): JIT es parte de JVM que solía mejorar el rendimiento de JVM. Compilará o traducirá dinámicamente el código de bytes de Java en código de máquina nativo durante el tiempo de ejecución.

(Editado)

Lea Máquina virtual de Java (JVM) en línea: https://riptutorial.com/es/java/topic/8110/maquina-virtual-de-java--jvm-

Capítulo 130: Método dinámico de envío

Introducción

¿Qué es el método dinámico de envío?

Dynamic Method Dispatch es un proceso en el que la llamada a un método anulado se resuelve en tiempo de ejecución en lugar de en tiempo de compilación. Cuando se llama a un método anulado por una referencia, Java determina qué versión de ese método se ejecutará según el tipo de objeto al que se refiere. Esto también se conoce como polimorfismo de tiempo de ejecución.

Veremos esto a través de un ejemplo.

Observaciones

- Enlace dinámico = Enlace tardío
- Las clases abstractas no pueden ser instanciadas, pero pueden ser subclasificadas (Base para una clase infantil)
- Un método abstracto es un método que se declara sin una implementación
- La clase abstracta puede contener una mezcla de métodos declarados con o sin una implementación
- Cuando una clase abstracta está subclasificada, la subclase generalmente proporciona implementaciones para todos los métodos abstractos en su clase principal. Sin embargo, si no lo hace, entonces la subclase también debe ser declarada abstracta
- El envío de métodos dinámicos es un mecanismo mediante el cual una llamada a un método anulado se resuelve en tiempo de ejecución. Así es como Java implementa el polimorfismo en tiempo de ejecución.
- Upcasting: Convertir un subtipo en un supertipo, hacia arriba en el árbol de herencia.
- Polimorfismo en tiempo de ejecución = polimorfismo dinámico

Examples

Método dinámico de envío - Código de ejemplo

Clase abstracta :

```
package base;

/*
Abstract classes cannot be instantiated, but they can be subclassed
*/
public abstract class ClsVirusScanner {

    //With One Abstract method
    public abstract void fnStartScan();

    protected void fnCheckForUpdateVersion(){
```

```
        System.out.println("Perform Virus Scanner Version Check");
    }

    protected void fnBootTimeScan(){
        System.out.println("Perform BootTime Scan");
    }
    protected void fnInternetSecutiry(){
        System.out.println("Scan for Internet Security");
    }

    protected void fnRealTimeScan(){
        System.out.println("Perform RealTime Scan");
    }

    protected void fnVirusMalwareScan(){
        System.out.println("Detect Virus & Malware");
    }
}
```

Método abstracto superior en la clase infantil:

```
import base.ClsVirusScanner;

//All the 3 child classes inherits the base class ClsVirusScanner
//Child Class 1
class ClsPaidVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnCheckForUpdateVersion();
        super.fnBootTimeScan();
        super.fnInternetSecutiry();
        super.fnRealTimeScan();
        super.fnVirusMalwareScan();
    }
};    //ClsPaidVersion IS-A ClsVirusScanner
//Child Class 2

class ClsTrialVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnInternetSecutiry();
        super.fnVirusMalwareScan();
    }
};    //ClsTrialVersion IS-A ClsVirusScanner

//Child Class 3
class ClsFreeVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnVirusMalwareScan();
    }
};     //ClsTrialVersion IS-A ClsVirusScanner
```

El enlace dinámico / tardío conduce al envío del método dinámico:

```
//Calling Class
public class ClsRunTheApplication {

    public static void main(String[] args) {
```

```
        final String VIRUS_SCANNER_VERSION = "TRIAL_VERSION";

        //Parent Refers Null
        ClsVirusScanner objVS=null;

        //String Cases Supported from Java SE 7
        switch (VIRUS_SCANNER_VERSION){
        case "FREE_VERSION":

            //Parent Refers Child Object 3
            //ClsFreeVersion IS-A ClsVirusScanner
            objVS = new ClsFreeVersion(); //Dynamic or Runtime Binding
            break;
        case "PAID_VERSION":

            //Parent Refers Child Object 1
            //ClsPaidVersion IS-A ClsVirusScanner
            objVS = new ClsPaidVersion(); //Dynamic or Runtime Binding
            break;
        case "TRIAL_VERSION":

            //Parent Refers Child Object 2
            objVS = new ClsTrialVersion(); //Dynamic or Runtime Binding
            break;
        }

        //Method fnStartScan() is the Version of ClsTrialVersion()
        objVS.fnStartScan();

    }
}
```

Resultado:

```
Scan for Internet Security
Detect Virus & Malware
```

Upcasting:

```
objVS = new ClsFreeVersion();
objVS = new ClsPaidVersion();
objVS = new ClsTrialVersion()
```

Lea Método dinámico de envío en línea: https://riptutorial.com/es/java/topic/9204/metodo-dinamico-de-envio

Capítulo 131: Métodos de la fábrica de recolección

Introducción

La llegada de Java 9 trae muchas características nuevas a la API de colecciones de Java, una de las cuales son los métodos de la fábrica de colecciones. Estos métodos permiten una fácil inicialización de colecciones **inmutables** , ya sean vacías o no vacías.

Tenga en cuenta que estos métodos de fábrica solo están disponibles para las siguientes interfaces: `List<E>` , `Set<E>` y `Map<K, V>`

Sintaxis

- `static <E> List<E> of()`
- `static <E> List<E> of(E e1)`
- `static <E> List<E> of(E e1, E e2)`
- `static <E> List<E> of(E e1, E e2, ..., E e9, E e10)`
- `static <E> List<E> of(E... elements)`
- `static <E> Set<E> of()`
- `static <E> Set<E> of(E e1)`
- `static <E> Set<E> of(E e1, E e2)`
- `static <E> Set<E> of(E e1, E e2, ..., E e9, E e10)`
- `static <E> Set<E> of(E... elements)`
- `static <K,V> Map<K,V> of()`
- `static <K,V> Map<K,V> of(K k1, V v1)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2)`
- `static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, ..., K k9, V v9, K k10, V v10)`
- `static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)`

Parámetros

Método con parámetro	Descripción
`List.of(E e)`	Un tipo genérico que puede ser una clase o interfaz.
`Set.of(E e)`	Un tipo genérico que puede ser una clase o interfaz.
`Map.of(K k, V v)`	Un par clave-valor de tipos genéricos que pueden ser una clase o interfaz.
`Map.of(Map.Entry<? extends K, ? extends V> entry)`	Una instancia de `Map.Entry` donde su clave puede ser `K` o uno de sus hijos, y su valor puede ser `V` o cualquiera de sus hijos.

Examples

Lista Ejemplos de métodos de fábrica

- `List<Integer> immutableEmptyList = List.of();`
  - Inicializa una `List<Integer>` vacía e inmutable `List<Integer>` .
- `List<Integer> immutableList = List.of(1, 2, 3, 4, 5);`
  - Inicializa una `List<Integer>` inmutable `List<Integer>` con cinco elementos iniciales.
- `List<Integer> mutableList = new ArrayList<>(immutableList);`
  - Inicializa una `List<Integer>` mutable `List<Integer>` de una `List<Integer>` inmutable `List<Integer>` .

Conjunto Ejemplos de métodos de fábrica

- `Set<Integer> immutableEmptySet = Set.of();`
  - Inicializa un conjunto vacío e inmutable `Set<Integer>` .
- `Set<Integer> immutableSet = Set.of(1, 2, 3, 4, 5);`
  - Inicializa un `Set<Integer>` inmutable `Set<Integer>` con cinco elementos iniciales.
- `Set<Integer> mutableSet = new HashSet<>(immutableSet);`
  - Inicializa un `Set<Integer>` mutable `Set<Integer>` partir de un `Set<Integer>` inmutable `Set<Integer>` .

Mapa Ejemplos de métodos de fábrica

- `Map<Integer, Integer> immutableEmptyMap = Map.of();`
  - Inicializa un `Map<Integer, Integer>` vacío e inmutable `Map<Integer, Integer>` .
- `Map<Integer, Integer> immutableMap = Map.of(1, 2, 3, 4);`
  - Inicializa un `Map<Integer, Integer>` inmutable `Map<Integer, Integer>` con dos entradas iniciales de clave-valor.
- `Map<Integer, Integer> immutableMap = Map.ofEntries(Map.entry(1, 2), Map.entry(3, 4));`
  - Inicializa un `Map<Integer, Integer>` inmutable `Map<Integer, Integer>` con dos entradas iniciales de clave-valor.
- `Map<Integer, Integer> mutableMap = new HashMap<>(immutableMap);`
  - Inicializa un `Map<Integer, Integer>` mutable `Map<Integer, Integer>` partir de un `Map<Integer, Integer>` inmutable `Map<Integer, Integer>` .

Lea Métodos de la fábrica de recolección en línea:
https://riptutorial.com/es/java/topic/9783/metodos-de-la-fabrica-de-recoleccion

Capítulo 132: Métodos predeterminados

Introducción

El método predeterminado introducido en Java 8 permite a los desarrolladores agregar nuevos métodos a una interfaz sin romper las implementaciones existentes de esta interfaz. Proporciona flexibilidad para permitir que la interfaz defina una implementación que se utilizará como predeterminada cuando una clase que implementa esa interfaz no puede proporcionar una implementación de ese método.

Sintaxis

- Nombre del método vacío público predeterminado () {/ * cuerpo del método * /}

Observaciones

Métodos predeterminados

- Puede usarse dentro de una interfaz, para introducir un comportamiento sin forzar las subclases existentes para implementarlo.
- Puede ser reemplazado por subclases o por una sub-interfaz.
- No se les permite anular los métodos en la clase java.lang.Object.
- Si una clase que implementa más de una interfaz, hereda los métodos predeterminados con firmas de métodos idénticas de cada una de las interfaces, debe invalidar y proporcionar su propia interfaz como si no fueran métodos predeterminados (como parte de la resolución de la herencia múltiple).
- A pesar de que están destinadas a introducir un comportamiento sin romper las implementaciones existentes, las subclases existentes con un método estático con la misma firma de método que el método por defecto recién introducido todavía se romperán. Sin embargo, esto es cierto incluso en el caso de introducir un método de instancia en una superclase.

Métodos estáticos

- Puede usarse dentro de una interfaz, principalmente destinada a ser utilizada como un método de utilidad para los métodos predeterminados.
- No puede ser reemplazado por subclases o por una sub-interfaz (está oculto para ellos). Sin embargo, como es el caso de los métodos estáticos, incluso ahora, cada clase o interfaz puede tener la suya propia.
- No se les permite anular los métodos de instancia en la clase java.lang.Object (como en la

actualidad es el caso de las subclases también).

A continuación se muestra una tabla que resume la interacción entre subclase y superclase.

-	SUPER_CLASS-INSTANCE-METHOD	SUPER_CLASS-STATIC-METHOD
SUB_CLASS-INSTANCE-METHOD	*anula*	*genera-compiletime-error*
SUB_CLASS-STATIC-METHOD	*genera-compiletime-error*	*se esconde*

A continuación se muestra una tabla que resume la interacción entre la interfaz y la clase de implementación.

-	INTERFACE-DEFAULT-METHOD	INTERFACE-ESTÁTICO-MÉTODO
IMPL_CLASS-INSTANCE-METHOD	*anula*	*se esconde*
IMPL_CLASS-STATIC-METHOD	*genera-compiletime-error*	*se esconde*

Referencias:

- http://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method
- https://docs.oracle.com/javase/tutorial/java/IandI/override.html

Examples

Uso básico de los métodos por defecto.

```
/**
 * Interface with default method
 */
public interface Printable {
    default void printString() {
        System.out.println( "default implementation" );
    }
}
```

```
/**
 * Class which falls back to default implementation of {@link #printString()}
 */
public class WithDefault
    implements Printable
{
}

/**
 * Custom implementation of {@link #printString()}
 */
public class OverrideDefault
    implements Printable {
    @Override
    public void printString() {
        System.out.println( "overridden implementation" );
    }
}
```

Las siguientes afirmaciones

```
    new WithDefault().printString();
    new OverrideDefault().printString();
```

Producirá esta salida:

```
default implementation
overridden implementation
```

Acceso a otros métodos de interfaz dentro del método predeterminado

También puede acceder a otros métodos de interfaz desde su método predeterminado.

```
public interface Summable {
    int getA();

    int getB();

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {
    @Override
    public int getA() {
        return 1;
    }

    @Override
    public int getB() {
        return 2;
    }
}
```

La siguiente declaración imprimirá *3* :

```
System.out.println(new Sum().calculateSum());
```

Los métodos predeterminados también podrían usarse junto con los métodos de interfaz estáticos:

```
public interface Summable {
    static int getA() {
        return 1;
    }

    static int getB() {
        return 2;
    }

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {}
```

La siguiente declaración también imprimirá 3:

```
System.out.println(new Sum().calculateSum());
```

Accediendo a métodos predeterminados sobrescritos desde la implementación de la clase

En las clases, `super.foo()` buscará solo en superclases. Si desea llamar a una implementación predeterminada desde una superinterfaz, necesita calificar a `super` con el nombre de la interfaz: `Fooable.super.foo()` .

```
public interface Fooable {
    default int foo() {return 3;}
}

public class A extends Object implements Fooable {
    @Override
    public int foo() {
        //return super.foo() + 1; //error: no method foo() in java.lang.Object
        return Fooable.super.foo() + 1; //okay, returns 4
    }
}
```

¿Por qué usar métodos predeterminados?

La respuesta simple es que le permite evolucionar una interfaz existente sin romper las implementaciones existentes.

Por ejemplo, tienes la interfaz `Swim` que publicaste hace 20 años.

```
public interface Swim {
```

```
    void backStroke();
}
```

Hicimos un gran trabajo, nuestra interfaz es muy popular, hay muchas implementaciones en todo el mundo y usted no tiene control sobre su código fuente.

```
public class FooSwimmer implements Swim {
    public void backStroke() {
         System.out.println("Do backstroke");
    }
}
```

Después de 20 años, ha decidido agregar una nueva funcionalidad a la interfaz, pero parece que nuestra interfaz está congelada porque romperá las implementaciones existentes.

Afortunadamente, Java 8 introduce una nueva característica llamada método predeterminado.

Ahora podemos agregar un nuevo método a la interfaz `Swim` .

```
public interface Swim {
    void backStroke();
    default void sideStroke() {
        System.out.println("Default sidestroke implementation. Can be overridden");
    }
}
```

Ahora todas las implementaciones existentes de nuestra interfaz pueden seguir funcionando. Pero lo más importante es que pueden implementar el método recién agregado en su propio tiempo.

Una de las razones más importantes de este cambio, y uno de sus usos más importantes, está en el marco de las Colecciones Java. Oracle no pudo agregar un método `foreach` a la interfaz existente de Iterable sin romper todo el código existente que implementó Iterable. Al agregar métodos predeterminados, la implementación existente de Iterable heredará la implementación predeterminada.

Clase, clase abstracta y prioridad de método de interfaz

Las implementaciones en clases, incluidas las declaraciones abstractas, tienen prioridad sobre todos los valores predeterminados de la interfaz.

- El método de clase abstracta tiene prioridad sobre el método predeterminado de interfaz .

```
public interface Swim {
    default void backStroke() {
        System.out.println("Swim.backStroke");
    }
}

public abstract class AbstractSwimmer implements Swim {
    public void backStroke() {
        System.out.println("AbstractSwimmer.backStroke");
```

```
    }
}

public class FooSwimmer extends AbstractSwimmer {
}
```

La siguiente declaración

```
new FooSwimmer().backStroke();
```

Producirá

```
AbstractSwimmer.backStroke
```

- El método de clase tiene prioridad sobre el método predeterminado de la interfaz

```
public interface Swim {
    default void backStroke() {
        System.out.println("Swim.backStroke");
    }
}

public abstract class AbstractSwimmer implements Swim {
}

public class FooSwimmer extends AbstractSwimmer {
    public void backStroke() {
        System.out.println("FooSwimmer.backStroke");
    }
}
```

La siguiente declaración

```
new FooSwimmer().backStroke();
```

Producirá

```
FooSwimmer.backStroke
```

Método predeterminado colisión de herencia múltiple

Considere el siguiente ejemplo:

```
public interface A {
    default void foo() { System.out.println("A.foo"); }
}

public interface B {
    default void foo() { System.out.println("B.foo"); }
}
```

Aquí hay dos interfaces que declaran `default` método `default foo` con la misma firma.

Si intentas `extend` estas dos interfaces en la nueva interfaz, debes elegir entre dos, porque Java te obliga a resolver esta colisión explícitamente.

Primero , puede declarar el método `foo` con la misma firma que el `abstract` , lo que anulará el comportamiento `A` y `B`

```
public interface ABExtendsAbstract extends A, B {
    @Override
    void foo();
}
```

Y cuando `implement ABExtendsAbstract` en la `class` deberás proporcionar la implementación `foo` :

```
public class ABExtendsAbstractImpl implements ABExtendsAbstract {
    @Override
    public void foo() { System.out.println("ABImpl.foo"); }
}
```

O **segundo** , puede proporcionar una implementación `default` completamente nueva. También puede reutilizar el código de los métodos `A` y `B` `foo` accediendo a los métodos predeterminados reemplazados desde la clase de implementación .

```
public interface ABExtends extends A, B {
    @Override
    default void foo() { System.out.println("ABExtends.foo"); }
}
```

Y cuando `implement ABExtends` en la `class not` tendrá que proporcionar la implementación `foo` :

```
public class ABExtendsImpl implements ABExtends {}
```

Lea Métodos predeterminados en línea: https://riptutorial.com/es/java/topic/113/metodos-predeterminados

Capítulo 133: Modelo de memoria de Java

Observaciones

El modelo de memoria de Java es la sección del JLS que especifica las condiciones bajo las cuales se garantiza que un hilo vea los efectos de las escrituras en memoria realizadas por otro hilo. La sección relevante en las ediciones recientes es "Modelo de memoria JLS 17.4" (en Java 8 , Java 7 , Java 6)

Hubo una revisión importante del modelo de memoria de Java en Java 5 que (entre otras cosas) cambió la forma en que funcionaba la `volatile` . Desde entonces, el modelo de memoria ha sido esencialmente sin cambios.

Examples

Motivación para el modelo de memoria.

Considere el siguiente ejemplo:

```
public class Example {
    public int a, b, c, d;

    public void doIt() {
       a = b + 1;
       c = d + 1;
    }
}
```

Si se usa esta clase es una aplicación de un solo hilo, entonces el comportamiento observable será exactamente el que usted esperaría. Por ejemplo:

```
public class SingleThreaded {
    public static void main(String[] args) {
        Example eg = new Example();
        System.out.println(eg.a + ", " + eg.c);
        eg.doIt();
        System.out.println(eg.a + ", " + eg.c);
    }
}
```

saldrá:

```
0, 0
1, 1
```

En lo que el hilo "principal" puede decir , las declaraciones en el método `main()` y el método `doIt()` se ejecutarán en el orden en que están escritas en el código fuente. Este es un requisito claro de la especificación del lenguaje Java (JLS).

Ahora considere la misma clase utilizada en una aplicación de subprocesos múltiples.

```
public class MultiThreaded {
    public static void main(String[] args) {
        final Example eg = new Example();
        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    eg.doIt();
                }
            }
        }).start();
        while (true) {
            System.out.println(eg.a + ", " + eg.c);
        }
    }
}
```

¿Qué imprimirá esto?

De hecho, según el JLS no es posible predecir que esto se imprimirá:

- Probablemente verá algunas líneas de `0, 0` para empezar.
- Entonces probablemente veas líneas como `N, N` o `N, N + 1` .
- Es posible que veas líneas como `N + 1, N`
- En teoría, podría incluso ver que las líneas `0, 0` continúan para siempre [1] .

1 - En la práctica, la presencia de las sentencias de `println` puede causar una sincronización inesperada y el `println` de la memoria caché. Es probable que oculte algunos de los efectos que podrían causar el comportamiento anterior.

Entonces, ¿cómo podemos explicar esto?

Reordenación de asignaciones

Una posible explicación de los resultados inesperados es que el compilador JIT ha cambiado el orden de las asignaciones en el método `doIt()` . El JLS requiere que las instrucciones *parezcan* ejecutarse en orden *desde la perspectiva del* hilo *actual* . En este caso, nada en el código del método `doIt()` puede observar el efecto de un reordenamiento (hipotético) de esas dos afirmaciones. Esto significa que al compilador JIT se le permitiría hacer eso.

¿Porqué haría eso?

En el hardware moderno típico, las instrucciones de la máquina se ejecutan utilizando un canal de instrucciones que permite que una secuencia de instrucciones se encuentre en diferentes etapas. Algunas fases de la ejecución de instrucciones toman más tiempo que otras, y las operaciones de memoria tienden a tomar más tiempo. Un compilador inteligente puede optimizar el rendimiento de las instrucciones de la tubería al ordenar las instrucciones para maximizar la cantidad de superposición. Esto puede llevar a la ejecución de partes de declaraciones fuera de orden. El JLS permite esto siempre que no afecte el resultado del cálculo *desde la perspectiva del subproceso actual* .

Efectos de cachés de memoria

Una segunda explicación posible es el efecto del almacenamiento en memoria caché. En una arquitectura de computadora clásica, cada procesador tiene un pequeño conjunto de registros y una mayor cantidad de memoria. El acceso a los registros es mucho más rápido que el acceso a la memoria principal. En las arquitecturas modernas, hay cachés de memoria que son más lentos que los registros, pero más rápidos que la memoria principal.

Un compilador explotará esto intentando mantener copias de variables en registros o en las memorias caché. Si no es *necesario* vaciar una variable en la memoria principal, o no es *necesario* leerla desde la memoria, hay ventajas significativas en el rendimiento al no hacer esto. En los casos en que el JLS no requiere que las operaciones de memoria sean visibles a otro hilo, es probable que el compilador JIT de Java no agregue las instrucciones de "barrera de lectura" y "barrera de escritura" que forzarán las lecturas y escrituras de la memoria principal. Una vez más, los beneficios de rendimiento de hacer esto son significativos.

Sincronización adecuada

Hasta ahora, hemos visto que el JLS permite al compilador JIT generar código que hace que el código de un solo subproceso sea más rápido al reordenar o evitar las operaciones de memoria. Pero, ¿qué sucede cuando otros hilos pueden observar el estado de las variables (compartidas) en la memoria principal?

La respuesta es que los otros subprocesos pueden observar estados variables que parecen imposibles ... según el orden del código de las declaraciones de Java. La solución a esto es usar la sincronización apropiada. Los tres enfoques principales son:

- Usando mutexes primitivos y las construcciones `synchronized` .
- Utilizando variables `volatile` .
- Uso de soporte de concurrencia de nivel superior; Por ejemplo, clases en los paquetes `java.util.concurrent` .

Pero incluso con esto, es importante entender dónde se necesita la sincronización y en qué efectos puede confiar. Aquí es donde entra en juego el modelo de memoria Java.

El modelo de memoria

El modelo de memoria de Java es la sección del JLS que especifica las condiciones bajo las cuales se garantiza que un hilo vea los efectos de las escrituras en memoria realizadas por otro hilo. El modelo de memoria se especifica con un cierto grado de *rigor formal* y (como resultado) requiere una lectura detallada y cuidadosa para comprender. Pero el principio básico es que ciertas construcciones crean una relación "sucede antes de" entre la escritura de una variable por un hilo y una lectura posterior de la misma variable por otro hilo. Si la relación "sucede antes" existe, el compilador JIT está *obligado* a generar código que garantice que la operación de lectura vea el valor escrito por la escritura.

Armado con esto, es posible razonar acerca de la coherencia de la memoria en un programa Java y decidir si esto será predecible y consistente para *todas las* plataformas de ejecución.

Relaciones de antes-antes

(La siguiente es una versión simplificada de lo que dice la especificación del lenguaje Java. Para una comprensión más profunda, debe leer la especificación en sí).

Las relaciones Happens-Before son la parte del modelo de memoria que nos permite entender y razonar sobre la visibilidad de la memoria. Como dice el JLS (JLS 17.4.5):

> "Se pueden ordenar dos *acciones* por una relación de *suceso-antes* . Si *ocurre* una acción *-antes de* otra, entonces la primera es visible y ordenada antes de la segunda".

¿Qué significa esto?

Comportamiento

Las acciones a las que se refiere la cita anterior se especifican en JLS 17.4.2 . Hay 5 tipos de acciones enumeradas definidas por la especificación:

- Leer: Leer una variable no volátil.
- Escribir: Escribir una variable no volátil.
- Acciones de sincronización:
  - Lectura volátil: Lectura de una variable volátil.
  - Escritura volátil: Escribiendo una variable volátil.
  - Bloquear. Bloqueo de un monitor
  - Desbloquear. Desbloqueo de un monitor.
  - Las primeras y últimas acciones (sintéticas) de un hilo.
  - Acciones que inician un hilo o detectan que un hilo ha terminado.
- Acciones externas. Una acción que tiene un resultado que depende del entorno en el que se encuentra el programa.
- Acciones de divergencia del hilo. Estos modelan el comportamiento de ciertos tipos de bucle infinito.

Orden del programa y orden de sincronización

Estos dos ordenamientos (JLS 17.4.3 y JLS 17.4.4) rigen la ejecución de sentencias en un Java

El orden del programa describe el orden de ejecución de la declaración dentro de un solo hilo.

El orden de sincronización describe el orden de ejecución de la sentencia para dos sentencias conectadas mediante una sincronización:

- Una acción de desbloqueo en el monitor se *sincroniza con* todas las acciones de bloqueo subsiguientes en ese monitor.
- Una escritura en una variable volátil se *sincroniza con* todas las lecturas posteriores de la misma variable por cualquier hilo.
- Una acción que inicia un hilo (es decir, la llamada a `Thread.start()`) se *sincroniza con* la primera acción en el hilo que comienza (es decir, la llamada al método `run()` del hilo).
- La inicialización predeterminada de los campos se *sincroniza con* la primera acción en cada hilo. (Vea el JLS para una explicación de esto.)
- La acción final en un hilo se *sincroniza con* cualquier acción en otro hilo que detecta la terminación; por ejemplo, la devolución de una llamada `join()` o `isTerminated()` llamada que devuelve `true` .
- Si un hilo interrumpe otro hilo, la llamada de interrupción en el primer hilo se *sincroniza con* el punto donde otro hilo detecta que el hilo se interrumpió.

Sucede antes de la orden

Este ordenamiento (JLS 17.4.5) es lo que determina si se garantiza que una escritura de memoria sea visible para una lectura de memoria posterior.

Más específicamente, se garantiza que una lectura de una variable `v` observe una escritura en `v` si y solo si *sucede la* `write(v)` *- antes de la* `read(v)` Y no hay escritura `v` a la `v` . Si hay escrituras intermedias, entonces la `read(v)` puede ver los resultados de ellas en lugar de la anterior.

Las reglas que definen el *suceso antes de* ordenar son las siguientes:

- **Regla Happens-Before # 1** : si x e y son acciones del mismo hilo y x aparece antes de y en el *orden del programa* , entonces x *sucede antes de* y.
- **Regla Happens-Before # 2** : hay un borde antes del evento desde el final de un constructor de un objeto hasta el inicio de un finalizador para ese objeto.
- **Happens-Before Rule # 3** - Si una acción x se *sincroniza con* una acción subsiguiente y, entonces x *sucede-* y.
- **Happens-Before Rule # 4** - Si x *sucede antes de* y y y *sucede antes de* z, entonces x *sucede antes de* z.

Además, varias clases en las bibliotecas estándar de Java se especifican para definir relaciones de *suceso antes* . Puede interpretar que esto significa que sucede de *alguna manera* , sin

necesidad de saber exactamente cómo se va a cumplir la garantía.

Razonamiento antes del razonamiento aplicado a algunos ejemplos.

Presentaremos algunos ejemplos para mostrar cómo aplicar el *suceso antes del* razonamiento para verificar que las escrituras sean visibles en las lecturas posteriores.

Código de un solo hilo

Como es de esperar, las escrituras siempre son visibles para lecturas posteriores en un programa de un solo hilo.

```
public class SingleThreadExample {
    public int a, b;

    public int add() {
       a = 1;         // write(a)
       b = 2;         // write(b)
       return a + b;  // read(a) followed by read(b)
    }
}
```

Por lo que sucede antes de la regla # 1:

1. La acción de `write(a)` *ocurre antes de* la acción *de* `write(b)` .
2. La acción de `write(b)` *ocurre antes de* la acción *de* `read(a)` .
3. La acción de `read(a)` *ocurre antes de* la acción *de* `read(a)` .

Por lo que sucede antes de la regla # 4:

4. `write(a)` *sucede antes de* `write(b)` y `write(b)` *sucede antes de* `read(a)` IMPLICA `write(a)` *sucede antes de* `read(a)` .
5. `write(b)` *sucede antes de* `read(a)` y `read(a)` *sucede antes de* `read(b)` IMPLICA `write(b)` *sucede antes de* `read(b)` .

Resumiendo:

6. La relación `write(a)` *sucede antes de* `read(a)` significa que se garantiza que la declaración a `a + b` ve el valor correcto de `a` .
7. La relación `write(b)` *sucede antes de* `read(b)` significa que se garantiza que la declaración a `a + b` ve el valor correcto de `b` .

Comportamiento de 'volátil' en un ejemplo con 2 hilos

Usaremos el siguiente código de ejemplo para explorar algunas implicaciones del modelo de memoria para `volatile.

```
public class VolatileExample {
    private volatile int a;
```

```
    private int b;         // NOT volatile

    public void update(int first, int second) {
       b = first;         // write(b)
       a = second;         // write-volatile(a)
    }

    public int observe() {
       return a + b;       // read-volatile(a) followed by read(b)
    }
}
```

Primero, considere la siguiente secuencia de declaraciones que involucran 2 hilos:

1. Se crea una instancia única de `VolatileExample` ; llámalo `ve`
2. `ve.update(1, 2)` se llama en un hilo, y
3. `ve.observe()` se llama en otro hilo.

Por lo que sucede antes de la regla # 1:

1. La acción de `write(a)` *ocurre antes de* la acción `volatile-write(a)` .
2. La acción de `volatile-read(a)` *ocurre antes de* la acción *de* `read(b)` .

Por lo que sucede antes de la regla 2:

3. La acción de `volatile-write(a)` en el primer subproceso *ocurre antes de* la acción de `volatile-read(a)` en el segundo subproceso.

Por lo que sucede antes de la regla # 4:

4. La acción de `write(b)` en el primer hilo *ocurre antes de* la acción de `read(b)` en el segundo hilo.

En otras palabras, para esta secuencia en particular, tenemos la garantía de que el segundo subproceso verá la actualización de la variable no volátil `b` realizada por el primer subproceso. Sin embargo, también debe quedar claro que si las asignaciones en el método de `update` fueran al revés, o el método de `observe()` leyera la variable `b` antes de `a` , entonces la cadena de *suceso antes* se rompería. La cadena también se rompería si `volatile-read(a)` en el segundo hilo no fuera posterior a la `volatile-write(a)` en el primer hilo.

Cuando se rompe la cadena, no hay *garantía de* que `observe()` verá el valor correcto de `b` .

Volátil con tres hilos.

Supongamos que agregamos un tercer hilo en el ejemplo anterior:

1. Se crea una instancia única de `VolatileExample` ; llámalo `ve`
2. `update` llamada a dos hilos:
   - `ve.update(1, 2)` se llama en un hilo,
   - `ve.update(3, 4)` se llama en el segundo hilo,
3. `ve.observe()` se llama posteriormente en un tercer hilo.

Para analizar esto completamente, debemos considerar todas las posibles interrelaciones de las declaraciones en el subproceso uno y el subproceso dos. En su lugar, consideraremos sólo dos de ellos.

Escenario # 1: supongamos que la `update(1, 2)` precede a la `update(3,4)` obtenemos esta secuencia:

```
write(b, 1), write-volatile(a, 2)     // first thread
write(b, 3), write-volatile(a, 4)     // second thread
read-volatile(a), read(b)             // third thread
```

En este caso, es fácil ver que hay una cadena de `write(b, 3)` *antes* de `write(b, 3)` para `read(b)` . Además, no hay que intervenir escribir a `b` . Por lo tanto, para este escenario, se garantiza que el tercer subproceso verá que `b` tiene valor `3` .

Escenario n. ° 2: supongamos que la `update(1, 2)` y la `update(3,4)` superponen y las condiciones se entrelazan de la siguiente manera:

```
write(b, 3)                           // second thread
write(b, 1)                           // first thread
write-volatile(a, 2)                  // first thread
write-volatile(a, 4)                  // second thread
read-volatile(a), read(b)             // third thread
```

Ahora, mientras hay una cadena de `write(b, 3)` *antes* de `write(b, 3)` para `read(b)` , hay una acción de `write(b, 1)` interviene por el otro hilo. Esto significa que no podemos estar seguros de qué valor `read(b)` .

(Aparte: Esto demuestra que no podemos confiar en la `volatile` para garantizar la visibilidad de las variables no volátiles, excepto en situaciones muy limitadas).

Cómo evitar tener que entender el modelo de memoria.

El modelo de memoria es difícil de entender y difícil de aplicar. Es útil si necesita razonar acerca de la corrección del código de subprocesos múltiples, pero no desea tener que hacer este razonamiento para cada aplicación de subprocesos múltiples que escriba.

Si adopta los siguientes principios al escribir código concurrente en Java, puede evitar en *gran medida* la necesidad de recurrir al *suceso antes del* razonamiento.

- Utilice estructuras de datos inmutables siempre que sea posible. Una clase inmutable implementada correctamente será segura para subprocesos y no presentará problemas de seguridad de subprocesos cuando la use con otras clases.
- Entender y evitar "publicación insegura".
- Use mutexes primitivos u objetos `Lock` para sincronizar el acceso al estado en objetos mutables que necesitan ser seguros para subprocesos [1] .
- Use `Executor` / `ExecutorService` o el framework de unión de fork en lugar de intentar crear

hilos de administración directamente.

- Utilice las clases `java.util.concurrent que proporcionan bloqueos avanzados, semáforos, cierres y barreras, en lugar de usar esperar / notificar / notificar a todos directamente.
- Utilice las versiones `java.util.concurrent` de mapas, conjuntos, listas, colas y deques en lugar de la sincronización externa de colecciones no concurrentes.

El principio general es tratar de usar las bibliotecas de concurrencia incorporadas de Java en lugar de "rodar su propia concurrencia". Puede confiar en que funcionen, si los usa correctamente.

---

1 - No todos los objetos deben estar a salvo de hilos. Por ejemplo, si un objeto u objetos están *limitados* a un hilo (es decir, solo es accesible a un hilo), entonces su seguridad no es relevante.

Lea Modelo de memoria de Java en línea: https://riptutorial.com/es/java/topic/6829/modelo-de-memoria-de-java

Capítulo 134: Modificación Bytecode

Examples

¿Qué es Bytecode?

Bytecode es el conjunto de instrucciones utilizadas por la JVM. Para ilustrar esto tomemos este programa Hello World.

```
public static void main(String[] args){
    System.out.println("Hello World");
}
```

Esto es en lo que se convierte cuando se compila en bytecode.

```
public static main([Ljava/lang/String; args)V
    getstatic java/lang/System out Ljava/io/PrintStream;
    ldc "Hello World"
    invokevirtual java/io/PrintStream print(Ljava/lang/String;)V
```

¿Cuál es la lógica detrás de esto?

getstatic : **recupera** el valor de un campo estático de una clase. En este caso, el *PrintStream "Out"* del *sistema* .

ldc - **Empuja** una constante en la pila. En este caso, el String "Hello World".

invokevirtual : **invoca** un método en una referencia cargada en la pila y coloca el resultado en la pila. Los parámetros del método también se toman de la pila.

Bueno, tiene que haber más derecho?

Hay 255 códigos de operación, pero no todos están implementados todavía. Una tabla con todos los códigos de operación actuales se puede encontrar aquí: listas de instrucciones de bytecode de Java .

¿Cómo puedo escribir / editar el bytecode?

Hay varias formas de escribir y editar el código de bytes. Puede utilizar un compilador, una biblioteca o un programa.

Para la escritura:

- Jazmín
- Krakatau

Para la edición:

- Bibliotecas
  - ASM
  - Javassista
  - BCEL - *No soporta Java 8+*
- Herramientas
  - Bytecode-Viewer
  - JBytedit
  - reJ - *No soporta Java 8+*
  - JBE - *No soporta Java 8+*

¡Me gustaría aprender más sobre el bytecode!

Probablemente hay una página de documentación específica específicamente para el código de bytes. Esta página se enfoca en la modificación del código de bytes usando diferentes bibliotecas y herramientas.

Cómo editar archivos jar con ASM

En primer lugar las clases del tarro deben ser cargadas. Usaremos tres métodos para este proceso:

- loadClasses (Archivo)
- readJar (JarFile, JarEntry, Mapa)
- getNode (byte [])

```
Map<String, ClassNode> loadClasses(File jarFile) throws IOException {
    Map<String, ClassNode> classes = new HashMap<String, ClassNode>();
    JarFile jar = new JarFile(jarFile);
    Stream<JarEntry> str = jar.stream();
    str.forEach(z -> readJar(jar, z, classes));
    jar.close();
    return classes;
}

Map<String, ClassNode> readJar(JarFile jar, JarEntry entry, Map<String, ClassNode> classes) {
    String name = entry.getName();
    try (InputStream jis = jar.getInputStream(entry)){
        if (name.endsWith(".class")) {
            byte[] bytes = IOUtils.toByteArray(jis);
            String cafebabe = String.format("%02X%02X%02X%02X", bytes[0], bytes[1], bytes[2],
bytes[3]);
            if (!cafebabe.toLowerCase().equals("cafebabe")) {
                // This class doesn't have a valid magic
```

```
                return classes;
            }
            try {
                ClassNode cn = getNode(bytes);
                classes.put(cn.name, cn);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return classes;
}

ClassNode getNode(byte[] bytes) {
    ClassReader cr = new ClassReader(bytes);
    ClassNode cn = new ClassNode();
    try {
        cr.accept(cn, ClassReader.EXPAND_FRAMES);
    } catch (Exception e) {
        e.printStackTrace();
    }
    cr = null;
    return cn;
}
```

Con estos métodos, cargar y cambiar un archivo jar se convierte en una simple cuestión de cambiar ClassNodes en un mapa. En este ejemplo, reemplazaremos todas las cadenas en el contenedor por unas en mayúsculas usando la API del árbol.

```
File jarFile = new File("sample.jar");
Map<String, ClassNode> nodes = loadClasses(jarFile);
// Iterate ClassNodes
for (ClassNode cn : nodes.values()){
    // Iterate methods in class
    for (MethodNode mn : cn.methods){
        // Iterate instructions in method
        for (AbstractInsnNode ain : mn.instructions.toArray()){
            // If the instruction is loading a constant value
            if (ain.getOpcode() == Opcodes.LDC){
                // Cast current instruction to Ldc
                // If the constant is a string then capitalize it.
                LdcInsnNode ldc = (LdcInsnNode) ain;
                if (ldc.cst instanceof String){
                    ldc.cst = ldc.cst.toString().toUpperCase();
                }
            }
        }
    }
}
```

Ahora que se han modificado todas las cadenas de ClassNode, debemos guardar los cambios. Para guardar los cambios y tener una salida de trabajo, hay que hacer algunas cosas:

- Exportar ClassNodes a bytes
- Cargar entradas de jar que no sean de clase *(Ej: Manifest.mf / otros recursos binarios en jar)*

como bytes
- Guarda todos los bytes en un nuevo jar

De la última parte de arriba, crearemos tres métodos.

- processNodes (Mapa <String, ClassNode> nodos)
- loadNonClasses (archivo jarFile)
- saveAsJar (Map <String, byte []> outBytes, String fileName)

Uso:

```
Map<String, byte[]> out = process(nodes, new HashMap<String, MappedClass>());
out.putAll(loadNonClassEntries(jarFile));
saveAsJar(out, "sample-edit.jar");
```

Los métodos utilizados:

```
static Map<String, byte[]> processNodes(Map<String, ClassNode> nodes, Map<String, MappedClass>
mappings) {
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    // Iterate nodes and add them to the map of <Class names , Class bytes>
    // Using Compute_Frames ensures that stack-frames will be re-calculated automatically
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name).getNewName() : cn.name,
cw.toByteArray());
    }
    return out;
}

static Map<String, byte[]> loadNonClasses(File jarFile) throws IOException {
    Map<String, byte[]> entries = new HashMap<String, byte[]>();
    ZipInputStream jis = new ZipInputStream(new FileInputStream(jarFile));
    ZipEntry entry;
    // Iterate all entries
    while ((entry = jis.getNextEntry()) != null) {
        try {
            String name = entry.getName();
            if (!name.endsWith(".class") && !entry.isDirectory()) {
                // Apache Commons - byte[] toByteArray(InputStream input)
                //
                // Add each entry to the map <Entry name , Entry bytes>
                byte[] bytes = IOUtils.toByteArray(jis);
                entries.put(name, bytes);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            jis.closeEntry();
        }
    }
    jis.close();
    return entries;
}

static void saveAsJar(Map<String, byte[]> outBytes, String fileName) {
    try {
```

```
        // Create jar output stream
        JarOutputStream out = new JarOutputStream(new FileOutputStream(fileName));
        // For each entry in the map, save the bytes
        for (String entry : outBytes.keySet()) {
            // Appent class names to class entries
            String ext = entry.contains(".") ? "" : ".class";
            out.putNextEntry(new ZipEntry(entry + ext));
            out.write(outBytes.get(entry));
            out.closeEntry();
        }
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Eso es. Todos los cambios se guardarán en "sample-edit.jar".

Cómo cargar un ClassNode como una clase

```
/**
 * Load a class by from a ClassNode
 *
 * @param cn
 *            ClassNode to load
 * @return
 */
public static Class<?> load(ClassNode cn) {
    ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
    return new ClassDefiner(ClassLoader.getSystemClassLoader()).get(cn.name.replace("/", "."),
cw.toByteArray());
}

/**
 * Classloader that loads a class from bytes.
 */
static class ClassDefiner extends ClassLoader {
    public ClassDefiner(ClassLoader parent) {
        super(parent);
    }

    public Class<?> get(String name, byte[] bytes) {
        Class<?> c = defineClass(name, bytes, 0, bytes.length);
        resolveClass(c);
        return c;
    }
}
```

Cómo cambiar el nombre de las clases en un archivo jar

```
public static void main(String[] args) throws Exception {
    File jarFile = new File("Input.jar");
    Map<String, ClassNode> nodes = JarUtils.loadClasses(jarFile);

    Map<String, byte[]> out = JarUtils.loadNonClassEntries(jarFile);
    Map<String, String> mappings = new HashMap<String, String>();
    mappings.put("me/example/ExampleClass", "me/example/ExampleRenamed");
```

```
    out.putAll(process(nodes, mappings));
    JarUtils.saveAsJar(out, "Input-new.jar");
}

static Map<String, byte[]> process(Map<String, ClassNode> nodes, Map<String, String> mappings)
{
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    Remapper mapper = new SimpleRemapper(mappings);
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        ClassVisitor remapper = new ClassRemapper(cw, mapper);
        cn.accept(remapper);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name) : cn.name,
cw.toByteArray());
    }
    return out;
}
```

SimpleRemapper es una clase existente en la biblioteca ASM. Sin embargo, solo permite que se cambien los nombres de clase. Si desea cambiar el nombre de campos y métodos, debe crear su propia implementación de la clase Remapper.

Javassist Basic

Javassist es una biblioteca de instrumentación de bytecode que le permite modificar el código de Java de inyección de bytecode que se convertirá en bytecode por Javassist y se agregará a la clase / método instrumentado en el tiempo de ejecución.

Permite escribir el primer transformador que realmente tome una clase hipotética "com.my.to.be.instrumented.MyClass" y agregar a las instrucciones de cada método una llamada de registro.

```
import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.security.ProtectionDomain;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;

public class DynamicTransformer implements ClassFileTransformer {

    public byte[] transform(ClassLoader loader, String className, Class classBeingRedefined,
        ProtectionDomain protectionDomain, byte[] classfileBuffer) throws
IllegalClassFormatException {

        byte[] byteCode = classfileBuffer;

        // into the transformer will arrive every class loaded so we filter
        // to match only what we need
        if (className.equals("com/my/to/be/instrumented/MyClass")) {

            try {
                // retrive default Javassist class pool
                ClassPool cp = ClassPool.getDefault();
                // get from the class pool our class with this qualified name
                CtClass cc = cp.get("com.my.to.be.instrumented.MyClass");
```

```
                // get all the methods of the retrieved class
                CtMethod[] methods = cc.getDeclaredMethods()
                for(CtMethod meth : methods) {
                    // The instrumentation code to be returned and injected
                    final StringBuffer buffer = new StringBuffer();
                    String name = meth.getName();
                    // just print into the buffer a log for example
                    buffer.append("System.out.println(\"Method " + name + " executed\" );");
                    meth.insertBefore(buffer.toString())
                }
                // create the byteclode of the class
                byteCode = cc.toBytecode();
                // remove the CtClass from the ClassPool
                cc.detach();
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }

        return byteCode;
    }
}
```

Ahora, para usar este transformador (para que nuestra JVM invoque la transformación del método en cada clase en el momento de la carga) necesitamos agregar este instrumento o esto con un agente:

```
import java.lang.instrument.Instrumentation;

public class EasyAgent {

    public static void premain(String agentArgs, Instrumentation inst) {

        // registers the transformer
        inst.addTransformer(new DynamicTransformer());
    }
}
```

El último paso para iniciar nuestro primer experimento de instrumentación es registrar esta clase de agente en la ejecución de la máquina JVM. La forma más fácil de hacerlo es registrarlo con una opción en el comando de shell:

```
java -javaagent:myAgent.jar MyJavaApplication
```

Como podemos ver, el proyecto de agente / transformador se agrega como un jar para la ejecución de cualquier aplicación llamada MyJavaApplication que se supone que contiene una clase llamada "com.my.to.be.instrumented.MyClass" para ejecutar nuestro código inyectado.

Lea Modificación Bytecode en línea: https://riptutorial.com/es/java/topic/3747/modificacion-bytecode

Capítulo 135: Modificadores sin acceso

Introducción

Los modificadores ***que no son de*** acceso ***no cambian la accesibilidad de las variables*** y los métodos, pero sí les proporcionan ***propiedades especiales*** .

Examples

final

`final` en Java puede referirse a variables, métodos y clases. Hay tres reglas simples:

- la variable final no puede ser reasignada
- El método final no puede ser anulado
- la clase final no puede ser extendida

Usos

Buena práctica de programación

Algunos desarrolladores consideran una buena práctica marcar una variable final cuando puedas. Si tiene una variable que no debe cambiarse, debe marcarla como final.

Un uso importante de `final` palabra clave `final` si para los parámetros del método. Si desea enfatizar que un método no cambia sus parámetros de entrada, marque las propiedades como finales.

```
public int sumup(final List<Integer> ints);
```

Esto enfatiza que el método de `sumup` no va a cambiar los `ints` .

Acceso a la clase interna

Si su clase interna anónima quiere acceder a una variable, la variable debe marcarse como `final`

```
  public IPrintName printName(){
    String name;
    return new IPrintName(){
        @Override
        public void printName(){
            System.out.println(name);
        }
    };
}
```

Esta clase no compila, ya que el `name` la variable, no es final.

Java SE 8

Efectivamente las variables finales son una excepción. Estas son variables locales que se escriben una sola vez y, por lo tanto, podrían ser definitivas. Efectivamente, también se puede acceder a las variables finales desde clases anónimas.

variable `final static`

Aunque el código a continuación es completamente legal cuando `final` variable `final foo` no es `static` , en caso de `static` no compilará:

```
class TestFinal {
    private final static List foo;

    public Test() {
        foo = new ArrayList();
    }
}
```

La razón es, repitámoslo de nuevo, *la variable final no puede ser reasignada* . Como `foo` es estático, se comparte entre todas las instancias de la clase `TestFinal` . Cuando se crea una nueva instancia de una clase `TestFinal` , se invoca su constructor y, por lo tanto, foo se reasigna y el compilador no lo permite. Una forma correcta de inicializar la variable `foo` en este caso es:

```
class TestFinal {
    private static final List foo = new ArrayList();
    //..
}
```

o utilizando un inicializador estático:

```
class TestFinal {
    private static final List foo;
    static {
        foo = new ArrayList();
    }
    //..
}
```

`final` métodos `final` son útiles cuando la clase base implementa alguna funcionalidad importante que la clase derivada no debe cambiar. También son más rápidos que los métodos no finales, porque no hay un concepto de tabla virtual involucrado.

Todas las clases de envoltorio en Java son finales, como `Integer` , `Long` , etc. Los creadores de estas clases no querían que nadie pudiera, por ejemplo, extender Integer a su propia clase y cambiar el comportamiento básico de la clase Integer. Uno de los requisitos para hacer que una clase sea inmutable es que las subclases no pueden anular los métodos. La forma más sencilla de hacer esto es declarar la clase como `final` .

volátil

El modificador `volatile` se utiliza en la programación de múltiples hilos. Si declara que un campo es `volatile` , es una señal a los subprocesos que deben leer el valor más reciente, no uno

almacenado en caché local. Además, se garantiza que `volatile` lecturas y escrituras `volatile` son atómicas (el acceso a un `long` o `double` no `volatile` no es atómico), lo que evita ciertos errores de lectura / escritura entre varios subprocesos.

```
public class MyRunnable implements Runnable
{
    private volatile boolean active;

    public void run(){ // run is called in one thread
        active = true;
        while (active){
            // some code here
        }
    }

    public void stop(){ // stop() is called from another thread
        active = false;
    }
}
```

estático

La palabra clave `static` se usa en una clase, método o campo para hacer que funcionen independientemente de cualquier instancia de la clase.

- Los campos estáticos son comunes a todas las instancias de una clase. No necesitan una instancia para acceder a ellos.
- Los métodos estáticos se pueden ejecutar sin una instancia de la clase en la que se encuentran. Sin embargo, solo pueden acceder a los campos estáticos de esa clase.
- Las clases estáticas pueden ser declaradas dentro de otras clases. No necesitan una instancia de la clase en la que están para ser instanciados.

```
public class TestStatic
{
    static int staticVariable;

    static {
        // This block of code is run when the class first loads
        staticVariable = 11;
    }

    int nonStaticVariable = 5;

    static void doSomething() {
        // We can access static variables from static methods
        staticVariable = 10;
    }

    void add() {
        // We can access both static and non-static variables from non-static methods
        nonStaticVariable += staticVariable;
    }

    static class StaticInnerClass {
        int number;
```

```
        public StaticInnerClass(int _number) {
            number = _number;
        }

        void doSomething() {
            // We can access number and staticVariable, but not nonStaticVariable
            number += staticVariable;
        }

        int getNumber() {
            return number;
        }
    }
}


// Static fields and methods
TestStatic object1 = new TestStatic();

System.out.println(object1.staticVariable); // 11
System.out.println(TestStatic.staticVariable); // 11

TestStatic.doSomething();

TestStatic object2 = new TestStatic();

System.out.println(object1.staticVariable); // 10
System.out.println(object2.staticVariable); // 10
System.out.println(TestStatic.staticVariable); // 10

object1.add();

System.out.println(object1.nonStaticVariable); // 15
System.out.println(object2.nonStaticVariable); // 10

// Static inner classes
StaticInnerClass object3 = new TestStatic.StaticInnerClass(100);
StaticInnerClass object4 = new TestStatic.StaticInnerClass(200);

System.out.println(object3.getNumber()); // 100
System.out.println(object4.getNumber()); // 200

object3.doSomething();

System.out.println(object3.getNumber()); // 110
System.out.println(object4.getNumber()); // 200
```

resumen

La abstracción es un proceso de ocultar los detalles de la implementación y mostrar solo la funcionalidad al usuario. Una clase abstracta nunca puede ser instanciada. Si una clase se declara como abstracta, el único propósito es que la clase se extienda.

```
abstract class Car
{
    abstract void tagLine();
}
```

```
class Honda extends Car
{
    void tagLine()
    {
        System.out.println("Start Something Special");
    }
}

class Toyota extends Car
{
    void tagLine()
    {
        System.out.println("Drive Your Dreams");
    }
}
```

sincronizado

El modificador sincronizado se usa para controlar el acceso de un método en particular o un bloque por varios subprocesos. Solo un hilo puede entrar en un método o un bloque que se declara como sincronizado. la palabra clave sincronizada funciona en el bloqueo intrínseco de un objeto, en el caso de un método sincronizado, los objetos actuales se bloquean y el método estático utiliza el objeto de clase. Cualquier hilo que intente ejecutar un bloque sincronizado debe adquirir primero el bloqueo del objeto.

```
class Shared
{
    int i;

    synchronized void SharedMethod()
    {
        Thread t = Thread.currentThread();

        for(int i = 0; i <= 1000; i++)
        {
            System.out.println(t.getName()+" : "+i);
        }
    }

    void SharedMethod2()
    {
        synchronized (this)
        {
            System.out.println("Thais access to currect object is synchronize "+this);
        }
    }
}

public class ThreadsInJava
{
    public static void main(String[] args)
    {
        final Shared s1 = new Shared();

        Thread t1 = new Thread("Thread - 1")
        {
            @Override
```

```
            public void run()
            {
                s1.SharedMethod();
            }
        };

        Thread t2 = new Thread("Thread - 2")
        {
            @Override
            public void run()
            {
                s1.SharedMethod();
            }
        };

        t1.start();

        t2.start();
    }
}
```

transitorio

Una variable que se declara como transitoria no se serializará durante la serialización de objetos.

```
public transient int limit = 55;   // will not persist
public int b; // will persist
```

strictfp

Java SE 1.2

El modificador strictfp se usa para cálculos de punto flotante. Este modificador hace que la variable de punto flotante sea más consistente en múltiples plataformas y garantiza que todos los cálculos de punto flotante se realicen de acuerdo con los estándares IEEE 754 para evitar errores de cálculo (errores de redondeo), desbordamientos y subflujos en la arquitectura de 32 bits y 64 bits. Esto no se puede aplicar en métodos abstractos, variables o constructores.

```
// strictfp keyword can be applied on methods, classes and interfaces.

strictfp class A{}

strictfp interface M{}

class A{
    strictfp void m(){}
}
```

Lea Modificadores sin acceso en línea: https://riptutorial.com/es/java/topic/4401/modificadores-sin-acceso

Capítulo 136: Módulos

Sintaxis

- requiere java.xml;
- requiere java.xml público; # expone el módulo a dependientes para su uso
- exportaciones com.example.foo; # dependientes pueden usar tipos públicos en este paquete
- exporta com.example.foo.impl a com.example.bar; # restringir el uso a un módulo

Observaciones

Se recomienda el uso de módulos, pero no es obligatorio, esto permite que el código existente continúe trabajando en Java 9. También permite una transición gradual al código modular.

Cualquier código no modular se coloca en un *módulo sin nombre* cuando se compila. Este es un módulo especial que puede usar tipos de todos los demás módulos pero **solo de paquetes que tienen una declaración de `exports`** .

Todos los paquetes en el *módulo sin nombre* se exportan automáticamente.

Las palabras clave, por ejemplo, el `module` etc., tienen un uso restringido dentro de la declaración del módulo, pero se pueden seguir utilizando como identificadores en otros lugares.

Examples

Definiendo un módulo básico.

Los módulos se definen en un archivo llamado `module-info.java` , denominado descriptor de módulo. Debe colocarse en la raíz del código fuente:

```
|-- module-info.java
|-- com
    |-- example
        |-- foo
            |-- Foo.java
        |-- bar
            |-- Bar.java
```

Aquí hay un descriptor de módulo simple:

```
module com.example {
    requires java.httpclient;
    exports com.example.foo;
}
```

El nombre del módulo debe ser único y se recomienda que use la misma notación de nombres de

DNS inverso que utilizan los paquetes para garantizar esto.

El módulo `java.base` , que contiene las clases básicas de Java, está visible de forma implícita en cualquier módulo y no necesita ser incluido.

La declaración de `requires` nos permite utilizar otros módulos, en el ejemplo se importa el módulo `java.httpclient` .

Un módulo también puede especificar qué paquetes `exports` y, por lo tanto, lo hace visible para otros módulos.

El paquete `com.example.foo` declarado en la cláusula de `exports` será visible para otros módulos. Cualquier `com.example.foo` de `com.example.foo` no se exportará, necesitan sus propias declaraciones de `export` .

A la inversa, `com.example.bar` , que no figura en las cláusulas de `exports` , no será visible para otros módulos.

Lea Módulos en línea: https://riptutorial.com/es/java/topic/5286/modulos

Capítulo 137: Moneda y dinero

Examples

Añadir moneda personalizada

JAR requeridos en classpath:

- javax.money:money-api:1.0 (api de dinero y moneda JSR354)
- org.javamoney: moneta: 1.0 (implementación de referencia)
- javax: anotacion-api: 1.2. (Anotaciones comunes utilizadas por la implementación de referencia)

```
// Let's create non-ISO currency, such as bitcoin

// At first, this will throw UnknownCurrencyException
MonetaryAmount moneys = Money.of(new BigDecimal("0.1"), "BTC");

// This happens because bitcoin is unknown to default currency
// providers
System.out.println(Monetary.isCurrencyAvailable("BTC")); // false

// We will build new currency using CurrencyUnitBuilder provided by org.javamoney.moneta
CurrencyUnit bitcoin = CurrencyUnitBuilder
    .of("BTC", "BtcCurrencyProvider") // Set currency code and currency provider name
    .setDefaultFractionDigits(2)      // Set default fraction digits
    .build(true);                     // Build new currency unit. Here 'true' means
                                      // currency unit is to be registered and
                                      // accessible within default monetary context

// Now BTC is available
System.out.println(Monetary.isCurrencyAvailable("BTC")); // True
```

Lea Moneda y dinero en línea: https://riptutorial.com/es/java/topic/8359/moneda-y-dinero

Capítulo 138: Motor de JavaScript Nashorn

Introducción

Nashorn es un motor de JavaScript desarrollado en Java por Oracle, y se ha lanzado con Java 8. Nashorn permite integrar Javascript en aplicaciones Java a través de JSR-223 y permite desarrollar aplicaciones de JavaScript independientes, y proporciona un mejor rendimiento en tiempo de ejecución y un mejor cumplimiento con el ECMA Especificación de Javascript normalizada.

Sintaxis

- ScriptEngineManager // Proporciona un mecanismo de descubrimiento e instalación para las clases de ScriptEngine; utiliza un SPI (Interfaz de Proveedor de Servicios)
- ScriptEngineManager.ScriptEngineManager () // constructor recomendado
- ScriptEngine // Proporciona la interfaz para el lenguaje de scripting
- ScriptEngine ScriptEngineManager.getEngineByName (String shortName) // Método de fábrica para la implementación dada
- Object ScriptEngine.eval (String script) // Ejecuta el script especificado
- Object ScriptEngine.eval (Reader Reader) // Carga y luego ejecuta un script desde la fuente especificada
- ScriptContext ScriptEngine.getContext () // Devuelve el proveedor predeterminado de enlaces, lectores y escritores
- void ScriptContext.setWriter (escritor escritor) // Establece el destino para enviar la salida del script a

Observaciones

Nashorn es un motor de JavaScript escrito en Java e incluido en Java 8. Todo lo que necesita está incluido en el paquete `javax.script` .

Tenga en cuenta que `ScriptEngineManager` proporciona una API genérica que le permite obtener motores de secuencias de comandos para varios lenguajes de secuencias de comandos (es decir, no solo Nashorn, no solo JavaScript).

Examples

Establecer variables globales

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Define a global variable
engine.put("textToPrint", "Data defined in Java.");
```

```
// Print the global variable
try {
    engine.eval("print(textToPrint);");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
// 'Data defined in Java.' printed on standard output
```

Hola Nashorn

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Execute an hardcoded script
try {
    engine.eval("print('Hello Nashorn!');");
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Hello Nashorn!' printed on standard output
```

Ejecutar archivo JavaScript

```
// Required imports
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngine;
import javax.script.ScriptException;
import java.io.FileReader;
import java.io.FileNotFoundException;

// Obtain an instance of the JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Load and execute a script from the file 'demo.js'
try {
    engine.eval(new FileReader("demo.js"));
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Script from file!' printed on standard output
```

demo.js :

```
print('Script from file!');
```

Salida de script de intercepción

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Setup a custom writer
StringWriter stringWriter = new StringWriter();
// Modify the engine context so that the custom writer is now the default
// output writer of the engine
engine.getContext().setWriter(stringWriter);

// Execute some script
try {
    engine.eval("print('Redirected text!');");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
// Nothing printed on standard output, but
// stringWriter.toString() contains 'Redirected text!'
```

Evaluar cadenas aritméticas

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("JavaScript");

//String to be evaluated
String str = "3+2*4+5";
//Value after doing Arithmetic operation with operator precedence will be 16

//Printing the value
try {
    System.out.println(engine.eval(str));
} catch (ScriptException ex) {
    ex.printStackTrace();
}

//Outcome:
//Value of the string after arithmetic evaluation is printed on standard output.
//In this case '16.0' will be printed on standard output.
```

Uso de objetos Java en JavaScript en Nashorn

Es posible pasar objetos Java al motor Nashorn para ser procesados en código Java. Al mismo tiempo, hay algunas construcciones específicas de JavaScript (y Nashorn), y no siempre está claro cómo funcionan con los objetos java.

A continuación hay una tabla que describe el comportamiento de los objetos Java nativos dentro de las construcciones de JavaScript.

Construcciones probadas:

1. Expresión en cláusula if. En la expresión JS, la cláusula if no tiene que ser booleana a diferencia de Java. Se evalúa como falso para los llamados valores falsos (nulo, indefinido, 0, cadenas vacías, etc.)
2. Para cada declaración, Nashorn tiene un tipo especial de bucle, para cada una, que puede iterar sobre diferentes objetos JS y Java.
3. Obtención del tamaño del objeto. En JS, los objetos tienen una longitud de propiedad, que devuelve el tamaño de una matriz o una cadena.

Resultados:

Tipo	Si	para cada	.longitud
Java nulo	falso	Sin iteraciones	**Excepción**
Java cadena vacía	falso	Sin iteraciones	0
Cadena de Java	cierto	Iteriza sobre caracteres de cadena	Longitud de la cuerda
Java Integer / Long	valor! = 0	Sin iteraciones	indefinido
Java ArrayList	cierto	Itera sobre los elementos	Longitud de la lista
Java HashMap	cierto	Iteriza sobre los valores.	nulo
Java HashSet	cierto	Iterar sobre elementos	indefinido

Recomendaciones:

- Es recomendable usar `if (some_string)` para verificar si una cadena no es nula y no está vacía
- `for each` puede usar de forma segura para iterar sobre cualquier colección, y no genera excepciones si la colección no es iterable, nula o indefinida
- Antes de obtener la longitud de un objeto, debe comprobarse si está nulo o no definido (lo mismo se aplica a cualquier intento de llamar a un método u obtener una propiedad del objeto Java)

Implementando una interfaz desde script

```
import java.io.FileReader;
import java.io.IOException;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class InterfaceImplementationExample {
    public static interface Pet {
        public void eat();
```

```
    }

    public static void main(String[] args) throws IOException {
        // Obtain an instance of JavaScript engine
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        try {
            //evaluate a script
            /* pet.js */
            /*
                var Pet = Java.type("InterfaceImplementationExample.Pet");

                new Pet() {
                    eat: function() { print("eat"); }
                }
            */

            Pet pet = (Pet) engine.eval(new FileReader("pet.js"));

            pet.eat();
        } catch (ScriptException ex) {
            ex.printStackTrace();
        }

        // Outcome:
        // 'eat' printed on standard output
    }
}
```

Establecer y obtener variables globales.

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

try {
    // Set value in the global name space of the engine
    engine.put("name","Nashorn");
    // Execute an hardcoded script
    engine.eval("var value='Hello '+name+'!';");
    // Get value
    String value=(String)engine.get("value");
    System.out.println(value);
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Hello Nashorn!' printed on standard output
```

Lea Motor de JavaScript Nashorn en línea: https://riptutorial.com/es/java/topic/166/motor-de-javascript-nashorn

Capítulo 139: NIO - Redes

Observaciones

`SelectionKey` define las diferentes operaciones e información seleccionables entre su Selector y Canal . En particular, el archivo adjunto se puede utilizar para almacenar información relacionada con la conexión.

El manejo de `OP_READ` es bastante sencillo. Sin embargo, se debe tener cuidado al tratar con `OP_WRITE` : la mayoría de las veces, los datos se pueden escribir en sockets para que el evento se siga disparando. Asegúrese de registrar `OP_WRITE` solo antes de que desee escribir datos (vea esa respuesta).

Además, `OP_CONNECT` debe cancelarse una vez que el Canal se haya conectado (porque, bueno, *está* conectado. Vea esto y lo que responde en SO). Por lo tanto, la eliminación de `OP_CONNECT` después de `finishConnect()` tuvo éxito.

Examples

Usando Selector para esperar eventos (ejemplo con OP_CONNECT)

NIO apareció en Java 1.4 e introdujo el concepto de "Canales", que se supone que son más rápidos que la E / S normal. En cuanto a la red, el `SelectableChannel` es el más interesante, ya que permite monitorear diferentes estados del Canal. Funciona de manera similar a la llamada al sistema C `select()` : nos despertamos cuando ocurren ciertos tipos de eventos:

- conexión recibida (`OP_ACCEPT`)
- conexión realizada (`OP_CONNECT`)
- Datos disponibles en lectura FIFO (`OP_READ`)
- los datos se pueden `OP_WRITE` para escribir FIFO (`OP_WRITE`)

Permite la separación entre la *detección de* E / S de socket (algo se puede leer / escribir / ...) y *realizar* la E / S (leer / escribir / ...). Especialmente, toda la detección de E / S se puede hacer en un solo subproceso para múltiples sockets (clientes), mientras que la realización de E / S se puede manejar en un conjunto de subprocesos o en cualquier otro lugar. Eso permite que una aplicación se adapte fácilmente al número de clientes conectados.

El siguiente ejemplo muestra lo básico:

1. Crear un `Selector`
2. Crear un `SocketChannel`
3. Registrar el `SocketChannel` al `Selector`
4. Bucle con el `Selector` para detectar eventos.

```
Selector sel = Selector.open(); // Create the Selector
SocketChannel sc = SocketChannel.open(); // Create a SocketChannel
```

```
sc.configureBlocking(false); // ... non blocking
sc.setOption(StandardSocketOptions.SO_KEEPALIVE, true); // ... set some options

// Register the Channel to the Selector for wake-up on CONNECT event and use some description
as an attachement
sc.register(sel, SelectionKey.OP_CONNECT, "Connection to google.com"); // Returns a
SelectionKey: the association between the SocketChannel and the Selector
System.out.println("Initiating connection");
if (sc.connect(new InetSocketAddress("www.google.com", 80)))
    System.out.println("Connected"); // Connected right-away: nothing else to do
else {
    boolean exit = false;
    while (!exit) {
        if (sel.select(100) == 0) // Did something happen on some registered Channels during
the last 100ms?
            continue; // No, wait some more

        // Something happened...
        Set<SelectionKey> keys = sel.selectedKeys(); // List of SelectionKeys on which some
registered operation was triggered
        for (SelectionKey k : keys) {
            System.out.println("Checking "+k.attachment());
            if (k.isConnectable()) { // CONNECT event
                System.out.print("Connected through select() on "+k.channel()+" -> ");
                if (sc.finishConnect()) { // Finish connection process
                    System.out.println("done!");
                    k.interestOps(k.interestOps() & ~SelectionKey.OP_CONNECT); // We are
already connected: remove interest in CONNECT event
                    exit = true;
                } else
                    System.out.println("unfinished...");
            }
            // TODO: else if (k.isReadable()) { ...
        }
        keys.clear(); // Have to clear the selected keys set once processed!
    }
}
System.out.print("Disconnecting ... ");
sc.shutdownOutput(); // Initiate graceful disconnection
// TODO: emtpy receive buffer
sc.close();
System.out.println("done");
```

Daría la siguiente salida:

```
Initiating connection
Checking Connection to google.com
Connected through 'select()' on java.nio.channels.SocketChannel[connection-pending
remote=www.google.com/216.58.208.228:80] -> done!
Disconnecting ... done
```

Lea NIO - Redes en línea: https://riptutorial.com/es/java/topic/5513/nio---redes

Capítulo 140: Nuevo archivo I / O

Sintaxis

- Paths.get (Cadena primero, Cadena ... más) // Crea una instancia de ruta por sus elementos de cadena
- Paths.get (URI uri) // Crea una instancia de Path por un URI

Examples

Creando caminos

La clase de `Path` se usa para representar de manera programática una ruta en el sistema de archivos (y, por lo tanto, puede apuntar a archivos y directorios, incluso a los que no existen)

Se puede obtener una ruta utilizando las `Paths` clase auxiliar:

```
Path p1 = Paths.get("/var/www");
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));
Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt");
Path p4 = Paths.get("/home", "arthur", "files", "diary.tex");
```

Recuperar información sobre una ruta

La información sobre una ruta se puede obtener utilizando los métodos de un objeto `Path` :

- `toString()` devuelve la representación de cadena de la ruta

  ```
  Path p1 = Paths.get("/var/www"); // p1.toString() returns "/var/www"
  ```

- `getFileName()` devuelve el nombre del archivo (o, más específicamente, el último elemento de la ruta)

  ```
  Path p1 = Paths.get("/var/www"); // p1.getFileName() returns "www"
  Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt"); // p3.getFileName()
  returns "HHGTDG.odt"
  ```

- `getNameCount()` devuelve el número de elementos que forman la ruta

  ```
  Path p1 = Paths.get("/var/www"); // p1.getNameCount() returns 2
  ```

- `getName(int index)` devuelve el elemento en el índice dado

  ```
  Path p1 = Paths.get("/var/www"); // p1.getName(0) returns "var", p1.getName(1) returns
  "www"
  ```

- `getParent()` devuelve la ruta del directorio padre

```
Path p1 = Paths.get("/var/www"); // p1.getParent().toString() returns "/var"
```

- `getRoot()` devuelve la raíz de la ruta

```
Path p1 = Paths.get("/var/www"); // p1.getRoot().toString() returns "/"
Path p3 = Paths.get("C:\\Users\\DentAr\\Documents\\HHGTDG.odt"); //
p3.getRoot().toString() returns "C:\\"
```

Manipulando caminos

Unirse a dos caminos

Las rutas se pueden unir usando el método `resolve()` . La ruta pasada tiene que ser una ruta parcial, que es una ruta que no incluye el elemento raíz.

```
Path p5 = Paths.get("/home/");
Path p6 = Paths.get("arthur/files");
Path joined = p5.resolve(p6);
Path otherJoined = p5.resolve("ford/files");
```

```
joined.toString() == "/home/arthur/files"
otherJoined.toString() == "/home/ford/files"
```

Normalizando un camino

Los caminos pueden contener los elementos `.` (que apunta al directorio en el que se encuentra actualmente) y `..` (que apunta al directorio principal).

Cuando se usa en un camino, `.` puede eliminarse en cualquier momento sin cambiar el destino de la ruta, y `..` puede eliminarse junto con el elemento anterior.

Con la API de rutas, esto se hace usando el método `.normalize()` :

```
Path p7 = Paths.get("/home/./arthur/../ford/files");
Path p8 = Paths.get("C:\\Users\\.\\..\\Program Files");
```

```
p7.normalize().toString() == "/home/ford/files"
p8.normalize().toString() == "C:\\Program Files"
```

Recuperando información utilizando el sistema de archivos.

Para interactuar con el sistema de archivos, utilice los métodos de la clase `Files` .

Comprobando la existencia

Para verificar la existencia del archivo o directorio al que apunta una ruta, utilice los siguientes métodos:

```
Files.exists(Path path)
```

y

```
Files.notExists(Path path)
```

`!Files.exists(path)` no necesariamente tiene que ser igual a `Files.notExists(path)` , porque hay tres escenarios posibles:

- Se verifica la existencia de un archivo o directorio (`exists` devuelve `true` y `notExists` devuelve `false` en este caso)
- La inexistencia de un archivo o directorio se verifica (`exists` devuelve `false` y `notExists` devuelve `true`)
- No se puede verificar la existencia ni la inexistencia de un archivo o un directorio (por ejemplo, debido a restricciones de acceso): ambos `exists` y los que no `nonExists` devuelven falso.

Comprobando si una ruta apunta a un archivo o directorio

Esto se hace usando `Files.isDirectory(Path path)` y `Files.isRegularFile(Path path)`

```
Path p1 = Paths.get("/var/www");
Path p2 = Paths.get("/home/testuser/File.txt");
```

```
Files.isDirectory(p1) == true
Files.isRegularFile(p1) == false

Files.isDirectory(p2) == false
Files.isRegularFile(p2) == true
```

Obteniendo propiedades

Esto se puede hacer usando los siguientes métodos:

```
Files.isReadable(Path path)
Files.isWritable(Path path)
Files.isExecutable(Path path)
```

```
Files.isHidden(Path path)
Files.isSymbolicLink(Path path)
```

Obteniendo el tipo MIME

```
Files.probeContentType(Path path)
```

Esto intenta obtener el tipo MIME de un archivo. Devuelve una cadena tipo MIME, como esta:

- `text/plain` para archivos de texto
- `text/html` para páginas HTML
- `application/pdf` para archivos PDF
- `image/png` para archivos PNG

Archivos de lectura

Los archivos se pueden leer en bytes y en línea usando la clase `Files` .

```
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));
byte[] content = Files.readAllBytes(p2);
List<String> linesOfContent = Files.readAllLines(p2);
```

`Files.readAllLines()` opcionalmente toma un conjunto de caracteres como parámetro (el valor predeterminado es `StandardCharsets.UTF_8`):

```
List<String> linesOfContent = Files.readAllLines(p2, StandardCharsets.ISO_8859_1);
```

Escribiendo archivos

Los archivos se pueden escribir de forma mordida y lineal utilizando la clase `Files`

```
Path p2 = Paths.get("/home/testuser/File.txt");
List<String> lines = Arrays.asList(
    new String[]{"First line", "Second line", "Third line"});

Files.write(p2, lines);
```

```
Files.write(Path path, byte[] bytes)
```

Los archivos existentes se anularán, se crearán archivos no existentes.

Lea Nuevo archivo I / O en línea: https://riptutorial.com/es/java/topic/5519/nuevo-archivo-i---o

Capítulo 141: Objetos inmutables

Observaciones

Los objetos inmutables tienen un estado fijo (no establecedores), por lo que todo el estado debe ser conocido en el momento de la creación del objeto.

Aunque no es técnicamente necesario, es una buena práctica hacer que todos los campos sean `final` . Esto hará que la clase inmutable sea segura para subprocesos (véase Java Concurrency in Practice, 3.4.1).

Los ejemplos muestran varios patrones que pueden ayudar a lograr esto.

Examples

Creación de una versión inmutable de un tipo mediante copia defensiva.

Algunos tipos y clases básicas en Java son fundamentalmente mutables. Por ejemplo, todos los tipos de matriz son mutables, y también lo son las clases como `java.util.Data` . Esto puede ser incómodo en situaciones donde se requiere un tipo inmutable.

Una forma de lidiar con esto es crear un envoltorio inmutable para el tipo mutable. Aquí hay una envoltura simple para una matriz de enteros

```
public class ImmutableIntArray {
    private final int[] array;

    public ImmutableIntArray(int[] array) {
        this.array = array.clone();
    }

    public int[] getValue() {
        return this.clone();
    }
}
```

Esta clase funciona utilizando *una copia defensiva* para aislar el estado mutable (el `int[]`) de cualquier código que pueda mutarlo:

- El constructor usa `clone()` para crear una copia distinta de la matriz de parámetros. Si la persona que llama al constructor posteriormente cambió la matriz de parámetros, no afectaría el estado de `ImmutableIntArray` .
- El método `getValue()` también usa `clone()` para crear la matriz que se devuelve. Si la persona que llama cambiara la matriz de resultados, no afectaría el estado de `ImmutableIntArray` .

También podríamos agregar métodos a `ImmutableIntArray` para realizar operaciones de solo

lectura en la matriz envuelta; por ejemplo, obtener su longitud, obtener el valor en un índice particular, y así sucesivamente.

Tenga en cuenta que un tipo de envoltorio inmutable implementado de esta manera no es compatible con el tipo original. No puedes simplemente sustituir el primero por el segundo.

La receta para una clase inmutable.

Un objeto inmutable es un objeto cuyo estado no se puede cambiar. Una clase inmutable es una clase cuyas instancias son inmutables por diseño e implementación. La clase de Java que se presenta más comúnmente como un ejemplo de inmutabilidad es java.lang.String .

El siguiente es un ejemplo estereotipado:

```
public final class Person {
    private final String name;
    private final String ssn;     // (SSN == social security number)

    public Person(String name, String ssn) {
        this.name = name;
        this.ssn = ssn;
    }

    public String getName() {
        return name;
    }

    public String getSSN() {
        return ssn;
    }
}
```

Una variante de esto es declarar el constructor como `private` y proporcionar un método de fábrica `public static` .

La *receta estándar* para una clase inmutable es la siguiente:

- Todas las propiedades deben establecerse en el (los) constructor (es) o en los métodos de fábrica.
- No debería haber setters.
- Si es necesario incluir configuradores por razones de compatibilidad de interfaz, no deberían hacer nada o lanzar una excepción.
- Todas las propiedades deben ser declaradas como `private` y `final` .
- Para todas las propiedades que son referencias a tipos mutables:
    - la propiedad debe inicializarse con una copia en profundidad del valor pasado a través del constructor, y
    - El captador de la propiedad debe devolver una copia profunda del valor de la propiedad.
- La clase debe declararse como `final` para evitar que alguien cree una subclase mutable de una clase inmutable.

Un par de otras cosas a tener en cuenta:

- La inmutabilidad no impide que el objeto sea anulable; Por ejemplo, `null` puede asignarse a una variable de `String` .
- Si las propiedades de una clase inmutable se declaran como `final` , las instancias son intrínsecamente seguras para subprocesos. Esto hace que las clases inmutables sean un buen bloque de construcción para implementar aplicaciones de subprocesos múltiples.

Fallas típicas de diseño que evitan que una clase sea inmutable.

Usando algunos configuradores, sin configurar todas las propiedades necesarias en el (los) constructor (es)

```
public final class Person { // example of a bad immutability
    private final String name;
    private final String surname;
    public Person(String name) {
        this.name = name;
      }
    public String getName() { return name;}
    public String getSurname() { return surname;}
    public void setSurname(String surname) { this.surname = surname); }
}
```

Es fácil mostrar que la clase `Person` no es inmutable:

```
Person person = new Person("Joe");
person.setSurname("Average"); // NOT OK, change surname field after creation
```

Para solucionarlo, simplemente elimine `setSurname()` y refactorice el constructor de la siguiente manera:

```
public Person(String name, String surname) {
    this.name = name;
    this.surname = surname;
  }
```

No marcando variables de instancia como privadas y finales.

Echa un vistazo a la siguiente clase:

```
public final class Person {
    public String name;
    public Person(String name) {
        this.name = name;
     }
    public String getName() {
        return name;
    }

}
```

El siguiente fragmento de código muestra que la clase anterior no es inmutable:

```
Person person = new Person("Average Joe");
person.name = "Magic Mike"; // not OK, new name for person after creation
```

Para solucionarlo, simplemente marque la propiedad del nombre como `private` y `final` .

Exponer un objeto mutable de la clase en un captador.

Echa un vistazo a la siguiente clase:

```
import java.util.List;
import java.util.ArrayList;
public final class Names {
    private final List<String> names;
    public Names(List<String> names) {
        this.names = new ArrayList<String>(names);
    }
    public List<String> getNames() {
        return names;
    }
    public int size() {
        return names.size();
    }
}
```

`Names` clase de `Names` parece inmutable a primera vista, pero no es como lo muestra el siguiente código:

```
List<String> namesList = new ArrayList<String>();
namesList.add("Average Joe");
Names names = new Names(namesList);
System.out.println(names.size()); // 1, only containing "Average Joe"
namesList = names.getNames();
namesList.add("Magic Mike");
System.out.println(names.size()); // 2, NOT OK, now names also contains "Magic Mike"
```

Esto sucedió porque un cambio en la Lista de referencia devuelto por `getNames()` puede modificar la lista real de `Names` .

Para solucionar este problema, simplemente evite devolver referencias que hagan referencia a objetos mutables de la clase, *ya sea* haciendo copias defensivas, de la siguiente manera:

```
public List<String> getNames() {
   return new ArrayList<String>(this.names); // copies elements
}
```

o diseñando captadores de manera que solo se devuelvan otros *objetos* y *primitivas inmutables* , de la siguiente manera:

```
public String getName(int index) {
    return names.get(index);
```

```
}
public int size() {
    return names.size();
}
```

Inyectar el constructor con objeto (s) que pueden modificarse fuera de la clase inmutable

Esta es una variación de la falla anterior. Echa un vistazo a la siguiente clase:

```
import java.util.List;
public final class NewNames {
    private final List<String> names;
    public Names(List<String> names) {
        this.names = names;
    }
    public String getName(int index) {
        return names.get(index);
    }
    public int size() {
        return names.size();
    }
}
```

Como los `Names` clase antes, también la clase `NewNames` parece inmutable a primera vista, pero no lo es, de hecho, el siguiente fragmento de código demuestra lo contrario:

```
List<String> namesList = new ArrayList<String>();
namesList.add("Average Joe");
NewNames names = new NewNames(namesList);
System.out.println(names.size()); // 1, only containing "Average Joe"
namesList.add("Magic Mike");
System.out.println(names.size()); // 2, NOT OK, now names also contains "Magic Mike"
```

Para corregir esto, como en el defecto anterior, simplemente haga copias defensivas del objeto sin asignarlo directamente a la clase inmutable, es decir, el constructor se puede cambiar de la siguiente manera:

```
    public Names(List<String> names) {
        this.names = new ArrayList<String>(names);
    }
```

Dejar que los métodos de la clase sean anulados

Echa un vistazo a la siguiente clase:

```
public class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
      }
    public String getName() { return name;}
}
```

`Person` clase de `Person` parece inmutable a primera vista, pero supongamos que se define una nueva subclase de `Person` :

```
public class MutablePerson extends Person {
    private String newName;
    public MutablePerson(String name) {
        super(name);
    }
    @Override
    public String getName() {
        return newName;
    }
    public void setName(String name) {
        newName = name;
    }
}
```

ahora la mutabilidad de `Person` (im) puede ser explotada a través del polimorfismo usando la nueva subclase:

```
Person person = new MutablePerson("Average Joe");
System.out.println(person.getName()); prints Average Joe
person.setName("Magic Mike"); // NOT OK, person has now a new name!
System.out.println(person.getName()); // prints Magic Mike
```

Para solucionar este problema, *o bien* marcar la clase como `final` por lo que no se puede ampliar *o* declarar la totalidad de su constructor (s) como `private` .

Lea Objetos inmutables en línea: https://riptutorial.com/es/java/topic/2807/objetos-inmutables

Capítulo 142: Objetos seguros

Sintaxis

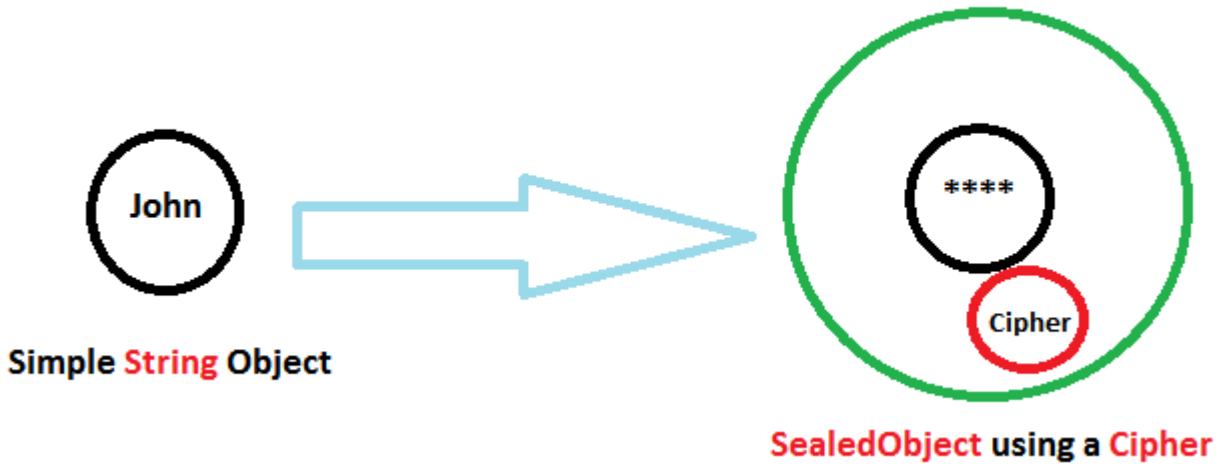
- SealedObject sealedObject = new SealedObject (obj, cipher);
- SignedObject signedObject = new SignedObject (obj, signKey, signEngine);

Examples

SealedObject (javax.crypto.SealedObject)

Esta clase permite que un programador cree un objeto y proteja su confidencialidad con un algoritmo criptográfico.

Dado cualquier objeto Serializable, se puede crear un **SealedObject** que encapsula el objeto original, en formato serializado (es decir, una "copia profunda"), y sella (encripta) su contenido serializado, utilizando un algoritmo criptográfico como AES, DES, para proteger Su confidencialidad. El contenido cifrado se puede descifrar más tarde (con el algoritmo correspondiente utilizando la clave de descifrado correcta) y deserializarse, lo que genera el objeto original.

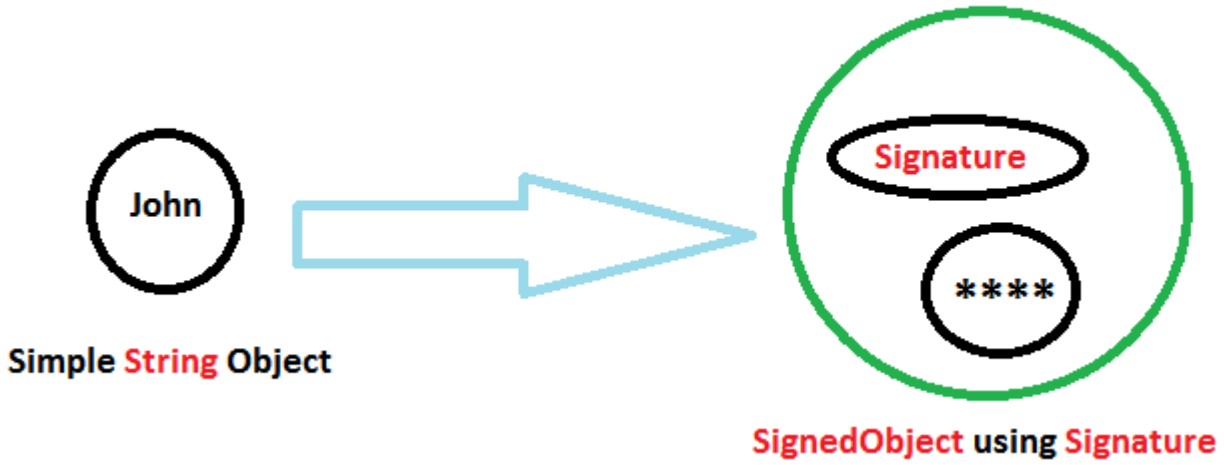



```
Serializable obj = new String("John");
// Generate key
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128);
SecretKey aesKey = kgen.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, aesKey);
SealedObject sealedObject = new SealedObject(obj, cipher);
System.out.println("sealedObject-" + sealedObject);
System.out.println("sealedObject Data-" + sealedObject.getObject(aesKey));
```

SignedObject (java.security.SignedObject)

SignedObject es una clase con el propósito de crear objetos de tiempo de ejecución auténticos cuya integridad no puede ser comprometida sin ser detectada.

Más específicamente, un SignedObject contiene otro objeto Serializable, el objeto firmado (a ser) y su firma.




```
//Create a key
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);
// create a private key
PrivateKey signingKey = keyGen.generateKeyPair().getPrivate();
// create a Signature
Signature signingEngine = Signature.getInstance("DSA");
signingEngine.initSign(signingKey);
// create a simple object
Serializable obj = new String("John");
// sign our object
SignedObject signedObject = new SignedObject(obj, signingKey, signingEngine);

System.out.println("signedObject-" + signedObject);
System.out.println("signedObject Data-" + signedObject.getObject());
```

Lea Objetos seguros en línea: https://riptutorial.com/es/java/topic/5528/objetos-seguros

Capítulo 143: Opcional

Introducción

`Optional` es un objeto contenedor que puede o no contener un valor no nulo. Si hay un valor presente, `isPresent()` devolverá `true` y `get()` devolverá el valor.

Se proporcionan métodos adicionales que dependen de la presencia del valor contenido, como `orElse()` , que devuelve un valor predeterminado si el valor no está presente, y `ifPresent()` que ejecuta un bloque de código si el valor está presente.

Sintaxis

- Optional.empty () // Crea una instancia opcional vacía.
- Optional.of (value) // Devuelve un Optional con el valor no nulo especificado. Se emitirá una NullPointerException si el valor pasado es nulo.
- Optional.ofNullable (valor) // Devuelve un opcional con el valor especificado que puede ser nulo.

Examples

Devolver valor predeterminado si Opcional está vacío

No solo use `Optional.get()` ya que puede lanzar `NoSuchElementException` . Los métodos `Optional.orElse(T)` y `Optional.orElseGet(Supplier<? extends T>)` proporcionan una manera de proporcionar un valor predeterminado en caso de que el Opcional esté vacío.

```
String value = "something";

return Optional.ofNullable(value).orElse("defaultValue");
// returns "something"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// returns "something" (never calls the getDefaultValue() method)
```

```
String value = null;

return Optional.ofNullable(value).orElse("defaultValue");
// returns "defaultValue"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// calls getDefaultValue() and returns its results
```

La diferencia crucial entre `orElse` y `orElseGet` es que este último solo se evalúa cuando el opcional está vacío, mientras que el argumento suministrado al anterior se evalúa incluso si el opcional no está vacío. Por lo tanto, el `orElse` solo debe usarse para constantes y nunca para suministrar valor en base a ningún tipo de cálculo.

Mapa

Use el método `map()` de `Optional` para trabajar con valores que podrían ser `null` sin hacer verificaciones `null` explícitas:

(Tenga en cuenta que las operaciones `map()` y `filter()` se evalúan inmediatamente, a diferencia de sus contrapartes de Stream que solo se evalúan en una *operación de terminal*).

Sintaxis:

```
public <U> Optional<U> map(Function<? super T,? extends U> mapper)
```

Ejemplos de código:

```
String value = null;

return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");
// returns "NONE"
```

```
String value = "something";

return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");
// returns "SOMETHING"
```

Dado que Optional.map () devuelve un opcional vacío cuando su función de asignación devuelve un valor nulo, puede encadenar varias operaciones map () como una forma de desreferenciación segura de nulos. Esto también se conoce como **encadenamiento Null-safe** .

Considere el siguiente ejemplo:

```
String value = foo.getBar().getBaz().toString();
```

Cualquiera de `getBar` , `getBaz` y `toString` puede lanzar una `NullPointerException` .

Aquí hay una forma alternativa de obtener el valor de `toString()` usando `Optional` :

```
String value = Optional.ofNullable(foo)
                       .map(Foo::getBar)
                       .map(Bar::getBaz)
                       .map(Baz::toString)
                       .orElse("");
```

Esto devolverá una cadena vacía si alguna de las funciones de mapeo devuelve un valor nulo.

A continuación se muestra otro ejemplo, pero ligeramente diferente. Imprimirá el valor solo si ninguna de las funciones de asignación devolvió un valor nulo.

```
Optional.ofNullable(foo)
        .map(Foo::getBar)
        .map(Bar::getBaz)
```

```
        .map(Baz::toString)
        .ifPresent(System.out::println);
```

Lanzar una excepción, si no hay valor.

Use el método `orElseThrow()` de `Optional` para obtener el valor contenido o lanzar una excepción, si no se ha establecido. Esto es similar a llamar a `get()` , excepto que permite tipos de excepción arbitrarios. El método toma un proveedor que debe devolver la excepción para ser lanzado.

En el primer ejemplo, el método simplemente devuelve el valor contenido:

```
Optional optional = Optional.of("something");

return optional.orElseThrow(IllegalArgumentException::new);
// returns "something" string
```

En el segundo ejemplo, el método lanza una excepción porque no se ha establecido un valor:

```
Optional optional = Optional.empty();

return optional.orElseThrow(IllegalArgumentException::new);
// throws IllegalArgumentException
```

También puede usar la sintaxis lambda si se necesita lanzar una excepción con el mensaje:

```
optional.orElseThrow(() -> new IllegalArgumentException("Illegal"));
```

Filtrar

`filter()` se usa para indicar que le gustaría el valor *solo si* coincide con su predicado.

Piense en ello como `if (!somePredicate(x)) { x = null; }` .

Ejemplos de código:

```
String value = null;
Optional.ofNullable(value) // nothing
        .filter(x -> x.equals("cool string"))// this is never run since value is null
        .isPresent(); // false
```

```
String value = "cool string";
Optional.ofNullable(value) // something
        .filter(x -> x.equals("cool string"))// this is run and passes
        .isPresent(); // true
```

```
String value = "hot string";
Optional.ofNullable(value) // something
        .filter(x -> x.equals("cool string"))// this is run and fails
        .isPresent(); // false
```

Usando contenedores opcionales para tipos de números primitivos

`OptionalDouble` , `OptionalInt` y `OptionalLong` funcionan como `Optional` , pero están diseñados específicamente para envolver tipos primitivos:

```
OptionalInt presentInt = OptionalInt.of(value);
OptionalInt absentInt = OptionalInt.empty();
```

Debido a que los tipos numéricos tienen un valor, no hay un manejo especial para el valor nulo. Los contenedores vacíos se pueden revisar con:

```
presentInt.isPresent(); // Is true.
absentInt.isPresent(); // Is false.
```

Del mismo modo, existen métodos abreviados para ayudar a la gestión del valor:

```
// Prints the value since it is provided on creation.
presentInt.ifPresent(System.out::println);

// Gives the other value as the original Optional is empty.
int finalValue = absentInt.orElseGet(this::otherValue);

// Will throw a NoSuchElementException.
int nonexistentValue = absentInt.getAsInt();
```

Ejecutar código solo si hay un valor presente

```
Optional<String> optionalWithValue = Optional.of("foo");
optionalWithValue.ifPresent(System.out::println);//Prints "foo".

Optional<String> emptyOptional = Optional.empty();
emptyOptional.ifPresent(System.out::println);//Does nothing.
```

Proporcionar perezosamente un valor predeterminado utilizando un Proveedor

El método *normal* `orElse` toma un `Object` , por lo que podría preguntarse por qué hay una opción para proporcionar un `Supplier` aquí (el método `orElseGet`).

Considerar:

```
String value = "something";
return Optional.ofNullable(value)
               .orElse(getValueThatIsHardToCalculate()); // returns "something"
```

Todavía llamaría a `getValueThatIsHardToCalculate()` aunque su resultado no se use porque el opcional no está vacío.

Para evitar esta penalización, debe suministrar un proveedor:

```
String value = "something";
return Optional.ofNullable(value)
               .orElseGet(() -> getValueThatIsHardToCalculate()); // returns "something"
```

De esta manera, solo se llamará a `getValueThatIsHardToCalculate()` si el `Optional` está vacío.

Mapa plano

`flatMap` es similar al `map` . La diferencia es descrita por el javadoc de la siguiente manera:

> Este método es similar al `map(Function)` , pero el asignador proporcionado es uno cuyo resultado ya es `Optional` y, si se invoca, `flatMap` no lo envuelve con un `Optional` adicional.

En otras palabras, cuando encadena una llamada de método que devuelve un `Optional` , el uso de `Optional.flatMap` evita la creación de `Optionals` anidados.

Por ejemplo, dadas las siguientes clases:

```
public class Foo {
    Optional<Bar> getBar(){
        return Optional.of(new Bar());
    }
}

public class Bar {
}
```

Si usa `Optional.map` , obtendrá un `Optional` anidado; es decir, `Optional<Optional<Bar>>` .

```
Optional<Optional<Bar>> nestedOptionalBar =
    Optional.of(new Foo())
        .map(Foo::getBar);
```

Sin embargo, si utiliza `Optional.flatMap` , obtendrá un `Optional` ; es decir, `Optional<Bar>` .

```
Optional<Bar> optionalBar =
    Optional.of(new Foo())
        .flatMap(Foo::getBar);
```

Lea Opcional en línea: https://riptutorial.com/es/java/topic/152/opcional

Capítulo 144: Operaciones de punto flotante de Java

Introducción

Los números de punto flotante son números que tienen partes fraccionarias (generalmente expresadas con un punto decimal). En Java, hay dos tipos primitivos para números de punto flotante que son `float` (usa 4 bytes) y `double` (usa 8 bytes). Esta página de documentación es para detallar con operaciones de ejemplos que se pueden hacer en puntos flotantes en Java.

Examples

Comparando valores de punto flotante

Debe tener cuidado al comparar valores de punto `float` (`float` o `double`) utilizando operadores relacionales: `==` `!=` , `<` Y así sucesivamente. Estos operadores dan resultados de acuerdo con las representaciones binarias de los valores de punto flotante. Por ejemplo:

```
public class CompareTest {
    public static void main(String[] args) {
        double oneThird = 1.0 / 3.0;
        double one = oneThird * 3;
        System.out.println(one == 1.0);      // prints "false"
    }
}
```

El cálculo `oneThird` ha introducido un pequeño error de redondeo, y cuando multiplicamos `oneThird` por `3` obtenemos un resultado que es ligeramente diferente a `1.0` .

Este problema de representaciones inexactas es más grave cuando intentamos mezclar el `double` y el `float` en los cálculos. Por ejemplo:

```
public class CompareTest2 {
    public static void main(String[] args) {
        float floatVal = 0.1f;
        double doubleVal = 0.1;
        double doubleValCopy = floatVal;

        System.out.println(floatVal);      // 0.1
        System.out.println(doubleVal);     // 0.1
        System.out.println(doubleValCopy); // 0.10000000149011612

        System.out.println(floatVal == doubleVal); // false
        System.out.println(doubleVal == doubleValCopy); // false
    }
}
```

Las representaciones de punto flotante utilizadas en Java para los tipos `float` y `double` tienen un

número limitado de dígitos de precisión. Para el tipo `float` , la precisión es de 23 dígitos binarios o aproximadamente 8 dígitos decimales. Para el tipo `double` , es de 52 bits o alrededor de 15 dígitos decimales. Además de eso, algunas operaciones aritméticas introducirán errores de redondeo. Por lo tanto, cuando un programa compara valores de punto flotante, es una práctica estándar definir un **delta aceptable** para la comparación. Si la diferencia entre los dos números es menor que el delta, se considera que son iguales. Por ejemplo

```
if (Math.abs(v1 - v2) < delta)
```

Ejemplo de comparación de Delta:

```
public class DeltaCompareExample {

    private static boolean deltaCompare(double v1, double v2, double delta) {
        // return true iff the difference between v1 and v2 is less than delta
        return Math.abs(v1 - v2) < delta;
    }

    public static void main(String[] args) {
        double[] doubles = {1.0, 1.0001, 1.0000001, 1.000000001, 1.0000000000001};
        double[] deltas = {0.01, 0.00001, 0.0000001, 0.0000000001, 0};

        // loop through all of deltas initialized above
        for (int j = 0; j < deltas.length; j++) {
            double delta = deltas[j];
            System.out.println("delta: " + delta);

            // loop through all of the doubles initialized above
            for (int i = 0; i < doubles.length - 1; i++) {
                double d1 = doubles[i];
                double d2 = doubles[i + 1];
                boolean result = deltaCompare(d1, d2, delta);

                System.out.println("" + d1 + " == " + d2 + " ? " + result);

            }

            System.out.println();
        }
    }
}
```

Resultado:

```
delta: 0.01
1.0 == 1.0001 ? true
1.0001 == 1.0000001 ? true
1.0000001 == 1.000000001 ? true
1.000000001 == 1.0000000000001 ? true

delta: 1.0E-5
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? true
1.000000001 == 1.0000000000001 ? true
```

```
delta: 1.0E-7
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? true
1.000000001 == 1.0000000000001 ? true

delta: 1.0E-10
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? false
1.000000001 == 1.0000000000001 ? false

delta: 0.0
1.0 == 1.0001 ? false
1.0001 == 1.0000001 ? false
1.0000001 == 1.000000001 ? false
1.000000001 == 1.0000000000001 ? false
```

También para la comparación de tipos primitivos `double` y `float` se puede usar el método de `compare` estática del tipo de boxeo correspondiente. Por ejemplo:

```
double a = 1.0;
double b = 1.0001;

System.out.println(Double.compare(a, b));//-1
System.out.println(Double.compare(b, a));//1
```

Finalmente, determinar qué deltas son más apropiadas para una comparación puede ser complicado. Un enfoque comúnmente utilizado es seleccionar valores delta que, según nuestra intuición, son correctos. Sin embargo, si conoce la escala y la precisión (verdadera) de los valores de entrada y los cálculos realizados, puede ser posible establecer límites matemáticamente sólidos sobre la precisión de los resultados y, por lo tanto, para los deltas. (Hay una rama formal de Matemáticas conocida como Análisis Numérico que solía enseñarse a científicos computacionales que cubrían este tipo de análisis).

OverFlow y UnderFlow

Tipo de datos **flotante**

El tipo de datos flotante es un punto flotante IEEE 754 de 32 bits de precisión simple.

Desbordamiento de `Float`

El valor máximo posible es `3.4028235e+38` , cuando supera este valor produce `Infinity`

```
float f = 3.4e38f;
float result = f*2;
System.out.println(result); //Infinity
```

`Float` **Bajo Flujo**

El valor mínimo es 1.4e-45f, cuando está por debajo de este valor, produce `0.0`

```
    float f = 1e-45f;
    float result = f/1000;
    System.out.println(result);
```

doble tipo de datos

El tipo de datos doble es un punto flotante IEEE 754 de doble precisión de `64-bit` .

`Double` **OverFlow**

El valor máximo posible es `1.7976931348623157e+308` , cuando supera este valor produce `Infinity`

```
double d = 1e308;
double result=d*2;
System.out.println(result); //Infinity
```

`Double` **flujo bajo**

El valor mínimo es 4.9e-324, cuando está por debajo de este valor, produce `0.0`

```
    double d = 4.8e-323;
    double result = d/1000;
    System.out.println(result); //0.0
```

Formato de los valores de punto flotante

Los números de punto flotante se pueden formatear como un número decimal utilizando `String.format` con el `String.format 'f'`

```
    //Two digits in fracttional part are rounded
    String format1 = String.format("%.2f", 1.2399);
    System.out.println(format1); // "1.24"

    // three digits in fractional part are rounded
    String format2 = String.format("%.3f", 1.2399);
    System.out.println(format2); // "1.240"

    //rounded to two digits, filled with zero
    String format3 = String.format("%.2f", 1.2);
    System.out.println(format3); // returns "1.20"

    //rounder to two digits
    String format4 = String.format("%.2f", 3.19999);
    System.out.println(format4); // "3.20"
```

Los números de punto flotante se pueden formatear como un número decimal usando `DecimalFormat`

```
   // rounded with one digit fractional part
    String format = new DecimalFormat("0.#").format(4.3200);
    System.out.println(format); // 4.3

   // rounded with two digit fractional part
```

```
    String format = new DecimalFormat("0.##").format(1.2323000);
    System.out.println(format); //1.23

    // formatting floating numbers to decimal number
    double dv = 123456789;
    System.out.println(dv); // 1.23456789E8
    String format =  new DecimalFormat("0").format(dv);
    System.out.println(format); //123456789
```

Adherencia estricta a la especificación IEEE

De forma predeterminada, las operaciones de punto flotante en `float` y `double` *no* se ajustan estrictamente a las reglas de la especificación IEEE 754. Se permite que una expresión use extensiones específicas de la implementación para el rango de estos valores; Básicamente permitiendo que sean *más* precisos de lo requerido.

`strictfp` desactiva este comportamiento. Se aplica a una clase, interfaz o método, y se aplica a todo lo que contiene, como clases, interfaces, métodos, constructores, inicializadores de variables, etc. Con `strictfp` , los valores intermedios de una expresión de punto flotante *deben* estar dentro de el conjunto de valores flotantes o el conjunto de valores dobles. Esto hace que los resultados de tales expresiones sean exactamente aquellos que predice la especificación IEEE 754.

Todas las expresiones constantes son implícitamente estrictas, incluso si no están dentro de un alcance `strictfp` .

Por lo tanto, `strictfp` tiene el efecto neto de que algunas veces los cálculos de casos de esquina sean *menos* precisos, y también puede hacer que las operaciones de punto flotante sean *más lentas* (ya que la CPU está haciendo más trabajo para garantizar que cualquier precisión adicional nativa no afecte el resultado). Sin embargo, también hace que los resultados sean exactamente iguales en todas las plataformas. Por lo tanto, es útil en cosas como programas científicos, donde la reproducibilidad es más importante que la velocidad.

```
public class StrictFP { // No strictfp -> default lenient
    public strictfp float strict(float input) {
        return input * input / 3.4f; // Strictly adheres to the spec.
                                     // May be less accurate and may be slower.
    }

    public float lenient(float input) {
        return input * input / 3.4f; // Can sometimes be more accurate and faster,
                                     // but results may not be reproducable.
    }

    public static final strictfp class Ops { // strictfp affects all enclosed entities
        private StrictOps() {}

        public static div(double dividend, double divisor) { // implicitly strictfp
            return dividend / divisor;
        }
    }
}
```

Lea Operaciones de punto flotante de Java en línea:
https://riptutorial.com/es/java/topic/6167/operaciones-de-punto-flotante-de-java

Capítulo 145: Paquetes

Introducción

El paquete en java se usa para agrupar clases e interfaces. Esto ayuda al desarrollador a evitar conflictos cuando hay un gran número de clases. Si usamos las clases de este paquete, podemos crear una clase / interfaz con el mismo nombre en diferentes paquetes. Mediante el uso de paquetes podemos importar la pieza de nuevo en otra clase. Hay muchos *paquetes incorporados en* java como> 1.java.util> 2.java.lang> 3.java.io Podemos definir nuestros propios *paquetes definidos por el usuario* .

Observaciones

Los paquetes proporcionan protección de acceso.

> La declaración del paquete debe ser la primera línea del código fuente. Solo puede haber un paquete en un archivo fuente.

Con la ayuda de paquetes se pueden evitar conflictos entre diferentes módulos.

Examples

Usando paquetes para crear clases con el mismo nombre

Primera prueba.clase:

```
package foo.bar

public class Test {

}
```

También Test.class en otro paquete

```
package foo.bar.baz

public class Test {

}
```

Lo anterior está bien porque las dos clases existen en paquetes diferentes.

Uso del alcance del paquete protegido

En Java, si no proporciona un modificador de acceso, el alcance predeterminado para las variables es el nivel protegido por paquete. Esto significa que las clases pueden acceder a las

variables de otras clases dentro del mismo paquete como si esas variables estuvieran disponibles públicamente.

```
package foo.bar

public class ExampleClass {
    double exampleNumber;
    String exampleString;

    public ExampleClass() {
        exampleNumber = 3;
        exampleString = "Test String";
    }
    //No getters or setters
}

package foo.bar

public class AnotherClass {
    ExampleClass clazz = new ExampleClass();

    System.out.println("Example Number: " + clazz.exampleNumber);
    //Prints Example Number: 3
    System.out.println("Example String: " + clazz.exampleString);
    //Prints Example String: Test String
}
```

Este método no funcionará para una clase en otro paquete:

```
package baz.foo

public class ThisShouldNotWork {
    ExampleClass clazz = new ExampleClass();

    System.out.println("Example Number: " + clazz.exampleNumber);
    //Throws an exception
    System.out.println("Example String: " + clazz.exampleString);
    //Throws an exception
}
```

Lea Paquetes en línea: https://riptutorial.com/es/java/topic/8273/paquetes

Capítulo 146: Polimorfismo

Introducción

El polimorfismo es uno de los principales conceptos de programación orientada a objetos (POO). La palabra polimorfismo se derivó de las palabras griegas "poli" y "morfos". Poli significa "muchos" y morfos significa "formas" (muchas formas).

Hay dos formas de realizar el polimorfismo. **Método de sobrecarga** y **método de anulación** .

Observaciones

`Interfaces` son otra forma de lograr el polimorfismo en Java, aparte de la herencia basada en clases. Las interfaces definen una lista de métodos que forman la API del programa. Las clases deben `implement` una `interface` anulando todos sus métodos.

Examples

Método de sobrecarga

La sobrecarga de métodos , también conocida como **sobrecarga de funciones** , es la capacidad de una clase para tener múltiples métodos con el mismo nombre, dado que difieren en el número o tipo de argumentos.

El compilador comprueba la ***firma*** del método para la sobrecarga del método.

Método de firma consiste en tres cosas:

1. Nombre del método
2. Numero de parametros
3. Tipos de parametros

Si estos tres son iguales para cualquiera de los dos métodos en una clase, el compilador arroja un **error de método duplicado** .

Este tipo de polimorfismo se denomina polimorfismo de *tiempo de compilación* o *estático* porque el compilador decide el método apropiado a ser llamado durante el tiempo de compilación basado en la lista de argumentos.

```
class Polymorph {

    public int add(int a, int b){
        return a + b;
    }

    public int add(int a, int b, int c){
        return a + b + c;
```

```
    }

    public float add(float a, float b){
        return a + b;
    }

    public static void main(String... args){
        Polymorph poly = new Polymorph();
        int a = 1, b = 2, c = 3;
        float d = 1.5, e = 2.5;

        System.out.println(poly.add(a, b));
        System.out.println(poly.add(a, b, c));
        System.out.println(poly.add(d, e));
    }

}
```

Esto resultará en:

```
2
6
4.000000
```

Los métodos sobrecargados pueden ser estáticos o no estáticos. Esto tampoco afecta la sobrecarga de métodos.

```
public class Polymorph {

    private static void methodOverloaded()
    {
        //No argument, private static method
    }

    private int methodOverloaded(int i)
    {
        //One argument private non-static method
        return i;
    }

    static int methodOverloaded(double d)
    {
        //static Method
        return 0;
    }

    public void methodOverloaded(int i, double d)
    {
        //Public non-static Method
    }
}
```

Además, si cambia el tipo de método de retorno, no podemos obtenerlo como método de sobrecarga.

```
public class Polymorph {
```

```
void methodOverloaded(){
    //No argument and No return type
}

int methodOverloaded(){
    //No argument and int return type
    return 0;
}
```

Método Anulando

El método de anulación es la capacidad de los subtipos para redefinir (anular) el comportamiento de sus supertipos.

En Java, esto se traduce en subclases que anulan los métodos definidos en la súper clase. En Java, todas las variables no primitivas son en realidad `references` , que son similares a los punteros a la ubicación del objeto real en la memoria. Las `references` solo tienen un tipo, que es el tipo con el que se declararon. Sin embargo, pueden apuntar a un objeto de su tipo declarado o cualquiera de sus subtipos.

Cuando se llama a un método en una `reference` , **se invoca** el **método** correspondiente **del objeto real al que se apunta** .

```
class SuperType {
    public void sayHello(){
        System.out.println("Hello from SuperType");
    }

    public void sayBye(){
        System.out.println("Bye from SuperType");
    }
}

class SubType extends SuperType {
    // override the superclass method
    public void sayHello(){
        System.out.println("Hello from SubType");
    }
}

class Test {
    public static void main(String... args){
        SuperType superType = new SuperType();
        superType.sayHello(); // -> Hello from SuperType

        // make the reference point to an object of the subclass
        superType = new SubType();
        // behaviour is governed by the object, not by the reference
        superType.sayHello(); // -> Hello from SubType

        // non-overridden method is simply inherited
        superType.sayBye(); // -> Bye from SuperType
    }
}
```

Reglas a tener en cuenta

Para anular un método en la subclase, el método de anulación (es decir, el que está en la subclase) **DEBE TENER** :

- mismo nombre
- mismo tipo de retorno en el caso de primitivas (se permite una subclase para las clases, esto también se conoce como tipos de retorno covariantes).
- mismo tipo y orden de parametros
- puede lanzar solo aquellas excepciones que están declaradas en la cláusula de lanzamientos del método de la superclase o excepciones que son subclases de las excepciones declaradas. También puede optar por NO lanzar ninguna excepción. Los nombres de los tipos de parámetros no importan. Por ejemplo, void methodX (int i) es igual que void methodX (int k)
- No podemos anular los métodos finales o estáticos. Lo único que podemos hacer es cambiar solo el cuerpo del método.

Agregar comportamiento agregando clases sin tocar el código existente

```
import java.util.ArrayList;
import java.util.List;

import static java.lang.System.out;

public class PolymorphismDemo {

    public static void main(String[] args) {
        List<FlyingMachine> machines = new ArrayList<FlyingMachine>();
        machines.add(new FlyingMachine());
        machines.add(new Jet());
        machines.add(new Helicopter());
        machines.add(new Jet());

        new MakeThingsFly().letTheMachinesFly(machines);
    }
}

class MakeThingsFly {
    public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
        for (FlyingMachine flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
    }
}

class FlyingMachine {
    public void fly() {
        out.println("No implementation");
    }
}

class Jet extends FlyingMachine {
    @Override
    public void fly() {
        out.println("Start, taxi, fly");
    }

    public void bombardment() {
```

```
        out.println("Fire missile");
    }
}

class Helicopter extends FlyingMachine {
    @Override
    public void fly() {
        out.println("Start vertically, hover, fly");
    }
}
```

Explicación

a) La clase `MakeThingsFly` puede trabajar con todo lo que es de tipo `FlyingMachine` .

b) El método `letTheMachinesFly` también funciona sin ningún cambio (!) cuando agrega una nueva clase, por ejemplo, `PropellerPlane` :

```
public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
        for (FlyingMachine flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
    }
}
```

Ese es el poder del polimorfismo. Puedes implementar el principio abierto-cerrado con él.

Funciones virtuales

Los métodos virtuales son métodos en Java que son no estáticos y sin la palabra clave Final en primer plano. Todos los métodos por defecto son virtuales en Java. Los métodos virtuales desempeñan funciones importantes en el polimorfismo porque las clases secundarias en Java pueden anular los métodos de sus clases primarias si la función que se reemplaza no es estática y tiene la misma firma de método.

Hay, sin embargo, algunos métodos que no son virtuales. Por ejemplo, si el método se declara privado o con la palabra clave final, entonces el método no es Virtual.

Considere el siguiente ejemplo modificado de herencia con métodos virtuales de esta publicación de StackOverflow ¿Cómo funcionan las funciones virtuales en C # y Java? :

```
public class A{
    public void hello(){
        System.out.println("Hello");
    }

    public void boo(){
        System.out.println("Say boo");

    }
}

public class B extends A{
     public void hello(){
```

```
        System.out.println("No");
     }

    public void boo(){
        System.out.println("Say haha");

    }
}
```

Si invocamos la clase B y llamamos hola () y boo (), obtendríamos "No" y "Diga jaja" como la salida resultante porque B anula los mismos métodos de A. Aunque el ejemplo anterior es casi exactamente el mismo que Al reemplazar los métodos, es importante entender que los métodos en la clase A son todos, por defecto, virtuales.

Además, podemos implementar métodos virtuales utilizando la palabra clave abstracta. Los métodos declarados con la palabra clave "resumen" no tienen una definición de método, lo que significa que el cuerpo del método aún no está implementado. Considere el ejemplo de arriba nuevamente, excepto que el método boo () se declara abstracto:

```
public class A{
   public void hello(){
        System.out.println("Hello");
    }

    abstract void boo();
}

public class B extends A{
     public void hello(){
        System.out.println("No");
     }

    public void boo(){
        System.out.println("Say haha");

    }
}
```

Si invocamos boo () desde B, la salida seguirá siendo "Say haha", ya que B hereda el método abstracto boo () y hace que boo () muestre "Say haha".

Fuentes utilizadas y lecturas adicionales:

¿Cómo funcionan las funciones virtuales en C # y Java?

Echa un vistazo a esta gran respuesta que proporciona información mucho más completa sobre las funciones virtuales:

¿Puedes escribir funciones / métodos virtuales en Java?

Polimorfismo y diferentes tipos de anulación.

De java tutorial

La definición del diccionario de polimorfismo se refiere a un principio en biología en el que un organismo o especie puede tener muchas formas o etapas diferentes. Este principio también puede aplicarse a la programación orientada a objetos y lenguajes como el lenguaje Java. ***Las subclases de una clase pueden definir sus propios comportamientos únicos y, sin embargo, compartir algunas de las mismas funciones de la clase principal.***

Eche un vistazo a este ejemplo para comprender los diferentes tipos de anulación.

1. La clase base no proporciona implementación y la subclase tiene que anular el método completo - (resumen)
2. La clase base proporciona implementación predeterminada y la subclase puede cambiar el comportamiento
3. La `super.methodName()` agrega extensión a la implementación de la clase base al llamar a `super.methodName()` como primera declaración
4. La clase base define la estructura del algoritmo (método de plantilla) y la subclase anulará una parte del algoritmo

fragmento de código:

```
import java.util.HashMap;

abstract class Game implements Runnable{

    protected boolean runGame = true;
    protected Player player1 = null;
    protected Player player2 = null;
    protected Player currentPlayer = null;

    public Game(){
        player1 = new Player("Player 1");
        player2 = new Player("Player 2");
        currentPlayer = player1;
        initializeGame();
    }

    /* Type 1: Let subclass define own implementation. Base class defines abstract method to
force
        sub-classes to define implementation
    */

    protected abstract void initializeGame();

    /* Type 2: Sub-class can change the behaviour. If not, base class behaviour is applicable
*/
    protected void logTimeBetweenMoves(Player player){
        System.out.println("Base class: Move Duration: player.PlayerActTime -
player.MoveShownTime");
    }

    /* Type 3: Base class provides implementation. Sub-class can enhance base class
implementation by calling
        super.methodName() in first line of the child class method and specific implementation
later */
    protected void logGameStatistics(){
        System.out.println("Base class: logGameStatistics:");
```

```
    }
    /* Type 4: Template method: Structure of base class can't be changed but sub-class can
some part of behaviour */
    protected void runGame() throws Exception{
        System.out.println("Base class: Defining the flow for Game:");
        while (runGame) {
            /*
            1. Set current player
            2. Get Player Move
            */
            validatePlayerMove(currentPlayer);
            logTimeBetweenMoves(currentPlayer);
            Thread.sleep(500);
            setNextPlayer();
        }
        logGameStatistics();
    }
    /* sub-part of the template method, which define child class behaviour */
    protected abstract void validatePlayerMove(Player p);

    protected void setRunGame(boolean status){
        this.runGame = status;
    }
    public void setCurrentPlayer(Player p){
        this.currentPlayer = p;
    }
    public void setNextPlayer(){
        if (currentPlayer == player1) {
            currentPlayer = player2;
        }else{
            currentPlayer = player1;
        }
    }
    public void run(){
        try{
            runGame();
        }catch(Exception err){
            err.printStackTrace();
        }
    }
}

class Player{
    String name;
    Player(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
}

/* Concrete Game implementation  */
class Chess extends Game{
    public Chess(){
        super();
    }
    public void initializeGame(){
        System.out.println("Child class: Initialized Chess game");
    }
    protected void validatePlayerMove(Player p){
```

```
        System.out.println("Child class: Validate Chess move:" + p.getName());
    }
    protected void logGameStatistics(){
        super.logGameStatistics();
        System.out.println("Child class: Add Chess specific logGameStatistics:");
    }
}
class TicTacToe extends Game{
    public TicTacToe(){
        super();
    }
    public void initializeGame(){
        System.out.println("Child class: Initialized TicTacToe game");
    }
    protected void validatePlayerMove(Player p){
        System.out.println("Child class: Validate TicTacToe move:" + p.getName());
    }
}

public class Polymorphism{
    public static void main(String args[]){
        try{

            Game game = new Chess();
            Thread t1 = new Thread(game);
            t1.start();
            Thread.sleep(1000);
            game.setRunGame(false);
            Thread.sleep(1000);

            game = new TicTacToe();
            Thread t2 = new Thread(game);
            t2.start();
            Thread.sleep(1000);
            game.setRunGame(false);

        }catch(Exception err){
            err.printStackTrace();
        }
    }
}
```

salida:

```
Child class: Initialized Chess game
Base class: Defining the flow for Game:
Child class: Validate Chess move:Player 1
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Child class: Validate Chess move:Player 2
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Base class: logGameStatistics:
Child class: Add Chess specific logGameStatistics:

Child class: Initialized TicTacToe game
Base class: Defining the flow for Game:
Child class: Validate TicTacToe move:Player 1
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Child class: Validate TicTacToe move:Player 2
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Base class: logGameStatistics:
```

Lea Polimorfismo en línea: https://riptutorial.com/es/java/topic/980/polimorfismo

Capítulo 147: Preferencias

Examples

Añadiendo oyentes de eventos

Hay dos tipos de eventos emitidos por un `Preferences` del objeto: `PreferenceChangeEvent` y `NodeChangeEvent` .

PreferenceChangeEvent

Un objeto `PreferenceChangeEvent` es emitido por un objeto `Properties` cada vez que cambia uno de los pares clave-valor del nodo. `PreferenceChangeEvent` s se puede escuchar con un `PreferenceChangeListener` :

Java SE 8

```
preferences.addPreferenceChangeListener(evt -> {
    String newValue = evt.getNewValue();
    String changedPreferenceKey = evt.getKey();
    Preferences changedNode = evt.getNode();
});
```

Java SE 8

```
preferences.addPreferenceChangeListener(new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        String newValue = evt.getNewValue();
        String changedPreferenceKey = evt.getKey();
        Preferences changedNode = evt.getNode();
    }
});
```

Este oyente no escuchará los pares clave-valor de los nodos secundarios.

NodeChangeEvent

Este evento se activará cada vez que se agregue o elimine un nodo secundario de un nodo `Properties` .

```
preferences.addNodeChangeListener(new NodeChangeListener() {
    @Override
    public void childAdded(NodeChangeEvent evt) {
        Preferences addedChild = evt.getChild();
        Preferences parentOfAddedChild = evt.getParent();
    }

    @Override
    public void childRemoved(NodeChangeEvent evt) {
```

```
        Preferences removedChild = evt.getChild();
        Preferences parentOfRemovedChild = evt.getParent();
    }
});
```

Obtención de subnodos de preferencias

`Preferences` objetos de `Preferences` siempre representan un nodo específico en un árbol de `Preferences` completo, como este:

```
/userRoot
├── com
│   └── mycompany
│       └── myapp
│           ├── darkApplicationMode=true
│           ├── showExitConfirmation=false
│           └── windowMaximized=true
└── org
    └── myorganization
        └── anotherapp
            ├── defaultFont=Helvetica
            ├── defaultSavePath=/home/matt/Documents
            └── exporting
                ├── defaultFormat=pdf
                └── openInBrowserAfterExport=false
```

Para seleccionar el nodo /com/mycompany/myapp :

1. Por convención, basado en el paquete de una clase:

   ```
   package com.mycompany.myapp;

   // ...

   // Because this class is in the com.mycompany.myapp package, the node
   // /com/mycompany/myapp will be returned.
   Preferences myApp = Preferences.userNodeForPackage(getClass());
   ```

2. Por camino relativo:

   ```
   Preferences myApp = Preferences.userRoot().node("com/mycompany/myapp");
   ```

   El uso de una ruta relativa (una ruta que no comienza con una /) hará que la ruta se resuelva en relación con el nodo principal en el que se resuelve. Por ejemplo, el siguiente ejemplo devolverá el nodo de la ruta /one/two/three/com/mycompany/myapp :

   ```
   Preferences prefix = Preferences.userRoot().node("one/two/three");
   Preferences myAppWithPrefix = prefix.node("com/mycompany/myapp");
   // prefix          is /one/two/three
   // myAppWithPrefix is /one/two/three/com/mycompany/myapp
   ```

3. Por camino absoluto:

   ```
   Preferences myApp = Preferences.userRoot().node("/com/mycompany/myapp");
   ```

   El uso de una ruta absoluta en el nodo raíz no será diferente de usar una ruta relativa. La diferencia es que, si se llama en un subnodo, la ruta se resolverá en relación con el nodo

raíz.

```
Preferences prefix = Preferences.userRoot().node("one/two/three");
Preferences myAppWitoutPrefix = prefix.node("/com/mycompany/myapp");
// prefix            is /one/two/three
// myAppWitoutPrefix is /com/mycompany/myapp
```

Coordinar las preferencias de acceso a través de múltiples instancias de aplicaciones

Todas las instancias de Preferences siempre son seguras para subprocesos en los subprocesos de una única Máquina Virtual de Java (JVM). Debido a que las Preferences se pueden compartir en múltiples JVM, existen métodos especiales que se ocupan de sincronizar los cambios en las máquinas virtuales.

Si tiene una aplicación que se supone que se ejecuta **solo** en una **instancia** , **no** se requiere **sincronización externa** .

Si tiene una aplicación que se ejecuta en **varias instancias** en un solo sistema y, por lo tanto, el acceso a las Preferences debe coordinarse entre las JVM del sistema, entonces se puede usar el **método sync()** de cualquier nodo de Preferences para garantizar que se realicen cambios en el nodo de Preferences visible para otras JVM en el sistema:

```
// Warning: don't use this if your application is intended
// to only run a single instance on a machine once
// (this is probably the case for most desktop applications)
try {
    preferences.sync();
} catch (BackingStoreException e) {
    // Deal with any errors while saving the preferences to the backing storage
    e.printStackTrace();
}
```

Preferencias de exportación

Preferences nodos de Preferences se pueden exportar a un documento XML que representa ese nodo. El árbol XML resultante se puede importar de nuevo. El documento XML resultante recordará si se exportó desde el usuario o las Preferences sistema.

Para exportar un solo nodo, pero **no sus nodos secundarios** :

Java SE 7

```
try (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
}
```

Java SE 7

```
OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
```

```
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException ignored) {}
    }
}
```

Para exportar un solo nodo **con sus nodos hijos** :

Java SE 7

```
try (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
}
```

Java SE 7

```
OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException ignored) {}
    }
}
```

Importando preferencias

Preferences nodos de Preferences se pueden importar desde un documento XML. La importación debe utilizarse junto con la funcionalidad de exportación de Preferences , ya que crea los documentos XML correspondientes correctos.

Los documentos XML recordarán si se exportaron desde el usuario o las Preferences sistema. Por lo tanto, se pueden importar de nuevo en sus respectivos árboles de Preferences , sin que tenga que averiguar o saber de dónde vienen. La función estática descubrirá automáticamente si el documento XML se exportó desde el usuario o las Preferences sistema y los importará automáticamente en el árbol desde el que se exportaron.

Java SE 7

```
try (InputStream is = ...) {
    // This is a static call on the Preferences class
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // Exception whilst reading data from the InputStream
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // Exception whilst parsing the XML document tree
    ipfe.printStackTrace();
}
```

Java SE 7

```
InputStream is = null;
try {
    is = ...;
    // This is a static call on the Preferences class
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // Exception whilst reading data from the InputStream
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // Exception whilst parsing the XML document tree
    ipfe.printStackTrace();
} finally {
    if (is != null) {
        try {
            is.close();
        } catch (IOException ignored) {}
    }
}
```

Eliminar oyentes de eventos

Los escuchas de eventos pueden eliminarse nuevamente de cualquier nodo de Properties , pero la instancia del oyente debe mantenerse alrededor para eso.

Java SE 8

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

PreferenceChangeListener listener = evt -> {
    System.out.println(evt.getKey() + " got new value " + evt.getNewValue());
};
preferences.addPreferenceChangeListener(listener);

//
// later...
//

preferences.removePreferenceChangeListener(listener);
```

Java SE 8

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

PreferenceChangeListener listener = new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        System.out.println(evt.getKey() + " got new value " + evt.getNewValue());
```

```
    }
};
preferences.addPreferenceChangeListener(listener);

//
// later...
//

preferences.removePreferenceChangeListener(listener);
```

Lo mismo se aplica para NodeChangeListener .

Obteniendo valores de preferencias

Un valor de un nodo de Preferences puede ser del tipo String , boolean , byte[] , double , float , int o long . Todas las invocaciones deben proporcionar un valor predeterminado, en caso de que el valor especificado no esté presente en el nodo Preferences .

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

String someString = preferences.get("someKey", "this is the default value");
boolean someBoolean = preferences.getBoolean("someKey", true);
byte[] someByteArray = preferences.getByteArray("someKey", new byte[0]);
double someDouble = preferences.getDouble("someKey", 887284.4d);
float someFloat = preferences.getFloat("someKey", 38723.3f);
int someInt = preferences.getInt("someKey", 13232);
long someLong = preferences.getLong("someKey", 2827637868234L);
```

Configuración de valores de preferencias

Para almacenar un valor en el nodo Preferences , se putXXX() uno de los métodos putXXX() . Un valor de un nodo de Preferences puede ser del tipo String , boolean , byte[] , double , float , int o long .

```
Preferences preferences = Preferences.userNodeForPackage(getClass());

preferences.put("someKey", "some String value");
preferences.putBoolean("someKey", false);
preferences.putByteArray("someKey", new byte[0]);
preferences.putDouble("someKey", 187398123.4454d);
preferences.putFloat("someKey", 298321.445f);
preferences.putInt("someKey", 77637);
preferences.putLong("someKey", 2873984729834L);
```

Usando preferencias

Preferences se pueden usar para almacenar la configuración del usuario que refleja la configuración personal de la aplicación del usuario, por ejemplo, la fuente de su editor, si prefieren que la aplicación se inicie en modo de pantalla completa, si marcó la casilla de verificación "No volver a mostrar esto" y demás. como eso.

```
public class ExitConfirmer {
    private static boolean confirmExit() {
        Preferences preferences = Preferences.userNodeForPackage(ExitConfirmer.class);
        boolean doShowDialog = preferences.getBoolean("showExitConfirmation", true); // true
is default value

        if (!doShowDialog) {
```

```
            return true;
        }

        //
        // Show a dialog here...
        //
        boolean exitWasConfirmed = ...; // whether the user clicked OK or Cancel
        boolean doNotShowAgain = ...; // get value from "Do not show again" checkbox

        if (exitWasConfirmed && doNotShowAgain) {
            // Exit was confirmed and the user chose that the dialog should not be shown again
            // Save these settings to the Preferences object so the dialog will not show again
next time
            preferences.putBoolean("showExitConfirmation", false);
        }

        return exitWasConfirmed;
    }

    public static void exit() {
        if (confirmExit()) {
            System.exit(0);
        }
    }
}
```

Lea Preferencias en línea: https://riptutorial.com/es/java/topic/582/preferencias

Capítulo 148: Procesamiento de argumentos de línea de comando

Sintaxis

- Public static void main (String [] args)

Parámetros

Parámetro	Detalles
args	Los argumentos de la línea de comandos. Suponiendo que el lanzador de Java invoca el método `main` , `args` no será nulo y no tendrá elementos `null` .

Observaciones

Cuando se lanza una aplicación Java normal usando el comando java (o equivalente), se llamará a un método main , pasando los argumentos desde la línea de comando en la matriz args .

Desafortunadamente, las bibliotecas de clase de Java SE no proporcionan ningún soporte directo para el procesamiento de argumentos de comando. Esto te deja dos alternativas:

- Implementar el procesamiento de argumentos a mano en Java.
- Hacer uso de una biblioteca de terceros.

Este tema intentará cubrir algunas de las bibliotecas de terceros más populares. Para obtener una lista extensa de las alternativas, consulte esta respuesta a la pregunta de StackOverflow "¿Cómo analizar los argumentos de la línea de comandos en Java?" .

Examples

Procesamiento de argumentos utilizando GWT ToolBase

Si desea analizar argumentos de línea de comando más complejos, por ejemplo, con parámetros opcionales, lo mejor es utilizar el enfoque GWT de Google. Todas las clases son públicas disponibles en:

https://gwt.googlesource.com/gwt/+/2.8.0-beta1/dev/core/src/com/google/gwt/util/tools/ToolBase.java

Un ejemplo para manejar la línea de comandos myprogram -dir "~/Documents" -port 8888 es:

```
public class MyProgramHandler extends ToolBase {
   protected File dir;
   protected int port;
   // getters for dir and port
   ...

   public MyProgramHandler() {
       this.registerHandler(new ArgHandlerDir() {
            @Override
            public void setDir(File dir) {
                this.dir = dir;
            }
       });
       this.registerHandler(new ArgHandlerInt() {
            @Override
```

```
            public String[] getTagArgs() {
               return new String[]{"port"};
            }
            @Override
            public void setInt(int value) {
               this.port = value;
            }
        });
    }
    public static void main(String[] args) {
       MyProgramHandler myShell = new MyProgramHandler();
       if (myShell.processArgs(args)) {
          // main program operation
          System.out.println(String.format("port: %d; dir: %s",
             myShell.getPort(), myShell.getDir()));
       }
       System.exit(1);
    }
}
```

ArgHandler también tiene un método isRequired() que se puede sobrescribir para decir que se requiere el argumento de la línea de comandos (el retorno predeterminado es false por lo que el argumento es opcional.

Procesando argumentos a mano

Cuando la sintaxis de la línea de comandos para una aplicación es simple, es razonable realizar el procesamiento de los argumentos del comando completamente en código personalizado.

En este ejemplo, presentaremos una serie de estudios de casos simples. En cada caso, el código generará mensajes de error si los argumentos son inaceptables, y luego llame a System.exit(1) para decirle al shell que el comando ha fallado. (Supondremos en cada caso que el código Java se invoca mediante un contenedor cuyo nombre es "myapp").

Un comando sin argumentos.

En este estudio de caso, el comando no requiere argumentos. El código ilustra que args.length nos da el número de argumentos de línea de comando.

```
public class Main {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.err.println("usage: myapp");
            System.exit(1);
        }
        // Run the application
        System.out.println("It worked");
    }
}
```

Un comando con dos argumentos.

En este estudio de caso, el comando requiere precisamente dos argumentos.

```
public class Main {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("usage: myapp <arg1> <arg2>");
            System.exit(1);
        }
```

```
        // Run the application
        System.out.println("It worked: " + args[0] + ", " + args[1]);
    }
}
```

Tenga en cuenta que si no args.length , el comando se bloquearía si el usuario lo ejecutara con muy pocos argumentos de línea de comandos.

Un comando con opciones de "bandera" y al menos un argumento

En este estudio de caso, el comando tiene un par de opciones de marca (opcionales) y requiere al menos un argumento después de las opciones.

```
package tommy;
public class Main {
    public static void main(String[] args) {
        boolean feelMe = false;
        boolean seeMe = false;
        int index;
        loop: for (index = 0; index < args.length; index++) {
            String opt = args[index];
            switch (opt) {
            case "-c":
               seeMe = true;
               break;
            case "-f":
               feelMe = true;
               break;
            default:
               if (!opts.isEmpty() && opts.charAt(0) == '-') {
                   error("Unknown option: '" + opt + "');
               }
               break loop;
            }
        }
        if (index >= args.length) {
            error("Missing argument(s)");
        }

        // Run the application
        // ...
    }

    private static void error(String message) {
        if (message != null) {
            System.err.println(message);
        }
        System.err.println("usage: myapp [-f] [-c] [ <arg> ...]");
        System.exit(1);
   }
}
```

Como puede ver, el procesamiento de los argumentos y las opciones se vuelve un tanto engorroso si la sintaxis del comando es complicada. Es recomendable utilizar una biblioteca de "análisis de línea de comando"; ver los otros ejemplos

Lea Procesamiento de argumentos de línea de comando en línea:
https://riptutorial.com/es/java/topic/4775/procesamiento-de-argumentos-de-linea-de-comando

Capítulo 149: Proceso

Observaciones

Tenga en cuenta que la API recomienda que, a partir de la versión 1.5, la forma preferida de crear un proceso es utilizar ProcessBuilder.start() .

Otra observación importante es que el valor de salida producido por waitFor depende del programa / script que se ejecuta. Por ejemplo, los códigos de salida producidos por **calc.exe** son diferentes de **notepad.exe** .

Examples

Ejemplo simple (versión de Java <1.5)

Este ejemplo llamará a la calculadora de windows. Es importante notar que el código de salida variará de acuerdo con el programa / script que se está llamando.

```
package process.example;

import java.io.IOException;

public class App {

    public static void main(String[] args) {
        try {
            // Executes windows calculator
            Process p = Runtime.getRuntime().exec("calc.exe");

            // Wait for process until it terminates
            int exitCode = p.waitFor();

            System.out.println(exitCode);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Usando la clase ProcessBuilder

La clase ProcessBuilder facilita el envío de un comando a través de la línea de comandos. Todo lo que requiere es una lista de cadenas que conforman los comandos que se ingresarán. Simplemente llame al método start () en su instancia de ProcessBuilder para ejecutar el comando.

Si tiene un programa llamado Add.exe que toma dos argumentos y los agrega, el código se vería así:

```
List<String> cmds = new ArrayList<>();
cmds.add("Add.exe"); //the name of the application to be run
cmds.add("1"); //the first argument
cmds.add("5"); //the second argument

ProcessBuilder pb = new ProcessBuilder(cmds);

//Set the working directory of the ProcessBuilder so it can find the .exe
```

```
//Alternatively you can just pass in the absolute file path of the .exe
File myWorkingDirectory = new File(yourFilePathNameGoesHere);
pb.workingDirectory(myWorkingDirectory);

try {
    Process p = pb.start();
} catch (IOException e) {
    e.printStackTrace();
}
```

Algunas cosas para tener en mente:

- La matriz de comandos debe ser una cadena String
- Los comandos deben estar en el orden (en la matriz) que serían si hiciera la llamada al programa en la misma línea de comandos (es decir, el nombre del archivo .exe no puede ir después del primer argumento).
- Al configurar el directorio de trabajo, debe pasar un objeto de archivo y no solo el nombre del archivo como una cadena

Bloqueo frente a llamadas no bloqueadas

En general, al realizar una llamada a la línea de comando, el programa enviará el comando y luego continuará su ejecución.

Sin embargo, es posible que desee esperar a que finalice el programa llamado antes de continuar con su propia ejecución (por ejemplo, el programa llamado escribirá los datos en un archivo y su programa necesita ese acceso para acceder a esos datos).

Esto se puede hacer fácilmente llamando al método waitFor() desde la instancia de Process devuelta.

Ejemplo de uso:

```
//code setting up the commands omitted for brevity...

ProcessBuilder pb = new ProcessBuilder(cmds);

try {
    Process p = pb.start();
    p.waitFor();
} catch (IOException e) {
    e.printStackTrack();
} catch (InterruptedException e) {
    e.printStackTrace();
}

//more lines of code here...
```

ch.vorburger.exec

Lanzar procesos externos desde Java utilizando la API java.lang.ProcessBuilder sin formato directamente puede ser un poco engorroso. La biblioteca de Apache Commons Exec lo hace un poco más fácil. La biblioteca ch.vorburger.exec se extiende más allá de Commons Exec para que sea realmente conveniente:

```
 ManagedProcess proc = new ManagedProcessBuilder("path-to-your-executable-binary")
     .addArgument("arg1")
     .addArgument("arg2")
     .setWorkingDirectory(new File("/tmp"))
     .setDestroyOnShutdown(true)
```

```
     .setConsoleBufferMaxLines(7000)
     .build();

proc.start();
int status = proc.waitForExit();
int status = proc.waitForExitMaxMsOrDestroy(3000);
String output = proc.getConsole();

proc.startAndWaitForConsoleMessageMaxMs("started!", 7000);
// use service offered by external process...
proc.destroy();
```

Pitfall: Runtime.exec, Process y ProcessBuilder no entienden la sintaxis de shell

Los métodos Runtime.exec(String ...) y Runtime.exec(String) permiten ejecutar un comando como un proceso externo [1] . En la primera versión, proporciona el nombre del comando y los argumentos del comando como elementos separados de la cadena de cadenas, y el tiempo de ejecución de Java solicita al sistema de tiempo de ejecución del sistema operativo que inicie el comando externo. La segunda versión es aparentemente fácil de usar, pero tiene algunas trampas.

En primer lugar, aquí hay un ejemplo de uso exec(String) se usa de manera segura:

```
Process p = Runtime.exec("mkdir /tmp/testDir");
p.waitFor();
if (p.exitValue() == 0) {
    System.out.println("created the directory");
}
```

Espacios en las rutas

Supongamos que generalizamos el ejemplo anterior para que podamos crear un directorio arbitrario:

```
Process p = Runtime.exec("mkdir " + dirPath);
// ...
```

Esto funcionará normalmente, pero fallará si dirPath es (por ejemplo) "/ home / user / My Documents". El problema es que exec(String) divide la cadena en un comando y argumentos simplemente buscando espacios en blanco. La cadena de comando:

```
"mkdir /home/user/My Documents"
```

se dividirá en:

```
"mkdir", "/home/user/My", "Documents"
```

y esto hará que el comando "mkdir" falle porque espera un argumento, no dos.

Ante esto, algunos programadores intentan agregar citas alrededor de la ruta de acceso. Esto tampoco funciona:

```
"mkdir \"/home/user/My Documents\""
```

se dividirá en:

```
"mkdir", "\"/home/user/My", "Documents\""
```

Los caracteres de comillas dobles adicionales que se agregaron para intentar "citar" los

espacios se tratan como cualquier otro carácter que no sea un espacio en blanco. De hecho, cualquier cosa que hagamos o escapemos de los espacios va a fallar.

La forma de solucionar este problema en particular es usar la sobrecarga exec(String ...) .

```
Process p = Runtime.exec("mkdir", dirPath);
// ...
```

Esto funcionará si dirpath incluye caracteres de espacio en blanco porque esta sobrecarga de exec no intenta dividir los argumentos. Las cadenas se pasan a la llamada del sistema exec sistema operativo tal como está.

Redireccionamiento, tuberías y otra sintaxis de shell.

Supongamos que queremos redireccionar la entrada o salida de un comando externo, o ejecutar una canalización. Por ejemplo:

```
Process p = Runtime.exec("find / -name *.java -print 2>/dev/null");
```

o

```
Process p = Runtime.exec("find source -name *.java | xargs grep package");
```

(El primer ejemplo enumera los nombres de todos los archivos Java en el sistema de archivos, y el segundo imprime las declaraciones del package [2] en los archivos Java en el árbol "fuente").

Estos no van a funcionar como se esperaba. En el primer caso, el comando "buscar" se ejecutará con "2> / dev / null" como un argumento de comando. No se interpretará como una redirección. En el segundo ejemplo, el carácter de canalización ("|") y las obras siguientes se asignarán al comando "buscar".

El problema aquí es que los métodos exec y ProcessBuilder no comprenden ninguna sintaxis de shell. Esto incluye redirecciones, conductos, expansión variable, globbing, etc.

En algunos casos (por ejemplo, redirección simple) puede lograr fácilmente el efecto deseado utilizando ProcessBuilder . Sin embargo, esto no es cierto en general. Un enfoque alternativo es ejecutar la línea de comandos en un shell; por ejemplo:

```
Process p = Runtime.exec("bash", "-c",
                         "find / -name *.java -print 2>/dev/null");
```

o

```
Process p = Runtime.exec("bash", "-c",
                         "find source -name \\*.java | xargs grep package");
```

Pero tenga en cuenta que en el segundo ejemplo, necesitábamos escapar del carácter comodín ("*") porque queremos que el comodín sea interpretado por "buscar" en lugar de por el shell.

Los comandos de shell incorporados no funcionan

Supongamos que los siguientes ejemplos no funcionarán en un sistema con un shell similar a UNIX:

```
Process p = Runtime.exec("cd", "/tmp");     // Change java app's home directory
```

o

```
Process p = Runtime.exec("export", "NAME=value");  // Export NAME to the java app's
environment
```

Hay un par de razones por las que esto no funciona:

1. En "cd" y "exportar" los comandos son comandos incorporados en el shell. No existen como ejecutables distintos.

2. Para que los shell construidos puedan hacer lo que se supone que deben hacer (por ejemplo, cambiar el directorio de trabajo, actualizar el entorno), deben cambiar el lugar donde reside ese estado. Para una aplicación normal (incluida una aplicación Java), el estado está asociado con el proceso de la aplicación. Así, por ejemplo, el proceso hijo que ejecutaría el comando "cd" no pudo cambiar el directorio de trabajo de su proceso "java" principal. De manera similar, un proceso exec no puede cambiar el directorio de trabajo para un proceso que lo sigue.

Este razonamiento se aplica a todos los comandos incorporados del shell.

---

1 - También puede usar `ProcessBuilder` , pero eso no es relevante hasta el punto de este ejemplo.

2 - Esto es un poco áspero y listo ... pero una vez más, las fallas de este enfoque no son relevantes para el ejemplo.

Lea Proceso en línea: https://riptutorial.com/es/java/topic/4682/proceso

Capítulo 150: Programación Concurrente (Hilos)

Introducción

La computación concurrente es una forma de computación en la cual varios cálculos se ejecutan simultáneamente en lugar de secuencialmente. El lenguaje Java está diseñado para admitir la programación concurrente mediante el uso de subprocesos. Los objetos y recursos pueden ser accedidos por múltiples hilos; cada subproceso puede potencialmente acceder a cualquier objeto en el programa y el programador debe garantizar que los accesos de lectura y escritura a los objetos estén correctamente sincronizados entre los subprocesos.

Observaciones

Tema (s) relacionado (s) en StackOverflow:

- Tipos atómicos
- Ejecutor, ExecutorService y grupos de subprocesos
- Extendiendo Thread versus implementando Runnable

Examples

Multihilo básico

Si tiene muchas tareas que ejecutar y todas estas tareas no dependen del resultado de las anteriores, puede usar **Multithreading** para su computadora para realizar todas estas tareas al mismo tiempo con más procesadores si su computadora puede. Esto puede hacer que la ejecución de su programa sea **más rápida** si tiene algunas grandes tareas independientes.

```
class CountAndPrint implements Runnable {

    private final String name;

    CountAndPrint(String name) {
        this.name = name;
    }

    /** This is what a CountAndPrint will do */
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.println(this.name + ": " + i);
        }
    }

    public static void main(String[] args) {
        // Launching 4 parallel threads
        for (int i = 1; i <= 4; i++) {
            // `start` method will call the `run` method
            // of CountAndPrint in another thread
            new Thread(new CountAndPrint("Instance " + i)).start();
        }

        // Doing some others tasks in the main Thread
        for (int i = 0; i < 10000; i++) {
            System.out.println("Main: " + i);
        }
    }
}
```

El código del método de ejecución de las distintas instancias de CountAndPrint se ejecutará en un orden no predecible. Un fragmento de una ejecución de muestra podría verse así:

```
Instance 4: 1
Instance 2: 1
Instance 4: 2
Instance 1: 1
Instance 1: 2
Main: 1
Instance 4: 3
Main: 2
Instance 3: 1
Instance 4: 4
...
```

Productor-consumidor

Un ejemplo simple de solución de problema productor-consumidor. Tenga en cuenta que las clases JDK (AtomicBoolean y BlockingQueue) se utilizan para la sincronización, lo que reduce la posibilidad de crear una solución no válida. Consulte Javadoc para varios tipos de BlockingQueue ; elegir una implementación diferente puede cambiar drásticamente el comportamiento de este ejemplo (como DelayQueue o Priority Queue).

```
public class Producer implements Runnable {

    private final BlockingQueue<ProducedData> queue;

    public Producer(BlockingQueue<ProducedData> queue) {
        this.queue = queue;
    }

    public void run() {
        int producedCount = 0;
        try {
            while (true) {
                producedCount++;
                //put throws an InterruptedException when the thread is interrupted
                queue.put(new ProducedData());
            }
        } catch (InterruptedException e) {
            // the thread has been interrupted: cleanup and exit
            producedCount--;
            //re-interrupt the thread in case the interrupt flag is neededed higher up
            Thread.currentThread().interrupt();
        }
        System.out.println("Produced " + producedCount + " objects");
    }
}

public class Consumer implements Runnable {

    private final BlockingQueue<ProducedData> queue;

    public Consumer(BlockingQueue<ProducedData> queue) {
        this.queue = queue;
    }

    public void run() {
        int consumedCount = 0;
        try {
            while (true) {
```

```
                //put throws an InterruptedException when the thread is interrupted
                ProducedData data = queue.poll(10, TimeUnit.MILLISECONDS);
                // process data
                consumedCount++;
            }
        } catch (InterruptedException e) {
            // the thread has been interrupted: cleanup and exit
            consumedCount--;
            //re-interrupt the thread in case the interrupt flag is needeed higher up
            Thread.currentThread().interrupt();
        }
        System.out.println("Consumed " + consumedCount + " objects");
    }
}


public class ProducerConsumerExample {
    static class ProducedData {
        // empty data object
    }

    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<ProducedData> queue = new ArrayBlockingQueue<ProducedData>(1000);
        // choice of queue determines the actual behavior: see various BlockingQueue
implementations

        Thread producer = new Thread(new Producer(queue));
        Thread consumer = new Thread(new Consumer(queue));

        producer.start();
        consumer.start();

        Thread.sleep(1000);
        producer.interrupt();
        Thread.sleep(10);
        consumer.interrupt();
    }
}
```

Usando ThreadLocal

Una herramienta útil en Java Concurrency es ThreadLocal : esto le permite tener una variable que será única para un hilo determinado. Por lo tanto, si el mismo código se ejecuta en subprocesos diferentes, estas ejecuciones no compartirán el valor, sino que cada subproceso tiene su propia variable que es *local al subproceso* .

Por ejemplo, esto se usa con frecuencia para establecer el contexto (como la información de autorización) del manejo de una solicitud en un servlet. Podrías hacer algo como esto:

```
private static final ThreadLocal<MyUserContext> contexts = new ThreadLocal<>();

public static MyUserContext getContext() {
    return contexts.get(); // get returns the variable unique to this thread
}

public void doGet(...) {
    MyUserContext context = magicGetContextFromRequest(request);
    contexts.put(context); // save that context to our thread-local - other threads
                           // making this call don't overwrite ours
    try {
```

```
        // business logic
    } finally {
        contexts.remove(); // 'ensure' removal of thread-local variable
    }
}
```

Ahora, en lugar de pasar MyUserContext a cada método, puede usar MyServlet.getContext() donde lo necesite. Ahora, por supuesto, esto introduce una variable que necesita ser documentada, pero es segura para subprocesos, lo que elimina muchas de las desventajas de usar una variable de alto alcance.

La ventaja clave aquí es que cada hilo tiene su propia variable local de hilo en ese contenedor de contexts . Siempre que lo use desde un punto de entrada definido (como exigir que cada servlet mantenga su contexto, o quizás agregando un filtro de servlet) puede confiar en que este contexto estará allí cuando lo necesite.

CountDownLatch

CountDownLatch

> Una ayuda de sincronización que permite que uno o más subprocesos esperen hasta que se complete un conjunto de operaciones que se están realizando en otros subprocesos.

1. Un CountDownLatch se inicializa con un recuento dado.
2. El bloque de métodos de espera hasta que el conteo actual llegue a cero debido a invocaciones del método countDown() , después de lo cual se liberan todos los subprocesos en espera y cualquier invocación posterior de espera regresa de inmediato.
3. Este es un fenómeno de un solo disparo: el conteo no se puede restablecer. Si necesita una versión que restablezca la cuenta, considere usar un CyclicBarrier .

Métodos clave:

```
public void await() throws InterruptedException
```

> Provoca que el subproceso actual espere hasta que el pestillo haya descendido hasta cero, a menos que se interrumpa el subproceso.

```
public void countDown()
```

> Reduce el conteo del pestillo, liberando todos los hilos en espera si el conteo llega a cero.

Ejemplo:

```
import java.util.concurrent.*;

class DoSomethingInAThread implements Runnable {
    CountDownLatch latch;
    public DoSomethingInAThread(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            System.out.println("Do some thing");
            latch.countDown();
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}
```

```
public class CountDownLatchDemo {
    public static void main(String[] args) {
        try {
            int numberOfThreads = 5;
            if (args.length < 1) {
                System.out.println("Usage: java CountDownLatchDemo numberOfThreads");
                return;
            }
            try {
                numberOfThreads = Integer.parseInt(args[0]);
            } catch(NumberFormatException ne) {

            }
            CountDownLatch latch = new CountDownLatch(numberOfThreads);
            for (int n = 0; n < numberOfThreads; n++) {
                Thread t = new Thread(new DoSomethingInAThread(latch));
                t.start();
            }
            latch.await();
            System.out.println("In Main thread after completion of " + numberOfThreads + "
threads");
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}
```

salida:

```
java CountDownLatchDemo 5
Do some thing
Do some thing
Do some thing
Do some thing
Do some thing
In Main thread after completion of 5 threads
```

Explicación:

1. CountDownLatch se inicializa con un contador de 5 en el subproceso principal
2. El hilo principal está esperando usando el método await() .
3. Se han creado cinco instancias de DoSomethingInAThread . Cada instancia decrementó el contador con el método countDown() .
4. Una vez que el contador se vuelve cero, el hilo principal se reanudará

Sincronización

En Java, hay un mecanismo integrado de bloqueo de nivel de idioma: el bloque synchronized , que puede usar cualquier objeto Java como un bloqueo intrínseco (es decir, cada objeto Java puede tener un monitor asociado).

Los bloqueos intrínsecos proporcionan atomicidad a grupos de afirmaciones. Para entender lo que eso significa para nosotros, echemos un vistazo a un ejemplo donde la synchronized es útil:

```
private static int t = 0;
private static Object mutex = new Object();

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread
```

```
count is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            synchronized (mutex) {
                t++;
                System.out.println(MessageFormat.format("t: {0}", t));
            }
        });
    }
    executorService.shutdown();
}
```

En este caso, si no fuera por el bloque synchronized , habría habido múltiples problemas de concurrencia involucrados. El primero sería con el operador de incremento posterior (no es atómico en sí mismo), y el segundo sería que observaríamos el valor de t después de que una cantidad arbitraria de otros subprocesos haya tenido la oportunidad de modificarlo. Sin embargo, desde que adquirimos un bloqueo intrínseco, no habrá condiciones de carrera aquí y la salida contendrá números del 1 al 100 en su orden normal.

Los bloqueos intrínsecos en Java son *exclusiones mutuas* (es decir, bloqueos de ejecución mutua). La ejecución mutua significa que si un subproceso ha adquirido el bloqueo, el segundo se verá obligado a esperar a que el primero lo libere antes de que pueda adquirir el bloqueo por sí mismo. Nota: Una operación que puede poner el hilo en el estado de espera (reposo) se denomina *operación de bloqueo* . Por lo tanto, la adquisición de un bloqueo es una operación de bloqueo.

Los bloqueos intrínsecos en Java son *reentrantes* . Esto significa que si un hilo intenta adquirir un bloqueo que ya posee, no lo bloqueará y lo logrará con éxito. Por ejemplo, el siguiente código *no se* bloqueará cuando se le llame:

```
public void bar(){
    synchronized(this){
        ...
    }
}
public void foo(){
    synchronized(this){
        bar();
    }
}
```

Además de synchronized bloques synchronized , también hay métodos synchronized .

Los siguientes bloques de código son prácticamente equivalentes (aunque el código de bytes parece ser diferente):

1. Bloque synchronized en this :

```
public void foo() {
    synchronized(this) {
        doStuff();
    }
}
```

2. método synchronized :

```
public synchronized void foo() {
    doStuff();
}
```

Igualmente para static métodos static , esto:

```
class MyClass {
    ...
    public static void bar() {
        synchronized(MyClass.class) {
            doSomeOtherStuff();
        }
    }
}
```

tiene el mismo efecto que este:

```
class MyClass {
    ...
    public static synchronized void bar() {
        doSomeOtherStuff();
    }
}
```

Operaciones atómicas

Una operación atómica es una operación que se ejecuta "todo a la vez", sin ninguna posibilidad de que otros hilos observen o modifiquen el estado durante la ejecución de la operación atómica.

Consideremos un **mal ejemplo** .

```
private static int t = 0;

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread
count is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            t++;
            System.out.println(MessageFormat.format("t: {0}", t));
        });
    }
    executorService.shutdown();
}
```

En este caso, hay dos cuestiones. El primer problema es que el operador de incremento posterior *no* es atómico. Se compone de varias operaciones: obtener el valor, agregar 1 al valor, establecer el valor. Es por eso que si ejecutamos el ejemplo, es probable que no veamos t: 100 en la salida; dos subprocesos pueden obtener el valor al mismo tiempo, incrementarlo y establecerlo: digamos que el valor de t es 10, y dos subprocesos están aumentando t. Ambos subprocesos establecerán el valor de t en 11, ya que el segundo subproceso observa el valor de t antes de que el primer subproceso haya terminado de incrementarlo.

El segundo tema es sobre cómo estamos observando t. Cuando estamos imprimiendo el valor de t, es posible que el valor ya haya sido cambiado por un hilo diferente después de la operación de incremento de este hilo.

Para solucionar esos problemas, usaremos el `java.util.concurrent.atomic.AtomicInteger` , que tiene muchas operaciones atómicas para que utilicemos.

```
private static AtomicInteger t = new AtomicInteger(0);

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread
count is for demonstration purposes.
```

```
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            int currentT = t.incrementAndGet();
            System.out.println(MessageFormat.format("t: {0}", currentT));
        });
    }
    executorService.shutdown();
}
```

El método incrementAndGet de AtomicInteger incrementa y devuelve el nuevo valor, eliminando así la condición de carrera anterior. Tenga en cuenta que, en este ejemplo, las líneas aún estarán fuera de orden porque no hacemos ningún esfuerzo por secuenciar las llamadas de println y que esto queda fuera del alcance de este ejemplo, ya que requeriría sincronización y el objetivo de este ejemplo es mostrar cómo usar AtomicInteger para eliminar las condiciones de la raza concernientes al estado.

Creación de un sistema básico de punto muerto.

Se produce un punto muerto cuando dos acciones en competencia esperan a que la otra termine, y por lo tanto ninguna de las dos lo hace. En java hay un bloqueo asociado a cada objeto. Para evitar la modificación simultánea realizada por varios subprocesos en un solo objeto, podemos usar palabras clave synchronized , pero todo tiene un costo. El uso incorrecto de una palabra clave synchronized puede llevar a sistemas atascados llamados sistemas bloqueados.

Considere que hay 2 subprocesos trabajando en 1 instancia, permite llamar a los subprocesos como primero y segundo, y digamos que tenemos 2 recursos R1 y R2. First adquiere R1 y también necesita R2 para completarse, mientras que Second adquiere R2 y necesita R1 para completarse.

digamos en el tiempo t = 0,

Primero tiene R1 y Segundo tiene R2. ahora First está esperando a R2 mientras que Second está esperando a R1. esta espera es indefinida y esto lleva a un punto muerto.

```
public class Example2 {

    public static void main(String[] args) throws InterruptedException {
        final DeadLock dl = new DeadLock();
        Thread t1 = new Thread(new Runnable() {

            @Override
            public void run() {
                // TODO Auto-generated method stub
                dl.methodA();
            }
        });

        Thread t2 = new Thread(new Runnable() {

            @Override
            public void run() {
                // TODO Auto-generated method stub
                try {
                    dl.method2();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        });
        t1.setName("First");
        t2.setName("Second");
```

```
        t1.start();
        t2.start();
    }
}

class DeadLock {

    Object mLock1 = new Object();
    Object mLock2 = new Object();


    public void methodA() {
        System.out.println("methodA wait for mLock1  " + Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("methodA mLock1 acquired   " +
Thread.currentThread().getName());
            try {
                Thread.sleep(100);
                method2();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
    public void method2() throws InterruptedException {
        System.out.println("method2 wait for mLock2  " + Thread.currentThread().getName());
        synchronized (mLock2) {
            System.out.println("method2  mLock2 acquired   " +
Thread.currentThread().getName());
            Thread.sleep(100);
            method3();
        }
    }
    public void method3() throws InterruptedException {
        System.out.println("method3  mLock1  "+ Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("method3   mLock1 acquired  " +
Thread.currentThread().getName());
        }
    }
}
```

Salida de este programa:

```
methodA wait for mLock1  First
method2 wait for mLock2  Second
method2  mLock2 acquired   Second
methodA mLock1 acquired   First
method3  mLock1  Second
method2 wait for mLock2  First
```

Pausa de ejecucion

Thread.sleep hace que el subproceso actual suspenda la ejecución durante un período específico. Este es un medio eficaz para hacer que el tiempo del procesador esté disponible para los otros subprocesos de una aplicación u otras aplicaciones que puedan estar ejecutándose en un sistema informático. Hay dos métodos de sleep sobrecargados en la clase Thread.

Uno que especifica el tiempo de reposo al milisegundo.

```
public static void sleep(long millis) throws InterruptedException
```

Una que especifica el tiempo de reposo al nanosegundo.

```
public static void sleep(long millis, int nanos)
```

Pausando Ejecución por 1 segundo

```
Thread.sleep(1000);
```

Es importante tener en cuenta que esto es una sugerencia para el programador del kernel del sistema operativo. Esto puede no ser necesariamente preciso, y algunas implementaciones ni siquiera consideran el parámetro de nanosegundo (posiblemente redondeando al milisegundo más cercano).

Se recomienda incluir una llamada a Thread.sleep en try / catch and catch InterruptedException .

Visualizar barreras de lectura / escritura mientras se usa sincronizado / volátil

Como sabemos, debemos usar palabras clave synchronized para hacer que la ejecución de un método o bloque sea exclusiva. Pero es posible que pocos de nosotros no estemos conscientes de un aspecto más importante del uso de palabras clave synchronized y volatile : *además de hacer una unidad de código atómico, también proporciona una barrera de lectura / escritura* . ¿Qué es esta barrera de lectura / escritura? Vamos a discutir esto usando un ejemplo:

```
class Counter {

  private Integer count = 10;

  public synchronized void incrementCount() {
    count++;
  }

  public Integer getCount() {
    return count;
  }
}
```

Supongamos que un subproceso *A* llama a incrementCount() primero y luego otro subproceso *B* llama a getCount() . En este escenario, no hay garantía de que B vea el valor actualizado del count . Todavía puede ver el count como 10 , incluso es posible que nunca vea el valor actualizado del count nunca.

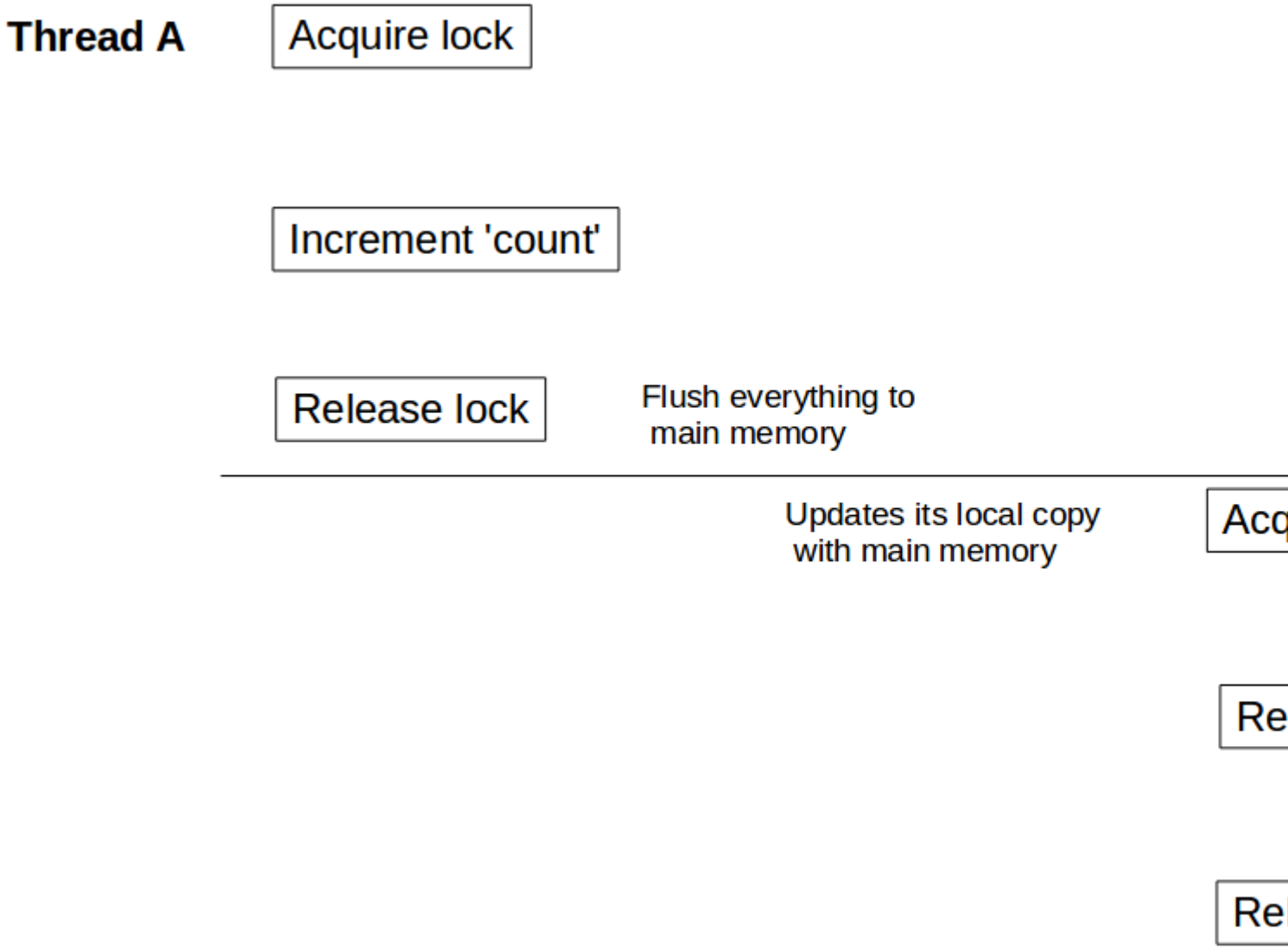
Para comprender este comportamiento, debemos comprender cómo se integra el modelo de memoria Java con la arquitectura de hardware. En Java, cada hilo tiene su propia pila de hilos. Esta pila contiene: pila de llamada a método y variable local creada en ese hilo. En un sistema de varios núcleos, es muy posible que dos subprocesos se ejecuten simultáneamente en núcleos separados. En tal escenario, es posible que parte de la pila de un hilo se encuentre dentro del registro / caché de un núcleo. Si dentro de un hilo, se accede a un objeto usando una palabra clave synchronized (o volatile), después del bloque synchronized ese hilo sincroniza su copia local de esa variable con la memoria principal. Esto crea una barrera de lectura / escritura y se asegura de que el hilo vea el último valor de ese objeto.

Pero en nuestro caso, dado que el subproceso B no ha utilizado el acceso sincronizado para el count , puede ser un valor referencial del count almacenado en el registro y puede que nunca vea las actualizaciones del subproceso A. Para asegurarse de que B ve el valor más reciente del recuento, necesitamos hacer getCount() sincronizado también.

```
public synchronized Integer getCount() {
```

```
  return count;
}
```

Ahora, cuando el subproceso A finaliza con el count actualizaciones, desbloquea la instancia de Counter , al mismo tiempo crea una barrera de escritura y borra todos los cambios realizados dentro de ese bloque en la memoria principal. De manera similar, cuando el subproceso B adquiere el bloqueo en la misma instancia de Counter , entra en la barrera de lectura y lee el valor del count desde la memoria principal y ve todas las actualizaciones.



El mismo efecto de visibilidad se volatile lecturas / escrituras volatile . Todas las variables actualizadas antes de escribir en volatile se vaciarán en la memoria principal y todas las lecturas después de la lectura de la variable volatile serán de la memoria principal.

Creando una instancia java.lang.Thread

Hay dos enfoques principales para crear un hilo en Java. En esencia, crear un hilo es tan fácil como escribir el código que se ejecutará en él. Los dos enfoques difieren en donde se define ese código.

En Java, un subproceso está representado por un objeto: una instancia de java.lang.Thread o su subclase. Entonces, el primer enfoque es crear esa subclase y anular el método *run ()* .

Nota : usaré *Thread* para referirme a la clase java.lang.Thread y *thread* para referirse al concepto lógico de threads.

```
class MyThread extends Thread {
```

```
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread running!");
        }
    }
}
```

Ahora que ya hemos definido el código que se ejecutará, el hilo se puede crear simplemente como:

```
MyThread t = new MyThread();
```

La clase Thread también contiene un constructor que acepta una cadena, que se utilizará como el nombre del hilo. Esto puede ser particularmente útil cuando se depura un programa multihilo.

```
class MyThread extends Thread {
    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread running! ");
        }
    }
}

MyThread t = new MyThread("Greeting Producer");
```

El segundo enfoque es definir el código utilizando java.lang.Runnable y su único método *run ()* . La clase Thread le permite ejecutar ese método en un hilo separado. Para lograr esto, cree el hilo usando un constructor que acepte una instancia de la interfaz Runnable .

```
Thread t = new Thread(aRunnable);
```

Esto puede ser muy poderoso cuando se combina con lambdas o referencias de métodos (solo Java 8):

```
Thread t = new Thread(operator::hardWork);
```

También puedes especificar el nombre del hilo.

```
Thread t = new Thread(operator::hardWork, "Pi operator");
```

Hablando de manera práctica, puede utilizar ambos enfoques sin preocupaciones. Sin embargo, la sabiduría general dice usar este último.

Para cada uno de los cuatro constructores mencionados, también hay una alternativa que acepta una instancia de java.lang.ThreadGroup como primer parámetro.

```
ThreadGroup tg = new ThreadGroup("Operators");
Thread t = new Thread(tg, operator::hardWork, "PI operator");
```

El ThreadGroup representa un conjunto de hilos. Solo puede agregar un Thread a un ThreadGroup usando un constructor de Thread . El ThreadGroup se puede utilizar para gestionar toda su rosca s juntos, así como el hilo se puede obtener información de su ThreadGroup .

Para resumir, el subproceso se puede crear con uno de estos constructores públicos:

```
Thread()
Thread(String name)
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(ThreadGroup group, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
```

El último nos permite definir el tamaño de pila deseado para el nuevo hilo.

A menudo, la legibilidad del código sufre al crear y configurar muchos subprocesos con las mismas propiedades o desde el mismo patrón. Ahí es cuando se puede usar java.util.concurrent.ThreadFactory . Esta interfaz le permite encapsular el procedimiento de crear el hilo a través del patrón de fábrica y su único método, *newThread (Runnable)* .

```
class WorkerFactory implements ThreadFactory {
    private int id = 0;

    @Override
    public Thread newThread(Runnable r) {
        return new Thread(r, "Worker " + id++);
    }
}
```

Interrupción de hilo / hilos de parada

Cada subproceso de Java tiene un indicador de interrupción, que inicialmente es falso. Interrumpir un hilo no es más que establecer esa bandera en verdadero. El código que se ejecuta en ese hilo puede verificar la bandera en ocasiones y actuar sobre ella. El código también puede ignorarlo por completo. Pero ¿por qué cada Hilo tiene una bandera así? Después de todo, tener una bandera booleana en un hilo es algo que podemos organizarnos siempre que lo necesitemos. Bueno, hay métodos que se comportan de una manera especial cuando se interrumpe el hilo en el que se están ejecutando. Estos métodos se denominan métodos de bloqueo. Estos son métodos que ponen el hilo en el estado WAITING o TIMED_WAITING. Cuando un subproceso se encuentra en este estado, al interrumpirlo, se generará una InterruptedException en el subproceso interrumpido, en lugar de que el indicador de interrupción se establezca en verdadero, y el subproceso se RUNNABLE nuevamente. El código que invoca un método de bloqueo está obligado a tratar con la excepción InterruptedException, ya que es una excepción comprobada. Entonces, y de ahí su nombre, una interrupción puede tener el efecto de interrumpir una ESPERA, terminándola de manera efectiva. Tenga en cuenta que no todos los métodos que están esperando de alguna manera (por ejemplo, bloqueando IO) responden a la interrupción de esa manera, ya que no ponen el hilo en un estado de espera. Por último, un subproceso que tiene establecido su indicador de interrupción, que ingresa a un método de bloqueo (es decir, intenta pasar al estado de espera), lanzará inmediatamente una InterruptedException y el indicador de interrupción se borrará.

Aparte de estas mecánicas, Java no asigna ningún significado semántico especial a la interrupción. El código es libre de interpretar una interrupción de la manera que quiera. Pero la mayoría de las veces, la interrupción se utiliza para señalar a un subproceso que debe dejar de ejecutarse lo antes posible. Pero, como debe quedar claro de lo anterior, depende del código de ese hilo reaccionar adecuadamente a esa interrupción para detener la ejecución. Detener un hilo es una colaboración. Cuando un hilo se interrumpe, su código de ejecución puede tener varios niveles de profundidad en el seguimiento de pila. La mayoría de los códigos no llama a un método de bloqueo, y termina lo suficientemente oportuno como para no retrasar demasiado la detención del hilo. El código que debería preocuparse principalmente de responder a la interrupción, es un código que se encuentra en una tarea de manejo de bucle hasta que no queda ninguno, o hasta que se establece un indicador que lo indica para detener ese bucle. Los bucles que manejan tareas posiblemente infinitas (es decir, se siguen ejecutando en principio) deben verificar el indicador de interrupción para salir del bucle. Para bucles finitos, la semántica puede dictar que todas las tareas deben terminarse antes de finalizar, o puede ser apropiado

dejar algunas tareas sin manejar. El código que llama a los métodos de bloqueo se verá obligado a lidiar con la excepción InterruptedException. Si es semánticamente posible, puede simplemente propagar la InterruptedException y declarar que se debe lanzar. Como tal, se convierte en un método de bloqueo en sí mismo en relación con sus llamadores. Si no puede propagar la excepción, al menos debería establecer el indicador interrumpido, para que las personas que llaman más arriba en la pila también sepan que el hilo fue interrumpido. En algunos casos, el método debe continuar esperando, independientemente de la InterruptedException, en cuyo caso debe demorar la configuración del indicador interrumpido hasta que, una vez que se termina de esperar, esto puede implicar establecer una variable local, que debe verificarse antes de salir del método para entonces interrumpe su hilo.

Ejemplos:

Ejemplo de código que deja de manejar tareas en caso de interrupción.

```
class TaskHandler implements Runnable {

    private final BlockingQueue<Task> queue;

    TaskHandler(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) { // check for interrupt flag, exit
loop when interrupted
            try {
                Task task = queue.take(); // blocking call, responsive to interruption
                handle(task);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // cannot throw InterruptedException (due
to Runnable interface restriction) so indicating interruption by setting the flag
            }
        }
    }

    private void handle(Task task) {
        // actual handling
    }
}
```

Ejemplo de código que retrasa la configuración de la bandera de interrupción hasta que esté completamente terminado:

```
class MustFinishHandler implements Runnable {

    private final BlockingQueue<Task> queue;

    MustFinishHandler(BlockingQueue<Task> queue) {
        this.queue = queue;
    }

    @Override
    public void run() {
        boolean shouldInterrupt = false;

        while (true) {
            try {
                Task task = queue.take();
                if (task.isEndOfTasks()) {
```

```
                    if (shouldInterrupt) {
                        Thread.currentThread().interrupt();
                    }
                    return;
                }
                handle(task);
            } catch (InterruptedException e) {
                shouldInterrupt = true; // must finish, remember to set interrupt flag when
we're done
            }
        }
    }

    private void handle(Task task) {
        // actual handling
    }
}
```

Ejemplo de código que tiene una lista fija de tareas pero que puede cerrarse antes cuando se interrumpe

```
class GetAsFarAsPossible implements Runnable {

    private final List<Task> tasks = new ArrayList<>();

    @Override
    public void run() {
        for (Task task : tasks) {
            if (Thread.currentThread().isInterrupted()) {
                return;
            }
            handle(task);
        }
    }

    private void handle(Task task) {
        // actual handling
    }
}
```

Ejemplo de productor / consumidor múltiple con cola global compartida

A continuación el código muestra múltiples programas de Productor / Consumidor. Tanto las hebras del productor como las del consumidor comparten la misma cola global.

```
import java.util.concurrent.*;
import java.util.Random;

public class ProducerConsumerWithES {
    public static void main(String args[]) {
        BlockingQueue<Integer> sharedQueue = new LinkedBlockingQueue<Integer>();

        ExecutorService pes = Executors.newFixedThreadPool(2);
        ExecutorService ces = Executors.newFixedThreadPool(2);

        pes.submit(new Producer(sharedQueue, 1));
        pes.submit(new Producer(sharedQueue, 2));
        ces.submit(new Consumer(sharedQueue, 1));
        ces.submit(new Consumer(sharedQueue, 2));
```

```
        pes.shutdown();
        ces.shutdown();
    }
}

/* Different producers produces a stream of integers continuously to a shared queue,
which is shared between all Producers and consumers */

class Producer implements Runnable {
    private final BlockingQueue<Integer> sharedQueue;
    private int threadNo;
    private Random random = new Random();
    public Producer(BlockingQueue<Integer> sharedQueue,int threadNo) {
        this.threadNo = threadNo;
        this.sharedQueue = sharedQueue;
    }
    @Override
    public void run() {
        // Producer produces a continuous stream of numbers for every 200 milli seconds
        while (true) {
            try {
                int number = random.nextInt(1000);
                System.out.println("Produced:" + number + ":by thread:"+ threadNo);
                sharedQueue.put(number);
                Thread.sleep(200);
            } catch (Exception err) {
                err.printStackTrace();
            }
        }
    }
}
/* Different consumers consume data from shared queue, which is shared by both producer and
consumer threads */
class Consumer implements Runnable {
    private final BlockingQueue<Integer> sharedQueue;
    private int threadNo;
    public Consumer (BlockingQueue<Integer> sharedQueue,int threadNo) {
        this.sharedQueue = sharedQueue;
        this.threadNo = threadNo;
    }
    @Override
    public void run() {
        // Consumer consumes numbers generated from Producer threads continuously
        while(true){
            try {
                int num = sharedQueue.take();
                System.out.println("Consumed: "+ num + ":by thread:"+threadNo);
            } catch (Exception err) {
               err.printStackTrace();
            }
        }
    }
}
```

salida:

```
Produced:69:by thread:2
Produced:553:by thread:1
Consumed: 69:by thread:1
Consumed: 553:by thread:2
Produced:41:by thread:2
```

```
Produced:796:by thread:1
Consumed: 41:by thread:1
Consumed: 796:by thread:2
Produced:728:by thread:2
Consumed: 728:by thread:1
```

y así .....

Explicación:

1. sharedQueue , que es un LinkedBlockingQueue se comparte entre todos los subprocesos Producer y Consumer.
2. Los hilos de producción producen un número entero por cada 200 milisegundos de forma continua y lo sharedQueue a sharedQueue
3. Hilo de Consumer consume entero de sharedQueue continuamente.
4. Este programa se implementa sin construcciones explícitamente synchronized o de Lock . BlockingQueue es la clave para lograrlo.

   Las implementaciones de BlockingQueue están diseñadas para ser utilizadas principalmente para colas productor-consumidor.

   Las implementaciones de BlockingQueue son seguras para subprocesos. Todos los métodos de colas logran sus efectos de forma atómica utilizando bloqueos internos u otras formas de control de concurrencia.

Acceso exclusivo de lectura / lectura simultánea

A veces se requiere que un proceso escriba y lea simultáneamente los mismos "datos".

La interfaz ReadWriteLock y su implementación ReentrantReadWriteLock permiten un patrón de acceso que se puede describir de la siguiente manera:

1. Puede haber cualquier número de lectores concurrentes de los datos. Si hay al menos un acceso de lector otorgado, entonces no es posible el acceso de un escritor.
2. Puede haber como máximo un único escritor para los datos. Si hay un acceso de escritor otorgado, entonces ningún lector puede acceder a los datos.

Una implementación podría verse como:

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class Sample {

// Our lock. The constructor allows a "fairness" setting, which guarantees the chronology of
lock attributions.
protected static final ReadWriteLock RW_LOCK = new ReentrantReadWriteLock();

// This is a typical data that needs to be protected for concurrent access
protected static int data = 0;

/** This will write to the data, in an exclusive access */
public static void writeToData() {
    RW_LOCK.writeLock().lock();
    try {
        data++;
    } finally {
        RW_LOCK.writeLock().unlock();
    }
}

public static int readData() {
    RW_LOCK.readLock().lock();
```

```
    try {
        return data;
    } finally {
        RW_LOCK.readLock().unlock();
    }
}

}
```

NOTA 1 : Este caso de uso preciso tiene una solución más limpia que usa AtomicInteger , pero lo que se describe aquí es un patrón de acceso, que funciona independientemente del hecho de que los datos aquí son un número entero que es una variante atómica.

NOTA 2 : El bloqueo en la parte de lectura es realmente necesario, aunque podría no parecerlo para el lector ocasional. De hecho, si no se bloquea en el lado del lector, cualquier cosa puede salir mal, entre las cuales:

1. No se garantiza que las escrituras de valores primitivos sean atómicas en todas las JVM, por lo que el lector podría ver, por ejemplo, que solo se escriben 32 bits de 64 bits si los data fueran de 64 bits.
2. La visibilidad de la escritura desde un subproceso que no la realizó está garantizada por la JVM solo si establecemos la *relación Suceder antes de* entre las escrituras y las lecturas. Esta relación se establece cuando tanto los lectores como los escritores usan sus respectivos bloqueos, pero no de otra manera

Java SE 8

En caso de que se requiera un mayor rendimiento, y bajo ciertos tipos de uso, hay un tipo de bloqueo más rápido disponible, llamado StampedLock , que entre otras cosas implementa un modo de bloqueo optimista. Este bloqueo funciona de manera muy diferente de ReadWriteLock , y esta muestra no es transponible.

Objeto ejecutable

La interfaz Runnable define un solo método, run() , destinado a contener el código ejecutado en el hilo.

El objeto Runnable se pasa al constructor Thread . Y se llama el método start() Thread.

Ejemplo

```
public class HelloRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println("Hello from a thread");
    }

    public static void main(String[] args) {
        new Thread(new HelloRunnable()).start();
    }
}
```

Ejemplo en Java8:

```
public static void main(String[] args) {
    Runnable r = () -> System.out.println("Hello world");
    new Thread(r).start();
}
```

Subclase ejecutable vs subproceso

El empleo de un objeto Runnable es más general, porque el objeto Runnable puede subclasificar una clase que no sea Thread .

Thread subclasificación de Thread es más fácil de usar en aplicaciones simples, pero está limitada por el hecho de que su clase de tarea debe ser descendiente de Thread .

Un objeto Runnable es aplicable a las API de administración de subprocesos de alto nivel.

Semáforo

Un semáforo es un sincronizador de alto nivel que mantiene un conjunto de *permisos* que pueden ser adquiridos y liberados por subprocesos. Un semáforo se puede imaginar como un contador de *permisos* que se reducirá cuando un hilo adquiera, y se incrementará cuando un hilo se libere. Si la cantidad de *permisos* es 0 cuando un subproceso intenta adquirir, el subproceso se bloqueará hasta que un permiso esté disponible (o hasta que el subproceso se interrumpa).

Un semáforo se inicializa como:

Semaphore semaphore = new Semaphore(1); // The int value being the number of permits

El constructor Semaphore acepta un parámetro booleano adicional para ser justos. Cuando se establece como falso, esta clase no ofrece garantías sobre el orden en que los subprocesos adquieren permisos. Cuando la imparcialidad se establece como verdadera, el semáforo garantiza que los subprocesos que invocan cualquiera de los métodos de adquisición se seleccionan para obtener permisos en el orden en que se procesó su invocación de dichos métodos. Se declara de la siguiente manera:

Semaphore semaphore = new Semaphore(1, true);

Ahora veamos un ejemplo de javadocs, donde Semaphore se usa para controlar el acceso a un conjunto de elementos. En este ejemplo se usa un semáforo para proporcionar una funcionalidad de bloqueo a fin de garantizar que siempre haya elementos que se obtendrán cuando se getItem() .

```
class Pool {
    /*
     * Note that this DOES NOT bound the amount that may be released!
     * This is only a starting value for the Semaphore and has no other
     * significant meaning UNLESS you enforce this inside of the
     * getNextAvailableItem() and markAsUnused() methods
     */
    private static final int MAX_AVAILABLE = 100;
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);

    /**
     * Obtains the next available item and reduces the permit count by 1.
     * If there are no items available, block.
     */
    public Object getItem() throws InterruptedException {
        available.acquire();
        return getNextAvailableItem();
    }

    /**
     * Puts the item into the pool and add 1 permit.
     */
    public void putItem(Object x) {
        if (markAsUnused(x))
            available.release();
    }

    private Object getNextAvailableItem() {
        // Implementation
```

```
    }

    private boolean markAsUnused(Object o) {
        // Implementation
    }
}
```

Agregue dos arreglos `int` usando un Threadpool

Un Threadpool tiene una cola de tareas, de las cuales cada una se ejecutará en uno de estos Threads.

El siguiente ejemplo muestra cómo agregar dos matrices int utilizando un Threadpool.

Java SE 8

```
int[] firstArray = { 2, 4, 6, 8 };
int[] secondArray = { 1, 3, 5, 7 };
int[] result = { 0, 0, 0, 0 };

ExecutorService pool = Executors.newCachedThreadPool();

// Setup the ThreadPool:
// for each element in the array, submit a worker to the pool that adds elements
for (int i = 0; i < result.length; i++) {
    final int worker = i;
    pool.submit(() -> result[worker] = firstArray[worker] + secondArray[worker] );
}

// Wait for all Workers to finish:
try {
    // execute all submitted tasks
    pool.shutdown();
    // waits until all workers finish, or the timeout ends
    pool.awaitTermination(12, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    pool.shutdownNow(); //kill thread
}

System.out.println(Arrays.toString(result));
```

Notas:

1. Este ejemplo es puramente ilustrativo. En la práctica, no habrá ningún aumento de velocidad mediante el uso de subprocesos para una tarea tan pequeña. Es probable que se produzca una ralentización, ya que los gastos generales de la creación y programación de tareas sobrecargarán el tiempo necesario para ejecutar una tarea.

2. Si estuviera utilizando Java 7 y anteriores, usaría clases anónimas en lugar de lambdas para implementar las tareas.

Obtenga el estado de todas las hebras iniciadas por su programa, excluyendo las hebras del sistema

Fragmento de código:

```
import java.util.Set;

public class ThreadStatus {
```

```
    public static void main(String args[]) throws Exception {
        for (int i = 0; i < 5; i++){
            Thread t = new Thread(new MyThread());
            t.setName("MyThread:" + i);
            t.start();
        }
        int threadCount = 0;
        Set<Thread> threadSet = Thread.getAllStackTraces().keySet();
        for (Thread t : threadSet) {
            if (t.getThreadGroup() == Thread.currentThread().getThreadGroup()) {
                System.out.println("Thread :" + t + ":" + "state:" + t.getState());
                ++threadCount;
            }
        }
        System.out.println("Thread count started by Main thread:" + threadCount);
    }
}

class MyThread implements Runnable {
    public void run() {
        try {
            Thread.sleep(2000);
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}
```

Salida:

```
Thread :Thread[MyThread:1,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:3,5,main]:state:TIMED_WAITING
Thread :Thread[main,5,main]:state:RUNNABLE
Thread :Thread[MyThread:4,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:0,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:2,5,main]:state:TIMED_WAITING
Thread count started by Main thread:6
```

Explicación:

Thread.getAllStackTraces().keySet() devuelve todos los Thread , incluidos los subprocesos de aplicaciones y los subprocesos del sistema. Si solo está interesado en el estado de los Thread , iniciado por su aplicación, itere el conjunto de subprocesos marcando el Grupo de subprocesos de un subproceso en particular con el subproceso principal del programa.

En ausencia de la condición de ThreadGroup anterior, el programa devuelve el estado de los subprocesos del sistema a continuación:

```
Reference Handler
Signal Dispatcher
Attach Listener
Finalizer
```

Callable y Futuro

Si bien Runnable proporciona un medio para envolver el código para que se ejecute en un subproceso diferente, tiene una limitación en cuanto a que no puede devolver un resultado de la ejecución. La única forma de obtener algún valor de retorno de la ejecución de un Runnable es asignar el resultado a una variable accesible en un ámbito fuera del Runnable .

Callable se introdujo en Java 5 como un par de Runnable . Callable es esencialmente el mismo, excepto que tiene un método de call lugar de run . El método de call tiene la capacidad adicional de devolver un resultado y también se le permite lanzar excepciones marcadas.

El resultado de un *envío de tarea callable* está disponible para ser pulsado a través de un futuro

Future puede considerarse un contenedor de clases que alberga el resultado de la computación Callable . El cálculo de la llamada puede continuar en otro hilo, y cualquier intento de tocar el resultado de un Future se bloqueará y solo devolverá el resultado una vez que esté disponible.

Interfaz invocable

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Futuro

```
interface Future<V> {
    V get();
    V get(long timeout, TimeUnit unit);
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
}
```

Usando ejemplo llamable y futuro:

```
public static void main(String[] args) throws Exception {
    ExecutorService es = Executors.newSingleThreadExecutor();

    System.out.println("Time At Task Submission : " + new Date());
    Future<String> result = es.submit(new ComplexCalculator());
    // the call to Future.get() blocks until the result is available.So we are in for about a
10 sec wait now
    System.out.println("Result of Complex Calculation is : " + result.get());
    System.out.println("Time At the Point of Printing the Result : " + new Date());
}
```

Nuestro Callable que hace un largo cálculo.

```
public class ComplexCalculator implements Callable<String> {

    @Override
    public String call() throws Exception {
        // just sleep for 10 secs to simulate a lengthy computation
        Thread.sleep(10000);
        System.out.println("Result after a lengthy 10sec calculation");
        return "Complex Result"; // the result
    }
}
```

Salida

```
Time At Task Submission : Thu Aug 04 15:05:15 EDT 2016
Result after a lengthy 10sec calculation
Result of Complex Calculation is : Complex Result
```

```
Time At the Point of Printing the Result : Thu Aug 04 15:05:25 EDT 2016
```

Otras operaciones permitidas en el futuro

Si bien get() es el método para extraer el resultado real, Future tiene provisión

- get(long timeout, TimeUnit unit) define el período de tiempo máximo durante la get(long timeout, TimeUnit unit) actual esperará un resultado;
- Para cancelar la tarea, cancele la llamada cancel(mayInterruptIfRunning) . La bandera mayInterrupt indica que la tarea debe interrumpirse si se inició y se está ejecutando en este momento;
- Para verificar si la tarea se completó / terminó llamando a isDone() ;
- Para verificar si la tarea prolongada se canceló, se cancela isCancelled() .

Bloqueos como ayudas de sincronización

Antes de la introducción del paquete concurrente de Java 5, el subproceso era más bajo. La introducción de este paquete proporcionaba varias ayudas / construcciones de programación concurrente de nivel superior.

Los bloqueos son mecanismos de sincronización de subprocesos que sirven esencialmente para el mismo propósito que los bloques sincronizados o palabras clave.

Bloqueo intrínseco

```
int count = 0; // shared among multiple threads

public void doSomething() {
    synchronized(this) {
        ++count; // a non-atomic operation
    }
}
```

Sincronización mediante Locks

```
int count = 0; // shared among multiple threads

Lock lockObj = new ReentrantLock();
public void doSomething() {
    try {
        lockObj.lock();
        ++count; // a non-atomic operation
    } finally {
        lockObj.unlock(); // sure to release the lock without fail
    }
}
```

Los bloqueos también tienen una funcionalidad disponible que el bloqueo intrínseco no ofrece, como el bloqueo pero que responde a la interrupción, o intenta bloquear, y no bloquear cuando no puede hacerlo.

Bloqueo, sensible a la interrupción.

```
class Locky {
    int count = 0; // shared among multiple threads

    Lock lockObj = new ReentrantLock();

    public void doSomething() {
        try {
```

```
            try {
                lockObj.lockInterruptibly();
                ++count; // a non-atomic operation
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // stopping
            }
        } finally {
            if (!Thread.currentThread().isInterrupted()) {
                lockObj.unlock(); // sure to release the lock without fail
            }
        }
    }
}
```

Solo hacer algo cuando se pueda bloquear.

```
public class Locky2 {
    int count = 0; // shared among multiple threads

    Lock lockObj = new ReentrantLock();

    public void doSomething() {
        boolean locked = lockObj.tryLock(); // returns true upon successful lock
        if (locked) {
            try {
                ++count; // a non-atomic operation
            } finally {
                lockObj.unlock(); // sure to release the lock without fail
            }
        }
    }
}
```

Hay varias variantes de bloqueo disponibles. Para más detalles, consulte los documentos api aquí

Lea Programación Concurrente (Hilos) en línea:
https://riptutorial.com/es/java/topic/121/programacion-concurrente--hilos-

Capítulo 151: Programación paralela con el framework Fork / Join.

Examples

Horquilla / Unir tareas en Java

El framework fork / join en Java es ideal para un problema que se puede dividir en partes más pequeñas y resolver en paralelo. Los pasos fundamentales de un problema fork / join son:

- Divide el problema en varias piezas.
- Resuelve cada una de las piezas en paralelo entre sí.
- Combine cada una de las sub-soluciones en una solución global

Una ForkJoinTask es la interfaz que define este problema. En general, se espera que usted subclasifique una de sus implementaciones abstractas (generalmente la Tarea Recursiva) en lugar de implementar la interfaz directamente.

En este ejemplo, vamos a sumar una colección de enteros, dividiendo hasta obtener un tamaño de lote de no más de diez.

```
import java.util.List;
import java.util.concurrent.RecursiveTask;

public class SummingTask extends RecursiveTask<Integer> {
    private static final int MAX_BATCH_SIZE = 10;

    private final List<Integer> numbers;
    private final int minInclusive, maxExclusive;

    public SummingTask(List<Integer> numbers) {
        this(numbers, 0, numbers.size());
    }

    // This constructor is only used internally as part of the dividing process
    private SummingTask(List<Integer> numbers, int minInclusive, int maxExclusive) {
        this.numbers = numbers;
        this.minInclusive = minInclusive;
        this.maxExclusive = maxExclusive;
    }

    @Override
    public Integer compute() {
        if (maxExclusive - minInclusive > MAX_BATCH_SIZE) {
            // This is too big for a single batch, so we shall divide into two tasks
            int mid = (minInclusive + maxExclusive) / 2;
            SummingTask leftTask = new SummingTask(numbers, minInclusive, mid);
            SummingTask rightTask = new SummingTask(numbers, mid, maxExclusive);

            // Submit the left hand task as a new task to the same ForkJoinPool
            leftTask.fork();

            // Run the right hand task on the same thread and get the result
            int rightResult = rightTask.compute();

            // Wait for the left hand task to complete and get its result
            int leftResult = leftTask.join();

            // And combine the result
            return leftResult + rightResult;
        } else {
```

```
            // This is fine for a single batch, so we will run it here and now
            int sum = 0;
            for (int i = minInclusive; i < maxExclusive; i++) {
                sum += numbers.get(i);
            }
            return sum;
        }
    }
}
```

Una instancia de esta tarea ahora se puede pasar a una instancia de ForkJoinPool .

```
// Because I am not specifying the number of threads
// it will create a thread for each available processor
ForkJoinPool pool = new ForkJoinPool();

// Submit the task to the pool, and get what is effectively the Future
ForkJoinTask<Integer> task = pool.submit(new SummingTask(numbers));

// Wait for the result
int result = task.join();
```

Lea Programación paralela con el framework Fork / Join. en línea:
https://riptutorial.com/es/java/topic/4245/programacion-paralela-con-el-framework-fork---join-

Capítulo 152: Publicación por entregas

Introducción

Java proporciona un mecanismo, denominado serialización de objetos, en el que un objeto se puede representar como una secuencia de bytes que incluye los datos del objeto, así como información sobre el tipo del objeto y los tipos de datos almacenados en el objeto.

Después de que un objeto serializado se haya escrito en un archivo, se puede leer del archivo y deserializar, es decir, la información de tipo y los bytes que representan el objeto y sus datos se pueden usar para recrear el objeto en la memoria.

Examples

Serialización básica en Java

¿Qué es la serialización?

La serialización es el proceso de convertir el estado de un objeto (incluidas sus referencias) en una secuencia de bytes, así como el proceso de reconstruir esos bytes en un objeto vivo en algún momento futuro. La serialización se utiliza cuando desea persistir el objeto. Java RMI también lo utiliza para pasar objetos entre JVM, ya sea como argumentos en una invocación de método de un cliente a un servidor o como valores de retorno de una invocación de método, o como excepciones lanzadas por métodos remotos. En general, la serialización se usa cuando queremos que el objeto exista más allá de la vida útil de la JVM.

java.io.Serializable es una interfaz de marcador (no tiene cuerpo). Solo se utiliza para "marcar" clases de Java como serializables.

El tiempo de ejecución de serialización asocia con cada clase serializable un número de versión, llamado serialVersionUID , que se usa durante la *des-* serialización para verificar que el remitente y el receptor de un objeto serializado hayan cargado clases para ese objeto que sean compatibles con respecto a la serialización. Si el receptor ha cargado una clase para el objeto que tiene un serialVersionUID diferente al de la clase del remitente correspondiente, la deserialización dará como resultado una InvalidClassException . Una clase serializable puede declarar su propio serialVersionUID declarando explícitamente un campo llamado serialVersionUID que debe ser static, final, y de tipo long :

ANY-ACCESS-MODIFIER static final long serialVersionUID = 1L;

Cómo hacer que una clase sea elegible para la serialización

Para conservar un objeto, la clase respectiva debe implementar la interfaz java.io.Serializable .

```
import java.io.Serializable;

public class SerialClass implements Serializable {

    private static final long serialVersionUID = 1L;
    private Date currentTime;

    public SerialClass() {
        currentTime = Calendar.getInstance().getTime();
    }

    public Date getCurrentTime() {
        return currentTime;
    }
}
```

Cómo escribir un objeto en un archivo

Ahora necesitamos escribir este objeto en un sistema de archivos. Usamos java.io.ObjectOutputStream para este propósito.

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;

public class PersistSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = new SerialClass(); //We will write this object to file system.
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(filename));
            out.writeObject(time); //Write byte stream to file system.
            out.close();
        } catch(IOException ex){
            ex.printStackTrace();
        }
     }
 }
```

Cómo recrear un objeto desde su estado serializado

El objeto almacenado puede leerse desde el sistema de archivos más tarde usando java.io.ObjectInputStream como se muestra a continuación:

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.io.java.lang.ClassNotFoundException;

 public class ReadSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = null;

        try {
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename));
            time = (SerialClass)in.readObject();
            in.close();
        } catch(IOException ex){
            ex.printStackTrace();
        } catch(ClassNotFoundException cnfe){
            cnfe.printStackTrace();
        }
        // print out restored time
        System.out.println("Restored time: " + time.getTime());
     }
 }
```

La clase serializada está en forma binaria. La deserialización puede ser problemática si la definición de la clase cambia: consulte el capítulo Control de versiones de objetos serializados de la Especificación de serialización de Java para obtener más detalles.

Serializar un objeto serializa el gráfico de objeto completo del cual es la raíz, y funciona correctamente en presencia de gráficos cíclicos. Se proporciona un método reset() para forzar que ObjectOutputStream olvide de los objetos que ya se han serializado.

Campos transitorios - Serialización

Serialización con Gson

La serialización con Gson es fácil y producirá JSON correcto.

```
public class Employe {

    private String firstName;
    private String lastName;
    private int age;
    private BigDecimal salary;
    private List<String> skills;

    //getters and setters
}
```

(Publicación por entregas)

```
//Skills
List<String> skills = new LinkedList<String>();
skills.add("leadership");
skills.add("Java Experience");

//Employe
Employe obj = new Employe();
obj.setFirstName("Christian");
obj.setLastName("Lusardi");
obj.setAge(25);
obj.setSalary(new BigDecimal("10000"));
obj.setSkills(skills);

//Serialization process
Gson gson = new Gson();
String json = gson.toJson(obj);
//{"firstName":"Christian","lastName":"Lusardi","age":25,"salary":10000,"skills":["leadership","Java
Experience"]}
```

Tenga en cuenta que no puede serializar objetos con referencias circulares, ya que esto resultará en una recursión infinita.

(Deserialización)

```
//it's very simple...
//Assuming that json is the previous String object....

Employe obj2 = gson.fromJson(json, Employe.class); // obj2 is just like obj
```

Serialización con Jackson 2

La siguiente es una implementación que demuestra cómo un objeto puede ser serializado en su cadena JSON correspondiente.

```
class Test {

    private int idx;
    private String name;

    public int getIdx() {
```

```
        return idx;
    }

    public void setIdx(int idx) {
        this.idx = idx;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Publicación por entregas:

```
Test test = new Test();
test.setIdx(1);
test.setName("abc");

ObjectMapper mapper = new ObjectMapper();

String jsonString;
try {
    jsonString = mapper.writerWithDefaultPrettyPrinter().writeValueAsString(test);
    System.out.println(jsonString);
} catch (JsonProcessingException ex) {
    // Handle Exception
}
```

Salida:

```
{
  "idx" : 1,
  "name" : "abc"
}
```

Si no lo necesita, puede omitir la impresora bonita predeterminada.

La dependencia utilizada aquí es la siguiente:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.6.3</version>
</dependency>
```

Serialización personalizada

En este ejemplo, queremos crear una clase que genere y genere en la consola, un número aleatorio entre un rango de dos enteros que se pasan como argumentos durante la inicialización.

```
    public class SimpleRangeRandom implements Runnable {
    private int min;
    private int max;

    private Thread thread;
```

```
    public SimpleRangeRandom(int min, int max){
        this.min = min;
        this.max = max;
        thread = new Thread(this);
        thread.start();
    }

    @Override
 private void WriteObject(ObjectOutputStreamout) throws IO Exception;
    private void ReadObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
    public void run() {
        while(true) {
            Random rand = new Random();
            System.out.println("Thread: " + thread.getId() + " Random:" + rand.nextInt(max -
min));
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    }
```

Ahora si queremos que esta clase sea serializable habrá algunos problemas. El subproceso es una de las ciertas clases de nivel de sistema que no son serializables. Así que tenemos que declarar el hilo como **transitorio** . Al hacer esto podremos serializar los objetos de esta clase pero aún tendremos un problema. Como puede ver en el constructor, establecemos los valores mínimo y máximo de nuestro aleatorizador y, a continuación, iniciamos el hilo que es responsable de generar e imprimir el valor aleatorio. Por lo tanto, al restaurar el objeto persistente llamando a ***readObject ()***, el constructor no se ejecutará nuevamente ya que no hay creación de un nuevo objeto. En ese caso, necesitamos desarrollar una **serialización personalizada** proporcionando dos métodos dentro de la clase. Esos métodos son:

```
private void writeObject(ObjectOutputStream out) throws IOException;
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

Por lo tanto, al agregar nuestra implementación en el ***readObject ()*** podemos iniciar y comenzar nuestro hilo:

```
class RangeRandom implements Serializable, Runnable {

private int min;
private int max;

private transient Thread thread;
//transient should be any field that either cannot be serialized e.g Thread or any field you
do not want serialized

public RangeRandom(int min, int max){
    this.min = min;
    this.max = max;
    thread = new Thread(this);
    thread.start();
}

@Override
public void run() {
    while(true) {
```

```
        Random rand = new Random();
        System.out.println("Thread: " + thread.getId() + " Random:" + rand.nextInt(max -
min));
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

private void writeObject(ObjectOutputStream oos) throws IOException {
    oos.defaultWriteObject();
}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    thread = new Thread(this);
    thread.start();
}
}
```

Aquí está el principal para nuestro ejemplo:

```
public class Main {
public static void main(String[] args) {
    System.out.println("Hello");
    RangeRandom rangeRandom = new RangeRandom(1,10);

    FileOutputStream fos = null;
    ObjectOutputStream out = null;
    try
    {
        fos = new FileOutputStream("test");
        out = new ObjectOutputStream(fos);
        out.writeObject(rangeRandom);
        out.close();
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }


    RangeRandom rangeRandom2 = null;
       FileInputStream fis = null;
       ObjectInputStream in = null;
       try
       {
             fis = new FileInputStream("test");
             in = new ObjectInputStream(fis);
           rangeRandom2 = (RangeRandom)in.readObject();
             in.close();
           }
       catch(IOException ex)
       {
             ex.printStackTrace();
           }
       catch(ClassNotFoundException ex)
       {
             ex.printStackTrace();
```

```
            }

}
}
```

Si ejecuta main, verá que hay dos subprocesos en ejecución para cada instancia de RangeRandom y eso se debe a que el método ***Thread.start ()*** ahora está en el constructor y en el ***readObject ()*** .

Versiones y serialVersionUID

Cuando implementa la interfaz java.io.Serializable para hacer que una clase sea serializable, el compilador busca un campo static final llamado serialVersionUID de tipo long . Si la clase no tiene este campo declarado explícitamente, el compilador creará uno de estos campos y le asignará un valor que sale de un cálculo dependiente de la implementación de serialVersionUID . Este cálculo depende de varios aspectos de la clase y sigue las Especificaciones de serialización de objetos proporcionadas por Sun. Pero, no se garantiza que el valor sea el mismo en todas las implementaciones del compilador.

Este valor se usa para verificar la compatibilidad de las clases con respecto a la serialización y esto se hace al deserializar un objeto guardado. El Tiempo de ejecución de serialización verifica que serialVersionUID lea de los datos sin serializar y que serialVersionUID declarado en la clase sea exactamente el mismo. Si ese no es el caso, lanza una InvalidClassException .

Es altamente recomendable que declare e inicialice explícitamente el campo estático, final de tipo long y que se llame 'serialVersionUID' en todas sus clases que desee que sean serializables en lugar de confiar en el cálculo predeterminado del valor para este campo, incluso si no va a hacerlo. usar versionamiento **El cálculo de 'serialVersionUID' es extremadamente sensible y puede variar de una implementación de compilador a otra y, por lo tanto, puede obtener la InvalidClassException incluso para la misma clase solo porque usó diferentes implementaciones de compilador en el remitente y el receptor finaliza el proceso de serialización.**

```
public class Example implements Serializable {
    static final long serialVersionUID = 1L /*or some other value*/;
    //...
}
```

Mientras que serialVersionUID sea el mismo, la serialización de Java puede manejar diferentes versiones de una clase. Cambios compatibles e incompatibles son;

Cambios compatibles

- **Adición de campos:** cuando la clase que se reconstituye tiene un campo que no aparece en el flujo, ese campo en el objeto se inicializará con el valor predeterminado para su tipo. Si se necesita una inicialización específica de la clase, la clase puede proporcionar un método readObject que puede inicializar el campo a valores no predeterminados.
- **Agregar clases:** la secuencia contendrá la jerarquía de tipos de cada objeto en la secuencia. La comparación de esta jerarquía en la secuencia con la clase actual puede detectar clases adicionales. Como no hay información en la ruta desde la cual inicializar el objeto, los campos de la clase se inicializarán a los valores predeterminados.
- **Eliminación de clases: la** comparación de la jerarquía de clases en la ruta con la de la clase actual puede detectar que una clase se ha eliminado. En este caso, los campos y objetos correspondientes a esa clase se leen de la secuencia. Los campos primitivos se descartan, pero los objetos a los que hace referencia la clase eliminada se crean, ya que se pueden consultar más adelante en la secuencia. Se recolectarán cuando el flujo se recolecta o se restablece.
- **Agregar métodos writeObject / readObject:** si la versión que lee la secuencia tiene estos métodos, se espera que readObject, como de costumbre, lea los datos necesarios escritos en la secuencia por la serialización predeterminada. Debe llamar a defaultReadObject primero antes de leer cualquier dato opcional. Se espera que el método writeObject llame a defaultWriteObject para escribir los datos requeridos y luego puede escribir datos opcionales.

- **Agregar java.io.Serializable:** Esto es equivalente a agregar tipos. No habrá valores en el flujo para esta clase, por lo que sus campos se inicializarán a los valores predeterminados. El soporte para la subclasificación de clases no serializables requiere que el supertipo de la clase tenga un constructor sin argumentos y la clase en sí misma se inicializará con los valores predeterminados. Si el constructor no-arg no está disponible, se lanza la excepción InvalidClassException.
- **Cambio del acceso a un campo:** los modificadores de acceso público, paquete, protegido y privado no afectan la capacidad de serialización para asignar valores a los campos.
- **Cambiar un campo de estático a no estático o transitorio a no transitorio:** cuando se confía en la serialización predeterminada para calcular los campos serializables, este cambio es equivalente a agregar un campo a la clase. El nuevo campo se escribirá en la secuencia, pero las clases anteriores ignorarán el valor, ya que la serialización no asignará valores a campos estáticos o transitorios.

Cambios incompatibles

- **Eliminar campos:** si se elimina un campo en una clase, la secuencia escrita no contendrá su valor. Cuando una clase anterior lee el flujo, el valor del campo se establecerá en el valor predeterminado porque no hay ningún valor disponible en el flujo. Sin embargo, este valor predeterminado puede perjudicar la capacidad de la versión anterior para cumplir su contrato.
- **Mover clases hacia arriba o hacia abajo en la jerarquía:** esto no se puede permitir ya que los datos en la secuencia aparecen en la secuencia incorrecta.
- **Cambiar un campo no estático a estático o un campo no transitorio a transitorio:** cuando se confía en la serialización predeterminada, este cambio es equivalente a eliminar un campo de la clase. Esta versión de la clase no escribirá esos datos en el flujo, por lo que no estará disponible para ser leída por versiones anteriores de la clase. Al igual que al eliminar un campo, el campo de la versión anterior se inicializará con el valor predeterminado, lo que puede hacer que la clase falle de manera inesperada.
- **Cambio del tipo declarado de un campo primitivo:** Cada versión de la clase escribe los datos con su tipo declarado. Las versiones anteriores de la clase que intenten leer el campo fallarán porque el tipo de datos en la secuencia no coincide con el tipo del campo.
- Cambiar el método writeObject o readObject para que ya no escriba o lea los datos de campo predeterminados o cambiarlo para que intente escribirlo o leerlo cuando la versión anterior no lo hizo. Los datos de campo predeterminados deben aparecer constantemente o no aparecer en la transmisión.
- Cambiar una clase de Serializable a Externalizable o viceversa es un cambio incompatible, ya que la transmisión contendrá datos que son incompatibles con la implementación de la clase disponible.
- Cambiar una clase de un tipo sin enumeración a un tipo de enumeración o viceversa, ya que la secuencia contendrá datos que son incompatibles con la implementación de la clase disponible.
- La eliminación de Serializable o Externalizable es un cambio incompatible, ya que cuando se escribe ya no proporcionará los campos necesarios para las versiones anteriores de la clase.
- Agregar el método writeReplace o readResolve a una clase es incompatible si el comportamiento produce un objeto que es incompatible con cualquier versión anterior de la clase.

Deserialización JSON personalizada con Jackson

Consumimos la API de descanso como un formato JSON y luego la descomprimimos en un POJO. El org.codehaus.jackson.map.ObjectMapper de Jackson "simplemente funciona" fuera de la caja y realmente no hacemos nada en la mayoría de los casos. Pero a veces necesitamos deserializador personalizado para satisfacer nuestras necesidades personalizadas y este tutorial lo guiará a través del proceso de creación de su propio deserializador personalizado.

Digamos que tenemos las siguientes entidades.

```
public class User {
    private Long id;
```

```
    private String name;
    private String email;

    //getter setter are omitted for clarity
}
```

Y

```
public class Program {
    private Long id;
    private String name;
    private User createdBy;
    private String contents;

    //getter setter are omitted for clarity
}
```

Primero serialicemos / ordenemos un objeto.

```
User user = new User();
user.setId(1L);
user.setEmail("example@example.com");
user.setName("Bazlur Rahman");

Program program = new Program();
program.setId(1L);
program.setName("Program @# 1");
program.setCreatedBy(user);
program.setContents("Some contents");

ObjectMapper objectMapper = new ObjectMapper();
```

String final json = objectMapper.writeValueAsString (programa); System.out.println (json);

El código anterior producirá siguiente JSON-

```
{
    "id": 1,
    "name": "Program @# 1",
    "createdBy": {
        "id": 1,
        "name": "Bazlur Rahman",
        "email": "example@example.com"
    },
    "contents": "Some contents"
}
```

Ahora puede hacer lo contrario muy fácilmente. Si tenemos este JSON, podemos unmarshal a un objeto de programa usando ObjectMapper de la siguiente manera:

Ahora, digamos, este no es el caso real, vamos a tener un JSON diferente de una API que no coincide con nuestra clase de Program .

```
{
"id": 1,
"name": "Program @# 1",
"ownerId": 1
"contents": "Some contents"
}
```

Mire la cadena JSON, como puede ver, tiene un campo diferente que es owenerId.

Ahora, si desea serializar este JSON como hicimos anteriormente, tendrá excepciones.

Hay dos formas de evitar excepciones y tener esto en serie:

Ignora los campos desconocidos.

Ignora el onwerId . Agregue la siguiente anotación en la clase Programa

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Program {}
```

Escribe deserializador personalizado

Pero hay casos en los que realmente necesita este campo owerId . Digamos que desea relacionarlo como un ID de la clase User .

En tal caso, necesita escribir un deserializador personalizado

Como puede ver, primero debe acceder a JsonNode desde JonsParser . Y luego puede extraer fácilmente información de un JsonNode usando el método get() . y tienes que asegurarte del nombre del campo. Debe ser el nombre exacto, error de ortografía causará excepciones.

Y, finalmente, debe registrar su ProgramDeserializer en el ObjectMapper .

```
ObjectMapper mapper = new ObjectMapper();
SimpleModule module = new SimpleModule();
module.addDeserializer(Program.class, new ProgramDeserializer());

mapper.registerModule(module);

String newJsonString = "{\"id\":1,\"name\":\"Program @# 1\",\"ownerId\":1,\"contents\":\"Some
contents\"}";
final Program program2 = mapper.readValue(newJsonString, Program.class);
```

Alternativamente, puede usar la anotación para registrar el deserializador directamente -

```
@JsonDeserialize(using = ProgramDeserializer.class)
public class Program {
}
```

Lea Publicación por entregas en línea: https://riptutorial.com/es/java/topic/767/publicacion-por-entregas

Capítulo 153: Puntos de referencia

Introducción

Escribir puntos de referencia de rendimiento en java no es tan simple como obtener System.currentTimeMillis() al principio y al final y calcular la diferencia. Para escribir puntos de referencia de rendimiento válidos, se deben usar las herramientas adecuadas.

Examples

Ejemplo simple de JMH

Una de las herramientas para escribir pruebas de referencia adecuadas es JMH . Digamos que queremos comparar el rendimiento de la búsqueda de un elemento en HashSet vs TreeSet .

La manera más fácil de conseguir JHM en su proyecto - es el uso de Maven y sombra plugin. También puedes ver pom.xml de los ejemplos de JHM .

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>3.0.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <finalName>/benchmarks</finalName>
                        <transformers>
                            <transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                                <mainClass>org.openjdk.jmh.Main</mainClass>
                            </transformer>
                        </transformers>
                        <filters>
                            <filter>
                                <artifact>*:*</artifact>
                                <excludes>
                                    <exclude>META-INF/*.SF</exclude>
                                    <exclude>META-INF/*.DSA</exclude>
                                    <exclude>META-INF/*.RSA</exclude>
                                </excludes>
                            </filter>
                        </filters>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

<dependencies>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-core</artifactId>
```

```
        <version>1.18</version>
    </dependency>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-generator-annprocess</artifactId>
        <version>1.18</version>
    </dependency>
</dependencies>
```

Después de esto necesitas escribir la clase de referencia en sí:

```
package benchmark;

import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;

import java.util.HashSet;
import java.util.Random;
import java.util.Set;
import java.util.TreeSet;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
public class CollectionFinderBenchmarkTest {
    private static final int SET_SIZE = 10000;

    private Set<String> hashSet;
    private Set<String> treeSet;

    private String stringToFind = "8888";

    @Setup
    public void setupCollections() {
        hashSet = new HashSet<>(SET_SIZE);
        treeSet = new TreeSet<>();

        for (int i = 0; i < SET_SIZE; i++) {
            final String value = String.valueOf(i);
            hashSet.add(value);
            treeSet.add(value);
        }

        stringToFind = String.valueOf(new Random().nextInt(SET_SIZE));
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public void testHashSet(Blackhole blackhole) {
        blackhole.consume(hashSet.contains(stringToFind));
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public void testTreeSet(Blackhole blackhole) {
        blackhole.consume(treeSet.contains(stringToFind));
    }
}
```

Por favor, tenga en cuenta este blackhole.consume() , volveremos más adelante. También

necesitamos la clase principal para ejecutar benchmark:

```
package benchmark;

import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

public class BenchmarkMain {
    public static void main(String[] args) throws RunnerException {
        final Options options = new OptionsBuilder()
                .include(CollectionFinderBenchmarkTest.class.getSimpleName())
                .forks(1)
                .build();

        new Runner(options).run();
    }
}
```

Y estamos todos listos. Solo necesitamos ejecutar el mvn package (creará benchmarks.jar en su carpeta /target) y ejecutamos nuestra prueba de referencia:

```
java -cp target/benchmarks.jar benchmark.BenchmarkMain
```

Y después de algunas iteraciones de calentamiento y cálculo, tendremos nuestros resultados:

```
# Run complete. Total time: 00:01:21

Benchmark                                  Mode  Cnt   Score    Error  Units
CollectionFinderBenchmarkTest.testHashSet  avgt   20   9.940 ±  0.270  ns/op
CollectionFinderBenchmarkTest.testTreeSet  avgt   20  98.858 ± 13.743  ns/op
```

Sobre eso blackhole.consume() . Si sus cálculos no cambian el estado de su aplicación, Java probablemente lo ignorará. Por lo tanto, para evitarlo, puede hacer que sus métodos de referencia devuelvan algo de valor o utilizar el objeto Blackhole para consumirlo.

Puede encontrar más información sobre cómo escribir puntos de referencia adecuados en el blog de Aleksey Shipilëv , en el blog de Jacob Jenkov y en el blog de java-performance: 1 , 2 .

Lea Puntos de referencia en línea: https://riptutorial.com/es/java/topic/9514/puntos-de-referencia

Capítulo 154: Recursion

Introducción

La recursión ocurre cuando un método se llama a sí mismo. Tal método se llama **recursivo** . Un método recursivo puede ser más conciso que un enfoque equivalente no recursivo. Sin embargo, para una recursión profunda, a veces una solución iterativa puede consumir menos espacio de pila finito de un hilo.

Este tema incluye ejemplos de recursión en Java.

Observaciones

Diseñando un Método Recursivo

Al diseñar un método recursivo, tenga en cuenta que necesita:

- **Caso base.** Esto definirá cuándo se detendrá su recursión y dará salida al resultado. El caso base en el ejemplo factorial es:

```
if (n <= 1) {
    return 1;
}
```

- **Llamada recursiva.** En esta declaración, vuelve a llamar al método con un parámetro cambiado. La llamada recursiva en el ejemplo factorial anterior es:

```
else {
    return n * factorial(n - 1);
}
```

Salida

En este ejemplo, calculamos el n-ésimo número factorial. Los primeros factoriales son:

```
0! = 1

1! = 1

2! = 1 x 2 = 2

3! = 1 x 2 x 3 = 6

4! = 1 x 2 x 3 x 4 = 24

...
```

Eliminación de Java y Tail-Call

Los compiladores de Java actuales (hasta Java 9 inclusive) no realizan la eliminación de la llamada de cola. Esto puede afectar el rendimiento de los algoritmos recursivos, y si la recursión es lo suficientemente profunda, puede provocar que StackOverflowError bloquee; Ver recursión profunda es problemática en Java

Examples

La idea básica de la recursión.

¿Qué es la recursión?

En general, la recursión es cuando una función se invoca a sí misma, ya sea directa o indirectamente. Por ejemplo:

```
// This method calls itself "infinitely"
public void useless() {
    useless();  // method calls itself (directly)
}
```

Condiciones para aplicar la recursividad a un problema:

Hay dos condiciones previas para usar funciones recursivas para resolver un problema específico:

1. Debe haber una condición base para el problema, que será el punto final de la recursión. Cuando una función recursiva alcanza la condición base, no hace más llamadas recursivas (más profundas).

2. Cada nivel de recursión debería estar intentando un problema más pequeño. La función recursiva divide el problema en partes cada vez más pequeñas. Suponiendo que el problema es finito, esto asegurará que la recursión termine.

En Java hay una tercera condición previa: no debería ser necesario repetir demasiado para resolver el problema; Ver recursión profunda es problemática en Java

Ejemplo

La siguiente función calcula los factoriales utilizando la recursividad. Observe cómo el método factorial llama a sí mismo dentro de la función. Cada vez que se llama a sí mismo, reduce el parámetro n en 1. Cuando n alcanza 1 (la condición base), la función no retrocederá más.

```
public int factorial(int n) {
    if (n <= 1) { // the base condition
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Esta no es una forma práctica de calcular factoriales en Java, ya que no tiene en cuenta el desbordamiento de enteros o el desbordamiento de la pila de llamadas (es decir, las excepciones de `StackOverflowError`) para valores grandes de `n` .

Cálculo del Número N. de Fibonacci

El siguiente método calcula el número Nth Fibonacci usando la recursión.

```
public int fib(final int n) {
    if (n > 2) {
        return fib(n - 2) + fib(n - 1);
    }
    return 1;
}
```

El método implementa un caso base (n <= 2) y un caso recursivo (n> 2). Esto ilustra el uso de la recursión para calcular una relación recursiva.

Sin embargo, aunque este ejemplo es ilustrativo, también es ineficiente: cada instancia individual del método llamará a la función en sí misma dos veces, lo que llevará a un crecimiento exponencial en el número de veces que se llama a la función a medida que N aumenta. La función anterior es O (2^N), pero una solución iterativa equivalente tiene complejidad O (N). Además, hay una expresión de "forma cerrada" que se puede evaluar en multiplicaciones de punto flotante O (N).

Cálculo de la suma de enteros de 1 a N

El siguiente método calcula la suma de enteros de 0 a N usando la recursión.

```
public int sum(final int n) {
    if (n > 0) {
        return n + sum(n - 1);
    } else {
        return n;
    }
}
```

Este método es O (N) y se puede reducir a un simple bucle usando la optimización de la llamada de cola. De hecho, hay una expresión de *forma cerrada* que calcula la suma en operaciones O(1) .

Cálculo de la enésima potencia de un número

El siguiente método calcula el valor de num elevado a la potencia de exp mediante recursión:

```
public long power(final int num, final int exp) {
    if (exp == 0) {
        return 1;
    }
    if (exp == 1) {
        return num;
    }
    return num * power(num, exp - 1);
}
```

Esto ilustra los principios mencionados anteriormente: el método recursivo implementa un caso base (dos casos, n = 0 y n = 1) que termina la recursión, y un caso recursivo que llama al método nuevamente. Este método es O (N) y se puede reducir a un simple bucle usando la optimización de la llamada de cola.

Invertir una cadena usando Recursión

A continuación se muestra un código recursivo para revertir una cadena

```
/**
 * Just a snippet to explain the idea of recursion
 *
 **/

public class Reverse {
    public static void main (String args[]) {
        String string = "hello world";
        System.out.println(reverse(string)); //prints dlrow olleh
    }

    public static String reverse(String s) {
        if (s.length() == 1) {
            return s;
        }

        return reverse(s.substring(1)) + s.charAt(0);
    }
}
```

Atravesando una estructura de datos de árbol con recursión

Considere la clase de nodo con 3 datos de miembros, puntero izquierdo izquierdo y puntero secundario derecho, como se muestra a continuación.

```
public class Node {
    public int data;
    public Node left;
    public Node right;

    public Node(int data){
        this.data = data;
    }
}
```

Podemos atravesar el árbol construido conectando el objeto de varias clases de Nodos como a continuación, el recorrido se denomina recorrido transversal del árbol.

```
public static void inOrderTraversal(Node root) {
        if (root != null) {
            inOrderTraversal(root.left); // traverse left sub tree
            System.out.print(root.data + " "); // traverse current node
            inOrderTraversal(root.right); // traverse right sub tree
        }
    }
```

Como se demostró anteriormente, utilizando la **recursión** podemos atravesar la **estructura de datos** del **árbol** sin usar ninguna otra estructura de datos que no sea posible con el enfoque **iterativo** .

Tipos de recursion

La recursión se puede clasificar como **Recursión de cabeza** o **Recursión de cola** , dependiendo de dónde se coloca la llamada de método recursivo.

En la **recursión de la cabeza** , la llamada recursiva, cuando ocurre, se produce antes que otro procesamiento en la función (piense que sucede en la parte superior, o cabeza, de la función).

En la **recursión de cola** , es lo contrario: el procesamiento se produce antes de la llamada recursiva. La elección entre los dos estilos recursivos puede parecer arbitraria, pero la elección puede hacer toda la diferencia.

Una función con una ruta con una sola llamada recursiva al comienzo de la ruta usa lo que se llama recursión de cabeza. La función factorial de una exposición anterior utiliza la recursión de la cabeza. Lo primero que hace una vez que determina que se necesita recursión es llamarse a sí mismo con el parámetro decrementado. Una función con una sola llamada recursiva al final de una ruta está utilizando la recursión de cola.

```
public void tail(int n)
{
   if(n == 1)
      return;
   else
      System.out.println(n);

   tail(n-1);
}
```

```
public void head(int n)
   {
     if(n == 0)
        return;
     else
        head(n-1);

      System.out.println(n);
    }
```

Si la llamada recursiva se produce al final de un método, se denomina tail recursion . La recursión de la cola es similar to a loop . El method executes all the statements before jumping

into the next recursive call .

Si la llamada recursiva se produce al beginning of a method, it is called a head recursion . El method saves the state before jumping into the next recursive call .

Referencia: La diferencia entre recursión de cabeza y cola.

StackOverflowError & recursion to loop

Si una llamada recursiva es "demasiado profunda", esto se traduce en un StackOverflowError . Java asigna un nuevo marco para cada llamada de método en la pila de su hilo. Sin embargo, el espacio de la pila de cada hilo es limitado. Demasiados fotogramas en la pila llevan al desbordamiento de pila (SO).

Ejemplo

```
public static void recursion(int depth) {
    if (depth > 0) {
        recursion(depth-1);
    }
}
```

Si se llama a este método con parámetros grandes (p. Ej., La recursion(50000) probablemente provocará un desbordamiento de pila. El valor exacto depende del tamaño de la pila de hilos, que a su vez depende de la construcción del hilo, los parámetros de la línea de comandos como -Xss , o el tamaño por defecto para la JVM.

Solución

Una recursión se puede convertir en un bucle almacenando los datos para cada llamada recursiva en una estructura de datos. Esta estructura de datos se puede almacenar en el montón en lugar de en la pila de hilos.

En general, los datos necesarios para restaurar el estado de invocación de un método se pueden almacenar en una pila y un bucle while se puede usar para "simular" las llamadas recursivas. Los datos que pueden requerirse incluyen:

- el objeto para el que se solicitó el método (solo métodos de instancia)
- los parámetros del método
- variables locales
- La posición actual en la ejecución o el método.

Ejemplo

La siguiente clase permite la impresión recursiva de una estructura de árbol hasta una profundidad específica.

```
public class Node {

    public int data;
    public Node left;
    public Node right;

    public Node(int data) {
        this(data, null, null);
    }

    public Node(int data, Node left, Node right) {
        this.data = data;
        this.left = left;
        this.right = right;
```

```
    }

    public void print(final int maxDepth) {
        if (maxDepth <= 0) {
            System.out.print("(...)");
        } else {
            System.out.print("(");
            if (left != null) {
                left.print(maxDepth-1);
            }
            System.out.print(data);
            if (right != null) {
                right.print(maxDepth-1);
            }
            System.out.print(")");
        }
    }

}
```

p.ej

```
Node n = new Node(10, new Node(20, new Node(50), new Node(1)), new Node(30, new Node(42),
null));
n.print(2);
System.out.println();
```

Huellas dactilares

```
(((...)20(...))10((...)30))
```

Esto podría ser convertido al siguiente bucle:

```
public class Frame {

    public final Node node;

    // 0: before printing anything
    // 1: before printing data
    // 2: before printing ")"
    public int state = 0;
    public final int maxDepth;

    public Frame(Node node, int maxDepth) {
        this.node = node;
        this.maxDepth = maxDepth;
    }

}
```

```
List<Frame> stack = new ArrayList<>();
stack.add(new Frame(n, 2)); // first frame = initial call

while (!stack.isEmpty()) {
    // get topmost stack element
    int index = stack.size() - 1;
    Frame frame = stack.get(index); // get topmost frame
    if (frame.maxDepth <= 0) {
        // termial case (too deep)
```

```
        System.out.print("(...)");
        stack.remove(index); // drop frame
    } else {
        switch (frame.state) {
            case 0:
                frame.state++;

                // do everything done before the first recursive call
                System.out.print("(");
                if (frame.node.left != null) {
                    // add new frame (recursive call to left and stop)
                    stack.add(new Frame(frame.node.left, frame.maxDepth - 1));
                    break;
                }
            case 1:
                frame.state++;

                // do everything done before the second recursive call
                System.out.print(frame.node.data);
                if (frame.node.right != null) {
                    // add new frame (recursive call to right and stop)
                    stack.add(new Frame(frame.node.right, frame.maxDepth - 1));
                    break;
                }
            case 2:
                // do everything after the second recursive call & drop frame
                System.out.print(")");
                stack.remove(index);
        }
    }
}
System.out.println();
```

Nota: Este es solo un ejemplo del enfoque general. A menudo, puede encontrar una forma mucho mejor de representar un marco y / o almacenar los datos del marco.

La recursión profunda es problemática en Java

Considere el siguiente método ingenuo para sumar dos números positivos usando la recursión:

```
public static int add(int a, int b) {
    if (a == 0) {
        return b;
    } else {
        return add(a - 1, b + 1);  // TAIL CALL
    }
}
```

Esto es algorítmicamente correcto, pero tiene un problema importante. Si llama a add con una gran a , se bloqueará con un StackOverflowError , en cualquier versión de Java hasta (al menos) Java 9.

En un lenguaje de programación funcional típico (y muchos otros lenguajes) el compilador optimiza la recursión de la cola . El compilador notaría que la llamada a add (en la línea marcada) es una llamada de cola y reescribiría la recursión como un bucle. Esta transformación se llama eliminación de cola.

Sin embargo, los compiladores de Java de la generación actual no realizan la eliminación de la llamada de cola. (Esto no es un descuido simple. Hay razones técnicas sustanciales para esto; vea más abajo). En su lugar, cada llamada recursiva de add hace que se asigne un nuevo marco en la pila del hilo. Por ejemplo, si llama a add(1000, 1) , tomará 1000 llamadas recursivas para

llegar a la respuesta 1001 .

El problema es que el tamaño de la pila de hilos de Java se fija cuando se crea el hilo. (Esto incluye el hilo "principal" en un programa de un solo hilo.) Si se asignan demasiados marcos de pila, la pila se desbordará. La JVM detectará esto y lanzará un StackOverflowError .

Un enfoque para lidiar con esto es simplemente usar una pila más grande. Hay opciones de JVM que controlan el tamaño predeterminado de una pila, y también puede especificar el tamaño de la pila como un parámetro del constructor Thread . Desafortunadamente, esto solo "quita" el desbordamiento de pila. Si necesita realizar un cálculo que requiera una pila aún mayor, entonces StackOverflowError regresa.

La solución real es identificar algoritmos recursivos donde es probable una recursión profunda y realizar *manualmente* la optimización de la llamada de cola en el nivel del código fuente. Por ejemplo, nuestro método de add se puede reescribir de la siguiente manera:

```
public static int add(int a, int b) {
    while (a != 0) {
       a = a - 1;
       b = b + 1;
    }
    return b;
}
```

(Obviamente, hay mejores maneras de agregar dos enteros. Lo anterior es simplemente para ilustrar el efecto de la eliminación manual de la llamada de la cola).

Por qué la eliminación de la llamada de cola no está implementada en Java (todavía)

Hay varias razones por las que agregar la eliminación de la llamada de cola a Java no es fácil. Por ejemplo:

- Algunos códigos podrían depender de StackOverflowError para (por ejemplo) colocar un límite en el tamaño de un problema computacional.
- Los administradores de seguridad de Sandbox a menudo se basan en analizar la pila de llamadas al decidir si permiten que un código sin privilegios realice una acción privilegiada.

Como John Rose explica en "Llamadas de cola en la máquina virtual" :

> *"Los efectos de eliminar el marco de la pila de la persona que llama son visibles para algunas API, en particular las verificaciones de control de acceso y el seguimiento de la pila. Es como si la persona que llamó hubiera llamado directamente a la persona llamada. Cualquier privilegio que posea la persona que llama se descarta después de que el control se transfiere al Sin embargo, la vinculación y la accesibilidad del método se calculan antes de la transferencia de control, y tienen en cuenta al llamante que llama ".*

En otras palabras, la eliminación de la llamada de cola podría hacer que un método de control de acceso piense erróneamente que un código de confianza estaba llamando a una API sensible a la seguridad.

Lea Recursion en línea: https://riptutorial.com/es/java/topic/914/recursion

Capítulo 155: Recursos (en classpath)

Introducción

Java permite la recuperación de recursos basados en archivos almacenados dentro de un JAR junto con las clases compiladas. Este tema se enfoca en cargar esos recursos y ponerlos a disposición de su código.

Observaciones

Un *recurso* son datos de tipo archivo con un nombre de ruta, que reside en la ruta de clase. El uso más común de los recursos es el agrupamiento de imágenes de aplicaciones, sonidos y datos de solo lectura (como la configuración predeterminada).

Se puede acceder a los recursos con los métodos ClassLoader.getResource y ClassLoader.getResourceAsStream . El caso de uso más común es tener recursos ubicados en el mismo paquete que la clase que los lee; los métodos Class.getResource y Class.getResourceAsStream sirven este caso de uso común.

La única diferencia entre un método getResource y un método getResourceAsStream es que el primero devuelve una URL, mientras que el segundo abre esa URL y devuelve un InputStream.

Los métodos de ClassLoader aceptan un nombre de recurso similar a una ruta como un argumento y buscan en cada ubicación en la ruta de clases del ClassLoader una entrada que coincida con ese nombre.

- Si una ubicación de classpath es un archivo .jar, una entrada jar con el nombre especificado se considera una coincidencia.
- Si una ubicación de classpath es un directorio, un archivo relativo debajo de ese directorio con el nombre especificado se considera una coincidencia.

El nombre del recurso es similar a la porción de ruta de una URL relativa. En *todas las plataformas,* utiliza barras diagonales (/) como separadores de directorios. No debe comenzar con una barra.

Los métodos de clase correspondientes son similares, excepto:

- El nombre del recurso puede comenzar con una barra inclinada, en cuyo caso esa barra inicial se elimina y el resto del nombre se pasa al método correspondiente de ClassLoader.
- Si el nombre del recurso no comienza con una barra diagonal, se trata en relación con la clase a la que se llama el método getResource o getResourceAsStream. El nombre del recurso real se convierte en *paquete* / *nombre* , donde *paquete* es el nombre del paquete al que pertenece la clase, con cada período reemplazado por una barra y *nombre* es el argumento original dado al método.

Por ejemplo:

```
package com.example;

public class ExampleApplication {
    public void readImage()
    throws IOException {

        URL imageURL = ExampleApplication.class.getResource("icon.png");

        // The above statement is identical to:
        // ClassLoader loader = ExampleApplication.class.getClassLoader();
        // URL imageURL = loader.getResource("com/example/icon.png");

        Image image = ImageIO.read(imageURL);
    }
```

```
}
```

Los recursos deben colocarse en paquetes con nombre, en lugar de en la raíz de un archivo .jar, por la misma razón que las clases se colocan en paquetes: para evitar colisiones entre varios proveedores. Por ejemplo, si hay varios archivos .jar en la ruta de clase, y más de uno de ellos contiene una entrada config.properties en su raíz, las llamadas a los métodos getResource o getResourceAsStream devolverán las config.properties de cualquiera que sea .jar aparece primero en el classpath Este no es un comportamiento predecible en entornos donde el orden de la ruta de clase no está bajo el control directo de la aplicación, como Java EE.

Todos los métodos getResource y getResourceAsStream devuelven un null si el recurso especificado no existe. Dado que los recursos deben agregarse a la aplicación en el momento de la compilación, sus ubicaciones deben ser conocidas cuando se escribe el código; un error al no encontrar un recurso en tiempo de ejecución suele ser el resultado de un error del programador.

Los recursos son de solo lectura. No hay forma de escribir en un recurso. Los desarrolladores novatos a menudo cometen el error de suponer que, dado que el recurso es un archivo físico independiente cuando se desarrolla en un IDE (como Eclipse), será seguro tratarlo como un archivo físico separado en el caso general. Sin embargo, esto no es correcto; las aplicaciones casi siempre se distribuyen como archivos .jar o archivos .war, y en tales casos, un recurso no será un archivo separado y no se podrá escribir. (El método getFile de la clase de URL *no* es una solución para esto; a pesar de su nombre, simplemente devuelve la parte de la ruta de una URL, que de ninguna manera se garantiza que sea un nombre de archivo válido).

No hay una forma segura de enumerar los recursos en tiempo de ejecución. Nuevamente, dado que los desarrolladores son responsables de agregar archivos de recursos a la aplicación en el momento de la compilación, los desarrolladores ya deben conocer sus rutas. Si bien hay soluciones alternativas, no son confiables y eventualmente fallarán.

Examples

Cargando una imagen de un recurso

Para cargar una imagen agrupada:

```
package com.example;

public class ExampleApplication {
    private Image getIcon() throws IOException {
        URL imageURL = ExampleApplication.class.getResource("icon.png");
        return ImageIO.read(imageURL);
    }
}
```

Cargando la configuración por defecto

Para leer las propiedades de configuración por defecto:

```
package com.example;

public class ExampleApplication {
    private Properties getDefaults() throws IOException {
        Properties defaults = new Properties();

        try (InputStream defaultsStream =
            ExampleApplication.class.getResourceAsStream("config.properties")) {

            defaults.load(defaultsStream);
        }
```

```
        return defaults;
    }
}
```

Cargando recursos del mismo nombre desde múltiples archivos JAR

El recurso con la misma ruta y nombre puede existir en más de un archivo JAR en la ruta de clase. Los casos comunes son recursos que siguen una convención o que son parte de una especificación de empaquetado. Ejemplos de tales recursos son

- META-INF / MANIFEST.MF
- META-INF / beans.xml (CDI Spec)
- Propiedades de ServiceLoader que contienen proveedores de implementación

Para obtener acceso a *todos* estos recursos en diferentes tarros, uno tiene que usar un ClassLoader, que tiene un método para esto. La Enumeration devuelta se puede convertir convenientemente en una List mediante una función de colecciones.

```
Enumeration<URL> resEnum = MyClass.class.getClassLoader().getResources("META-
INF/MANIFEST.MF");
ArrayList<URL> resources = Collections.list(resEnum);
```

Encontrar y leer recursos usando un cargador de clases

La carga de recursos en Java comprende los siguientes pasos:

1. Encontrar la Class o ClassLoader que encontrará el recurso.
2. Encontrar el recurso.
3. Obtención del flujo de bytes para el recurso.
4. Lectura y procesamiento del flujo de bytes.
5. Cierre de la corriente de bytes.

Los últimos tres pasos se realizan normalmente al pasar la URL a un método de biblioteca o un constructor para cargar el recurso. Normalmente utilizarás un método getResource en este caso. También es posible leer los datos del recurso en el código de la aplicación. En este caso, normalmente getResourceAsStream .

Rutas de recursos absolutos y relativos

Los recursos que se pueden cargar desde la ruta de clase se indican mediante una *ruta* . La sintaxis de la ruta es similar a la ruta de un archivo UNIX / Linux. Consiste en nombres simples separados por caracteres de barra (/). Una *ruta relativa* comienza con un nombre y una *ruta absoluta* comienza con un separador.

Como describen los ejemplos de Classpath, la classpath de una JVM define un espacio de nombres superponiendo los espacios de nombres de los directorios y los archivos JAR o ZIP en la classpath. Cuando se resuelve una ruta absoluta, los cargadores de clases interpretan la inicial / como la raíz del espacio de nombres. Por el contrario, una ruta relativa *puede* resolverse en relación con cualquier "carpeta" en el espacio de nombres. La carpeta utilizada dependerá del objeto que utilice para resolver la ruta.

Obtención de una Clase o Classloader

Un recurso se puede ubicar utilizando un objeto Class o un objeto ClassLoader . Un objeto de Class puede resolver rutas relativas, por lo que normalmente usará una de estas si tiene un recurso relativo (clase). Hay varias formas de obtener un objeto de Class . Por ejemplo:

- Un *literal de clase* le dará el objeto de Class para cualquier clase que pueda nombrar en el código fuente de Java; por ejemplo, String.class te da el objeto Class para el tipo String .

- El Object.getClass() le dará el objeto Class para el tipo od cualquier objeto; por ejemplo, "hello".getClass() es otra forma de obtener la Class del tipo String .

- El método Class.forName(String) (si es necesario) cargará dinámicamente una clase y devolverá su objeto Class ; por ejemplo, Class.forName("java.lang.String") .

Un objeto ClassLoader se obtiene normalmente llamando a getClassLoader() en un objeto Class . También es posible obtener el cargador de clases predeterminado de la JVM utilizando el método estático ClassLoader.getSystemClassLoader() .

Los metodos de get

Una vez que tenga una instancia de Class o ClassLoader , puede encontrar un recurso, usando uno de los siguientes métodos:

Métodos	Descripción
ClassLoader.getResource(path) ClassLoader.getResources(path)	Devuelve una URL que representa la ubicación del recurso con la ruta dada.
ClassLoader.getResources(path) Class.getResources(path)	Devuelve una Enumeration<URL> proporciona las URL que se pueden usar para ubicar el recurso foo.bar ; vea abajo.
ClassLoader.getResourceAsStream(path) Class.getResourceStream(path)	Devuelve un InputStream desde el cual puede leer el contenido del recurso foo.bar como una secuencia de bytes.

Notas:

- La principal diferencia entre las versiones ClassLoader y Class de los métodos se encuentra en la forma en que se interpretan las rutas relativas.
    - Los métodos de Class resuelven una ruta relativa en la "carpeta" que corresponde al paquete de clases.
    - Los métodos de ClassLoader tratan las rutas relativas como si fueran absolutas; es decir, resolverlos en la "carpeta raíz" del espacio de nombres de classpath.

- Si no se puede encontrar el recurso (o recursos) solicitado, los methods return getResource y getResourceAsStream methods return valor nulo , and the methods return an empty getResources methods return an empty Enumeración methods return an empty .

- Las URL devueltas se podrán resolver mediante URL.toStream() . Pueden ser file: URL u otras URL convencionales, pero si el recurso reside en un archivo JAR, serán jar: URL que identifican el archivo JAR y un recurso específico dentro de él.

- Si su código utiliza un método getResourceAsStream (o URL.toStream()) para obtener un InputStream , es responsable de cerrar el objeto de flujo. Si no se cierra el flujo, podría producirse una fuga de recursos.

Lea Recursos (en classpath) en línea: https://riptutorial.com/es/java/topic/2433/recursos--en-classpath-

Capítulo 156: Redes

Sintaxis

- nuevo Socket ("localhost", 1234); // Se conecta a un servidor en la dirección "localhost" y el puerto 1234
- nuevo SocketServer ("localhost", 1234); // Crea un servidor de socket que puede escuchar nuevos sockets en la dirección localhost y el puerto 1234
- socketServer.accept (); // Acepta un nuevo objeto Socket que puede usarse para comunicarse con el cliente

Examples

Comunicación básica de cliente y servidor mediante un socket

Servidor: Iniciar y esperar las conexiones entrantes.

```
//Open a listening "ServerSocket" on port 1234.
ServerSocket serverSocket = new ServerSocket(1234);

while (true) {
    // Wait for a client connection.
    // Once a client connected, we get a "Socket" object
    // that can be used to send and receive messages to/from the newly
    // connected client
    Socket clientSocket = serverSocket.accept();

    // Here we'll add the code to handle one specific client.
}
```

Servidor: Manejo de clientes.

Manejaremos cada cliente en un subproceso separado para que múltiples clientes puedan interactuar con el servidor al mismo tiempo. Esta técnica funciona bien siempre que el número de clientes sea bajo (<< 1000 clientes, según la arquitectura del sistema operativo y la carga esperada de cada subproceso).

```
new Thread(() -> {
    // Get the socket's InputStream, to read bytes from the socket
    InputStream in = clientSocket.getInputStream();
    // wrap the InputStream in a reader so you can read a String instead of bytes
    BufferedReader reader = new BufferedReader(
            new InputStreamReader(in, StandardCharsets.UTF_8));
    // Read text from the socket and print line by line
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
    }).start();
```

Cliente: Conectarse al servidor y enviar un mensaje.

```
// 127.0.0.1 is the address of the server (this is the localhost address; i.e.
// the address of our own machine)
// 1234 is the port that the server will be listening on
Socket socket = new Socket("127.0.0.1", 1234);

// Write a string into the socket, and flush the buffer
```

```
OutputStream outStream = socket.getOutputStream();
PrintWriter writer = new PrintWriter(
        new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
writer.println("Hello world!");
writer.flush();
```

Cierre de enchufes y manejo de excepciones.

Los ejemplos anteriores dejaron algunas cosas para facilitar su lectura.

1. Al igual que los archivos y otros recursos externos, es importante que le informemos al sistema operativo cuando hayamos terminado con ellos. Cuando hayamos terminado con un socket, llame a socket.close() para cerrarlo correctamente.

2. Los sockets manejan las operaciones de E / S (entrada / salida) que dependen de una variedad de factores externos. Por ejemplo, ¿qué pasa si el otro lado se desconecta de repente? ¿Qué pasa si hay un error de red? Estas cosas están más allá de nuestro control. Esta es la razón por la que muchas operaciones de socket pueden generar excepciones, especialmente IOException .

Por lo tanto, un código más completo para el cliente sería algo como esto:

```
// "try-with-resources" will close the socket once we leave its scope
try (Socket socket = new Socket("127.0.0.1", 1234)) {
    OutputStream outStream = socket.getOutputStream();
    PrintWriter writer = new PrintWriter(
            new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
    writer.println("Hello world!");
    writer.flush();
} catch (IOException e) {
    //Handle the error
}
```

Servidor básico y cliente - ejemplos completos

Servidor:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Server {
    public static void main(String args[]) {
        try (ServerSocket serverSocket = new ServerSocket(1234)) {
            while (true) {
                // Wait for a client connection.
                Socket clientSocket = serverSocket.accept();

                // Create and start a thread to handle the new client
                new Thread(() -> {
                    try {
                        // Get the socket's InputStream, to read bytes
                        // from the socket
                        InputStream in = clientSocket.getInputStream();
                        // wrap the InputStream in a reader so you can
                        // read a String instead of bytes
                        BufferedReader reader = new BufferedReader(
```

```
                            new InputStreamReader(in, StandardCharsets.UTF_8));
                        // Read from the socket and print line by line
                        String line;
                        while ((line = reader.readLine()) != null) {
                            System.out.println(line);
                        }
                    }
                    catch (IOException e) {
                        e.printStackTrace();
                    } finally {
                        // This finally block ensures the socket is closed.
                        // A try-with-resources block cannot be used because
                        // the socket is passed into a thread, so it isn't
                        // created and closed in the same block
                        try {
                            clientSocket.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                }).start();
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

Cliente:

```
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Client {
    public static void main(String args[]) {
        try (Socket socket = new Socket("127.0.0.1", 1234)) {
            // We'll reach this code once we've connected to the server

            // Write a string into the socket, and flush the buffer
            OutputStream outStream = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(
                    new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
            writer.println("Hello world!");
            writer.flush();
        } catch (IOException e) {
            // Exception should be handled.
            e.printStackTrace();
        }
    }
}
```

Cargando TrustStore y KeyStore desde InputStream

```
public class TrustLoader {

    public static void main(String args[]) {
        try {
                //Gets the inputstream of a a trust store file under ssl/rpgrenadesClient.jks
                //This path refers to the ssl folder in the jar file, in a jar file in the
same directory
                //as this jar file, or a different directory in the same directory as the jar
file
                InputStream stream =
TrustLoader.class.getResourceAsStream("/ssl/rpgrenadesClient.jks");
                //Both trustStores and keyStores are represented by the KeyStore object
                KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
                //The password for the trustStore
                char[] trustStorePassword = "password".toCharArray();
                //This loads the trust store into the object
                trustStore.load(stream, trustStorePassword);

                //This is defining the SSLContext so the trust store will be used
                //Getting default SSLContext to edit.
                SSLContext context = SSLContext.getInstance("SSL");
                //TrustMangers hold trust stores, more than one can be added
                TrustManagerFactory factory =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
                //Adds the truststore to the factory
                factory.init(trustStore);
                //This is passed to the SSLContext init method
                TrustManager[] managers = factory.getTrustManagers();
                context.init(null, managers, null);
                //Sets our new SSLContext to be used.
                SSLContext.setDefault(context);
            } catch (KeyStoreException | IOException | NoSuchAlgorithmException
                    | CertificateException | KeyManagementException ex) {
                //Handle error
                ex.printStackTrace();
            }

    }
}
```

La iniciación de un Almacén de claves funciona de la misma manera, excepto que reemplaza cualquier palabra Trust en un nombre de objeto con Key . Además, la matriz KeyManager[] debe pasar al primer argumento de SSLContext.init . Eso es SSLContext.init(keyMangers, trustMangers, null)

Ejemplo de socket: leer una página web utilizando un socket simple

```
import java.io.*;
import java.net.Socket;

public class Main {

    public static void main(String[] args) throws IOException {//We don't handle Exceptions in
this example
        //Open a socket to stackoverflow.com, port 80
        Socket socket = new Socket("stackoverflow.com",80);

        //Prepare input, output stream before sending request
        OutputStream outStream = socket.getOutputStream();
        InputStream inStream = socket.getInputStream();
```

```
        BufferedReader reader = new BufferedReader(new InputStreamReader(inStream));
        PrintWriter writer = new PrintWriter(new BufferedOutputStream(outStream));

        //Send a basic HTTP header
        writer.print("GET / HTTP/1.1\nHost:stackoverflow.com\n\n");
        writer.flush();

        //Read the response
        System.out.println(readFully(reader));

        //Close the socket
        socket.close();
    }

    private static String readFully(Reader in) {
        StringBuilder sb = new StringBuilder();
        int BUFFER_SIZE=1024;
        char[] buffer = new char[BUFFER_SIZE]; // or some other size,
        int charsRead = 0;
        while ( (charsRead  = rd.read(buffer, 0, BUFFER_SIZE)) != -1) {
          sb.append(buffer, 0, charsRead);
        }
    }
}
```

Debería obtener una respuesta que comience con HTTP/1.1 200 OK , que indica una respuesta HTTP normal, seguida del resto del encabezado HTTP, seguida de la página web sin procesar en formato HTML.

Tenga en cuenta que el método readFully() es importante para evitar una excepción prematura de EOF. A la última línea de la página web le puede faltar una devolución, para indicar el final de la línea, luego readLine() se quejará, por lo que uno debe leerla a mano o usar métodos de utilidad de Apache commons-io IOUtils

Este ejemplo pretende ser una simple demostración de conectarse a un recurso existente mediante un socket, no es una forma práctica de acceder a páginas web. Si necesita acceder a una página web utilizando Java, es mejor utilizar una biblioteca de cliente HTTP existente, como el Cliente HTTP de Apache o el Cliente HTTP de Google.

Comunicación básica cliente / servidor mediante UDP (datagrama)

Client.java

```
import java.io.*;
import java.net.*;

public class Client{
    public static void main(String [] args) throws IOException{
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress address = InetAddress.getByName(args[0]);

        String ex = "Hello, World!";
        byte[] buf = ex.getBytes();

        DatagramPacket packet = new DatagramPacket(buf,buf.length, address, 4160);
        clientSocket.send(packet);
    }
}
```

En este caso, pasamos la dirección del servidor, a través de un argumento (args[0]). El puerto que estamos utilizando es 4160.

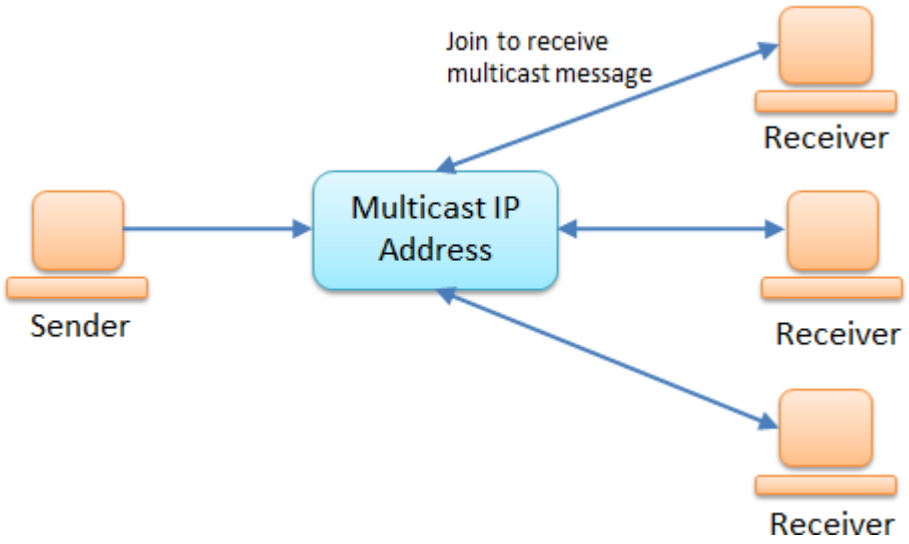
Servidor.java

```
import java.io.*;
import java.net.*;

public class Server{
    public static void main(String [] args) throws IOException{
        DatagramSocket serverSocket = new DatagramSocket(4160);

        byte[] rbuf = new byte[256];
        DatagramPacket packet = new DatagramPacket(rbuf, rbuf.length);
        serverSocket.receive(packet);
        String response = new String(packet.getData());
        System.out.println("Response: " + response);
    }
}
```

En el lado del servidor, declare un DatagramSocket en el mismo puerto al que enviamos nuestro mensaje (4160) y espere una respuesta.

Multidifusión

La multidifusión es un tipo de Datagram Socket. A diferencia de los Datagramas normales, la Multidifusión no maneja cada cliente individualmente, en lugar de eso, lo envía a una dirección IP y todos los clientes suscritos recibirán el mensaje.



Código de ejemplo para un servidor:

```
public class Server {

    private DatagramSocket serverSocket;

    private String ip;

    private int port;

    public Server(String ip, int port) throws SocketException, IOException{
        this.ip = ip;
        this.port = port;
        // socket used to send
        serverSocket = new DatagramSocket();
    }
```

```
    public void send() throws IOException{
        // make datagram packet
        byte[] message = ("Multicasting...").getBytes();
        DatagramPacket packet = new DatagramPacket(message, message.length,
            InetAddress.getByName(ip), port);
        // send packet
        serverSocket.send(packet);
    }

    public void close(){
        serverSocket.close();
    }
}
```

Código de ejemplo para un lado del cliente:

```
public class Client {

    private MulticastSocket socket;

    public Client(String ip, int port) throws IOException {

        // important that this is a multicast socket
        socket = new MulticastSocket(port);

        // join by ip
        socket.joinGroup(InetAddress.getByName(ip));
    }

    public void printMessage() throws IOException{
        // make datagram packet to recieve
        byte[] message = new byte[256];
        DatagramPacket packet = new DatagramPacket(message, message.length);

        // recieve the packet
        socket.receive(packet);
        System.out.println(new String(packet.getData()));
    }

    public void close(){
        socket.close();
    }
}
```

Código para ejecutar el servidor:

```
public static void main(String[] args) {
    try {
        final String ip = args[0];
        final int port = Integer.parseInt(args[1]);
        Server server = new Server(ip, port);
        server.send();
        server.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Código para ejecutar un cliente:

```
public static void main(String[] args) {
    try {
        final String ip = args[0];
        final int port = Integer.parseInt(args[1]);
        Client client = new Client(ip, port);
        client.printMessage();
        client.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Primero ejecute el Cliente: El Cliente debe suscribirse a la IP antes de que pueda comenzar a recibir paquetes. Si inicia el servidor y llama al método send() , y luego printMessage() un cliente (& call printMessage()). Nada ocurrirá porque el cliente se conectó después de que se envió el mensaje.

Deshabilite temporalmente la verificación SSL (para propósitos de prueba)

En ocasiones, en un entorno de desarrollo o prueba, es posible que la cadena de certificados SSL no esté completamente establecida (aún).

Para continuar con el desarrollo y las pruebas, puede desactivar la verificación SSL mediante programación instalando un administrador de confianza "confiable":

```
try {
   // Create a trust manager that does not validate certificate chains
   TrustManager[] trustAllCerts = new TrustManager[] {
      new X509TrustManager() {
       public X509Certificate[] getAcceptedIssuers() {
           return null;
       }
       public void checkClientTrusted(X509Certificate[] certs, String authType) {
       }
       public void checkServerTrusted(X509Certificate[] certs, String authType) {
       }
      }
   };

   // Install the all-trusting trust manager
   SSLContext sc = SSLContext.getInstance("SSL");
   sc.init(null, trustAllCerts, new java.security.SecureRandom());
   HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

   // Create all-trusting host name verifier
   HostnameVerifier allHostsValid = new HostnameVerifier() {
       public boolean verify(String hostname, SSLSession session) {
           return true;
       }
   };

   // Install the all-trusting host verifier
   HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);
} catch (NoSuchAlgorithmException | KeyManagementException e) {
    e.printStackTrace();
}
```

Descargando un archivo usando el canal

Si el archivo ya existe, se sobrescribirá!

```
String fileName     = "file.zip";                   // name of the file
String urlToGetFrom = "http://www.mywebsite.com/"; // URL to get it from
String pathToSaveTo = "C:\\Users\\user\\";          // where to put it

//If the file already exists, it will be overwritten!

//Opening OutputStream to the destination file
try (ReadableByteChannel rbc =
      Channels.newChannel(new URL(urlToGetFrom + fileName).openStream()) ) {
    try ( FileChannel channel =
        new FileOutputStream(pathToSaveTo + fileName).getChannel(); ) {
      channel.transferFrom(rbc, 0, Long.MAX_VALUE);
    }
    catch (FileNotFoundException e) { /* Output directory not found */ }
    catch (IOException e)           { /* File IO error */ }
}
catch (MalformedURLException e)     { /* URL is malformed */ }
catch (IOException e)               { /* IO error connecting to website */ }
```

Notas

- No dejes los bloques de captura vacíos!
- En caso de error, compruebe si el archivo remoto existe
- Esta es una operación de bloqueo, puede llevar mucho tiempo con archivos grandes

Lea Redes en línea: https://riptutorial.com/es/java/topic/149/redes

Capítulo 157: Referencias de objetos

Observaciones

Esto debería ayudarlo a comprender una "excepción de puntero nulo": uno de ellos obtiene uno porque una referencia de objeto es nula, pero el código del programa espera que el programa use algo en esa referencia de objeto. Sin embargo, eso merece su propio tema ...

Examples

Referencias de objetos como parámetros del método.

Este tema explica el concepto de una *referencia de objeto* ; Está dirigido a personas que son nuevas en la programación en Java. Ya debe estar familiarizado con algunos términos y significados: definición de clase, método principal, instancia de objeto y la llamada de métodos "en" un objeto, y pasar parámetros a los métodos.

```
public class Person {

  private String name;

  public void setName(String name) { this.name = name; }

  public String getName() { return name; }

  public static void main(String [] arguments) {
    Person person = new Person();
    person.setName("Bob");

    int i = 5;
    setPersonName(person, i);

    System.out.println(person.getName() + " " + i);
  }

  private static void setPersonName(Person person, int num) {
    person.setName("Linda");
    num = 99;
  }
}
```

Para ser plenamente competente en la programación de Java, debería poder explicar este ejemplo a alguien que no esté en lo alto de su cabeza. Sus conceptos son fundamentales para entender cómo funciona Java.

Como puede ver, tenemos un main que crea una instancia de un objeto para la person variable y llama a un método para establecer el campo de name en ese objeto en "Bob" . Luego llama a otro método y pasa a la person como uno de dos parámetros; el otro parámetro es una variable entera, establecida en 5.

El método llamado establece el valor del name en el objeto pasado a "Linda", y establece la variable entera pasada a 99, luego regresa.

Entonces, ¿qué se imprimiría?

```
Linda 5
```

Entonces, ¿por qué el cambio realizado en person tiene efecto en main , pero el cambio realizado en el entero no lo hace?

Cuando se realiza la llamada, el método principal pasa una *referencia de objeto* por person al método setPersonName ; cualquier cambio que setAnotherName haga a ese objeto es parte de ese objeto, por lo que esos cambios aún son parte de ese objeto cuando el método regresa.

Otra forma de decir lo mismo: la person apunta a un objeto (almacenado en el montón, si está interesado). Cualquier cambio que el método haga a ese objeto se realiza "en ese objeto", y no se ve afectado por si el método que realiza el cambio todavía está activo o ha regresado. Cuando el método vuelve, cualquier cambio realizado en el objeto todavía se almacena en ese objeto.

Contrasta esto con el entero que se pasa. Dado que este es un int *primitivo* (y no una instancia de objeto Integer), se pasa "por valor", lo que significa que su valor se proporciona al método, no un puntero al entero original pasado. El método puede cambiarlo por el método propósitos propios, pero eso no afecta a la variable utilizada cuando se realiza la llamada al método.

En Java, todas las primitivas se pasan por valor. Los objetos se pasan por referencia, lo que significa que un puntero al objeto se pasa como parámetro a cualquier método que los tome.

Esto significa algo menos obvio: no es posible que un método llamado cree un *nuevo* objeto y lo devuelva como uno de los parámetros. La única forma en que un método puede devolver un objeto que se crea, directa o indirectamente, mediante la llamada al método, es como un valor de retorno del método. Primero veamos cómo eso no funcionaría, y luego cómo funcionaría.

Agreguemos otro método a nuestro pequeño ejemplo aquí:

```
private static void getAnotherObjectNot(Person person) {
  person = new Person();
  person.setName("George");
}
```

Y, de nuevo en la parte main , debajo de la llamada a setAnotherName , pongamos una llamada a este método y otra llamada println:

```
getAnotherObjectNot(person);
System.out.println(person.getName());
```

Ahora el programa se imprimiría:

```
Linda 5
Linda
```

¿Qué pasó con el objeto que tenía George? Bueno, el parámetro que se pasó fue un indicador a Linda; cuando el método getAnotherObjectNot creó un nuevo objeto, reemplazó la referencia al objeto de Linda con una referencia al objeto de George. El objeto de Linda todavía existe (en el montón), el método main todavía puede acceder a él, pero el método getAnotherObjectNot no podría hacer nada con él después de eso, porque no tiene ninguna referencia a él. Parece que el autor del código destinado al método crea un nuevo objeto y lo devuelve, pero si es así, no funciona.

Si eso es lo que el escritor quería hacer, tendría que devolver el objeto recién creado desde el método, algo como esto:

```
private static Person getAnotherObject() {
  Person person = new Person();
  person.setName("Mary");
  return person;
}
```

Entonces llámalo así:

```
Person mary;
mary = getAnotherObject();
System.out.println(mary.getName());
```

Y la salida del programa completo ahora sería:

```
Linda 5
Linda
Mary
```

Aquí está el programa completo, con ambas adiciones:

```
public class Person {
  private String name;

  public void setName(String name) { this.name = name; }
  public String getName() { return name; }

  public static void main(String [] arguments) {
    Person person = new Person();
    person.setName("Bob");

    int i = 5;
    setPersonName(person, i);
    System.out.println(person.getName() + " " + i);

    getAnotherObjectNot(person);
    System.out.println(person.getName());

    Person person;
    person = getAnotherObject();
    System.out.println(person.getName());
  }

  private static void setPersonName(Person person, int num) {
    person.setName("Linda");
    num = 99;
  }

  private static void getAnotherObjectNot(Person person) {
    person = new Person();
    person.setMyName("George");
  }

  private static person getAnotherObject() {
    Person person = new Person();
    person.setMyName("Mary");
    return person;
  }
}
```

Lea Referencias de objetos en línea: https://riptutorial.com/es/java/topic/5454/referencias-de-objetos

Capítulo 158: Registro (java.util.logging)

Examples

Usando el registrador predeterminado

Este ejemplo muestra cómo usar la api de registro predeterminada.

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class MyClass {

    // retrieve the logger for the current class
    private static final Logger LOG = Logger.getLogger(MyClass.class.getName());

    public void foo() {
        LOG.info("A log message");
        LOG.log(Level.INFO, "Another log message");

        LOG.fine("A fine message");

        // logging an exception
        try {
            // code might throw an exception
        } catch (SomeException ex) {
            // log a warning printing "Something went wrong"
            // together with the exception message and stacktrace
            LOG.log(Level.WARNING, "Something went wrong", ex);
        }

        String s = "Hello World!";

        // logging an object
        LOG.log(Level.FINER, "String s: {0}", s);

        // logging several objects
        LOG.log(Level.FINEST, "String s: {0} has length {1}", new Object[]{s, s.length()});
    }

}
```

Niveles de registro

Java Logging Api tiene 7 niveles . Los niveles en orden descendente son:

- SEVERE (valor más alto)
- WARNING
  - INFO
  - CONFIG
  - FINE
  - FINER
  - FINEST (valor más bajo)

El nivel predeterminado es INFO (pero esto depende del sistema y utiliza una máquina virtual).

Nota : También hay niveles OFF (se pueden usar para desactivar el registro) y ALL (el opuesto de OFF).

Código de ejemplo para esto:

```
import java.util.logging.Logger;

public class Levels {
    private static final Logger logger = Logger.getLogger(Levels.class.getName());

    public static void main(String[] args) {

        logger.severe("Message logged by SEVERE");
        logger.warning("Message logged by WARNING");
        logger.info("Message logged by INFO");
        logger.config("Message logged by CONFIG");
        logger.fine("Message logged by FINE");
        logger.finer("Message logged by FINER");
        logger.finest("Message logged by FINEST");

        // All of above methods are really just shortcut for
        // public void log(Level level, String msg):
        logger.log(Level.FINEST, "Message logged by FINEST");
    }
}
```

Por defecto, ejecutar esta clase solo generará mensajes con un nivel más alto que CONFIG :

```
Jul 23, 2016 9:16:11 PM LevelsExample main
SEVERE: Message logged by SEVERE
Jul 23, 2016 9:16:11 PM LevelsExample main
WARNING: Message logged by WARNING
Jul 23, 2016 9:16:11 PM LevelsExample main
INFO: Message logged by INFO
```

Registro de mensajes complejos (eficientemente)

Veamos una muestra de registro que puede ver en muchos programas:

```
public class LoggingComplex {

    private static final Logger logger =
        Logger.getLogger(LoggingComplex.class.getName());

    private int total = 50, orders = 20;
    private String username = "Bob";

    public void takeOrder() {
        // (...) making some stuff
        logger.fine(String.format("User %s ordered %d things (%d in total)",
                                  username, orders, total));
        // (...) some other stuff
    }

    // some other methods and calculations
}
```

El ejemplo anterior se ve perfectamente bien, pero muchos programadores olvidan que Java VM es una máquina de pila. Esto significa que todos los parámetros del método se calculan **antes de** ejecutar el método.

Este hecho es crucial para el registro en Java, especialmente para registrar algo en niveles bajos, como FINE , FINER , FINEST que son desactivados por defecto. Veamos el takeOrder() Java

para el método takeOrder() .

El resultado para javap -c LoggingComplex.class es algo como esto:

```
public void takeOrder();
    Code:
       0: getstatic     #27 // Field logger:Ljava/util/logging/Logger;
       3: ldc           #45 // String User %s ordered %d things (%d in total)
       5: iconst_3
       6: anewarray     #3  // class java/lang/Object
       9: dup
      10: iconst_0
      11: aload_0
      12: getfield      #40 // Field username:Ljava/lang/String;
      15: aastore
      16: dup
      17: iconst_1
      18: aload_0
      19: getfield      #36 // Field orders:I
      22: invokestatic  #47 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
      25: aastore
      26: dup
      27: iconst_2
      28: aload_0
      29: getfield      #34 // Field total:I
      32: invokestatic  #47 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
      35: aastore
      36: invokestatic  #53 // Method
java/lang/String.format:(Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;
      39: invokevirtual #59 // Method java/util/logging/Logger.fine:(Ljava/lang/String;)V
      42: return
```

La línea 39 ejecuta el registro real. Todo el trabajo anterior (cargar variables, crear nuevos objetos, concatenar cadenas en el método de format) puede ser en vano si el nivel de registro se establece más alto que FINE (y por defecto lo es). Dicho registro puede ser muy ineficiente y consumir recursos innecesarios de memoria y procesador.

Por eso debe preguntar si el nivel que desea usar está habilitado.

La forma correcta debe ser:

```
public void takeOrder() {
    // making some stuff
    if (logger.isLoggable(Level.FINE)) {
        // no action taken when there's no need for it
        logger.fine(String.format("User %s ordered %d things (%d in total)",
                                  username, orders, total));
    }
    // some other stuff
}
```

Desde Java 8:

La clase Logger tiene métodos adicionales que toman un Supplier<String> como parámetro, que simplemente puede proporcionar un lambda:

```
public void takeOrder() {
    // making some stuff
    logger.fine(() -> String.format("User %s ordered %d things (%d in total)",
            username, orders, total));
    // some other stuff
}
```

El método de get() Proveedores get() en este caso, la lambda, solo se llama cuando el nivel correspondiente está habilitado y por lo tanto la construcción if ya no es necesaria.

Lea Registro (java.util.logging) en línea: https://riptutorial.com/es/java/topic/2010/registro--java-util-logging-

Capítulo 159: Seguridad y criptografía

Examples

Calcular los hash criptográficos

Para calcular los hashes de bloques de datos relativamente pequeños utilizando diferentes algoritmos:

```
final MessageDigest md5 = MessageDigest.getInstance("MD5");
final MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
final MessageDigest sha256 = MessageDigest.getInstance("SHA-256");

final byte[] data = "FOO BAR".getBytes();

System.out.println("MD5    hash: " + DatatypeConverter.printHexBinary(md5.digest(data)));
System.out.println("SHA1   hash: " + DatatypeConverter.printHexBinary(sha1.digest(data)));
System.out.println("SHA256 hash: " + DatatypeConverter.printHexBinary(sha256.digest(data)));
```

Produce esta salida:

```
MD5    hash: E99E768582F6DD5A3BA2D9C849DF736E
SHA1   hash: 0135FAA6323685BA8A8FF8D3F955F0C36949D8FB
SHA256 hash: 8D35C97BCD902B96D1B551741BBE8A7F50BB5A690B4D0225482EAA63DBFB9DED
```

Es posible que haya algoritmos adicionales disponibles según su implementación de la plataforma Java.

Generar datos criptográficamente aleatorios

Para generar muestras de datos criptográficamente aleatorios:

```
final byte[] sample = new byte[16];

new SecureRandom().nextBytes(sample);

System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

Produce una salida similar a:

```
Sample: E4F14CEA2384F70B706B53A6DF8C5EFE
```

Tenga en cuenta que la llamada a nextBytes() puede bloquearse mientras se recopila la entropía según el algoritmo que se esté utilizando.

Para especificar el algoritmo y el proveedor:

```
final byte[] sample = new byte[16];
final SecureRandom randomness = SecureRandom.getInstance("SHA1PRNG", "SUN");

randomness.nextBytes(sample);

System.out.println("Provider: " + randomness.getProvider());
System.out.println("Algorithm: " + randomness.getAlgorithm());
System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

Produce una salida similar a:

```
Provider: SUN version 1.8
Algorithm: SHA1PRNG
Sample: C80C44BAEB352FD29FBBE20489E4C0B9
```

Generar pares de claves públicas / privadas

Para generar pares de claves utilizando diferentes algoritmos y tamaños de clave:

```
final KeyPairGenerator dhGenerator = KeyPairGenerator.getInstance("DiffieHellman");
final KeyPairGenerator dsaGenerator = KeyPairGenerator.getInstance("DSA");
final KeyPairGenerator rsaGenerator = KeyPairGenerator.getInstance("RSA");

dhGenerator.initialize(1024);
dsaGenerator.initialize(1024);
rsaGenerator.initialize(2048);

final KeyPair dhPair = dhGenerator.generateKeyPair();
final KeyPair dsaPair = dsaGenerator.generateKeyPair();
final KeyPair rsaPair = rsaGenerator.generateKeyPair();
```

Es posible que haya algoritmos adicionales y tamaños de clave disponibles en su implementación de la plataforma Java.

Para especificar una fuente de aleatoriedad para usar al generar las claves:

```
final KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");

generator.initialize(2048, SecureRandom.getInstance("SHA1PRNG", "SUN"));

final KeyPair pair = generator.generateKeyPair();
```

Calcular y verificar firmas digitales

Para calcular una firma:

```
final PrivateKey privateKey = keyPair.getPrivate();
final byte[] data = "FOO BAR".getBytes();
final Signature signer = Signature.getInstance("SHA1withRSA");

signer.initSign(privateKey);
signer.update(data);

final byte[] signature = signer.sign();
```

Tenga en cuenta que el algoritmo de firma debe ser compatible con el algoritmo utilizado para generar el par de claves.

Para verificar una firma:

```
final PublicKey publicKey = keyPair.getPublic();
final Signature verifier = Signature.getInstance("SHA1withRSA");

verifier.initVerify(publicKey);
verifier.update(data);

System.out.println("Signature: " + verifier.verify(signature));
```

Produce esta salida:

```
Signature: true
```

Cifrar y descifrar datos con claves públicas / privadas

Para cifrar los datos con una clave pública:

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
rsa.update(message.getBytes());
final byte[] result = rsa.doFinal();

System.out.println("Message: " + message);
System.out.println("Encrypted: " + DatatypeConverter.printHexBinary(result));
```

Produce una salida similar a:

```
Message: Hello
Encrypted: 5641FBB9558ECFA9ED...
```

Tenga en cuenta que al crear el objeto Cipher , debe especificar una transformación que sea compatible con el tipo de clave que se está utilizando. (Consulte Nombres de algoritmos estándar JCA para obtener una lista de las transformaciones admitidas). Para los datos de cifrado RSA, la longitud de message.getBytes() debe ser menor que el tamaño de la clave. Vea esta respuesta SO para detalles.

Para descifrar los datos:

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
rsa.update(cipherText);
final String result = new String(rsa.doFinal());

System.out.println("Decrypted: " + result);
```

Produce la siguiente salida:

```
Decrypted: Hello
```

Lea Seguridad y criptografía en línea: https://riptutorial.com/es/java/topic/7529/seguridad-y-criptografia

Capítulo 160: Seguridad y criptografía

Introducción

Las prácticas de seguridad en Java se pueden separar en dos categorías amplias, vagamente definidas; Seguridad de la plataforma Java, y programación Java segura.

Las prácticas de seguridad de la plataforma Java se ocupan de gestionar la seguridad y la integridad de la JVM. Incluye temas como la gestión de proveedores de JCE y políticas de seguridad.

Las prácticas de programación de Java seguras se refieren a las mejores formas de escribir programas de Java seguros. Incluye temas como el uso de números aleatorios y la criptografía, y la prevención de vulnerabilidades.

Observaciones

Si bien los ejemplos deben hacerse claramente, algunos temas que deben cubrirse son:

1. El concepto / estructura del proveedor de JCE
2. Elemento de lista

Examples

La jce

La extensión de criptografía de Java (JCE) es un marco integrado en la JVM para permitir a los desarrolladores utilizar la criptografía de forma fácil y segura en sus programas. Para ello, proporciona una interfaz simple y portátil a los programadores, mientras utiliza un sistema de proveedores JCE para implementar de forma segura las operaciones criptográficas subyacentes.

Claves y gestión de claves

Si bien la JCE asegura las operaciones criptográficas y la generación de claves, es responsabilidad del desarrollador administrar realmente sus claves. Más información debe ser proporcionada aquí.

Una de las mejores prácticas comúnmente aceptadas para manejar claves en tiempo de ejecución es almacenarlas solo como matrices de byte , y nunca como cadenas. Esto se debe a que las cadenas Java son inmutables y no se pueden "borrar" o "borrar a cero" manualmente en la memoria; mientras que una referencia a una cadena puede ser eliminada, la cadena exacta permanecerá en la memoria hasta que su segmento de memoria sea recolectado y reutilizado. Un atacante tendría una ventana grande en la que podría volcar la memoria del programa y encontrar fácilmente la clave. Por el contrario, las matrices de byte son mutables y sus contenidos pueden sobrescribirse en su lugar; es una buena idea 'poner a cero' sus llaves tan pronto como ya no las necesite.

Vulnerabilidades comunes de Java

Contenido necesita

Preocupaciones de redes

Contenido necesita

Aleatoriedad y tu

Contenido necesita

Para la mayoría de las aplicaciones, la clase java.utils.Random es una fuente perfectamente fina de datos "aleatorios". Si necesita elegir un elemento aleatorio de una matriz, o generar una

cadena aleatoria, o crear un identificador "único" temporal, probablemente debería usar Random .

Sin embargo, muchos sistemas criptográficos dependen de la aleatoriedad para su seguridad, y la aleatoriedad proporcionada por Random simplemente no es de la calidad suficiente. Para cualquier operación criptográfica que requiera una entrada aleatoria, debe usar SecureRandom en SecureRandom lugar.

Hashing y Validación

Se necesita más información.

Una función hash criptográfica es un miembro de una clase de funciones con tres propiedades vitales; Consistencia, singularidad e irreversibilidad.

Consistencia: dados los mismos datos, una función hash siempre devolverá el mismo valor. Es decir, si X = Y, f (x) siempre será igual a f (y) para la función hash f.

Unicidad: no hay dos entradas para una función hash que resultarán en la misma salida. Es decir, si X! = Y, f (x)! = F (y), para cualquier valor de X e Y.

Irreversibilidad: es imprácticamente difícil, si no imposible, "revertir" una función hash. Es decir, dado solo f (X), no debería haber manera de encontrar la X original antes de poner todos los valores posibles de X a través de la función f (fuerza bruta). No debería haber una función f1 tal que f1 (f (X)) = X.

Muchas funciones carecen de al menos uno de estos atributos. Por ejemplo, se sabe que MD5 y SHA1 tienen colisiones, es decir, dos entradas que tienen la misma salida, por lo que carecen de singularidad. Algunas funciones que actualmente se consideran seguras son SHA-256 y SHA-512.

Lea Seguridad y criptografía en línea: https://riptutorial.com/es/java/topic/9371/seguridad-y-criptografia

Capítulo 161: ServiceLoader

Observaciones

ServiceLoader se puede usar para obtener instancias de clases que amplían un tipo dado (= servicio) que se especifican en un archivo empaquetado en un archivo .jar . El servicio que se extiende / implementa a menudo es una interfaz, pero esto no es necesario.

Las clases de extensión / implementación deben proporcionar un constructor de argumento cero para que ServiceLoader ejemplifique.

Para que ServiceLoader descubra, debe ServiceLoader un archivo de texto con el nombre del nombre de tipo completo del servicio implementado dentro del directorio META-INF/services en el archivo jar. Este archivo contiene un nombre completo de una clase que implementa el servicio por línea.

Examples

Servicio de registrador

El siguiente ejemplo muestra cómo crear una instancia de una clase para el registro a través del ServiceLoader .

Servicio

```
package servicetest;

import java.io.IOException;

public interface Logger extends AutoCloseable {

    void log(String message) throws IOException;
}
```

Implementaciones del servicio.

La siguiente implementación simplemente escribe el mensaje en System.err

```
package servicetest.logger;

import servicetest.Logger;

public class ConsoleLogger implements Logger {

    @Override
    public void log(String message) {
        System.err.println(message);
    }

    @Override
    public void close() {
    }

}
```

La siguiente implementación escribe los mensajes en un archivo de texto:

```
package servicetest.logger;
```

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import servicetest.Logger;

public class FileLogger implements Logger {

    private final BufferedWriter writer;

    public FileLogger() throws IOException {
        writer = new BufferedWriter(new FileWriter("log.txt"));
    }

    @Override
    public void log(String message) throws IOException {
        writer.append(message);
        writer.newLine();
    }

    @Override
    public void close() throws IOException {
        writer.close();
    }

}
```

META-INF / services / servicetest.Logger

El archivo META-INF/services/servicetest.Logger enumera los nombres de las implementaciones de Logger .

```
servicetest.logger.ConsoleLogger
servicetest.logger.FileLogger
```

Uso

El siguiente método main escribe un mensaje a todos los registradores disponibles. Los registradores se ServiceLoader instancias utilizando ServiceLoader .

```
public static void main(String[] args) throws Exception {
    final String message = "Hello World!";

    // get ServiceLoader for Logger
    ServiceLoader<Logger> loader = ServiceLoader.load(servicetest.Logger.class);

    // iterate through instances of available loggers, writing the message to each one
    Iterator<Logger> iterator = loader.iterator();
    while (iterator.hasNext()) {
        try (Logger logger = iterator.next()) {
            logger.log(message);
        }
    }
}
```

Ejemplo simple de ServiceLoader

El ServiceLoader es un mecanismo integrado simple y fácil de usar para la carga dinámica de implementaciones de interfaz. Con el cargador de servicios, que proporciona medios para la

creación de instancias (pero no el cableado), se puede construir un mecanismo de inyección de dependencia simple en Java SE. Con la interfaz y la implementación de ServiceLoader, la separación se vuelve natural y los programas se pueden extender convenientemente. En realidad, una gran cantidad de API de Java están implementadas basadas en el ServiceLoader

Los conceptos básicos son

- Operando en *interfaces* de servicios
- Obtención de implementaciones del servicio a través de ServiceLoader
- Proporcionando implementación de servicios

Comencemos con la interfaz y la pongamos en un jar, llamado, por ejemplo, accounting-api.jar

```
package example;

public interface AccountingService {

  long getBalance();
}
```

Ahora proporcionamos una implementación de ese servicio en un jar llamado accounting-impl.jar , que contiene una implementación del servicio

```
package example.impl;
import example.AccountingService;

public interface DefaultAccountingService implements AccouningService {

  public long getBalance() {
    return balanceFromDB();
  }

  private long balanceFromDB(){
    ...
  }
}
```

Además, accounting-impl.jar contiene un archivo que declara que este contenedor proporciona una implementación de AccountingService . El archivo debe tener una ruta que comience con META-INF/services/ y debe tener el mismo nombre que el nombre *completo* de la interfaz:

- META-INF/services/example.AccountingService

El contenido del archivo es el nombre *completo* de la implementación:

```
example.impl.DefaultAccountingService
```

Dado que ambos archivos están en el classpath del programa, que consume el AccountingService , se puede obtener una instancia del Servicio utilizando el ServiceLauncher

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
AccountingService service = loader.next();
long balance = service.getBalance();
```

Como el ServiceLoader es un Iterable , admite múltiples proveedores de implementación, desde donde el programa puede elegir:

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
for(AccountingService service : loader) {
   //...
```

```
}
```

Tenga en cuenta que al invocar next() creará una nueva instancia. Si desea reutilizar una instancia, debe utilizar el método iterator iterator() del ServiceLoader o el bucle for-each como se muestra arriba.

Lea ServiceLoader en línea: https://riptutorial.com/es/java/topic/5433/serviceloader

Capítulo 162: Servicio de impresión de Java

Introducción

La API de Java Print Service proporciona funcionalidades para descubrir servicios de impresión y enviar solicitudes de impresión para ellos.

Incluye atributos de impresión extensibles basados en los atributos estándar especificados en el Protocolo de Impresión de Internet (IPP) 1.1 de la Especificación IETF, RFC 2911 .

Examples

Descubriendo los servicios de impresión disponibles.

Para descubrir todos los servicios de impresión disponibles, podemos usar la clase PrintServiceLookup . Veamos cómo:

```
import javax.print.PrintService;
import javax.print.PrintServiceLookup;

public class DiscoveringAvailablePrintServices {

    public static void main(String[] args) {
        discoverPrintServices();
    }

    public static void discoverPrintServices() {
        PrintService[] allPrintServices = PrintServiceLookup.lookupPrintServices(null, null);

        for (Printservice printService : allPrintServices) {
            System.out.println("Print service name: " + printService.getName());
        }
    }

}
```

Este programa, cuando se ejecuta en un entorno Windows, imprimirá algo como esto:

```
Print service name: Fax
Print service name: Microsoft Print to PDF
Print service name: Microsoft XPS Document Viewer
```

Descubriendo el servicio de impresión predeterminado

Para descubrir el servicio de impresión predeterminado, podemos usar la clase PrintServiceLookup . Veamos cómo ::

```
import javax.print.PrintService;
import javax.print.PrintServiceLookup;

public class DiscoveringDefaultPrintService {

    public static void main(String[] args) {
        discoverDefaultPrintService();
    }

    public static void discoverDefaultPrintService() {
        PrintService defaultPrintService = PrintServiceLookup.lookupDefaultPrintService();
```

```
        System.out.println("Default print service name: " + defaultPrintService.getName());
    }

}
```

Creación de un trabajo de impresión desde un servicio de impresión

Un trabajo de impresión es una solicitud de imprimir algo en un servicio de impresión específico. Consiste, básicamente, en:

- los datos que se imprimirán (consulte Creación del documento que se imprimirá)
- un conjunto de atributos

Después de seleccionar la instancia de servicio de impresión correcta, podemos solicitar la creación de un trabajo de impresión:

```
DocPrintJob printJob = printService.createPrintJob();
```

La interfaz DocPrintJob nos proporciona el método de print :

```
printJob.print(doc, pras);
```

El argumento doc es un Doc : los datos que se imprimirán.

Y el argumento pras es una interfaz PrintRequestAttributeSet : un conjunto de PrintRequestAttribute . Son ejemplos de atributos de solicitud de impresión:

- cantidad de copias (1, 2 etc),
- orientación (vertical u horizontal)
- cromacia (monocromo, color)
- Calidad (calado, normal, alta)
- lados (de una cara, de dos caras, etc.)
- y así...

El método de impresión puede lanzar una PrintException .

Construyendo el Doc que será impreso.

Doc es una interfaz y la API del servicio de impresión Java proporciona una implementación simple llamada SimpleDoc .

Cada instancia de Doc se compone básicamente de dos aspectos:

- el propio contenido de datos de impresión (un correo electrónico, una imagen, un documento, etc.)
- el formato de datos de impresión, denominado DocFlavor (tipo MIME + clase de representación).

Antes de crear el objeto Doc , debemos cargar nuestro documento desde algún lugar. En el ejemplo, cargaremos un archivo específico del disco:

```
FileInputStream pdfFileInputStream = new FileInputStream("something.pdf");
```

Entonces, ahora tenemos que elegir un DocFlavor que coincida con nuestro contenido. La clase DocFlavor tiene un montón de constantes para representar los tipos de datos más habituales. Vamos a elegir el INPUT_STREAM.PDF uno:

```
DocFlavor pdfDocFlavor = DocFlavor.INPUT_STREAM.PDF;
```

Ahora, podemos crear una nueva instancia de SimpleDoc :

```
Doc doc = new SimpleDoc(pdfFileInputStream, pdfDocFlavor , null);
```

El objeto doc ahora se puede enviar a la solicitud de trabajo de impresión (consulte Creación de un trabajo de impresión desde un servicio de impresión).

Definiendo atributos de solicitud de impresión

A veces necesitamos determinar algunos aspectos de la solicitud de impresión. Los llamaremos *atributo* .

Son ejemplos de atributos de solicitud de impresión:

- cantidad de copias (1, 2 etc),
- orientación (vertical u horizontal)
- cromacia (monocromo, color)
- Calidad (calado, normal, alta)
- lados (de una cara, de dos caras, etc.)
- y así...

Antes de elegir uno de ellos y qué valor tendrá cada uno, primero debemos crear un conjunto de atributos:

```
PrintRequestAttributeSet pras = new HashPrintRequestAttributeSet();
```

Ahora podemos añadirlos. Algunos ejemplos son:

```
pras.add(new Copies(5));
pras.add(MediaSize.ISO_A4);
pras.add(OrientationRequested.PORTRAIT);
pras.add(PrintQuality.NORMAL);
```

El objeto pras ahora se puede enviar a la solicitud de trabajo de impresión (consulte Creación de un trabajo de impresión desde un servicio de impresión).

Escuchar cambio de estado de solicitud de trabajo de impresión

Para la mayoría de los clientes de impresión, es extremadamente útil saber si un trabajo de impresión ha finalizado o ha fallado.

La API del servicio de impresión de Java proporciona algunas funcionalidades para informarse sobre estos escenarios. Todo lo que tenemos que hacer es:

- proporcionar una implementación para la interfaz PrintJobListener y
- Registre esta implementación en el trabajo de impresión.

Cuando cambie el estado del trabajo de impresión, se nos notificará. Podemos hacer lo que sea necesario, por ejemplo:

- actualizar una interfaz de usuario,
- iniciar otro proceso de negocio,
- grabar algo en la base de datos,
- o simplemente inicie sesión.

En el siguiente ejemplo, registraremos cada cambio de estado del trabajo de impresión:

```
import javax.print.event.PrintJobEvent;
```

```
import javax.print.event.PrintJobListener;

public class LoggerPrintJobListener implements PrintJobListener {

    // Your favorite Logger class goes here!
    private static final Logger LOG = Logger.getLogger(LoggerPrintJobListener.class);


    public void printDataTransferCompleted(PrintJobEvent pje) {
        LOG.info("Print data transfer completed ;) ");
    }

    public void printJobCompleted(PrintJobEvent pje) {
        LOG.info("Print job completed =) ");
    }

    public void printJobFailed(PrintJobEvent pje) {
        LOG.info("Print job failed =( ");
    }

    public void printJobCanceled(PrintJobEvent pje) {
        LOG.info("Print job canceled :| ");
    }

    public void printJobNoMoreEvents(PrintJobEvent pje) {
        LOG.info("No more events to the job ");
    }

    public void printJobRequiresAttention(PrintJobEvent pje) {
        LOG.info("Print job requires attention :O ");
    }
}
```

Finalmente, podemos agregar nuestra implementación de escucha de trabajo de impresión en el trabajo de impresión antes de la solicitud de impresión, de la siguiente manera:

```
DocPrintJob printJob = printService.createPrintJob();

printJob.addPrintJobListener(new LoggerPrintJobListener());

printJob.print(doc, pras);
```

El argumento *PrintJobEvent pje*

Observe que cada método tiene un argumento PrintJobEvent pje . No lo usamos en este ejemplo por razones de simplicidad, pero puede usarlo para explorar el estado. Por ejemplo:

```
pje.getPrintJob().getAttributes();
```

PrintJobAttributeSet una instancia de objeto PrintJobAttributeSet y puede ejecutarlos de una manera para cada uno.

Otra forma de lograr el mismo objetivo.

Otra opción para lograr el mismo objetivo es extender la clase PrintJobAdapter , como su nombre lo indica, es un adaptador para PrintJobListener . Implementando la interfaz obligatoriamente tenemos que implementar todos ellos. La ventaja de esta manera es que debemos anular solo los métodos que queremos. Vamos a ver cómo funciona:

```
import javax.print.event.PrintJobEvent;
import javax.print.event.PrintJobAdapter;

public class LoggerPrintJobAdapter extends PrintJobAdapter {

    // Your favorite Logger class goes here!
    private static final Logger LOG = Logger.getLogger(LoggerPrintJobAdapter.class);

    public void printJobCompleted(PrintJobEvent pje) {
        LOG.info("Print job completed =) ");
    }

    public void printJobFailed(PrintJobEvent pje) {
        LOG.info("Print job failed =( ");
    }
}
```

Tenga en cuenta que solo anulamos algunos métodos específicos.

De la misma manera en el ejemplo que implementa la interfaz PrintJobListener , agregamos el oyente al trabajo de impresión antes de enviarlo a imprimir:

```
printJob.addPrintJobListener(new LoggerPrintJobAdapter());

printJob.print(doc, pras);
```

Lea Servicio de impresión de Java en línea:
https://riptutorial.com/es/java/topic/10178/servicio-de-impresion-de-java

Capítulo 163: Singletons

Introducción

Un singleton es una clase que solo tiene una sola instancia. Para obtener más información sobre el *patrón de diseño* Singleton, consulte el tema Singleton en la etiqueta Patrones de diseño .

Examples

Enum Singleton

Java SE 5

```
public enum Singleton {
    INSTANCE;

    public void execute (String arg) {
        // Perform operation here
    }
}
```

Las enumeraciones tienen constructores privados, son definitivas y proporcionan la maquinaria de serialización adecuada. También son muy concisos y están perezosamente iniciados de manera segura para los hilos.

La JVM ofrece una garantía de que los valores de enumeración no se instanciarán más de una vez, lo que otorga al patrón de enumeración de enumeración una defensa muy fuerte contra los ataques de reflexión.

Lo *que no* protege contra el patrón de enumeración es que otros desarrolladores agreguen físicamente más elementos al código fuente. Por consiguiente, si elige este estilo de implementación para sus singletons, es imperativo que documente claramente que no se deben agregar nuevos valores a esas enumeraciones.

Esta es la forma recomendada de implementar el patrón de singleton, como lo explica Joshua Bloch en Effective Java.

Hilo seguro Singleton con doble control de bloqueo

Este tipo de Singleton es seguro para subprocesos y evita el bloqueo innecesario después de que se haya creado la instancia de Singleton.

Java SE 5

```
public class MySingleton {

    // instance of class
    private static volatile MySingleton instance = null;

    // Private constructor
    private MySingleton() {
        // Some code for constructing object
    }

    public static MySingleton getInstance() {
        MySingleton result = instance;

        //If the instance already exists, no locking is necessary
        if(result == null) {
```

```
            //The singleton instance doesn't exist, lock and check again
            synchronized(MySingleton.class) {
                result = instance;
                if(result == null) {
                    instance = result = new MySingleton();
                }
            }
        }
        return result;
    }
}
```

Se debe enfatizar: en las versiones anteriores a Java SE 5, la implementación anterior es incorrecta y debe evitarse. No es posible implementar el bloqueo de doble comprobación correctamente en Java antes de Java 5.

Singleton sin uso de Enum (inicialización impaciente)

```
public class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Se puede argumentar que este ejemplo es *efectivamente* una inicialización perezosa. La sección 12.4.1 de la especificación del lenguaje Java establece:

> Una clase o tipo de interfaz T se inicializará inmediatamente antes de la primera aparición de cualquiera de los siguientes:
>
> - T es una clase y se crea una instancia de T
> - T es una clase y se invoca un método estático declarado por T
> - Se asigna un campo estático declarado por T
> - Se usa un campo estático declarado por T y el campo no es una variable constante
> - T es una clase de nivel superior y se ejecuta una declaración de afirmación anidada léxicamente dentro de T.

Por lo tanto, mientras no haya otros campos estáticos o métodos estáticos en la clase, la instancia de Singleton no se inicializará hasta que se invoque el método getInstance() primera vez.

Inicialización perezosa segura para subprocesos utilizando la clase de soporte | Implementación de Bill Pugh Singleton

```
public class Singleton {
    private static class InstanceHolder {
        static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return InstanceHolder.INSTANCE;
    }

    private Singleton() {}
}
```

Esto inicializa la variable INSTANCE en la primera llamada a Singleton.getInstance() , aprovechando las garantías de seguridad de subprocesos del idioma para la inicialización estática sin necesidad de sincronización adicional.

Esta implementación también se conoce como patrón singleton de Bill Pugh. [Wiki]

Extendiendo singleton (herencia singleton)

En este ejemplo, la clase base Singleton proporciona el método getMessage() que devuelve "Hello world!" mensaje.

Es subclases UppercaseSingleton y LowercaseSingleton reemplazar el método getMessage () para proporcionar una representación apropiada del mensaje.

```
//Yeah, we'll need reflection to pull this off.
import java.lang.reflect.*;

/*
Enumeration that represents possible classes of singleton instance.
If unknown, we'll go with base class - Singleton.
*/
enum SingletonKind {
    UNKNOWN,
    LOWERCASE,
    UPPERCASE
}

//Base class
class Singleton{

    /*
    Extended classes has to be private inner classes, to prevent extending them in
    uncontrolled manner.
     */
    private class UppercaseSingleton extends Singleton {

        private UppercaseSingleton(){
            super();
        }

        @Override
        public String getMessage() {
            return super.getMessage().toUpperCase();
        }
    }

    //Another extended class.
    private class LowercaseSingleton extends Singleton
    {
        private LowercaseSingleton(){
            super();
        }

        @Override
        public String getMessage() {
            return super.getMessage().toLowerCase();
        }
    }

    //Applying Singleton pattern
    private static SingletonKind kind = SingletonKind.UNKNOWN;
```

```
    private static Singleton instance;

    /*
    By using this method prior to getInstance() method, you effectively change the
    type of singleton instance to be created.
     */
    public static void setKind(SingletonKind kind) {
        Singleton.kind = kind;
    }

    /*
    If needed, getInstance() creates instance appropriate class, based on value of
    singletonKind field.
     */
    public static Singleton getInstance()
        throws  NoSuchMethodException,
                IllegalAccessException,
                InvocationTargetException,
                InstantiationException {

        if(instance==null){
            synchronized (Singleton.class){
                if(instance==null){
                    Singleton singleton = new Singleton();
                    switch (kind){
                        case UNKNOWN:

                            instance = singleton;
                            break;

                        case LOWERCASE:

                            /*
                             I can't use simple

                             instance = new LowercaseSingleton();

                             because java compiler won't allow me to use
                             constructor of inner class in static context,
                             so I use reflection API instead.

                             To be able to access inner class by reflection API,
                             I have to create instance of outer class first.
                             Therefore, in this implementation, Singleton cannot be
                             abstract class.
                             */

                            //Get the constructor of inner class.
                            Constructor<LowercaseSingleton> lcConstructor =

LowercaseSingleton.class.getDeclaredConstructor(Singleton.class);

                            //The constructor is private, so I have to make it accessible.
                            lcConstructor.setAccessible(true);

                            // Use the constructor to create instance.
                            instance = lcConstructor.newInstance(singleton);

                            break;
```

```
                        case UPPERCASE:

                            //Same goes here, just with different type
                            Constructor<UppercaseSingleton> ucConstructor =

UppercaseSingleton.class.getDeclaredConstructor(Singleton.class);
                            ucConstructor.setAccessible(true);
                            instance = ucConstructor.newInstance(singleton);
                        }
                    }
                }
            }
        return instance;
    }

    //Singletons state that is to be used by subclasses
    protected String message;

    //Private constructor prevents external instantiation.
    private Singleton()
    {
        message = "Hello world!";
    }

    //Singleton's API. Implementation can be overwritten by subclasses.
    public String getMessage() {
        return message;
    }
}

//Just a small test program
public class ExtendingSingletonExample {

    public static void main(String args[]){

        //just uncomment one of following lines to change singleton class

        //Singleton.setKind(SingletonKind.UPPERCASE);
        //Singleton.setKind(SingletonKind.LOWERCASE);

        Singleton singleton = null;
        try {
            singleton = Singleton.getInstance();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        }
        System.out.println(singleton.getMessage());
    }
}
```

Lea Singletons en línea: https://riptutorial.com/es/java/topic/130/singletons

Capítulo 164: Sockets de Java

Introducción

Los sockets son una interfaz de red de bajo nivel que ayuda a crear una conexión entre dos programas, principalmente clientes que pueden estar o no ejecutándose en la misma máquina.

La programación de sockets es uno de los conceptos de redes más utilizados.

Observaciones

Hay dos tipos de tráfico de protocolo de Internet:
1. TCP - Protocolo de control de transmisión 2. UDP - Protocolo de datagramas de usuario

TCP es un protocolo orientado a la conexión.
UDP es un protocolo sin conexión.

TCP es adecuado para aplicaciones que requieren alta confiabilidad, y el tiempo de transmisión es relativamente menos crítico.

UDP es adecuado para aplicaciones que necesitan una transmisión rápida y eficiente, como los juegos. La naturaleza sin estado de UDP también es útil para servidores que responden a pequeñas consultas de un gran número de clientes.

En palabras más simples:
Use TCP cuando no pueda permitirse perder datos y cuando el tiempo para enviar y recibir datos no importe. Use UDP cuando no pueda perder tiempo y cuando la pérdida de datos no importe.

Existe una garantía absoluta de que los datos transferidos permanecen intactos y llegan en el mismo orden en que se enviaron en caso de TCP.
mientras que no hay garantía de que los mensajes o paquetes enviados alcancen en absoluto en UDP.

Examples

Un simple servidor TCP con respaldo

Nuestro servidor TCP echo back será un hilo separado. Es simple como es un comienzo. Solo devolverá lo que sea que envíe, pero en mayúscula.

```
public class CAPECHOServer extends Thread{

    // This class implements server sockets. A server socket waits for requests to come
    // in over the network only when it is allowed through the local firewall
    ServerSocket serverSocket;

    public CAPECHOServer(int port, int timeout){
        try {
            // Create a new Server on specified port.
            serverSocket = new ServerSocket(port);
            // SoTimeout is basiacally the socket timeout.
            // timeout is the time until socket timeout in milliseconds
            serverSocket.setSoTimeout(timeout);
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Override
    public void run(){
```

```
        try {
            // We want the server to continuously accept connections
            while(!Thread.interrupted()){

            }
            // Close the server once done.
            serverSocket.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

}
```

Ahora a aceptar conexiones. Vamos a actualizar el método de ejecución.

```
@Override
public void run(){
    while(!Thread.interrupted()){
        try {
            // Log with the port number and machine ip
            Logger.getLogger((this.getClass().getName())).log(Level.INFO, "Listening for
Clients at {0} on {1}", new Object[]{serverSocket.getLocalPort(),
InetAddress.getLocalHost().getHostAddress()});
            Socket client = serverSocket.accept();  // Accept client conncetion
            // Now get DataInputStream and DataOutputStreams
            DataInputStream istream = new DataInputStream(client.getInputStream()); // From
client's input stream
            DataOutputStream ostream = new DataOutputStream(client.getOutputStream());
            // Important Note
            /*
                The server's input is the client's output
                The client's input is the server's output
            */
            // Send a welcome message
            ostream.writeUTF("Welcome!");

            // Close the connection
            istream.close();
            ostream.close();
            client.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    // Close the server once done

    try {
        serverSocket.close();
    } catch (IOException ex) {
        Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Ahora, si puede abrir telnet e intentar conectarse, verá un mensaje de bienvenida.

Debe conectarse con el puerto que especificó y la dirección IP.

Deberías ver un resultado similar a este:

```
Welcome!

Connection to host lost.
```

Bueno, la conexión se perdió porque la terminamos. A veces tendríamos que programar nuestro propio cliente TCP. En este caso, necesitamos que un cliente solicite información del usuario y la envíe a través de la red para recibir la información en mayúsculas.

Si el servidor envía los datos primero, el cliente debe leer los datos primero.

```
public class CAPECHOClient extends Thread{

Socket server;
Scanner key; // Scanner for input

    public CAPECHOClient(String ip, int port){
        try {
            server = new Socket(ip, port);
            key = new Scanner(System.in);
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Override
    public void run(){
        DataInputStream istream = null;
        DataOutputStream ostream = null;
        try {
            istream = new DataInputStream(server.getInputStream()); // Familiar lines
            ostream = new DataOutputStream(server.getOutputStream());
            System.out.println(istream.readUTF());  // Print what the server sends
            System.out.print(">");
            String tosend = key.nextLine();
            ostream.writeUTF(tosend);   // Send whatever the user typed to the server
            System.out.println(istream.readUTF());  // Finally read what the server sends
before exiting.
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
        } finally {
            try {
                istream.close();
                ostream.close();
                server.close();
            } catch (IOException ex) {
                Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

Ahora actualiza el servidor

```
ostream.writeUTF("Welcome!");

String inString = istream.readUTF();    // Read what the user sent
String outString = inString.toUpperCase();  // Change it to caps
ostream.writeUTF(outString);

// Close the connection
istream.close();
```

Y ahora ejecuta el servidor y el cliente, deberías tener una salida similar a esta

```
Welcome!
>
```

Lea Sockets de Java en línea: https://riptutorial.com/es/java/topic/9923/sockets-de-java

Capítulo 165: SortedMap

Introducción

Introducción al mapa ordenado.

Examples

Introducción al mapa ordenado.

Punto clave :-

- La interfaz SortedMap extiende el mapa.
- las entradas se mantienen en orden ascendente de claves.

Métodos de mapa ordenados:

- Comparador comparador ().
- Objeto firstKey ().
- SortedMap headMap (objeto final).
- Objeto lastKey ().
- SortedMap subMap (inicio de objeto, final de objeto).
- SortedMap tailMap (inicio de objeto).

Ejemplo

```
public static void main(String args[]) {
      // Create a hash map
      TreeMap tm = new TreeMap();

      // Put elements to the map
      tm.put("Zara", new Double(3434.34));
      tm.put("Mahnaz", new Double(123.22));
      tm.put("Ayan", new Double(1378.00));
      tm.put("Daisy", new Double(99.22));
      tm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = tm.entrySet();

      // Get an iterator
      Iterator i = set.iterator();

      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();

      // Deposit 1000 into Zara's account
      double balance = ((Double)tm.get("Zara")).doubleValue();
      tm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " + tm.get("Zara"));
   }
```

Lea SortedMap en línea: https://riptutorial.com/es/java/topic/10748/sortedmap

Capítulo 166: StringBuffer

Introducción

Introducción a la clase Java StringBuffer.

Examples

Clase de búfer de cadena

Puntos clave :-

- utilizado para crear una cadena mutable (modificable).
- **Mutable** : - Que se puede cambiar.
- es seguro para subprocesos, es decir, múltiples subprocesos no pueden acceder a él simultáneamente.

Métodos: -

- apéndice StringBuffer sincronizado al público (String s)
- Inserción StringBuffer sincronizada con el público (desplazamiento int, String s)
- Reemplazo de StringBuffer sincronizado al público (int startIndex, int endIndex, String str)
- Eliminar StringBuffer sincronizado público (int startIndex, int endIndex)
- StringBuffer invertido público sincronizado ()
- capacidad pública int ()
- vacío público garantizarCapacidad (int mínimoCapacidad)
- char char pública (índice int)
- public int length ()
- Subcadena de cadena pública (int beginIndex)
- subcadena de cadena pública (int beginIndex, int endIndex)

Ejemplo que muestra la diferencia entre la implementación de String y String Buffer: -

```
class Test {
 public static void main(String args[])
 {
  String str = "study";
  str.concat("tonight");
  System.out.println(str);      // Output: study

  StringBuffer strB = new StringBuffer("study");
  strB.append("tonight");
  System.out.println(strB);    // Output: studytonight
 }
}
```

Lea StringBuffer en línea: https://riptutorial.com/es/java/topic/10757/stringbuffer

Capítulo 167: StringBuilder

Introducción

La clase StringBuilder de Java se utiliza para crear una cadena mutable (modificable). La clase StringBuilder de Java es la misma que la clase StringBuffer, excepto que no está sincronizada. Está disponible desde JDK 1.5.

Sintaxis

- nuevo StringBuilder ()
- nuevo StringBuilder (capacidad int)
- nuevo StringBuilder (CharSequence seq)
- Nuevo StringBuilder (constructor StringBuilder)
- nuevo StringBuilder (cadena de cadena)
- Nuevo StringJoiner (delimitador de CharSequence)
- Nuevo StringJoiner (delimitador de CharSequence, prefijo de CharSequence, sufijo de CharSequence)

Observaciones

La creación de un nuevo StringBuilder con tipo char como parámetro daría como resultado que se llame al constructor con int capacity argumento y no a la String string argumento:

```
StringBuilder v = new StringBuilder('I'); //'I' is a character, "I" is a String.
System.out.println(v.capacity()); --> output 73
System.out.println(v.toString()); --> output nothing
```

Examples

Repetir una cadena n veces

Problema: Cree una String contenga n repeticiones de una String s .

El enfoque trivial consistiría repetidamente en concatenar la String

```
final int n = ...
final String s = ...
String result = "";

for (int i = 0; i < n; i++) {
    result += s;
}
```

Esto crea n nuevas instancias de cadena que contienen de 1 a n repeticiones de s resultan en un tiempo de ejecución de O(s.length() * n²) = O(s.length() * (1+2+...+(n-1)+n)) .

Para evitar esto, debe utilizarse StringBuilder , que permite crear la String en O(s.length() * n) lugar:

```
final int n = ...
final String s = ...

StringBuilder builder = new StringBuilder();

for (int i = 0; i < n; i++) {
    builder.append(s);
}

String result = builder.toString();
```

Comparando StringBuffer, StringBuilder, Formatter y StringJoiner

Las clases StringBuffer , StringBuilder , Formatter y StringJoiner son clases de utilidad de Java SE que se usan principalmente para ensamblar cadenas a partir de otra información:

- La clase StringBuffer ha estado presente desde Java 1.0 y proporciona una variedad de métodos para construir y modificar un "búfer" que contiene una secuencia de caracteres.
- La clase StringBuilder se agregó en Java 5 para abordar los problemas de rendimiento con la clase StringBuffer original. Las API para las dos clases son esencialmente las mismas. La principal diferencia entre StringBuffer y StringBuilder es que el primero es seguro para subprocesos y está sincronizado y el segundo no.

Este ejemplo muestra cómo se puede usar StringBuilder :

```
int one = 1;
String color = "red";
StringBuilder sb = new StringBuilder();
sb.append("One=").append(one).append(", Color=").append(color).append('\n');
System.out.print(sb);
// Prints "One=1, Colour=red" followed by an ASCII newline.
```

(La clase StringBuffer se usa de la misma manera: simplemente cambie StringBuilder a StringBuffer en la parte superior)

Las clases StringBuffer y StringBuilder son adecuadas tanto para ensamblar como para modificar cadenas; es decir, proporcionan métodos para reemplazar y eliminar caracteres, así como agregarlos en varios. Las dos clases de remisión son específicas a la tarea de ensamblar cadenas.

- La clase Formatter se agregó en Java 5 y está modelada de manera flexible en la función sprintf en la biblioteca estándar de C. Toma una cadena de *formato* con *especificadores de formato* incrustados y una secuencia de otros argumentos, y genera una cadena convirtiendo los argumentos en texto y sustituyéndolos en lugar de los especificadores de formato. Los detalles de los especificadores de formato dicen cómo los argumentos se convierten en texto.
- La clase StringJoiner se agregó en Java 8. Es un formateador de propósito especial que formatea sucintamente una secuencia de cadenas con separadores entre ellas. Está diseñado con una API *fluida* y puede usarse con flujos de Java 8.

Aquí hay algunos ejemplos típicos de uso del Formatter :

```
// This does the same thing as the StringBuilder example above
int one = 1;
String color = "red";
Formatter f = new Formatter();
System.out.print(f.format("One=%d, colour=%s%n", one, color));
// Prints "One=1, Colour=red" followed by the platform's line separator
```

```
// The same thing using the `String.format` convenience method
System.out.print(String.format("One=%d, color=%s%n", one, color));
```

La clase StringJoiner no es ideal para la tarea anterior, por lo que aquí hay un ejemplo de StringJoiner formatear una matriz de cadenas.

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
for (String s : new String[]{"A", "B", "C"}) {
    sj.add(s);
}
System.out.println(sj);
// Prints "[A, B, C]"
```

Los casos de uso para las 4 clases se pueden resumir:

- StringBuilder adecuado para cualquier montaje de cadena O tarea de modificación de cadena.
- Utilice StringBuffer (solo) cuando necesite una versión segura de subprocesos de StringBuilder .
- Formatter proporciona una funcionalidad de formato de cadena mucho más rica, pero no es tan eficiente como StringBuilder . Esto se debe a que cada llamada a Formatter.format(...) implica:
    - analizando la cadena de format ,
    - creando y rellenando una matriz de *varargs* , y
    - Autoboxing cualquier tipo de argumentos primitivos.
- StringJoiner proporciona un formateo StringJoiner y eficiente de una secuencia de cadenas con separadores, pero no es adecuado para otras tareas de formateo.

Lea StringBuilder en línea: https://riptutorial.com/es/java/topic/1037/stringbuilder

Capítulo 168: sun.misc.Unsafe

Observaciones

La clase Unsafe permite que un programa haga cosas que no están permitidas por el compilador de Java. Los programas normales deben evitar el uso Unsafe .

ADVERTENCIAS

1. Si comete un error al utilizar las API no Unsafe , es probable que sus aplicaciones causen que la JVM se bloquee y / o muestre síntomas que son difíciles de diagnosticar.
2. La API Unsafe está sujeta a cambios sin previo aviso. Si lo usa en su código, es posible que deba volver a escribir el código al cambiar las versiones de Java.

Examples

Instalar sun.misc.Unsafe a través de la reflexión

```
public static Unsafe getUnsafe() {
    try {
        Field unsafe = Unsafe.class.getDeclaredField("theUnsafe");
        unsafe.setAccessible(true);
        return (Unsafe) unsafe.get(null);
    } catch (IllegalAccessException e) {
        // Handle
    } catch (IllegalArgumentException e) {
        // Handle
    } catch (NoSuchFieldException e) {
        // Handle
    } catch (SecurityException e) {
        // Handle
    }
}
```

sun.misc.Unsafe tiene un constructor privado, y el getUnsafe() estático getUnsafe() con una comprobación del cargador de clases para garantizar que el código se haya cargado con el cargador de clases principal. Por lo tanto, un método para cargar la instancia es usar la reflexión para obtener el campo estático.

Creación de una instancia de sun.misc.Unsafe a través de bootclasspath

```
public class UnsafeLoader {
    public static Unsafe loadUnsafe() {
        return Unsafe.getUnsafe();
    }
}
```

Si bien este ejemplo se compilará, es probable que falle en tiempo de ejecución a menos que la clase Insegura se haya cargado con el cargador de clases primario. Para asegurarse de que esto suceda, la JVM debe cargarse con los argumentos apropiados, como:

```
java -Xbootclasspath:$JAVA_HOME/jre/lib/rt.jar:./UnsafeLoader.jar foo.bar.MyApp
```

La clase foo.bar.MyApp puede usar UnsafeLoader.loadUnsafe() .

Obtención de instancia de inseguro

Inseguro se almacena como un campo privado al que no se puede acceder directamente. El constructor es privado y el único método para acceder a public static Unsafe getUnsafe() tiene acceso privilegiado. Mediante el uso de la reflexión, hay una solución alternativa para que los campos privados sean accesibles:

```
public static final Unsafe UNSAFE;

static {
    Unsafe unsafe = null;

    try {
        final PrivilegedExceptionAction<Unsafe> action = () -> {
            final Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);

            return (Unsafe) f.get(null);
        };

        unsafe = AccessController.doPrivileged(action);
    } catch (final Throwable t) {
        throw new RuntimeException("Exception accessing Unsafe", t);
    }

    UNSAFE = unsafe;
}
```

Usos de inseguros

Algunos usos de inseguros son los siguientes:

Utilizar	API
Asignación, reasignación y desasignación fuera de almacenamiento / memoria directa	`allocateMemory(bytes)` , `reallocateMemory(address, bytes)` y `freeMemory(address)`
Vallas de memoria	`loadFence()` , `storeFence()` , `fullFence()`
Hilo de corriente de estacionamiento	`park(isAbsolute, time)` , `unpark(thread)`
Campo directo y / o acceso a memoria.	`get*` y `put*` familia de métodos.
Lanzar excepciones sin marcar	`throwException(e)`
CAS y operaciones atómicas	Familia de métodos `compareAndSwap*`
Configuración de la memoria	`setMemory`
Operaciones volátiles o concurrentes.	`get*Volatile` , `put*Volatile` , `putOrdered*`

La familia de métodos get y put son relativos a un objeto dado. Si el objeto es nulo, se trata como una dirección absoluta.

```
// Putting a value to a field
protected static long fieldOffset = UNSAFE.objectFieldOffset(getClass().getField("theField"));
```

```
UNSAFE.putLong(this, fieldOffset , newValue);

// Puting an absolute value
 UNSAFE.putLong(null, address, newValue);
 UNSAFE.putLong(address, newValue);
```

Algunos métodos solo se definen para int y longs. Puedes usar estos métodos en flotantes y dobles usando floatToRawIntBits , intBitsToFloat, doubleToRawLongBits , longBitsToDouble`

Lea sun.misc.Unsafe en línea: https://riptutorial.com/es/java/topic/6771/sun-misc-unsafe

Capítulo 169: súper palabra clave

Examples

Uso de palabras clave super con ejemplos

super palabra clave desempeña un papel importante en tres lugares

1. Nivel de constructor
2. Nivel de método
3. Nivel variable

Nivel de constructor

super palabra clave super se utiliza para llamar al constructor de la clase padre. Este constructor puede ser constructor por defecto o constructor parametrizado.

- Constructor por defecto: super();
- Constructor parametrizado: super(int no, double amount, String name);

```
class Parentclass
{
   Parentclass(){
      System.out.println("Constructor of Superclass");
   }
}
class Subclass extends Parentclass
{
   Subclass(){
    /* Compile adds super() here at the first line
     * of this constructor implicitly
     */
    System.out.println("Constructor of Subclass");
   }
   Subclass(int n1){
    /* Compile adds super() here at the first line
     * of this constructor implicitly
     */
    System.out.println("Constructor with arg");
   }
   void display(){
    System.out.println("Hello");
   }
   public static void main(String args[]){
    // Creating object using default constructor
    Subclass obj= new Subclass();
    //Calling sub class method
       obj.display();
       //Creating object 2 using arg constructor
       Subclass obj2= new Subclass(10);
       obj2.display();
  }
}
```

Nota : super() debe ser la primera declaración en el constructor, de lo contrario obtendremos el mensaje de error de compilación.

Nivel de método

super palabra clave super también se puede utilizar en caso de anulación del método. super palabra clave super se puede utilizar para invocar o llamar al método de clase padre.

```
class Parentclass
{
   //Overridden method
   void display(){
    System.out.println("Parent class method");
   }
}
class Subclass extends Parentclass
{
   //Overriding method
   void display(){
    System.out.println("Child class method");
   }
   void printMsg(){
    //This would call Overriding method
    display();
    //This would call Overridden method
    super.display();
   }
   public static void main(String args[]){
    Subclass obj= new Subclass();
    obj.printMsg();
   }
}
```

Nota : Si no hay una modificación de método, entonces no necesitamos usar la palabra clave super para llamar al método de clase principal.

Nivel variable

super se utiliza para referirse a la variable de instancia de clase primaria inmediata. En caso de herencia, puede haber posibilidad de que la clase base y la clase derivada puedan tener miembros de datos similares. Para diferenciar entre el miembro de datos de la clase base / padre y la clase derivada / hijo, en el contexto de la clase derivada, los datos de la clase base los miembros deben estar precedidos por una super palabra clave.

```
//Parent class or Superclass
class Parentclass
{
    int num=100;
}
//Child class or subclass
class Subclass extends Parentclass
{
    /* I am declaring the same variable
     * num in child class too.
     */
    int num=110;
    void printNumber(){
     System.out.println(num); //It will print value 110
     System.out.println(super.num); //It will print value 100
    }
    public static void main(String args[]){
       Subclass obj= new Subclass();
       obj.printNumber();
```

```
    }
}
```

Nota : Si no escribimos una super palabra clave antes del nombre del miembro de datos de la clase base, se hará referencia como miembro de datos de la clase actual y el miembro de datos de la clase base se ocultará en el contexto de la clase derivada.

Lea súper palabra clave en línea: https://riptutorial.com/es/java/topic/5764/super-palabra-clave

Capítulo 170: Tabla de picadillo

Introducción

Hashtable es una clase en las colecciones Java que implementa la interfaz Map y extiende la clase de diccionario.

Contiene solo elementos únicos y su sincronizado.

Examples

Tabla de picadillo

```
import java.util.*;
public class HashtableDemo {
   public static void main(String args[]) {
   // create and populate hash table
   Hashtable<Integer, String> map = new Hashtable<Integer, String>();
   map.put(101,"C Language");
   map.put(102, "Domain");
   map.put(104, "Databases");
   System.out.println("Values before remove: "+ map);
   // Remove value for key 102
   map.remove(102);
   System.out.println("Values after remove: "+ map);
   }
}
```

Lea Tabla de picadillo en línea: https://riptutorial.com/es/java/topic/10709/tabla-de-picadillo

Capítulo 171: ThreadLocal

Observaciones

Se utiliza mejor para objetos que dependen de elementos internos durante la invocación de una llamada, pero de lo contrario no tienen estado, como SimpleDateFormat , Marshaller

Para el uso Random ThreadLocal, considere usar ThreadLocalRandom

Examples

ThreadLocal Java 8 inicialización funcional

```
public static class ThreadLocalExample
{
    private static final ThreadLocal<SimpleDateFormat> format =
        ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyyMMdd_HHmm"));

    public String formatDate(Date date)
    {
        return format.get().format(date);
    }
}
```

Uso básico de ThreadLocal

Java ThreadLocal se utiliza para crear variables locales de subprocesos. Se sabe que los subprocesos de un Objeto comparten sus variables, por lo que la variable no es segura para subprocesos. Podemos usar la sincronización para la seguridad de los subprocesos, pero si queremos evitar la sincronización, ThreadLocal nos permite crear variables que son locales para el subproceso, es decir, solo ese subproceso puede leer o escribir en esas variables, por lo que los otros subprocesos ejecutan el mismo fragmento de código. no podrá acceder a las variables ThreadLocal de los demás.

Esto puede ser usado, podemos usar variables ThreadLocal . en situaciones en las que tiene un grupo de subprocesos como, por ejemplo, en un servicio web. Por ejemplo, la creación de un objeto SimpleDateFormat cada vez que se realiza cada solicitud requiere mucho tiempo y no se puede crear uno estático, ya que SimpleDateFormat no es seguro para subprocesos, por lo que podemos crear un ThreadLocal para que podamos realizar operaciones seguras para subprocesos sin la sobrecarga de crear SimpleDateFormat cada hora.

La siguiente pieza de código muestra cómo se puede usar:

Cada hilo tiene su propia variable ThreadLocal y pueden usar los métodos get() y set() para obtener el valor predeterminado o cambiar su valor local a Thread.

ThreadLocal instancias de ThreadLocal suelen ser campos estáticos privados en clases que desean asociar el estado con un hilo.

Aquí hay un pequeño ejemplo que muestra el uso de ThreadLocal en el programa java y demuestra que cada hilo tiene su propia copia de la variable ThreadLocal .

```
package com.examples.threads;

import java.text.SimpleDateFormat;
import java.util.Random;

public class ThreadLocalExample implements Runnable{
```

```
    // SimpleDateFormat is not thread-safe, so give one to each thread
 // SimpleDateFormat is not thread-safe, so give one to each thread
    private static final ThreadLocal<SimpleDateFormat> formatter = new
ThreadLocal<SimpleDateFormat>(){
        @Override
        protected SimpleDateFormat initialValue()
        {
            return new SimpleDateFormat("yyyyMMdd HHmm");
        }
    };

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalExample obj = new ThreadLocalExample();
        for(int i=0 ; i<10; i++){
            Thread t = new Thread(obj, ""+i);
            Thread.sleep(new Random().nextInt(1000));
            t.start();
        }
    }

    @Override
    public void run() {
        System.out.println("Thread Name= "+Thread.currentThread().getName()+" default
Formatter = "+formatter.get().toPattern());
        try {
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        formatter.set(new SimpleDateFormat());

        System.out.println("Thread Name= "+Thread.currentThread().getName()+" formatter =
"+formatter.get().toPattern());
    }

}
```

Salida:

```
Thread Name= 0 default Formatter = yyyyMMdd HHmm

Thread Name= 1 default Formatter = yyyyMMdd HHmm

Thread Name= 0 formatter = M/d/yy h:mm a

Thread Name= 2 default Formatter = yyyyMMdd HHmm

Thread Name= 1 formatter = M/d/yy h:mm a

Thread Name= 3 default Formatter = yyyyMMdd HHmm

Thread Name= 4 default Formatter = yyyyMMdd HHmm

Thread Name= 4 formatter = M/d/yy h:mm a

Thread Name= 5 default Formatter = yyyyMMdd HHmm

Thread Name= 2 formatter = M/d/yy h:mm a

Thread Name= 3 formatter = M/d/yy h:mm a

Thread Name= 6 default Formatter = yyyyMMdd HHmm

Thread Name= 5 formatter = M/d/yy h:mm a
```

Thread Name= 6 formatter = M/d/yy h:mm a

Thread Name= 7 default Formatter = yyyyMMdd HHmm

Thread Name= 8 default Formatter = yyyyMMdd HHmm

Thread Name= 8 formatter = M/d/yy h:mm a

Thread Name= 7 formatter = M/d/yy h:mm a

Thread Name= 9 default Formatter = yyyyMMdd HHmm

Thread Name= 9 formatter = M/d/yy h:mm a

Como podemos ver en la salida, Thread-0 ha cambiado el valor de formateador pero aún así el formateador predeterminado de thread-2 es el mismo que el valor inicializado.

Múltiples hilos con un objeto compartido.

En este ejemplo, solo tenemos un objeto, pero se comparte entre / ejecuta en diferentes subprocesos. El uso normal de los campos para guardar el estado no sería posible porque el otro hilo también lo vería (o probablemente no lo vería).

```
public class Test {
    public static void main(String[] args) {
        Foo foo = new Foo();
        new Thread(foo, "Thread 1").start();
        new Thread(foo, "Thread 2").start();
    }
}
```

En Foo contamos desde cero. En lugar de guardar el estado en un campo, almacenamos nuestro número actual en el objeto ThreadLocal, al que se puede acceder de forma estática. Tenga en cuenta que la sincronización en este ejemplo no está relacionada con el uso de ThreadLocal, sino que garantiza una mejor salida de la consola.

```
public class Foo implements Runnable {
    private static final int ITERATIONS = 10;
    private static final ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>() {
        @Override
        protected Integer initialValue() {
            return 0;
        }
    };

    @Override
    public void run() {
        for (int i = 0; i < ITERATIONS; i++) {
            synchronized (threadLocal) {
                //Although accessing a static field, we get our own (previously saved) value.
                int value = threadLocal.get();
                System.out.println(Thread.currentThread().getName() + ": " + value);

                //Update our own variable
                threadLocal.set(value + 1);

                try {
                    threadLocal.notifyAll();
                    if (i < ITERATIONS - 1) {
                        threadLocal.wait();
                    }
                } catch (InterruptedException ex) {
```

```
                    }
                }
            }
        }
    }
```

Desde la salida podemos ver que cada hilo cuenta por sí mismo y no usa el valor del otro:

```
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
Thread 1: 5
Thread 2: 5
Thread 1: 6
Thread 2: 6
Thread 1: 7
Thread 2: 7
Thread 1: 8
Thread 2: 8
Thread 1: 9
Thread 2: 9
```

Lea ThreadLocal en línea: https://riptutorial.com/es/java/topic/2001/threadlocal

Capítulo 172: Tipo de conversión

Sintaxis

- `TargetType target = (SourceType) fuente;`

Examples

Fundición primitiva no numérica

El tipo boolean no se puede convertir a / desde ningún otro tipo primitivo.

Se puede convertir un char a / desde cualquier tipo numérico utilizando las asignaciones de punto de código especificadas por Unicode. Un char se representa en la memoria como un valor entero de 16 bits sin signo (2 bytes), por lo que la conversión a byte (1 byte) eliminará 8 de esos bits (esto es seguro para los caracteres ASCII). Los métodos de utilidad de la clase de Character utilizan int (4 bytes) para transferir a / desde valores de punto de código, pero un short (2 bytes) también sería suficiente para almacenar un punto de código Unicode.

```
int badInt   = (int)  true; // Compiler error: incompatible types

char char1   = (char)   65; // A
byte byte1   = (byte)  'A'; // 65
short short1 = (short) 'A'; // 65
int int1     = (int)   'A'; // 65

char char2   = (char) 8253; // ?
byte byte2   = (byte)  '?'; // 61 (truncated code-point into the ASCII range)
short short2 = (short) '?'; // 8253
int int2     = (int)   '?'; // 8253
```

Fundición primitiva numérica

Los primitivos numéricos se pueden convertir de dos maneras. *La* conversión *implícita* ocurre cuando el tipo de origen tiene un rango más pequeño que el tipo de destino.

```
//Implicit casting
byte byteVar = 42;
short shortVar = byteVar;
int intVar = shortVar;
long longVar = intvar;
float floatVar = longVar;
double doubleVar = floatVar;
```

La conversión *explícita* se debe realizar cuando el tipo de origen tiene un rango mayor que el tipo de destino.

```
//Explicit casting
double doubleVar = 42.0d;
float floatVar = (float) doubleVar;
long longVar = (long) floatVar;
int intVar = (int) longVar;
short shortVar = (short) intVar;
byte byteVar = (byte) shortVar;
```

Cuando se lanzan primitivas de punto flotante (float , double) a primitivas de números enteros, el número se **redondea hacia abajo** .

Fundición de objetos

Al igual que con los primitivos, los objetos pueden ser lanzados tanto explícita como implícitamente.

La conversión implícita ocurre cuando el tipo de origen extiende o implementa el tipo de destino (conversión a una superclase o interfaz).

La conversión explícita se debe realizar cuando el tipo de origen es extendido o implementado por el tipo de destino (conversión a un subtipo). Esto puede producir una excepción de tiempo de ejecución (ClassCastException) cuando el objeto que se está emitiendo no es del tipo de destino (o del subtipo del destino).

```
Float floatVar = new Float(42.0f);
Number n = floatVar;                //Implicit (Float implements Number)
Float floatVar2 = (Float) n;        //Explicit
Double doubleVar = (Double) n;      //Throws exception (the object is not Double)
```

Promoción Numérica Básica

```
 static void testNumericPromotion() {

    char char1 = 1, char2 = 2;
    short short1 = 1, short2 = 2;
    int int1 = 1, int2 = 2;
    float float1 = 1.0f, float2 = 2.0f;

    // char1 = char1 + char2;      // Error: Cannot convert from int to char;
    // short1 = short1 + short2;   // Error: Cannot convert from int to short;
    int1 = char1 + char2;          // char is promoted to int.
    int1 = short1 + short2;        // short is promoted to int.
    int1 = char1 + short2;         // both char and short promoted to int.
    float1 = short1 + float2;      // short is promoted to float.
    int1 = int1 + int2;            // int is unchanged.
}
```

Probar si un objeto puede ser lanzado usando instanceof

Java proporciona el operador instanceof para probar si un objeto es de un tipo determinado o una subclase de ese tipo. El programa puede elegir lanzar o no ese objeto en consecuencia.

```
Object obj = Calendar.getInstance();
long time = 0;

if(obj instanceof Calendar)
{
    time = ((Calendar)obj).getTime();
}
if(obj instanceof Date)
{
    time = ((Date)obj).getTime(); // This line will never be reached, obj is not a Date type.
}
```

Lea Tipo de conversión en línea: https://riptutorial.com/es/java/topic/1392/tipo-de-conversion

Capítulo 173: Tipos atómicos

Introducción

Los tipos atómicos de Java son tipos mutables simples que proporcionan operaciones básicas que son seguras para subprocesos y atómicas sin tener que recurrir al bloqueo. Están diseñados para usarse en casos en que el bloqueo sería un cuello de botella concurrente, o donde existe riesgo de bloqueo o bloqueo.

Parámetros

Parámetro	Descripción
conjunto	Conjunto volátil del campo.
obtener	Lectura volátil del campo.
lazySet	Esta es una operación ordenada por la tienda del campo.
compareAndSet	Si el valor es el valor de expiración, entonces envíelo al nuevo valor
getAndSet	obtener el valor actual y actualizar

Observaciones

Muchos en esencialmente combinaciones de lecturas o escrituras volátiles y operaciones CAS . La mejor manera de entender esto es mirar el código fuente directamente. Por ejemplo , AtomicInteger , Unsafe.getAndSet

Examples

Creando Tipos Atómicos

Para código simple de múltiples hilos, el uso de la sincronización es aceptable. Sin embargo, el uso de la sincronización tiene un impacto de vida, y a medida que la base de código se vuelve más compleja, aumenta la probabilidad de que termine con un punto muerto , un hambre o un bloqueo de vida .

En casos de concurrencia más compleja, el uso de variables atómicas suele ser una mejor alternativa, ya que permite acceder a una variable individual de una manera segura para subprocesos sin la sobrecarga de usar métodos sincronizados o bloques de código.

Creando un tipo AtomicInteger :

```
AtomicInteger aInt = new AtomicInteger() // Create with default value 0

AtomicInteger aInt = new AtomicInteger(1) // Create with initial value 1
```

Del mismo modo para otros tipos de instancia.

```
AtomicIntegerArray aIntArray = new AtomicIntegerArray(10) // Create array of specific length
AtomicIntegerArray aIntArray = new AtomicIntegerArray(new int[] {1, 2, 3}) // Initialize array
with another array
```

Del mismo modo para otros tipos atómicos.

Hay una notable excepción de que no hay tipos float y double . Estos se pueden simular mediante el uso de Float.floatToIntBits(float) y Float.intBitsToFloat(int) para float , así como Double.doubleToLongBits(double) y Double.longBitsToDouble(long) para dobles.

Si está dispuesto a usar sun.misc.Unsafe , puede usar cualquier variable primitiva como atómica utilizando la operación atómica en sun.misc.Unsafe . Todos los tipos primitivos deben convertirse o codificarse en int o longs para usarlos de esta manera. Para más sobre esto vea: sun.misc.Unsafe .

Motivación para los tipos atómicos

La forma sencilla de implementar aplicaciones de subprocesos múltiples es utilizar las primitivas de sincronización y bloqueo integradas de Java; por ejemplo, la palabra clave synchronized . El siguiente ejemplo muestra cómo podemos usar la synchronized para acumular cuentas.

```
public class Counters {
    private final int[] counters;

    public Counters(int nosCounters) {
        counters = new int[nosCounters];
    }

    /**
     * Increments the integer at the given index
     */
    public synchronized void count(int number) {
        if (number >= 0 && number < counters.length) {
            counters[number]++;
        }
    }

    /**
     * Obtains the current count of the number at the given index,
     * or if there is no number at that index, returns 0.
     */
    public synchronized int getCount(int number) {
        return (number >= 0 && number < counters.length) ? counters[number] : 0;
    }
}
```

Esta implementación funcionará correctamente. Sin embargo, si tiene un gran número de subprocesos que realizan muchas llamadas simultáneas en el mismo objeto de Counters , la sincronización puede ser un cuello de botella. Específicamente:

1. Cada llamada de método synchronized comenzará con el subproceso actual adquiriendo el bloqueo para la instancia de Counters .
2. El hilo mantendrá el bloqueo mientras verifica el valor del number y actualiza el contador.
3. Finalmente, el desbloqueará el bloqueo, permitiendo el acceso de otros hilos.

Si un hilo intenta adquirir el bloqueo mientras otro lo retiene, el hilo que intenta iniciarse se bloqueará (detendrá) en el paso 1 hasta que se libere el bloqueo. Si hay varios hilos en espera, uno de ellos lo obtendrá y los otros continuarán bloqueados.

Esto puede llevar a un par de problemas:

- Si hay mucha *disputa* por el bloqueo (es decir, un montón de subprocesos intenta adquirirlo), entonces se pueden bloquear algunos subprocesos durante mucho tiempo.
- Cuando se bloquea un subproceso esperando el bloqueo, el sistema operativo normalmente intentará cambiar la ejecución a un subproceso diferente. Este *cambio de contexto* incurre en un impacto de rendimiento relativamente grande en el procesador.

- Cuando hay varios subprocesos bloqueados en el mismo bloqueo, no hay ninguna garantía de que ninguno de ellos se trate de manera "justa" (es decir, se garantiza que cada subproceso está programado para ejecutarse). Esto puede conducir a la *hambruna de hilos* .

¿Cómo se implementan los tipos atómicos?

Comencemos reescribiendo el ejemplo anterior usando contadores de AtomicInteger :

```
public class Counters {
    private final AtomicInteger[] counters;

    public Counters(int nosCounters) {
        counters = new AtomicInteger[nosCounters];
        for (int i = 0; i < nosCounters; i++) {
            counters[i] = new AtomicInteger();
        }
    }

    /**
     * Increments the integer at the given index
     */
    public void count(int number) {
        if (number >= 0 && number < counters.length) {
            counters[number].incrementAndGet();
        }
    }

    /**
     * Obtains the current count of the object at the given index,
     * or if there is no number at that index, returns 0.
     */
    public int getCount(int number) {
        return (number >= 0 && number < counters.length) ?
                counters[number].get() : 0;
    }
}
```

Hemos reemplazado el int[] con un AtomicInteger[] , y lo AtomicInteger[] inicializado con una instancia en cada elemento. También hemos agregado llamadas a incrementAndGet() y get() en lugar de operaciones en valores int .

Pero lo más importante es que podemos eliminar la palabra clave synchronized porque ya no se requiere el bloqueo. Esto funciona porque las operaciones incrementAndGet() y get() son *atómicas* y *seguras para subprocesos* . En este contexto, significa que:

- Cada contador en la matriz solo será *observable* en el estado "antes" de una operación (como un "incremento") o en el estado "después".
- Suponiendo que la operación ocurra en el tiempo T , ningún hilo podrá ver el estado "antes" después del tiempo T

Además, mientras que dos subprocesos podrían intentar actualizar la misma instancia de AtomicInteger al mismo tiempo, las implementaciones de las operaciones aseguran que solo se produzca un incremento a la vez en la instancia dada. Esto se hace sin bloqueo, lo que a menudo resulta en un mejor rendimiento.

¿Cómo funcionan los tipos atómicos?

Los tipos atómicos generalmente se basan en instrucciones de hardware especializadas en el conjunto de instrucciones de la máquina de destino. Por ejemplo, los conjuntos de instrucciones basadas en Intel proporcionan una instrucción CAS (Comparar e Intercambiar) que realizará una secuencia específica de operaciones de memoria de forma atómica.

Estas instrucciones de bajo nivel se utilizan para implementar operaciones de alto nivel en las API de las respectivas clases AtomicXxx . Por ejemplo, (de nuevo, en pseudocódigo tipo C):

```
private volatile num;

int increment() {
  while (TRUE) {
    int old = num;
    int new = old + 1;
    if (old == compare_and_swap(&num, old, new)) {
      return new;
    }
  }
}
```

Si no hay contención en el AtomicXxxx , la prueba if tendrá éxito y el bucle finalizará inmediatamente. Si hay contención, entonces el if fallará para todos menos uno de los subprocesos, y "girarán" en el bucle durante un pequeño número de ciclos del bucle. En la práctica, el giro es más rápido en órdenes de magnitud (excepto en niveles de contención *irrealmente altos* , donde la sincronización funciona mejor que las clases atómicas porque cuando falla la operación CAS, el reintento solo agregará más contención) que suspender el hilo y cambiar a otro uno.

Incidentalmente, las instrucciones CAS suelen ser utilizadas por la JVM para implementar el *bloqueo no controlado* . Si la JVM puede ver que un bloqueo no está bloqueado actualmente, intentará usar un CAS para adquirir el bloqueo. Si el CAS tiene éxito, entonces no hay necesidad de realizar la costosa programación de subprocesos, el cambio de contexto, etc. Para obtener más información sobre las técnicas utilizadas, consulte Bloqueo sesgado en HotSpot .

Lea Tipos atómicos en línea: https://riptutorial.com/es/java/topic/5963/tipos-atomicos

Capítulo 174: Tipos de datos de referencia

Examples

Creando un tipo de referencia

```
Object obj = new Object(); // Note the 'new' keyword
```

Dónde:

- Object es un tipo de referencia.
- obj es la variable en la que se almacena la nueva referencia.
- Object() es la llamada a un constructor de Object .

Lo que pasa:

- Se asigna espacio en la memoria para el objeto.
- El constructor Object() se llama para inicializar ese espacio de memoria.
- La dirección de la memoria se almacena en obj , de modo que hace *referencia* al objeto recién creado.

Esto es diferente de los primitivos:

```
int i = 10;
```

Donde el valor real 10 se almacena en i .

Desreferenciación

La desreferenciación ocurre con el . operador:

```
Object obj = new Object();
String text = obj.toString(); // 'obj' is dereferenced.
```

La desreferenciación *sigue* la dirección de memoria almacenada en una referencia, al lugar en la memoria donde reside el objeto real. Cuando se ha encontrado un objeto, se llama al método solicitado (toString en este caso).

Cuando una referencia tiene el valor null , la desreferenciación da como resultado una excepción NullPointerException :

```
Object obj = null;
obj.toString(); // Throws a NullpointerException when this statement is executed.
```

null indica la ausencia de un valor, es decir, *siguiendo* las indicaciones de la dirección de memoria a ninguna parte. Así que no hay ningún objeto sobre el cual se pueda llamar al método solicitado.

Lea Tipos de datos de referencia en línea: https://riptutorial.com/es/java/topic/1046/tipos-de-datos-de-referencia

Capítulo 175: Tipos de datos primitivos

Introducción

Los 8 tipos de datos primitivos byte , short , int , long , char , boolean , float y double son los tipos que almacenan la mayoría de los datos numéricos sin procesar en los programas Java.

Sintaxis

- int aInt = 8; // La parte definitoria (número) de esta declaración int se llama un literal.
- int hexInt = 0x1a; // = 26; Puedes definir literales con valores hexadecimales prefijados con *0x* .
- int binInt = 0b11010; // = 26; También puedes definir literales binarios; prefijado con *0b* .
- long goodLong = 10000000000L; // Por defecto, los literales enteros son de tipo int. Al agregar la L al final del literal, le está diciendo al compilador que el literal es largo. Sin esto, el compilador arrojaría un error de "Número entero demasiado grande".
- doble aDouble = 3.14; // Los literales de punto flotante son de tipo doble de forma predeterminada.
- flotar aFloat = 3.14F; // Por defecto, este literal habría sido un doble (y causó un error de "Tipos incompatibles"), pero al agregar una F le decimos al compilador que es un flotador.

Observaciones

Java tiene 8 *tipos de datos primitivos* , a saber, boolean , byte , short , char , int , long , float y double . (Todos los demás tipos son tipos de *referencia* . Esto incluye todos los tipos de matriz y los tipos / clases de objetos integrados que tienen un significado especial en el lenguaje Java; por ejemplo, String , Class y Throwable y sus subclases).

El resultado de todas las operaciones (suma, resta, multiplicación, etc.) en un tipo primitivo es al menos un int , por lo que sumar un short a un short produce un int , al igual que agregar un byte a un byte , o un char a un char . Si desea asignar el resultado de eso a un valor del mismo tipo, debe convertirlo. p.ej

```
byte a = 1;
byte b = 2;
byte c = (byte) (a + b);
```

No lanzar la operación dará como resultado un error de compilación.

Esto se debe a la siguiente parte de la especificación del lenguaje Java, §2.11.1 :

> Un compilador codifica cargas de valores literales de tipos byte y short usando instrucciones de la Máquina Virtual Java que extienden sus valores de signo a valores de tipo int en tiempo de compilación o tiempo de ejecución. Las cargas de valores literales de los tipos boolean y char se codifican utilizando instrucciones que extienden el literal literalmente a un valor de tipo int en tiempo de compilación o tiempo de ejecución. [..]. Por lo tanto, la mayoría de las operaciones en valores de tipos reales boolean , byte , char y short se realizan correctamente mediante instrucciones que operan en valores de tipo computacional int .

La razón detrás de esto también se especifica en esa sección:

> Dado el **tamaño del código de operación de un byte de** la Máquina Virtual Java, los tipos de codificación en **códigos de operación** ejercen presión sobre el diseño de su

conjunto de instrucciones. Si cada instrucción escrita admitiera todos los tipos de datos en tiempo de ejecución de la Máquina Virtual Java, habría más instrucciones de las que podrían representarse en un byte . [...] Se pueden usar instrucciones separadas para convertir entre tipos de datos no compatibles y compatibles según sea necesario.

Examples

El primitivo int

Un tipo de datos primitivos como int contiene valores directamente en la variable que lo está utilizando, mientras que una variable que se declaró utilizando Integer contiene una referencia al valor.

De acuerdo con la API de Java : "La clase Integer ajusta un valor del tipo primitivo int en un objeto. Un objeto de tipo Integer contiene un campo único cuyo tipo es int".

Por defecto, int es un entero con signo de 32 bits. Puede almacenar un valor mínimo de -2^{31} , y un valor máximo de $2^{31} - 1$.

```
int example = -42;
int myInt = 284;
int anotherInt = 73;

int addedInts = myInt + anotherInt; // 284 + 73 = 357
int subtractedInts = myInt - anotherInt; // 284 - 73 = 211
```

Si necesita almacenar un número fuera de este rango, en long debe usar long . Si se excede el rango de valores de int produce un desbordamiento de enteros, lo que hace que el valor que exceda el rango se agregue al sitio opuesto del rango (positivo se convierte en negativo y viceversa). El valor es ((value - MIN_VALUE) % RANGE) + MIN_VALUE , o ((value + 2147483648) % 4294967296) - 2147483648

```
int demo = 2147483647; //maximum positive integer
System.out.println(demo); //prints 2147483647
demo = demo + 1; //leads to an integer overflow
System.out.println(demo); // prints -2147483648
```

Los valores máximo y mínimo de int se pueden encontrar en:

```
int high = Integer.MAX_VALUE;    // high == 2147483647
int low = Integer.MIN_VALUE;     // low == -2147483648
```

El valor predeterminado de un int es *0*

```
int defaultInt;    // defaultInt == 0
```

El primitivo corto

Un short es un entero con signo de 16 bits. Tiene un valor mínimo de -2^{15} (-32,768), y un valor máximo de 2^{15} -1 (32,767)

```
short example = -48;
short myShort = 987;
short anotherShort = 17;

short addedShorts = (short) (myShort + anotherShort); // 1,004
short subtractedShorts = (short) (myShort - anotherShort); // 970
```

Los valores máximo y mínimo de short se pueden encontrar en:

```
short high = Short.MAX_VALUE;        // high == 32767
short low = Short.MIN_VALUE;         // low == -32768
```

El valor predeterminado de un short es *0*

```
short defaultShort;    // defaultShort == 0
```

El primitivo largo

De forma predeterminada, long es un entero con signo de 64 bits (en Java 8, puede ser firmado o sin signo). Firmado, puede almacenar un valor mínimo de -2^{63}, y un valor máximo de $2^{63} - 1$, y sin firmar puede almacenar un valor mínimo de 0 y un valor máximo de $2^{64} - 1$

```
long example = -42;
long myLong = 284;
long anotherLong = 73;

//an "L" must be appended to the end of the number, because by default,
//numbers are assumed to be the int type. Appending an "L" makes it a long
//as 549755813888 (2 ^ 39) is larger than the maximum value of an int (2^31 - 1),
//"L" must be appended
long bigNumber = 549755813888L;

long addedLongs = myLong + anotherLong; // 284 + 73 = 357
long subtractedLongs = myLong - anotherLong; // 284 - 73 = 211
```

Los valores máximo y mínimo de long se pueden encontrar en:

```
long high = Long.MAX_VALUE;    // high == 9223372036854775807L
long low = Long.MIN_VALUE;     // low == -9223372036854775808L
```

El valor predeterminado de un long es *0L*.

```
long defaultLong;    // defaultLong == 0L
```

Nota: la letra "L" adjunta al final del literal long distingue entre mayúsculas y minúsculas, sin embargo, es una buena práctica usar mayúscula, ya que es más fácil distinguirla del dígito uno:

```
2L == 2l;            // true
```

Advertencia: Java almacena en caché instancias de objetos enteros del rango -128 a 127. El razonamiento se explica aquí:
https://blogs.oracle.com/darcy/entry/boxing_and_caches_integer_valueof

Se pueden encontrar los siguientes resultados:

```
Long val1 = 127L;
Long val2 = 127L;

System.out.println(val1 == val2); // true

Long val3 = 128L;
Long val4 = 128L;

System.out.println(val3 == val4); // false
```

Para comparar correctamente los valores de 2 objetos largos, use el siguiente código (desde Java 1.7 en adelante):

```
Long val3 = 128L;
Long val4 = 128L;

System.out.println(Objects.equal(val3, val4)); // true
```

La comparación de un largo primitivo con un objeto largo no dará como resultado un falso negativo como la comparación de 2 objetos con == hace.

El primitivo booleano

Un boolean puede almacenar uno de dos valores, ya sea true o false

```
boolean foo = true;
System.out.println("foo = " + foo);                // foo = true

boolean bar = false;
System.out.println("bar = " + bar);                // bar = false

boolean notFoo = !foo;
System.out.println("notFoo = " + notFoo);          // notFoo = false

boolean fooAndBar = foo && bar;
System.out.println("fooAndBar = " + fooAndBar);    // fooAndBar = false

boolean fooOrBar = foo || bar;
System.out.println("fooOrBar = " + fooOrBar);      // fooOrBar = true

boolean fooXorBar = foo ^ bar;
System.out.println("fooXorBar = " + fooXorBar);    // fooXorBar = true
```

El valor predeterminado de un boolean es *falso*

```
boolean defaultBoolean;    // defaultBoolean == false
```

El byte primitivo

Un byte es un entero con signo de 8 bits. Puede almacenar un valor mínimo de -2^7 (-128), y un valor máximo de $2^7 - 1$ (127)

```
byte example = -36;
byte myByte = 96;
byte anotherByte = 7;

byte addedBytes = (byte) (myByte + anotherByte); // 103
byte subtractedBytes = (byte) (myBytes - anotherByte); // 89
```

Los valores máximo y mínimo de byte se pueden encontrar en:

```
byte high = Byte.MAX_VALUE;        // high == 127
byte low = Byte.MIN_VALUE;         // low == -128
```

El valor predeterminado de un byte es *0*

```
byte defaultByte;    // defaultByte == 0
```

El flotador primitivo

Un float es un número de punto flotante IEEE 754 de 32 bits de precisión simple. Por defecto, los decimales se interpretan como dobles. Para crear un float , simplemente agregue una f al literal decimal.

```
double doubleExample = 0.5;      // without 'f' after digits = double
float floatExample = 0.5f;       // with 'f' after digits    = float

float myFloat = 92.7f;           // this is a float...
float positiveFloat = 89.3f;     // it can be positive,
float negativeFloat = -89.3f;    // or negative
float integerFloat = 43.0f;      // it can be a whole number (not an int)
float underZeroFloat = 0.0549f;  // it can be a fractional value less than 0
```

Los flotadores manejan las cinco operaciones aritméticas comunes: suma, resta, multiplicación, división y módulo.

Nota: Lo siguiente puede variar ligeramente como resultado de errores de punto flotante. Algunos resultados se han redondeado con fines de claridad y legibilidad (es decir, el resultado impreso del ejemplo de adición fue en realidad 34.600002).

```
// addition
float result = 37.2f + -2.6f;  // result: 34.6

// subtraction
float result = 45.1f - 10.3f;    // result: 34.8

// multiplication
float result = 26.3f * 1.7f;   // result: 44.71

// division
float result = 37.1f / 4.8f;   // result: 7.729166

// modulus
float result = 37.1f % 4.8f;   // result: 3.4999971
```

Debido a la forma en que se almacenan los números de punto flotante (es decir, en forma binaria), muchos números no tienen una representación exacta.

```
float notExact = 3.1415926f;
System.out.println(notExact); // 3.1415925
```

Si bien el uso de float está bien para la mayoría de las aplicaciones, no se debe usar float ni double para almacenar representaciones exactas de números decimales (como cantidades monetarias), o números donde se requiera mayor precisión. En su lugar, se debe utilizar la clase BigDecimal .

El valor predeterminado de un float es *0.0f* .

```
float defaultFloat;    // defaultFloat == 0.0f
```

Un float es preciso para aproximadamente un error de 1 en 10 millones.

Nota: Float.POSITIVE_INFINITY , Float.NEGATIVE_INFINITY , Float.NaN son valores float . NaN significa resultados de operaciones que no se pueden determinar, como dividir 2 valores infinitos. Además, 0f y -0f son diferentes, pero == produce verdadero:

```
float f1 = 0f;
float f2 = -0f;
```

```
System.out.println(f1 == f2); // true
System.out.println(1f / f1); // Infinity
System.out.println(1f / f2); // -Infinity
System.out.println(Float.POSITIVE_INFINITY / Float.POSITIVE_INFINITY); // NaN
```

El doble primitivo

Un double es un número de punto flotante IEEE 754 de doble precisión de 64 bits.

```
double example = -7162.37;
double myDouble = 974.21;
double anotherDouble = 658.7;

double addedDoubles = myDouble + anotherDouble; // 315.51
double subtractedDoubles = myDouble - anotherDouble; // 1632.91

double scientificNotationDouble = 1.2e-3;    // 0.0012
```

Debido a la forma en que se almacenan los números de punto flotante, muchos números no tienen una representación exacta.

```
double notExact = 1.32 - 0.42; // result should be 0.9
System.out.println(notExact); // 0.9000000000000001
```

Si bien el uso del double está bien para la mayoría de las aplicaciones, no se debe usar ni float ni double para almacenar números precisos, como la moneda. En su lugar, se debe usar la clase BigDecimal

El valor predeterminado de un double es *0.0d*

```
public double defaultDouble;    // defaultDouble == 0.0
```

Nota: Double.POSITIVE_INFINITY , Double.NEGATIVE_INFINITY , Double.NaN son valores double . NaN significa resultados de operaciones que no se pueden determinar, como dividir 2 valores infinitos. Además, 0d y -0d son diferentes, pero == produce verdadero:

```
double d1 = 0d;
double d2 = -0d;
System.out.println(d1 == d2); // true
System.out.println(1d / d1); // Infinity
System.out.println(1d / d2); // -Infinity
System.out.println(Double.POSITIVE_INFINITY / Double.POSITIVE_INFINITY); // NaN
```

El primitivo char

Un char puede almacenar un solo carácter Unicode de 16 bits. Un literal de carácter está encerrado entre comillas simples

```
char myChar = 'u';
char myChar2 = '5';
char myChar3 = 65; // myChar3 == 'A'
```

Tiene un valor mínimo de \u0000 (0 en la representación decimal, también llamado el *carácter nulo*) y un valor máximo de \uffff (65,535).

El valor predeterminado de un char es \u0000 .

```
char defaultChar;    // defaultChar == \u0000
```

Para definir un valor de char of ' se debe utilizar una secuencia de escape (carácter precedido por una barra invertida):

```
char singleQuote = '\'';
```

También hay otras secuencias de escape:

```
char tab = '\t';
char backspace = '\b';
char newline = '\n';
char carriageReturn = '\r';
char formfeed = '\f';
char singleQuote = '\'';
char doubleQuote = '\"'; // escaping redundant here; '"' would be the same; however still
allowed
char backslash = '\\';
char unicodeChar = '\uXXXX' // XXXX represents the Unicode-value of the character you want to
display
```

Puedes declarar un char de cualquier carácter Unicode.

```
char heart = '\u2764';
System.out.println(Character.toString(heart)); // Prints a line containing "❤".
```

También es posible añadir a un char . Por ejemplo, para iterar en cada letra minúscula, puede hacer lo siguiente:

```
for (int i = 0; i <= 26; i++) {
    char letter = (char) ('a' + i);
    System.out.println(letter);
}
```

Representación de valor negativo

Java y la mayoría de los otros lenguajes almacenan números integrales negativos en una representación llamada notación de *complemento a 2* .

Para una representación binaria única de un tipo de datos que utiliza n bits, los valores se codifican así:

Los bits n-1 menos significativos almacenan un número integral positivo x en representación integral. El valor más significativo almacena un poco con el valor s . El valor representado por esos bits es

$x - s * 2^{n-1}$

es decir, si el bit más significativo es 1, entonces un valor que es solo 1 mayor que el número que podría representar con los otros bits ($2^{n-2} + 2^{n-3} + ... + 2^{1} + 2^{0} = 2^{n-1} - 1$) se resta permitiendo una representación binaria única para cada valor de $- 2^{n-1}$ (s = 1; x = 0) a $2^{n-1} - 1$ (s = 0; x = $2^{n-1} - 1$).

Esto también tiene el efecto secundario agradable, que puede agregar las representaciones binarias como si fueran números binarios positivos:

$$v1 = x1 - s1 * 2^{n-1}$$
$$v2 = x2 - s2 * 2^{n-1}$$

s1	s2	x1 + x2 desbordamiento	resultado adicional
0	0	No	x1 + x2 = v1 + v2
0	0	Sí	demasiado grande para ser representada con el tipo de datos (desbordamiento)
0	1	No	$x1 + x2 - 2^{n-1} = x1 + x2 - s2 * 2^{n-1}$ $= v1 + v2$
0	1	Sí	$(x1 + x2) \textbf{ mod } 2^{n-1} = x1 + x2 - 2^{n-1}$ $= v1 + v2$
1	0	*	ver arriba (intercambiar sumas)
1	1	No	demasiado pequeño para ser representado con el tipo de datos ($x1 + x2 - 2^{n} < -2^{n-1}$; subflujo)
1	1	Sí	$(x1 + x2) \textbf{ mod } 2^{n-1} - 2^{n-1} = (x1 + x2 - 2^{n-1}) - 2^{n-1}$ $= (x1 - s1 * 2^{n-1}) + (x2 - s2 * 2^{n-1})$ $= v1 + v2$

Tenga en cuenta que este hecho hace que sea fácil encontrar la representación binaria del inverso aditivo (es decir, el valor negativo):

Observe que agregar el complemento bit a bit al número resulta en que todos los bits son 1. Ahora agregue 1 para hacer que el valor se desborde y obtendrá el elemento neutral 0 (todos los bits 0).

Por lo que el valor negativo de un número i se puede calcular utilizando (ignorando posible promoción a int aquí)

```
(~i) + 1
```

Ejemplo: tomando el valor negativo de 0 (byte):

El resultado de negar 0 , es 11111111 . Al sumar 1 se obtiene un valor de 100000000 (9 bits). Debido a que un byte solo puede almacenar 8 bits, el valor situado más a la izquierda se trunca y el resultado es 00000000

Original	Proceso	Resultado
0 (00000000)	Negar	-0 (11111111)
11111111	Agrega 1 al binario	100000000
100000000	Truncar a 8 bits	00000000 (-0 es igual a 0)

Consumo de memoria de primitivas vs. primitivas en caja

Primitivo	Tipo de caja	Tamaño de memoria de primitivo / en caja
booleano	Booleano	1 byte / 16 bytes
byte	Byte	1 byte / 16 bytes
corto	Corto	2 bytes / 16 bytes
carbonizarse	Carbonizarse	2 bytes / 16 bytes
En t	Entero	4 bytes / 16 bytes
largo	Largo	8 bytes / 16 bytes
flotador	Flotador	4 bytes / 16 bytes
doble	Doble	8 bytes / 16 bytes

Los objetos en caja siempre requieren 8 bytes para la administración de tipo y memoria, y como el tamaño de los objetos es siempre un múltiplo de 8, *todos los* tipos en caja *requieren un total de 16 bytes* . *Además* , cada uso de un objeto en caja implica almacenar una referencia que representa otros 4 u 8 bytes, según las opciones de JVM y JVM.

En las operaciones de uso intensivo de datos, el consumo de memoria puede tener un gran impacto en el rendimiento. El consumo de memoria aumenta aún más cuando se usan matrices: una matriz float[5] solo requerirá 32 bytes; mientras que un Float[5] almacena 5 valores distintos de valores nulos requerirá un total de 112 bytes (en 64 bits sin punteros comprimidos, esto aumenta a 152 bytes).

Cachés de valor en caja

Las sobrecargas de espacio de los tipos encajonados se pueden mitigar hasta cierto punto por los cachés de valores encajonados. Algunos de los tipos en caja implementan un caché de instancias. Por ejemplo, de forma predeterminada, la clase Integer almacenará en caché las instancias para representar números en el rango de −128 a +127 . Sin embargo, esto no reduce el costo adicional derivado de la indirección de memoria adicional.

Si crea una instancia de un tipo encuadrado ya sea mediante autofijación o llamando al método static valueOf(primitive) , el sistema de tiempo de ejecución intentará utilizar un valor almacenado en caché. Si su aplicación utiliza muchos valores en el rango que se almacena en la memoria caché, esto puede reducir sustancialmente la penalización de memoria al usar tipos en caja. Ciertamente, si está creando instancias de valor en caja "a mano", es mejor usar valueOf lugar de new . (La new operación siempre crea una nueva instancia). Sin embargo, si la mayoría de sus valores *no* están en el rango almacenado en la memoria caché, puede ser más rápido llamar new y guardar la búsqueda de caché.

Primitivas de conversión

En Java, podemos convertir entre valores enteros y valores de punto flotante. Además, dado que cada carácter corresponde a un número en la codificación Unicode, char tipos se pueden convertir a y desde los tipos de enteros y de coma flotante. boolean es el único tipo de datos primitivo que no se puede convertir ao desde cualquier otro tipo de datos primitivos.

Hay dos tipos de conversiones: *ampliar la conversión* y *reducir la conversión* .

Una *conversión de ampliación se* produce cuando un valor de un tipo de datos se convierte en un valor de otro tipo de datos que ocupa más bits que el anterior. No hay problema de pérdida de datos en este caso.

En consecuencia, una *conversión de reducción se* produce cuando un valor de un tipo de datos se convierte en un valor de otro tipo de datos que ocupa menos bits que el anterior. La pérdida de datos puede ocurrir en este caso.

Java realiza *ampliaciones de conversiones* automáticamente. Pero si desea realizar una *conversión de restricción* (si está seguro de que no se produzcan pérdidas de datos), entonces se puede forzar Java para realizar la conversión usando una construcción del lenguaje conocido como un cast .

Ampliación de la conversión:

```
int a = 1;
double d = a;    // valid conversion to double, no cast needed (widening)
```

Reducción de la conversión:

```
double d = 18.96
int b = d;       // invalid conversion to int, will throw a compile-time error
int b = (int) d; // valid conversion to int, but result is truncated (gets rounded down)
                 // This is type-casting
                 // Now, b = 18
```

Tipos primitivos Cheatsheet

Tabla que muestra el rango de tamaño y valores de todos los tipos primitivos:

tipo de datos	representación numérica	rango de valores	valor por defecto
booleano	n / A	falso y verdadero	falso
byte	Firmado de 8 bits	-2^{7} a $2^{7} - 1$	0
		-128 a +127	
corto	Firmado de 16 bits	-2^{15} a $2^{15} - 1$	0
		-32,768 a +32,767	
En t	32 bits firmado	-2^{31} a $2^{31} - 1$	0
		-2,147,483,648 a +2,147,483,647	
largo	Firmado de 64 bits	-2^{63} a $2^{63} - 1$	0L
		-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807	
flotador	Punto flotante de 32 bits	1.401298464e-45 a 3.402823466e + 38 (positivo o negativo)	0.0F
doble	Punto flotante de 64 bits	4.94065645841246544e-324d a 1.79769313486231570e + 308d (positivo o	0.0D

tipo de datos	representación numérica	rango de valores	valor por defecto
		negativo)	
carbonizarse	16 bits sin firmar	0 a 2^{16} - 1	0
		0 a 65,535	

Notas:

1. Los mandatos de la Especificación del lenguaje Java que los tipos integrales firmados (byte a long) usan una representación binaria de complemento a dos, y los tipos de punto flotante usan representaciones binarias estándar de punto flotante IEE 754.
2. Java 8 y versiones posteriores proporcionan métodos para realizar operaciones aritméticas no firmadas en int y long . Si bien estos métodos permiten que un programa *trate los* valores de los tipos respectivos como no firmados, los tipos siguen siendo tipos con signo.
3. El punto flotante más pequeño que se muestra arriba es *subnormal* ; Es decir, tienen menos precisión que un valor *normal* . Los números normales más pequeños son 1.175494351e - 38 y 2.2250738585072014e - 308
4. Un char representa convencionalmente una *unidad de código* Unicode / UTF-16.
5. Aunque un valor boolean contiene solo un bit de información, su tamaño en memoria varía según la implementación de la Máquina Virtual Java (vea el tipo booleano).

Lea Tipos de datos primitivos en línea: https://riptutorial.com/es/java/topic/148/tipos-de-datos-primitivos

Capítulo 176: Tipos de referencia

Examples

Diferentes tipos de referencia

java.lang.ref paquete java.lang.ref proporciona clases de objetos de referencia, que admiten un grado limitado de interacción con el recolector de basura.

Java tiene cuatro tipos principales de referencia diferentes. Son:

- Fuerte referencia
- Referencia débil
- Referencia suave
- Referencia fantasma

1. fuerte referencia

Esta es la forma habitual de crear objetos.

```
MyObject myObject = new MyObject();
```

El titular de la variable mantiene una fuerte referencia al objeto creado. Mientras esta variable esté MyObject y mantenga este valor, el recolector de basura no recopilará la instancia de MyObject .

2. Referencia débil

Cuando no desea mantener un objeto por más tiempo y necesita borrar / liberar la memoria asignada para un objeto tan pronto como sea posible, esta es la manera de hacerlo.

```
WeakReference myObjectRef = new WeakReference(MyObject);
```

Simplemente, una referencia débil es una referencia que no es lo suficientemente fuerte como para obligar a un objeto a permanecer en la memoria. Las referencias débiles le permiten aprovechar la capacidad del recolector de basura para determinar la accesibilidad para usted, por lo que no tiene que hacerlo usted mismo.

Cuando necesite el objeto que creó, simplemente use el método .get() :

```
myObjectRef.get();
```

El siguiente código ejemplificará esto:

```
WeakReference myObjectRef = new WeakReference(MyObject);
System.out.println(myObjectRef.get()); // This will print the object reference address
System.gc();
System.out.println(myObjectRef.get()); // This will print 'null' if the GC cleaned up the
object
```

3. Referencia suave

Las referencias blandas son ligeramente más fuertes que las débiles. Puede crear un objeto de referencia suave de la siguiente manera:

```
SoftReference myObjectRef = new SoftReference(MyObject);
```

Pueden aferrarse a la memoria con más fuerza que la referencia débil. Si tiene suficientes recursos / recursos de memoria, el recolector de basura no limpiará las referencias blandas con

tanto entusiasmo como las referencias débiles.

Las referencias blandas son útiles para usar en el almacenamiento en caché. Puede crear objetos de referencia suave como un caché, donde se guardarán hasta que se agote la memoria. Cuando su memoria no puede proporcionar suficientes recursos, el recolector de basura eliminará las referencias blandas.

```
SoftReference myObjectRef = new SoftReference(MyObject);
System.out.println(myObjectRef.get()); // This will print the reference address of the Object
System.gc();
System.out.println(myObjectRef.get()); // This may or may not print the reference address of
the Object
```

4. Referencia fantasma

Este es el tipo de referencia más débil. Si creó una referencia de objeto utilizando la Referencia fantasma, el método get() siempre devolverá nulo.

El uso de esta referencia es que "los objetos de referencia fantasma, que se ponen en cola después de que el recopilador determina que sus referentes pueden ser reclamados. Las referencias fantasma se usan más a menudo para programar acciones de limpieza premente de una manera más flexible que la posible Mecanismo de finalización de Java ". - De Phantom Reference Javadoc de Oracle.

Puede crear un objeto de referencia fantasma de la siguiente manera:

```
PhantomReference myObjectRef = new PhantomReference(MyObject);
```

Lea Tipos de referencia en línea: https://riptutorial.com/es/java/topic/4017/tipos-de-referencia

Capítulo 177: Tokenizador de cuerdas

Introducción

La clase java.util.StringTokenizer te permite dividir una cadena en tokens. Es una forma sencilla de romper la cuerda.

El conjunto de delimitadores (los caracteres que separan los tokens) se puede especificar en el momento de la creación o por token.

Examples

StringTokenizer dividido por el espacio

```
import java.util.StringTokenizer;
public class Simple{
 public static void main(String args[]){
   StringTokenizer st = new StringTokenizer("apple ball cat dog"," ");
     while (st.hasMoreTokens()) {
         System.out.println(st.nextToken());
     }
   }
}
```

Salida:

manzana

bola

gato

perro

StringTokenizer Dividir por comas ','

```
public static void main(String args[]) {
        StringTokenizer st = new StringTokenizer("apple,ball cat,dog", ",");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
```

Salida:

manzana

bola de gato

perro

Lea Tokenizador de cuerdas en línea: https://riptutorial.com/es/java/topic/10563/tokenizador-de-cuerdas

Capítulo 178: TreeMap y TreeSet

Introducción

TreeMap y TreeSet son colecciones básicas de Java agregadas en Java 1.2. TreeMap es una implementación de Map **ordenada** y **mutable** . De manera similar, TreeSet es una implementación de Set **ordenada** y **mutable** .

TreeMap se implementa como un árbol rojo-negro, que proporciona tiempos de acceso O(log n) . TreeSet se implementa utilizando un TreeMap con valores ficticios.

Ambas colecciones **no** son seguras para subprocesos.

Examples

TreeMap de un tipo de Java simple

Primero, creamos un mapa vacío, e insertamos algunos elementos en él:

Java SE 7

```
TreeMap<Integer, String> treeMap = new TreeMap<>();
```

Java SE 7

```
TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();
```

```
treeMap.put(10, "ten");
treeMap.put(4, "four");
treeMap.put(1, "one");
treeSet.put(12, "twelve");
```

Una vez que tenemos algunos elementos en el mapa, podemos realizar algunas operaciones:

```
System.out.println(treeMap.firstEntry()); // Prints 1=one
System.out.println(treeMap.lastEntry()); // Prints 12=twelve
System.out.println(treeMap.size()); // Prints 4, since there are 4 elemens in the map
System.out.println(treeMap.get(12)); // Prints twelve
System.out.println(treeMap.get(15)); // Prints null, since the key is not found in the map
```

También podemos iterar sobre los elementos del mapa usando un iterador o un bucle foreach. Tenga en cuenta que las entradas se imprimen de acuerdo con su orden natural , no el orden de inserción:

Java SE 7

```
for (Entry<Integer, String> entry : treeMap.entrySet()) {
    System.out.print(entry + " "); //prints 1=one 4=four 10=ten 12=twelve
}
```

```
Iterator<Entry<Integer, String>> iter = treeMap.entrySet().iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " "); //prints 1=one 4=four 10=ten 12=twelve
}
```

TreeSet de un tipo de Java simple

Primero, creamos un conjunto vacío, e insertamos algunos elementos en él:

Java SE 7

```
TreeSet<Integer> treeSet = new TreeSet<>();
```

Java SE 7

```
TreeSet<Integer> treeSet = new TreeSet<Integer>();
```

```
treeSet.add(10);
treeSet.add(4);
treeSet.add(1);
treeSet.add(12);
```

Una vez que tenemos algunos elementos en el conjunto, podemos realizar algunas operaciones:

```
System.out.println(treeSet.first()); // Prints 1
System.out.println(treeSet.last()); // Prints 12
System.out.println(treeSet.size()); // Prints 4, since there are 4 elemens in the set
System.out.println(treeSet.contains(12)); // Prints true
System.out.println(treeSet.contains(15)); // Prints false
```

También podemos iterar sobre los elementos del mapa usando un iterador o un bucle foreach. Tenga en cuenta que las entradas se imprimen de acuerdo con su orden natural , no el orden de inserción:

Java SE 7

```
for (Integer i : treeSet) {
    System.out.print(i + " "); //prints 1 4 10 12
}
```

```
Iterator<Integer> iter = treeSet.iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " "); //prints 1 4 10 12
}
```

TreeMap / TreeSet de un tipo de Java personalizado

Dado que TreeMap sy TreeSet s mantienen llaves / elementos de acuerdo con su orden natural . Por TreeMap tanto, TreeMap claves TreeMap y los elementos TreeSet tienen que ser comparables entre sí.

Digamos que tenemos una clase personalizada de Person :

```
public class Person  {

    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constuctors, getters, setters and various methods
}
```

Si lo almacenamos como está en un TreeSet (o una clave en un TreeMap):

```
TreeSet<Person2> set = ...
set.add(new Person(1,"first","last",Date.from(Instant.now())));
```

Entonces nos encontraríamos con una excepción como esta:

```
Exception in thread "main" java.lang.ClassCastException: Person cannot be cast to
java.lang.Comparable
    at java.util.TreeMap.compare(TreeMap.java:1294)
    at java.util.TreeMap.put(TreeMap.java:538)
    at java.util.TreeSet.add(TreeSet.java:255)
```

Para solucionarlo, supongamos que queremos ordenar instancias de Person función del orden de sus private int id (private int id). Podríamos hacerlo de una de dos maneras:

1. Una solución es modificar Person para que implemente la interfaz Comparable :

```
public class Person implements Comparable<Person> {
    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constuctors, getters, setters and various methods

    @Override
    public int compareTo(Person o) {
        return Integer.compare(this.id, o.id); //Compare by id
    }
}
```

2. Otra solución es proporcionar al TreeSet un Comparador :

Java SE 8

```
TreeSet<Person> treeSet = new TreeSet<>((personA, personB) -> Integer.compare(personA.getId(),
personB.getId()));
```

```
TreeSet<Person> treeSet = new TreeSet<>(new Comparator<Person>(){
    @Override
    public int compare(Person personA, Person personB) {
        return Integer.compare(personA.getId(), personB.getId());
    }
});
```

Sin embargo, hay dos advertencias para ambos enfoques:

1. Es **muy importante** no modificar los campos utilizados para ordenar una vez que una instancia se haya insertado en un TreeSet / TreeMap . En el ejemplo anterior, si cambiamos el id de una persona que ya está insertada en la colección, podríamos encontrar un comportamiento inesperado.

2. Es importante implementar la comparación de manera adecuada y consistente. Según el Javadoc :

   El implementador debe asegurarse de que sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) para todas las x y y. (Esto implica que x.compareTo(y) debe lanzar una excepción si y.compareTo(x) lanza una excepción).

   El implementador también debe asegurarse de que la relación sea transitiva: (x.compareTo(y)>0 && y.compareTo(z)>0) implica x.compareTo(z)>0 .

   Finalmente, el implementador debe asegurarse de que x.compareTo(y)==0 implica que

sgn(x.compareTo(z)) == sgn(y.compareTo(z)) , para todas las z.

TreeMap y TreeSet Thread Safety

TreeMap y TreeSet **no** son colecciones seguras para subprocesos, por lo que se debe tener cuidado para garantizar que se utilicen en programas de subprocesos múltiples.

Tanto TreeMap como TreeSet son seguros cuando se leen, incluso al mismo tiempo, por varios subprocesos. Por lo tanto, si se crearon y rellenaron con un solo subproceso (por ejemplo, al inicio del programa), y solo luego se leyeron, pero no se modificaron por varios subprocesos, no hay razón para la sincronización o el bloqueo.

Sin embargo, si se lee y se modifica al mismo tiempo, o si se modifica al mismo tiempo por más de un hilo, la colección puede generar una ConcurrentModificationException o comportarse de manera inesperada. En estos casos, es imperativo sincronizar / bloquear el acceso a la colección utilizando uno de los siguientes métodos:

1. Utilizando Collections.synchronizedSorted.. :

```
SortedSet<Integer> set = Collections.synchronizedSortedSet(new TreeSet<Integer>());
SortedMap<Integer,String> map = Collections.synchronizedSortedMap(new
TreeMap<Integer,String>());
```

   Esto proporcionará una implementación SortedSet / SortedMap respaldada por la colección real y sincronizada en algún objeto de exclusión mutua . Tenga en cuenta que esto sincronizará todos los accesos de lectura y escritura a la colección en un solo bloqueo, por lo que incluso las lecturas simultáneas no serían posibles.

2. Al sincronizar manualmente en algún objeto, como la propia colección:

```
 TreeSet<Integer> set = new TreeSet<>();
```

   ...

```
//Thread 1
synchronized (set) {
    set.add(4);
}
```

   ...

```
//Thread 2
synchronized (set) {
    set.remove(5);
}
```

3. Mediante el uso de un bloqueo, como ReentrantReadWriteLock :

```
 TreeSet<Integer> set = new TreeSet<>();
 ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
```

   ...

```
 //Thread 1
 lock.writeLock().lock();
 set.add(4);
 lock.writeLock().unlock();
```

...

```
//Thread 2
lock.readLock().lock();
set.contains(5);
lock.readLock().unlock();
```

A diferencia de los métodos de sincronización anteriores, el uso de ReadWriteLock permite que varios subprocesos lean del mapa simultáneamente.

Lea TreeMap y TreeSet en línea: https://riptutorial.com/es/java/topic/9905/treemap-y-treeset

Capítulo 179: Usando la palabra clave estática

Sintaxis

- public static int myVariable; // Declarar una variable estática
- public static myMethod () {} // Declarar un método estático
- public static final final doble MY_CONSTANT; // Declarar una variable constante que se comparte entre todas las instancias de la clase
- final pública doble MY_CONSTANT; // Declarar una variable constante específica para esta instancia de la clase (mejor se usa en un constructor que genera una constante diferente para cada instancia)

Examples

Usando static para declarar constantes

Como la palabra clave static se usa para acceder a los campos y métodos sin una clase instanciada, se puede usar para declarar constantes para usar en otras clases. Estas variables se mantendrán constantes en cada instanciación de la clase. Por convención, las variables static siempre son ALL_CAPS y usan guiones bajos en lugar de camel. ex:

```
 static E STATIC_VARIABLE_NAME
```

Como las constantes no pueden cambiar, la static también se puede usar con el modificador final :

Por ejemplo, para definir la constante matemática de pi:

```
public class MathUtilities {

    static final double PI = 3.14159265358

}
```

Que se puede usar en cualquier clase como una constante, por ejemplo:

```
public class MathCalculations {

    //Calculates the circumference of a circle
    public double calculateCircumference(double radius) {
        return (2 * radius * MathUtilities.PI);
    }

}
```

Usando estática con esto

Static proporciona un método o almacenamiento de variables que *no se* asigna para cada instancia de la clase. Más bien, la variable estática se comparte entre todos los miembros de la clase. Incidentalmente, tratar de tratar la variable estática como un miembro de la instancia de clase dará como resultado una advertencia:

```
public class Apple {
    public static int test;
    public int test2;
}

Apple a = new Apple();
```

```
a.test = 1; // Warning
Apple.test = 1; // OK
Apple.test2 = 1; // Illegal: test2 is not static
a.test2 = 1; // OK
```

Los métodos que se declaran estáticos se comportan de la misma manera, pero con una restricción adicional:

¡No puedes usar this palabra clave en ellos!

```
public class Pineapple {

    private static int numberOfSpikes;
    private int age;

    public static getNumberOfSpikes() {
        return this.numberOfSpikes; // This doesn't compile
    }


    public static getNumberOfSpikes() {
        return numberOfSpikes; // This compiles
    }

}
```

En general, es mejor declarar métodos genéricos que se aplican a diferentes instancias de una clase (como los métodos de clonación) static , mientras que los métodos como equals() no son estáticos. El main método de un programa Java es siempre estática, lo que significa que la palabra clave this no se puede utilizar dentro de main() .

Referencia a miembro no estático del contexto estático

Las variables y los métodos estáticos no forman parte de una instancia. Siempre habrá una sola copia de esa variable, sin importar cuántos objetos cree de una clase en particular.

Por ejemplo, es posible que desee tener una lista inmutable de constantes, sería una buena idea mantenerla estática e inicializarla solo una vez dentro de un método estático. Esto le daría un aumento significativo de rendimiento si está creando varias instancias de una clase en particular de forma regular.

Además, también puedes tener un bloque estático en una clase. Puede usarlo para asignar un valor predeterminado a una variable estática. Se ejecutan solo una vez cuando la clase se carga en la memoria.

La variable de instancia como el nombre sugiere depende de una instancia de un objeto en particular, viven para servir a sus caprichos. Puedes jugar con ellos durante un ciclo de vida particular de un objeto.

Todos los campos y métodos de una clase utilizados dentro de un método estático de esa clase deben ser estáticos o locales. Si intenta utilizar variables o métodos de instancia (no estáticos), su código no se compilará.

```
public class Week {
    static int daysOfTheWeek = 7; // static variable
    int dayOfTheWeek; // instance variable

    public static int getDaysLeftInWeek(){
        return Week.daysOfTheWeek-dayOfTheWeek; // this will cause errors
    }
```

```
    public int getDaysLeftInWeek(){
        return Week.daysOfTheWeek-dayOfTheWeek; // this is valid
    }

    public static int getDaysLeftInTheWeek(int today){
        return Week.daysOfTheWeek-today; // this is valid
    }

}
```

Lea Usando la palabra clave estática en línea:
https://riptutorial.com/es/java/topic/2253/usando-la-palabra-clave-estatica

Capítulo 180: Usando otros lenguajes de scripting en Java

Introducción

Java en sí mismo es un lenguaje extremadamente poderoso, pero su poder se puede extender aún más gracias a JSR223 (Solicitud de especificación de Java 223) que presenta un motor de script

Observaciones

La API de scripting de Java permite que los scripts externos interactúen con Java

La API de scripting puede permitir la interacción entre el script y java. Los lenguajes de scripting deben tener una implementación de Script Engine en el classpath.

Por defecto, JavaScript (también conocido como ECMAScript) es proporcionado por nashorn de forma predeterminada. Cada Script Engine tiene un contexto de script donde todas las variables, funciones y métodos se almacenan en enlaces. En ocasiones, es posible que desee utilizar varios contextos, ya que admiten la redirección de la salida a un Escritor con búfer y el error a otro.

Hay muchas otras bibliotecas de motor de script como Jython y JRuby. Mientras estén en la ruta de clase, puede evaluar el código.

Podemos usar enlaces para exponer variables en el script. Necesitamos varios enlaces en algunos casos, ya que la exposición de las variables al motor básicamente consiste en exponer las variables solo a ese motor, a veces es necesario exponer ciertas variables como el entorno del sistema y la ruta que es la misma para todos los motores del mismo tipo. En ese caso, requerimos un enlace que es un alcance global. Exponer las variables que lo exponen a todos los motores de script creados por el mismo EngineFactory

Examples

Evaluando un archivo javascript en modo de script de nashorn

```
public class JSEngine {

    /*
    * Note Nashorn is only available for Java-8 onwards
    * You can use rhino from ScriptEngineManager.getEngineByName("js");
    */

    ScriptEngine engine;
    ScriptContext context;
    public Bindings scope;

    // Initialize the Engine from its factory in scripting mode
    public JSEngine(){
        engine = new NashornScriptEngineFactory().getScriptEngine("-scripting");
        // Script context is an interface so we need an implementation of it
        context = new SimpleScriptContext();
        // Create bindings to expose variables into
        scope = engine.createBindings();
    }

    // Clear the bindings to remove the previous variables
    public void newBatch(){
        scope.clear();
    }

    public void execute(String file){
        try {
```

```
            // Get a buffered reader for input
            BufferedReader br = new BufferedReader(new FileReader(file));
            // Evaluate code, with input as bufferdReader
            engine.eval(br);
        } catch (FileNotFoundException ex) {
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        } catch (ScriptException ex) {
            // Script Exception is basically when there is an error in script
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    public void eval(String code){
        try {
            // Engine.eval basically treats any string as a line of code and evaluates it,
executes it
            engine.eval(code);
        } catch (ScriptException ex) {
            // Script Exception is basically when there is an error in script
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    // Apply the bindings to the context and set the engine's default context
    public void startBatch(int SCP){
        context.setBindings(scope, SCP);
        engine.setContext(context);
    }

    // We use the invocable interface to access methods from the script
    // Invocable is an optional interface, please check if your engine implements it
    public Invocable invocable(){
        return (Invocable)engine;
    }

}
```

Ahora el método principal

```
public static void main(String[] args) {
    JSEngine jse = new JSEngine();
    // Create a new batch probably unecessary
    jse.newBatch();
    // Expose variable x into script with value of hello world
    jse.scope.put("x", "hello world");
    // Apply the bindings and start the batch
    jse.startBatch(ScriptContext.ENGINE_SCOPE);
    // Evaluate the code
    jse.eval("print(x);");
}
```

Su salida debe ser similar a esta
hello world

Como se puede ver la variable expuesta x se ha impreso. Ahora probando con un archivo.

Aquí tenemos test.js

```
print(x);
function test(){
    print("hello test.js:test");
```

```
}
test();
```

Y el método principal actualizado.

```
public static void main(String[] args) {
    JSEngine jse = new JSEngine();
    // Create a new batch probably unecessary
    jse.newBatch();
    // Expose variable x into script with value of hello world
    jse.scope.put("x", "hello world");
    // Apply the bindings and start the batch
    jse.startBatch(ScriptContext.ENGINE_SCOPE);
    // Evaluate the code
    jse.execute("./test.js");
}
```

Suponiendo que test.js está en el mismo directorio que su aplicación, debería tener una salida similar a esta

```
hello world
hello test.js:test
```

Lea Usando otros lenguajes de scripting en Java en línea:
https://riptutorial.com/es/java/topic/9926/usando-otros-lenguajes-de-scripting-en-java

Capítulo 181: Usando ThreadPoolExecutor en aplicaciones MultiThreaded.

Introducción

Al crear una aplicación basada en datos y rendimiento, puede ser muy útil completar las tareas que requieren mucho tiempo de manera asíncrona y tener varias tareas ejecutándose simultáneamente. Este tema presentará el concepto de usar ThreadPoolExecutors para completar múltiples tareas de sincronización simultánea.

Examples

Realización de tareas asíncronas donde no se necesita un valor de retorno utilizando una instancia de clase ejecutable

Es posible que algunas aplicaciones deseen crear las llamadas tareas "Incendio y olvido" que pueden activarse periódicamente y no es necesario devolver ningún tipo de valor devuelto al finalizar la tarea asignada (por ejemplo, depuración de archivos temporales antiguos, registros rotativos, guardado automático) estado).

En este ejemplo, crearemos dos clases: una que implementa la interfaz Runnable y otra que contiene un método main ().

AsyncMaintenanceTaskCompleter.java

```
import lombok.extern.java.Log;

import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncMaintenanceTaskCompleter implements Runnable {
    private int taskNumber;

    public AsyncMaintenanceTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    public void run() {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("Task %d is sleeping for %d seconds", taskNumber,
timeout));
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));

        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
    }
}
```

AsyncExample1

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class AsyncExample1 {
    public static void main(String[] args){
        ExecutorService executorService = Executors.newCachedThreadPool();
```

```
        for(int i = 0; i < 10; i++){
            executorService.execute(new AsyncMaintenanceTaskCompleter(i));
        }
        executorService.shutdown();
    }
}
```

La ejecución de AsyncExample1.main () dio como resultado el siguiente resultado:

```
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 8 is sleeping for 18 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 6 is sleeping for 4 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 2 is sleeping for 6 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 3 is sleeping for 4 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 9 is sleeping for 14 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 4 is sleeping for 9 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 5 is sleeping for 10 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 0 is sleeping for 7 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 1 is sleeping for 9 seconds
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run
INFO: Task 7 is sleeping for 8 seconds
Dec 28, 2016 2:21:07 PM AsyncMaintenanceTaskCompleter run
INFO: Task 6 is done sleeping
Dec 28, 2016 2:21:07 PM AsyncMaintenanceTaskCompleter run
INFO: Task 3 is done sleeping
Dec 28, 2016 2:21:09 PM AsyncMaintenanceTaskCompleter run
INFO: Task 2 is done sleeping
Dec 28, 2016 2:21:10 PM AsyncMaintenanceTaskCompleter run
INFO: Task 0 is done sleeping
Dec 28, 2016 2:21:11 PM AsyncMaintenanceTaskCompleter run
INFO: Task 7 is done sleeping
Dec 28, 2016 2:21:12 PM AsyncMaintenanceTaskCompleter run
INFO: Task 4 is done sleeping
Dec 28, 2016 2:21:12 PM AsyncMaintenanceTaskCompleter run
INFO: Task 1 is done sleeping
Dec 28, 2016 2:21:13 PM AsyncMaintenanceTaskCompleter run
INFO: Task 5 is done sleeping
Dec 28, 2016 2:21:17 PM AsyncMaintenanceTaskCompleter run
INFO: Task 9 is done sleeping
Dec 28, 2016 2:21:21 PM AsyncMaintenanceTaskCompleter run
INFO: Task 8 is done sleeping

Process finished with exit code 0
```

Observaciones de la nota: hay varias cosas que se deben tener en cuenta en la salida anterior,

1. Las tareas no se ejecutaron en un orden predecible.
2. Dado que cada tarea estuvo durmiendo durante un tiempo (pseudo) aleatorio, no se completaron necesariamente en el orden en que se invocaron.

Realización de tareas asíncronas donde se necesita un valor de retorno utilizando una instancia de clase invocable

A menudo es necesario ejecutar una tarea de larga duración y utilizar el resultado de esa tarea una vez que se haya completado.

En este ejemplo, crearemos dos clases: una que implementa la interfaz <T> invocable (donde T es el tipo que queremos devolver) y otra que contiene un método main ().

AsyncValueTypeTaskCompleter.java

```
import lombok.extern.java.Log;

import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncValueTypeTaskCompleter implements Callable<Integer> {
    private int taskNumber;

    public AsyncValueTypeTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    @Override
    public Integer call() throws Exception {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("Task %d is sleeping", taskNumber));
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));

        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
        return timeout;
    }
}
```

AsyncExample2.java

```
import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

@Log
public class AsyncExample2 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<Integer>> futures = new ArrayList<>();
        for (int i = 0; i < 10; i++){
            Future<Integer> submittedFuture = executorService.submit(new
AsyncValueTypeTaskCompleter(i));
            futures.add(submittedFuture);
        }
        executorService.shutdown();
        while(!futures.isEmpty()){
            for(int j = 0; j < futures.size(); j++){
```

```
                Future<Integer> f = futures.get(j);
                if(f.isDone()){
                    try {
                        int timeout = f.get();
                        log.info(String.format("A task just completed after sleeping for %d
seconds", timeout));
                        futures.remove(f);
                    } catch (InterruptedException | ExecutionException e) {
                        log.warning(e.getMessage());
                    }
                }
            }
        }
    }
}
```

La ejecución de AsyncExample2.main () dio como resultado el siguiente resultado:

```
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 7 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 8 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 2 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 1 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 4 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 9 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 0 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 6 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 5 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 3 is sleeping
Dec 28, 2016 3:07:16 PM AsyncValueTypeTaskCompleter call
INFO: Task 8 is done sleeping
Dec 28, 2016 3:07:16 PM AsyncExample2 main
INFO: A task just completed after sleeping for 1 seconds
Dec 28, 2016 3:07:17 PM AsyncValueTypeTaskCompleter call
INFO: Task 2 is done sleeping
Dec 28, 2016 3:07:17 PM AsyncExample2 main
INFO: A task just completed after sleeping for 2 seconds
Dec 28, 2016 3:07:17 PM AsyncValueTypeTaskCompleter call
INFO: Task 9 is done sleeping
Dec 28, 2016 3:07:17 PM AsyncExample2 main
INFO: A task just completed after sleeping for 2 seconds
Dec 28, 2016 3:07:19 PM AsyncValueTypeTaskCompleter call
INFO: Task 3 is done sleeping
Dec 28, 2016 3:07:19 PM AsyncExample2 main
INFO: A task just completed after sleeping for 4 seconds
Dec 28, 2016 3:07:20 PM AsyncValueTypeTaskCompleter call
INFO: Task 0 is done sleeping
Dec 28, 2016 3:07:20 PM AsyncExample2 main
INFO: A task just completed after sleeping for 5 seconds
Dec 28, 2016 3:07:21 PM AsyncValueTypeTaskCompleter call
INFO: Task 5 is done sleeping
Dec 28, 2016 3:07:21 PM AsyncExample2 main
```

```
INFO: A task just completed after sleeping for 6 seconds
Dec 28, 2016 3:07:25 PM AsyncValueTypeTaskCompleter call
INFO: Task 1 is done sleeping
Dec 28, 2016 3:07:25 PM AsyncExample2 main
INFO: A task just completed after sleeping for 10 seconds
Dec 28, 2016 3:07:27 PM AsyncValueTypeTaskCompleter call
INFO: Task 6 is done sleeping
Dec 28, 2016 3:07:27 PM AsyncExample2 main
INFO: A task just completed after sleeping for 12 seconds
Dec 28, 2016 3:07:29 PM AsyncValueTypeTaskCompleter call
INFO: Task 7 is done sleeping
Dec 28, 2016 3:07:29 PM AsyncExample2 main
INFO: A task just completed after sleeping for 14 seconds
Dec 28, 2016 3:07:31 PM AsyncValueTypeTaskCompleter call
INFO: Task 4 is done sleeping
Dec 28, 2016 3:07:31 PM AsyncExample2 main
INFO: A task just completed after sleeping for 16 seconds
```

Observaciones de la Nota:

Hay varias cosas a tener en cuenta en la salida de arriba,

1. Cada llamada a ExecutorService.submit () devolvió una instancia de Future, que se almacenó en una lista para su uso posterior
2. Future contiene un método llamado isDone () que se puede usar para verificar si nuestra tarea se ha completado antes de intentar verificar su valor de retorno. Si se llama al método Future.get () en un futuro que aún no se ha realizado, se bloqueará el subproceso actual hasta que la tarea se haya completado, lo que posiblemente anulará muchos de los beneficios obtenidos al realizar la tarea de forma asíncrona.
3. Se llamó al método executorService.shutdown () antes de verificar los valores de retorno de los objetos Futuros. Esto no es obligatorio, pero se hizo de esta manera para demostrar que es posible. El método executorService.shutdown () no impide la finalización de tareas que ya se han enviado a ExecutorService, sino que evita que se agreguen nuevas tareas a la cola.

Definición de tareas asíncronas en línea usando Lambdas

Si bien un buen diseño de software a menudo maximiza la reutilización del código, a veces puede ser útil definir tareas asíncronas en línea en su código a través de expresiones Lambda para maximizar la legibilidad del código.

En este ejemplo, crearemos una sola clase que contenga un método main (). Dentro de este método, usaremos expresiones Lambda para crear y ejecutar instancias de Callable y Runnable <T>.

AsyncExample3.java

```
import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

@Log
public class AsyncExample3 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<Integer>> futures = new ArrayList<>();
        for(int i = 0; i < 5; i++){
            final int index = i;
            executorService.execute(() -> {
                int timeout = getTimeout();
```

```
                log.info(String.format("Runnable %d has been submitted and will sleep for %d
seconds", index, timeout));
                try {
                    TimeUnit.SECONDS.sleep(timeout);
                } catch (InterruptedException e) {
                    log.warning(e.getMessage());
                }
                log.info(String.format("Runnable %d has finished sleeping", index));
            });
            Future<Integer> submittedFuture = executorService.submit(() -> {
                int timeout = getTimeout();
                log.info(String.format("Callable %d will begin sleeping", index));
                try {
                    TimeUnit.SECONDS.sleep(timeout);
                } catch (InterruptedException e) {
                    log.warning(e.getMessage());
                }
                log.info(String.format("Callable %d is done sleeping", index));
                return timeout;
            });
            futures.add(submittedFuture);
        }
        executorService.shutdown();
        while(!futures.isEmpty()){
            for(int j = 0; j < futures.size(); j++){
                Future<Integer> f = futures.get(j);
                if(f.isDone()){
                    try {
                        int timeout = f.get();
                        log.info(String.format("A task just completed after sleeping for %d
seconds", timeout));
                        futures.remove(f);
                    } catch (InterruptedException | ExecutionException e) {
                        log.warning(e.getMessage());
                    }
                }
            }
        }
    }

    public static int getTimeout(){
        return ThreadLocalRandom.current().nextInt(1, 20);
    }
}
```

Observaciones de la Nota:

Hay varias cosas a tener en cuenta en la salida de arriba,

1. Las expresiones Lambda tienen acceso a variables y métodos que están disponibles para el ámbito en el que están definidas, pero todas las variables deben ser finales (o efectivamente finales) para su uso dentro de una expresión lambda.
2. No tenemos que especificar si nuestra expresión Lambda es Callable o Runnable <T> explícitamente, el tipo de retorno se deduce automáticamente por el tipo de retorno.

Lea Usando ThreadPoolExecutor en aplicaciones MultiThreaded. en línea: https://riptutorial.com/es/java/topic/8646/usando-threadpoolexecutor-en-aplicaciones-multithreaded-

Capítulo 182: Varargs (Argumento Variable)

Observaciones

Un argumento del método "varargs" permite a las personas que llaman de ese método especificar múltiples argumentos del tipo designado, cada uno como un argumento separado. Se especifica en la declaración del método mediante tres períodos ASCII (...) después del tipo base.

El método en sí recibe esos argumentos como una única matriz, cuyo tipo de elemento es el tipo del argumento varargs. La matriz se crea automáticamente (aunque las personas que llaman todavía pueden pasar una matriz explícita en lugar de pasar múltiples valores como argumentos de métodos separados).

Reglas para varargs:

1. Varargs debe ser el último argumento.
2. Sólo puede haber un Varargs en el método.

Debe seguir las reglas anteriores, de lo contrario el programa dará un error de compilación.

Examples

Especificando un parámetro varargs

```
void doSomething(String... strings) {
    for (String s : strings) {
        System.out.println(s);
    }
}
```

Los tres períodos posteriores al tipo de parámetro final indican que el argumento final se puede pasar como una matriz o como una secuencia de argumentos. Varargs solo se puede utilizar en la posición del argumento final.

Trabajando con los parámetros de Varargs

Utilizando varargs como parámetro para una definición de método, es posible pasar una matriz o una secuencia de argumentos. Si se pasa una secuencia de argumentos, se convierten automáticamente en una matriz.

Este ejemplo muestra una matriz y una secuencia de argumentos que se pasan al método printVarArgArray() , y cómo se tratan de manera idéntica en el código dentro del método:

```
public class VarArgs {

    // this method will print the entire contents of the parameter passed in

    void printVarArgArray(int... x) {
        for (int i = 0; i < x.length; i++) {
            System.out.print(x[i] + ",");
        }
    }

    public static void main(String args[]) {
        VarArgs obj = new VarArgs();

        //Using an array:
        int[] testArray = new int[]{10, 20};
        obj.printVarArgArray(testArray);
```

```
        System.out.println(" ");

        //Using a sequence of arguments
        obj.printVarArgArray(5, 6, 5, 8, 6, 31);
    }
}
```

Salida:

```
10,20,
5,6,5,8,6,31
```

Si define el método de esta manera, dará errores de tiempo de compilación.

```
void method(String... a, int... b , int c){} //Compile time error (multiple varargs )

void method(int... a, String b){} //Compile time error (varargs must be the last argument
```

Lea Varargs (Argumento Variable) en línea: https://riptutorial.com/es/java/topic/1948/varargs--argumento-variable-

Capítulo 183: Visibilidad (control de acceso a los miembros de una clase)

Sintaxis

- nombre de tipo público [= valor];
- nombre de tipo privado [= valor];
- nombre de tipo protegido [= valor];
- escriba nombre [= valor];
- nombre de la clase pública {
- nombre de la clase{

Observaciones

Desde el tutorial de Java :

Los modificadores de nivel de acceso determinan si otras clases pueden usar un campo en particular o invocar un método en particular. Hay dos niveles de control de acceso:

- En el nivel superior: public o *paquete privado* (sin modificador explícito).
- A nivel de miembro: public , private , protected o *paquete privado* (sin modificador explícito).

Una clase puede ser declarada con el modificador public , en cuyo caso esa clase es visible para todas las clases en cualquier lugar. Si una clase no tiene modificador (el valor predeterminado, también conocido como *paquete-privado*), es visible solo dentro de su propio paquete.

En el nivel de miembro, también puede usar el modificador public o ningún modificador (*paquete-privado*) al igual que con las clases de nivel superior, y con el mismo significado. Para los miembros, hay dos modificadores de acceso adicionales: private y protected . El modificador private especifica que solo se puede acceder al miembro en su propia clase. El modificador protected especifica que solo se puede acceder al miembro dentro de su propio paquete (como con el *paquete privado*) y, además, mediante una subclase de su clase en otro paquete.

La siguiente tabla muestra el acceso a los miembros permitido por cada modificador.

Niveles de acceso:

Modificador	Clase	Paquete	Subclase	Mundo
public	Y	Y	Y	Y
protected	Y	Y	Y	norte
sin modificador	Y	Y	norte	norte
private	Y	norte	norte	norte

Examples

Miembros de la interfaz

```
public interface MyInterface {
    public void foo();
    int bar();
```

```
    public String TEXT = "Hello";
    int ANSWER = 42;

    public class X {
    }

    class Y {
    }
}
```

Los miembros de la interfaz siempre tienen visibilidad pública, incluso si se omite public palabra clave public . Entonces, tanto foo() , bar() , TEXT , ANSWER , X e Y tienen visibilidad pública. Sin embargo, el acceso aún puede estar limitado por la interfaz que contiene, ya que MyInterface tiene visibilidad pública, se puede acceder a sus miembros desde cualquier lugar, pero si MyInterface hubiera tenido visibilidad del paquete, sus miembros solo serían accesibles desde el mismo paquete.

Visibilidad pública

Visible para la clase, paquete y subclase.

Veamos un ejemplo con la prueba de clase.

```
public class Test{
    public int number = 2;

    public Test(){

    }
}
```

Ahora intentemos crear una instancia de la clase. En este ejemplo, **podemos** acceder al number porque es public .

```
public class Other{

    public static void main(String[] args){
        Test t = new Test();
        System.out.println(t.number);
    }

}
```

Visibilidad privada

private visibilidad private permite que una variable solo sea accesible por su clase. A menudo se utilizan en **conjunto** con captadores y setters public .

```
class SomeClass {
    private int variable;

    public int getVariable() {
        return variable;
    }

    public void setVariable(int variable) {
        this.variable = variable;
    }
```

```
}

public class SomeOtherClass {
    public static void main(String[] args) {
        SomeClass sc = new SomeClass();

        // These statement won't compile because SomeClass#variable is private:
        sc.variable = 7;
        System.out.println(sc.variable);

        // Instead, you should use the public getter and setter:
        sc.setVariable(7);
        System.out.println(sc.getVariable());
    }
}
```

Visibilidad del paquete

Sin **modificador** , el valor predeterminado es la visibilidad del paquete. *De la documentación de Java,* "[visibilidad del paquete] indica si las clases en el mismo paquete que la clase (independientemente de su origen) tienen acceso al miembro". En este ejemplo de javax.swing ,

```
package javax.swing;
public abstract class JComponent extends Container … {
    …
    static boolean DEBUG_GRAPHICS_LOADED;
    …
}
```

DebugGraphics está en el mismo paquete, por lo que DEBUG_GRAPHICS_LOADED es accesible.

```
package javax.swing;
public class DebugGraphics extends Graphics {
    …
    static {
        JComponent.DEBUG_GRAPHICS_LOADED = true;
    }
    …
}
```

Este artículo da algunos antecedentes sobre el tema.

Visibilidad Protegida

Causas de visibilidad protegida significa que este miembro es visible para su paquete, junto con cualquiera de sus subclases.

Como ejemplo:

```
package com.stackexchange.docs;
public class MyClass{
    protected int variable; //This is the variable that we are trying to access
    public MyClass(){
        variable = 2;
    };
}
```

Ahora extenderemos esta clase e intentaremos acceder a uno de sus miembros protected .

```
package some.other.pack;
import com.stackexchange.docs.MyClass;
public class SubClass extends MyClass{
    public SubClass(){
        super();
        System.out.println(super.variable);
    }
}
```

También podría acceder a un miembro protected sin extenderlo si accede desde el mismo paquete.

Tenga en cuenta que este modificador solo funciona en miembros de una clase, no en la clase en sí.

Resumen de los Modificadores de Acceso de los Miembros de la Clase

Modificador de acceso	Visibilidad	Herencia
Privado	Solo clase	No puede ser heredado
Sin modificador / paquete	En paquete	Disponible si subclase en paquete
Protegido	En paquete	Disponible en subclase
Público	En todos lados	Disponible en subclase

Hubo una vez un modificador private protected (ambas palabras clave a la vez) que podría aplicarse a métodos o variables para hacerlos accesibles desde una subclase fuera del paquete, pero hacerlos privados a las clases en ese paquete. Sin embargo, esto fue eliminado en la versión de Java 1.0 .

Lea Visibilidad (control de acceso a los miembros de una clase) en línea: https://riptutorial.com/es/java/topic/134/visibilidad--control-de-acceso-a-los-miembros-de-una-clase-

Capítulo 184: XJC

Introducción

XJC es una herramienta de Java SE que compila un archivo de esquema XML en clases Java completamente anotadas.

Se distribuye dentro del paquete JDK y se encuentra en la ruta /bin/xjc .

Sintaxis

- xjc [opciones] archivo de esquema / URL / dir / jar ... [-b bindinfo] ...

Parámetros

Parámetro	Detalles
archivo de esquema	El archivo de esquema xsd para convertir a java

Observaciones

La herramienta XJC está disponible como parte del JDK. Permite crear un código java anotado con anotaciones JAXB adecuadas para (des) ordenación.

Examples

Generando código Java desde un simple archivo XSD

Esquema XSD (schema.xsd)

El siguiente esquema xml (xsd) define una lista de usuarios con el name y la reputation atributos.

```
<?xml version="1.0"?>

<xs:schema version="1.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:ns="http://www.stackoverflow.com/users"
           elementFormDefault="qualified"
           targetNamespace="http://www.stackoverflow.com/users">
    <xs:element name="users" type="ns:Users"/>

    <xs:complexType name="Users">
        <xs:sequence>
            <xs:element type="ns:User" name="user" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="User">
        <xs:attribute name="name" use="required" type="xs:string"/>
        <xs:attribute name="reputation" use="required">
            <xs:simpleType>
                <xs:restriction base="xs:int">
                    <xs:minInclusive value="1"/>
                </xs:restriction>
```

```
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
</xs:schema>
```

Usando xjc

Esto requiere que la ruta a la herramienta xjc (binarios JDK) esté en la variable de ruta del sistema operativo.

La generación de código se puede iniciar utilizando

```
xjc schema.xsd
```

Esto generará archivos java en el directorio de trabajo.

Archivos de resultados

Habrá algunos comentarios adicionales, pero básicamente los archivos Java generados se ven así:

```
package com.stackoverflow.users;

import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Users", propOrder = {
    "user"
})
public class Users {

    protected List<User> user;

    public List<User> getUser() {
        if (user == null) {
            user = new ArrayList<User>();
        }
        return this.user;
    }

}
```

```
package com.stackoverflow.users;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "User")
public class User {

    @XmlAttribute(name = "name", required = true)
    protected String name;
    @XmlAttribute(name = "reputation", required = true)
```

```
    protected int reputation;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public int getReputation() {
        return reputation;
    }

    public void setReputation(int value) {
        this.reputation = value;
    }

}
```

```
package com.stackoverflow.users;

import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.bind.annotation.XmlRegistry;
import javax.xml.namespace.QName;

@XmlRegistry
public class ObjectFactory {

    private final static QName _Users_QNAME = new QName("http://www.stackoverflow.com/users",
"users");

    public ObjectFactory() {
    }

    public Users createUsers() {
        return new Users();
    }

    public User createUser() {
        return new User();
    }

    @XmlElementDecl(namespace = "http://www.stackoverflow.com/users", name = "users")
    public JAXBElement<Users> createUsers(Users value) {
        return new JAXBElement<Users>(_Users_QNAME, Users.class, null, value);
    }

}
```

package-info.java

```
@javax.xml.bind.annotation.XmlSchema(namespace = "http://www.stackoverflow.com/users",
elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
package com.stackoverflow.users;
```

Lea XJC en línea: https://riptutorial.com/es/java/topic/4538/xjc

Capítulo 185: XOM - Modelo de objetos XML

Examples

Leyendo un archivo XML

Para cargar los datos XML con XOM , tendrá que crear un Builder desde el cual puede construirlo en un Document .

```
Builder builder = new Builder();
Document doc = builder.build(file);
```

Para obtener el elemento raíz, el padre primario más alto en el archivo xml, necesita usar getRootElement() en la instancia del Document .

```
Element root = doc.getRootElement();
```

Ahora, la clase Element tiene muchos métodos prácticos que hacen que la lectura de xml sea realmente fácil. Algunos de los más útiles se enumeran a continuación:

- getChildElements(String name) : devuelve una instancia de Elements que actúa como una matriz de elementos
- getFirstChildElement(String name) : devuelve el primer elemento secundario con esa etiqueta.
- getValue() : devuelve el valor dentro del elemento.
- getAttributeValue(String name) : devuelve el valor de un atributo con el nombre especificado.

Cuando llama a getChildElements() obtiene una instancia de Elements . A partir de esto, puede recorrer y llamar al método get(int index) para recuperar todos los elementos que contiene.

```
Elements colors = root.getChildElements("color");
for (int q = 0; q < colors.size(); q++){
    Element color = colors.get(q);
}
```

Ejemplo: Aquí hay un ejemplo de leer un archivo XML:

Archivo XML:

```
<example>
    <person>
        <name>
            <first>Dan</first>
            <last>Smith</last>
        </name>
        <age unit="years">23</age>
        <fav_color>green</fav_color>
    </person>
    <person>
        <name>
            <first>Bob</first>
            <last>Autry</last>
        </name>
        <age unit="months">3</age>
        <fav_color>N/A</fav_color>
    </person>
</example>
```

Código para leerlo e imprimirlo:

```
import java.io.File;
import java.io.IOException;
import nu.xom.Builder;
import nu.xom.Document;
import nu.xom.Element;
import nu.xom.Elements;
import nu.xom.ParsingException;

public class XMLReader {

    public static void main(String[] args) throws ParsingException, IOException{
        File file = new File("insert path here");
        // builder builds xml data
        Builder builder = new Builder();
        Document doc = builder.build(file);

        // get the root element <example>
        Element root = doc.getRootElement();

        // gets all element with tag <person>
        Elements people = root.getChildElements("person");

        for (int q = 0; q < people.size(); q++){
            // get the current person element
            Element person = people.get(q);

            // get the name element and its children: first and last
            Element nameElement = person.getFirstChildElement("name");
            Element firstNameElement = nameElement.getFirstChildElement("first");
            Element lastNameElement = nameElement.getFirstChildElement("last");

            // get the age element
            Element ageElement = person.getFirstChildElement("age");
```

```
            // get the favorite color element
            Element favColorElement = person.getFirstChildElement("fav_color");

            String fName, lName, ageUnit, favColor;
            int age;

            try {
                fName = firstNameElement.getValue();
                lName = lastNameElement.getValue();
                age = Integer.parseInt(ageElement.getValue());
                ageUnit = ageElement.getAttributeValue("unit");
                favColor = favColorElement.getValue();

                System.out.println("Name: " + lName + ", " + fName);
                System.out.println("Age: " + age + " (" + ageUnit + ")");
                System.out.println("Favorite Color: " + favColor);
                System.out.println("----------------");

            } catch (NullPointerException ex){
                ex.printStackTrace();
            } catch (NumberFormatException ex){
                ex.printStackTrace();
            }
        }
    }

}
```

Esto se imprimirá en la consola:

```
Name: Smith, Dan
Age: 23 (years)
Favorite Color: green
----------------
Name: Autry, Bob
Age: 3 (months)
Favorite Color: N/A
----------------
```

Escribir en un archivo XML

Escribir en un archivo XML utilizando XOM es muy similar a leerlo, excepto que en este caso estamos creando instancias en lugar de recuperarlas de la raíz.

Para hacer un nuevo elemento use el `Element(String name)` constructor `Element(String name)` . Querrá crear un elemento raíz para poder agregarlo fácilmente a un `Document` .

```
Element root = new Element("root");
```

La clase `Element` tiene algunos métodos prácticos para editar elementos. Se enumeran a continuación:

- `appendChild(String name)` - esto básicamente establecerá el valor del elemento a nombre.
- `appendChild(Node node)` : esto hará que el node el elemento principal. (Los elementos son nodos para que pueda analizar elementos).
- `addAttribute(Attribute attribute)` : agregará un atributo al elemento.

La clase `Attribute` tiene un par de constructores diferentes. El más simple es el `Attribute(String name, String value)` .

Una vez que haya agregado todos sus elementos a su elemento raíz, puede convertirlo en un Document . Document tomará un Element como argumento en su constructor.

Puede usar un Serializer para escribir su XML en un archivo. Deberá crear un nuevo flujo de salida para analizar en el constructor de Serializer .

```
FileOutputStream fileOutputStream = new FileOutputStream(file);
Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
serializer.setIndent(4);
serializer.write(doc);
```

Ejemplo

Código:

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import nu.xom.Attribute;
import nu.xom.Builder;
import nu.xom.Document;
import nu.xom.Element;
import nu.xom.Elements;
import nu.xom.ParsingException;
import nu.xom.Serializer;

public class XMLWriter{

    public static void main(String[] args) throws UnsupportedEncodingException,
            IOException{
        // root element <example>
        Element root = new Element("example");

        // make a array of people to store
        Person[] people = {new Person("Smith", "Dan", "years", "green", 23),
            new Person("Autry", "Bob", "months", "N/A", 3)};

        // add all the people
        for (Person person : people){

            // make the main person element <person>
            Element personElement = new Element("person");

            // make the name element and it's children: first and last
            Element nameElement = new Element("name");
            Element firstNameElement = new Element("first");
            Element lastNameElement = new Element("last");

            // make age element
            Element ageElement = new Element("age");

            // make favorite color element
            Element favColorElement = new Element("fav_color");

            // add value to names
            firstNameElement.appendChild(person.getFirstName());
            lastNameElement.appendChild(person.getLastName());

            // add names to name
            nameElement.appendChild(firstNameElement);
```

```
            nameElement.appendChild(lastNameElement);

            // add value to age
            ageElement.appendChild(String.valueOf(person.getAge()));

            // add unit attribute to age
            ageElement.addAttribute(new Attribute("unit", person.getAgeUnit()));

            // add value to favColor
            favColorElement.appendChild(person.getFavoriteColor());

            // add all contents to person
            personElement.appendChild(nameElement);
            personElement.appendChild(ageElement);
            personElement.appendChild(favColorElement);

            // add person to root
            root.appendChild(personElement);
        }

        // create doc off of root
        Document doc = new Document(root);

        // the file it will be stored in
        File file = new File("out.xml");
        if (!file.exists()){
            file.createNewFile();
        }

        // get a file output stream ready
        FileOutputStream fileOutputStream = new FileOutputStream(file);

        // use the serializer class to write it all
        Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
        serializer.setIndent(4);
        serializer.write(doc);
    }

    private static class Person {

        private String lName, fName, ageUnit, favColor;
        private int age;

        public Person(String lName, String fName, String ageUnit, String favColor, int age){
            this.lName = lName;
            this.fName = fName;
            this.age = age;
            this.ageUnit = ageUnit;
            this.favColor = favColor;
        }

        public String getLastName() { return lName; }
        public String getFirstName() { return fName; }
        public String getAgeUnit() { return ageUnit; }
        public String getFavoriteColor() { return favColor; }
        public int getAge() { return age; }
    }

}
```

Este será el contenido de "out.xml":

```xml
<?xml version="1.0" encoding="UTF-8"?>
<example>
    <person>
        <name>
            <first>Dan</first>
            <last>Smith</last>
        </name>
        <age unit="years">23</age>
        <fav_color>green</fav_color>
    </person>
    <person>
        <name>
            <first>Bob</first>
            <last>Autry</last>
        </name>
        <age unit="months">3</age>
        <fav_color>N/A</fav_color>
    </person>
</example>
```

Lea XOM - Modelo de objetos XML en línea: https://riptutorial.com/es/java/topic/5091/xom---modelo-de-objetos-xml

Creditos

S. No	Capítulos	Contributors
1	Empezando con el lenguaje Java	aa_oo, Aaqib Akhtar, abhinav, Abhishek Jain, Abob, acdcjunior, Adeel Ansari, adsalpha, AER, akhilsk, Akshit Soota, Alex A, alphaloop, altomnr, Amani Kilumanga, AndroidMechanic, Ani Menon , ankit dassor, Ankur Anand, antonio, Arkadiy, Ashish Ahuja, Ben Page, Blachshma, bpoiss, Burkhard, Carlton, Charlie H, Coffeehouse Coder, c□LDS□EED, Community, Configure, CraftedCart, dabansal, Daksh Gupta, Dan Hulme, Dan Morenus, DarkV1, David G. , David Grinberg, David Newcomb, DeepCoder, Do Nhu Vy, Draken, Durgpal Singh, Dushko Jovanovski, E_net4, Edvin Tenovimas, Emil Sierżęga, Emre Bolat, enrico.bacis, Eran, explv, fgb, Francesco Menzani, Functino, garg10may, Gautam Jose, GingerHead, Grzegorz Górkiewicz, iliketocode, ɹɯɐƃoʅ ɐɪzuɐʞ, intboolstring, ipsi, J F, James Taylor, Jason, JavaHopper, Javant, javydreamercsw, Jean Vitor, Jean-François Savard, Jeffrey Brett Coleman, Jeffrey Lin, Jens Schauder, John Fergus, John Riddick, John Slegers, Jojodmo, JonasCz, Jonathan, Jonny Henly, Jorn Vernee, kaartic, Lambda Ninja, LostAvatar, madx, Magisch, Makoto, manetsus, Marc, Mark Adelsberger, Maroun Maroun, Matt, Matt, mayojava, Mitch Talmadge, mnoronha, Mrunal Pagnis, Mukund B, Mureinik, NageN, Nathan Arthur, nevster, Nithanim, Nuri Tasdemir, nyarasha, ochi, OldMcDonald, Onur, Ortomala Lokni, OverCoder, P.J.Meisch, Pavneet_Singh, Petter Friberg, philnate, Phrancis, Pops, ppeterka, Přemysl Šťastný, Pritam Banerjee, Radek Postołowicz, Radouane ROUFID, Rafael Mello, Rakitić, Ram, RamenChef, rekire, René Link, Reut Sharabani, Richard Hamilton, Ronnie Wang, ronnyfm, Ross Drew, RotemDev, Ryan Hilbert, SachinSarawgi, Sanandrea, Sandeep Chatterjee, Sayakiss, ShivBuyya, Shoe, Siguza, solidcell, stackptr, Stephen C, Stephen Leppik, sudo, Sumurai8, Snađoшfa□, tbodt, The Coder, ThePhantomGamer, Thisaru Guruge, Thomas Gerot, ThomasThiebaud, ThunderStruct, tonirush, Tushar Mudgal, Unihedron, user1133275, user124993, uzaif, Vaibhav Jain, Vakerrian, vasili111, Victor Stafusa, Vin, VinayVeluri, Vogel612, vorburger, Wilson, worker_bee, Yash Jain, Yury Fedorov, Zachary David Saunders, Ze Rubeus
2	Acceso nativo de Java	Ezekiel Baniaga, Stephan, Stephen C
3	Afirmando	Jonathan, Makoto, rajah9, RamenChef, The Guy with The Hat, Uri Agassi
4	Agentes de Java	Display Name, mnoronha
5	Ajuste de rendimiento de Java	Gene Marin, jatanp, Stephen C, Vogel612
6	Análisis XML utilizando las API de JAXP	GPI
7	Anotaciones	Ad Infinitum, Alon .G., Andrei Maieras, Andrii Abramov, bruno, Conrad.Dean, Dariusz, Demon Coldmist, Drizzt321, Dushko Jovanovski, fabian, faraa, GhostCat, hd84335, Hendrik Ebbers, J Atkin, Jorn Vernee, Kapep, Malt, MasterBlaster, matt freake, Nolequen, Ortomala Lokni, Ram, shmosel, Stephen C, Umberto Raimondi, Vogel612, ΦXocę □ Переúра ツ

8	Apache Commons Lang	Jonathan Barbero
9	API de apilación	manouti
10	API de reflexión	Ali786, ArcticLord, Aurasphere, Blubberguy22, Bohemian, Christophe Weis, Drizzt321, fabian, hd84335, Joeri Hendrickx, Luan Nico, madx, Michael Myers, Onur, Petter Friberg, RamenChef , Ravindra babu, Squidward, Stephen C, Tony BenBrahim, Universal Electricity, ΦXocę Переúра ツ
11	AppDynamics y TIBCO BusinessWorks Instrumentation para una fácil integración	Alexandre Grimaud
12	Applets	ArcticLord, Enigo, MadProgrammer, ppeterka
13	Archivo I / O	Alper Fırat Kaya, Arthur, assylias, ata, Aurasphere, Burkhard, Conrad.Dean, Daniel M., Enigo, FlyingPiMonster, Gerald Mücke, Gubbel, Hay, hd84335, Jabir, James Jensen, Jason Sturges, Jordy Baylac, leaqui, mateuscb, MikaelF, Moshiour, Myridium, Nicktar, Peter Gordon, Petter Friberg, ppeterka, RAnders00, RobAu, rokonoid, Sampada, sebkur, ShivBuyya, Squidward, Stephen C, still_learning, Tilo, Tobias Friedinger, TuringTux, Will Hardwick-Smith
14	Archivos JAR Multi-Release	manouti
15	Arrays	3442, 416E64726577, A Boschman, A.M.K, A_Arnold, Abhishek Jain, Abubakkar, acdcjunior, Ad Infinitum, Addis, Adrian Krebs, AER, afzalex, agilob, Alan, Alex Shesterov, Alexandru, altomnr, Amani Kilumanga, Andrew Tobilko, Andrii Abramov, AndroidMechanic, Anil, ankidaemon, ankit dassor, anotherGatsby, antonio, Ares, Arthur, Ashish Ahuja, assylias, AstroCB, baao, Beggs, Berzerk, Big Fan, BitNinja, bjb568, Blubberguy22, Bob Rivers, bpoiss, Bryan, BudsNanKis, Burkhard, bwegs, c1phr, Cache Staheli, Cerbrus, Charitha, Charlie H, Chris Midgley, Christophe Weis, Christopher Schneider, Codebender, coder-croc, Cold Fire, Colin Pickard, Community, Confiqure, CptEric, Daniel Käfer, Daniel Stradowski, Dariusz, DarkV1, David G., DeepCoder, Devid Farinelli, Dhrubajyoti Gogoi, Dmitry Ginzburg, dorukayhan , Duh-Wayne-101, Durgpal Singh, DVarga, Ed Cottrell, Edvin Tenovimas, Eilit, eisbehr, Elad, Emil Sierżęga, Emre Bolat, Eng.Fouad, enrico.bacis, Eran, Erik Minarini, Etki, explv, fabian, fedorqui, Filip Haglund, Forest White, fracz, Franck Dernoncourt, Functino, futureelite7, Gal Dreiman, gar, Gene Marin, GingerHead, granmirupa, Grexis, Grzegorz Sancewicz, Gubbel, Guilherme Torres Castro, Gustavo Coelho, hhj8i, Hiren, Idos, ihatecsv, iliketocode, Ilya, Ilyas Mimouni, intboolstring , Irfan, J Atkin, jabbathehutt1234, JakeD, James Taylor, Jamie, Jamie Rees, Janez Kuhar, Jared Rummler, Jargonius, Jason Sturges, JavaHopper, Javant, Jeeter, Jeffrey Bosboom, Jens Schauder, Jérémie Bolduc, Jeutnarg, jhnance, Jim Garrison, jitendra varshney, jmattheis, Joffrey, Johannes, johannes_preiser, John Slegers, JohnB, Jojodmo, Jonathan, Jordi Castilla, Jorn, Jorn Vernee, Josh, JStef, JudgingNotJudging, Justin, Kapep, KartikKannapur, Kayathiri, Kaz Wolfe, Kenster, Kevin Thorne, Lambda Ninja, Liju Thomas, llamositopia, Loris Securo, Luan Nico, Lucas Paolillo, maciek, Magisch, Makoto, Makyen, Malt, Marc, Markus, Marvin, MasterBlaster, Matas Vaitkevicius, matsve, Matt, Matt, Matthias Braun, Maxim

		Kreschishin, Maxim Plevako, Maximillian Laumeister, MC Emperor, Menasheh, Michael Piefel, michaelbahr, Miljen Mikic, Minhas Kamal, Mitch Talmadge, Mohamed Fadhl, Muhammed Refaat, Muntasir, Mureinik, Mzzzzzz, NageN, Nathaniel Ford, Nayuki, nicael, Nigel Nop, niyasc, noɥʇʎɗʎzɐɹƆ, Nuri Tasdemir, Ocracoke, OldMcDonald, Onur, orccrusher99, Ortomala Lokni, Panda, Paolo Forgia, Paul Bellora, Paweł Albecki, PeerNet, Peter Gordon, phatfingers, Pimgd, Piyush, ppeterka, Přemysl Šťastný, PSN, Pujan Srivastava, QoP, Radiodef, Radouane ROUFID, Raidri, Rajesh, Rakitić, Ram, RamenChef, Ravi Chandra, René Link, Reut Sharabani, Richard Hamilton, Robert Columbia, rolfedh, rolve, Roman Cherepanov, roottraveller, Ross, Ryan Hilbert, Sam Hazleton, sandbo00, Saurabh, Sayakiss, sebkur, Sergii Bishyr, sevenforce, shmosel, Shoe, Siguza, Simulant, Slayther, Smi, solidcell, Spencer Wieczorek, Squidward, stackptr, stark, Stephen C, Stephen Leppik, Sualeh Fatehi, sudo, Sumurai8, Sunnyok, syb0rg, tbodt, tdelev, tharkay, Thomas, ThunderStruct, Tol182, ʇolɐəz əɥʇ qoq, tpunt, Travis J, Tunaki, Un3qual, Unihedron, user6653173, uzaif, vasili111, VedX, Ven, Victor G., Vikas Gupta, vincentvanjoe, Vogel612, Wilson, Winter, X.lophix, YCF_L, Yohanes Khosiawan □□□, yuku, Yury Fedorov, zamonier, Φ Xocę □ Пepeúpa ツ
16	Audio	Dac Saunders, Petter Friberg, RamenChef, TNT, tonirush, Tot Zam, Vogel612
17	Autoboxing	17slim, Anony-Mousse, Bob Rivers, Chuck Daniels, cshubhamrao, fabian, hd84335, J Atkin, janos, kaartic, Kirill Sokolov, Luan Nico, Nayuki, piyush_baderia, Ram, RamenChef, Saagar Jha, Stephen C, Unihedron, Vladimir Vagaytsev
18	Banderas JVM	Confiqure, RamenChef
19	BigDecimal	alain.janinm, Christian, Dth, Enigo, ggolding, Harish Gyanani, John Nash, Loris Securo, Łukasz Piaszczyk, Manish Kothari, mszymborski, RamenChef, sudo, xwoker
20	BigInteger	Alek Mieczkowski, Alex Shesterov, Amani Kilumanga, Andrii Abramov, azurefrog, Byte1518, dimo414, dorukayhan, Emil Sierżęga, fabian, GPI, Ha., hd84335, janos, Kaushal28, Maarten Bodewes, Makoto, matt freake, Md. Nasir Uddin Bhuiyan, Nufail, Pritam Banerjee, Ruslan Bes, ShivBuyya, Stendika, Vogel612
21	ByteBuffer	Community, Jon Ericson, Jorn Vernee, Tarık Yılmaz, Tomasz Bawor, victorantunes, Vogel612
22	Calendario y sus subclases	Bob Rivers, cdm, kann, Makoto, mnoronha, ppeterka, Ram, VGR
23	Características de Java SE 7	compuhosny, RamenChef
24	Características de Java SE 8	compuhosny, RamenChef, sun-solar-arrow
25	Cargadores de clases	FFY00, Flow, Holger, Makoto, Stephen C
26	Cifrado RSA	Dennis Kriechel, Drunix, iqbal_cs, Maarten Bodewes, Nicktar, Shog9
27	Clase - Reflexión de Java	gobes, KIRAN KUMAR MATAM

28	Clase de fecha	A_Arnold, alain.janinm, arcy, Bob Rivers, Christian Wilkie, explv, Jabir, Jean-Baptiste Yunès, John Smith, Matt Clark, Miles, NamshubWriter, Nicktar, Nishant123, Ph0bi4, ppeterka, Ralf Kleberhoff, Ram, skia.heliou, Squidward, Stephen C, Vinod Kumar Kashyap
29	Clase de objetos Métodos y constructor	A Boschman, Ad Infinitum, Andrii Abramov, Ani Menon, anuvab1911 , Arthur Noseda, augray, Brett Kail, Burkhard, CaffeineToCode, Chris Midgley, cricket_007, Dariusz, Elazar, Emil Sierżęga, Enigo, fabian, fgb, Floern, fzzfzzfzz, hd84335, intboolstring, james large, JamesENL, Jens Schauder, John Slegers, Jorn Vernee , kstandell, Lahiru Ashan, Laurel, Miljen Mikic, mnoronha, mykey, NageN, Nayuki, Nicktar, Pace, Petter Friberg, Radouane ROUFID, Ram, Robert Columbia, Ronnie Wang, shmosel, Stephen C, TNT
30	Clase de propiedades	17slim, Arthur, J Atkin, Jabir, KIRAN KUMAR MATAM, Marvin, peterh, Stephen C, VGR, vorburger
31	Clase EnumSet	KIRAN KUMAR MATAM
32	Clase inmutable	Mykola Yashchenko
33	Clase interna local	KIRAN KUMAR MATAM
34	Clases anidadas e internas	ChemicalFlash, DimaSan, fgb, hd84335, Mshnik, RamenChef, Sandesh, sargue, Slava Babin, Stephen C, tynn
35	Clases y objetos	Community, Confiqure, Daniel LIn, Dave Ranjan, EJP, eveysky, fabian, Jens Schauder, Kevin Johnson, KIRAN KUMAR MATAM, MasterBlaster, Mureinik, Rakitić, Ram, RamenChef, Ryan Cocuzzo, Salman Kazmi, Tyler Zika
36	Clonación de objetos	Ayush Bansal, Christophe Weis, Jonathan
37	Codificación de caracteres	Ilya
38	Código oficial de Oracle estándar	Ahmed Ashour, aioobe, akhilsk, alex s, Andrii Abramov, Cassio Mazzochi Molin, Dan Whitehouse, Enigo, erickson, f_puras, fabian, giucal, hd84335, J.D. Sandifer, Lahiru Ashan, Mac70, NamshubWriter, Nicktar, Petter Friberg, Pradatta, Pritam Banerjee, RamenChef, sanjaykumar81, Santa Claus, Santhosh Ramanan, VGR
39	Colas y deques	Ad Infinitum, Alek Mieczkowski, Androbin, DimaSan, engineercoding, ppeterka, RamenChef, rd22, Samk, Stephen C
40	Colecciones	4castle, A_Arnold, Ad Infinitum, Alek Mieczkowski, alex s, altomnr, Andy Thomas, Anony-Mousse, Ashok Felix, Aurasphere, Bob Rivers, ced-b, ChandrasekarG, Chirag Parmar, clinomaniac, Codebender, Craig Gidney, Daniel Stradowski, dcod, DimaSan, Dušan Rychnovský, Enigo, Eran, fabian, fgb, GPI, Grzegorz Górkiewicz, ionyx, Jabir, Jan Vladimir Mostert, KartikKannapur, Kenster, KIRAN KUMAR MATAM, koder23, KudzieChase, Makoto, Maroun Maroun, Martin Frank, Matsemann, Mike H, Mo.Ashfaq, Mrunal Pagnis, mystarrocks, Oleg Sklyar, Pablo, Paweł Albecki, Petter Friberg, philnate, Polostor, Poonam, Powerlord, ppeterka , Prasad Reddy, Radiodef, rajadilipkolli, rd22, rdonuk, Ruslan Bes, Samk, SjB, Squidward, Stephen C, Stephen Leppik, Unihedron , user2296600, user3105453, Vasiliy Vlasov, Vasily Kabunov, VatsalSura, vsminkov, webo80, xploreraj

41	Colecciones alternativas	mnoronha, ppeterka, Viacheslav Vedenin
42	Colecciones concurrentes	GPI, Kenster, Powerlord, user2296600
43	Comandos de tiempo de ejecución	RamenChef
44	Comparable y comparador	Andrii Abramov, Conrad.Dean, Daniel Nugent, fabian, GPI, Hazem Farahat, JAVAC, Mshnik, Nolequen, Petter Friberg, Prateek Agarwal, sebkur, Stephen C
45	Comparación de C ++	John DiFini
46	Compilador de Java - 'javac'	CraftedCart, Jatin Balodhi, Mark Stewart, nishizawa23, Stephen C, Snađošfa□, Tom Gijselinck
47	CompletableFuture	Adowrath, Kishore Tulsiani, WillShackleford
48	Conjuntos	A_Arnold, atom, ced-b, Chirag Parmar, Daniel Stradowski, demongolem, DimaSan, fabian, Kaushal28, Kenster
49	Constructores	Andrii Abramov, Asiat, BrunoDM, ced-b, Codebender, Dylan, George Bailey, Jeremy, Ralf Kleberhoff, RamenChef, Thomas Gerot , tynn, Vogel612
50	Convertir hacia y desde cuerdas	Chirag Parmar, DarkV1, Gihan Chathuranga, Jabir, JonasCz, Kaushal28, Lachlan Dowding, Laurel, Maarten Bodewes, Matt Clark , PSo, RamenChef, Shaan, Stephen C, still_learning
51	Corrientes	4castle, Abubakkar, acdcjunior, Aimee Borda, Akshit Soota, Amitay Stern, Andrew Tobilko, Andrii Abramov, ArsenArsen, Bart Kummel, berko, Blubberguy22, bpoiss, Brendan B, Burkhard, Cerbrus, Charlie H, Claudio, Community, Conrad.Dean, Constantine, Daniel Käfer, Daniel M., Daniel Stradowski, Dariusz, David G., DonyorM, Dth, Durgpal Singh, Dushko Jovanovski, DVarga, dwursteisen, Eirik Lygre, enrico.bacis, Eran, explv, Fildor, Gal Dreiman, gontard, GreenGiant, Grzegorz Oledzki, Hank D, Hulk, iliketocode, ItachiUchiha, izikovic, J Atkin, Jamie Rees, JavaHopper, Jean-François Savard, John Slegers, Jon Erickson, Jonathan, Jorn Vernee, Jude Niroshan, JudgingNotJudging, Justin, Kapep, Kip, LisaMM, Makoto, Malt, malteo, Marc, MasterBlaster, Matt, Matt, Matt S., Matthieu, Michael Piefel, MikeW, Mitch Talmadge, Mureinik, Muto, Naresh Kumar, Nathaniel Ford, Nuri Tasdemir, OldMcDonald, Oleg L., omiel, Ortomala Lokni, Pawan, Paweł Albecki, Petter Friberg, Philipp Wendler, philnate, Pirate_Jack, ppeterka, Radnyx, Radouane ROUFID, Rajesh Kumar, Rakitić, RamenChef, Ranadip Dutta, ravthiru, reto, Reut Sharabani, RobAu, Robin, Roland Illig, Ronnie Wang, rrampage, RudolphEst, sargue, Sergii Bishyr , sevenforce, Shailesh Kumar Dayananda, shmosel, Shoe, solidcell, Spina, Squidward, SRJ, stackptr, stark, Stefan Dollase, Stephen C, Stephen Leppik, Steve K, Sugan, suj1th, thiagogcm, tpunt, Tunaki, Unihedron, user1133275, user1803551, Valentino, vincentvanjoe, vsnyc, Wilson, Ze Rubeus, zwl
52	Creando imágenes programáticamente	alain.janinm, Dariusz, kajacx, Kenster, mnoronha
53	Desmontaje y Descompilación.	ipsi, mnoronha

54	Despliegue de Java	garg10may, nishizawa23, Pseudonym Patel, RamenChef, Smit, Stephen C
55	Dividir una cadena en partes de longitud fija	Bohemian
56	Documentando el código de Java	Blubberguy22, Burkhard, Caleb Brinkman, Carter Brainerd, Community, Do Nhu Vy, Emil Sierżęga, George Bailey, Gerald Mücke, hd84335, ipsi, Kevin Thorne, Martijn Woudstra, Mitch Talmadge, Nagesh Lakinepally, PizzaFrog, Radouane ROUFID, RamenChef, sargue, Stephan, Stephen C, Trevor Sears, Universal Electricity
57	E / S de consola	Aaron Franke, Ani Menon, Erkan Haspulat, Francesco Menzani, jayantS, Lankymart, Loris Securo, manetsus, Olivier Grégoire, Petter Friberg, rolve, Saagar Jha, Stephen C
58	Ediciones, versiones, lanzamientos y distribuciones de Java	Gal Dreiman, screab, Stephen C
59	Ejecutor, ExecutorService y grupos de subprocesos	Andrii Abramov, Cache Staheli, Fildor, hd84335, Jens Schauder, JonasCz, noscreenname, Olivier Grégoire, philnate, Ravindra babu, Shettyh, Stephen C, Suminda Sirinath S. Dharmasena, sv3k, tones, user1121883, Vlad-HC, Vogel612
60	El classpath	Aaron Digulla, GPI, K'', Kenster, Ruslan Ulanov, Stephen C, trashgod
61	El Comando de Java - 'java' y 'javaw'	4444, Ben, mnoronha, Stephen C, Vogel612
62	Encapsulacion	Adam Ratzman, Adil, Daniel M., Drayke, VISHWANATH N P
63	Enchufes	Ordiel
64	Enum a partir del número	Sugan
65	Enums	1d0m3n30, A Boschman, aioobe, Amani Kilumanga, Andreas Fester, Andrew Sklyarevsky, Andrew Tobilko, Andrii Abramov, Anony-Mousse, bcosynot, Bob Rivers, coder-croc, Community, Constantine, Daniel Käfer, Daniel M., Danilo Guimaraes, DVarga, Emil Sierżęga, enrico.bacis, f_puras, fabian, Gal Dreiman, Gene Marin, Grexis, Grzegorz Oledzki, ipsi, J Atkin, Jared Hooper, javac, Jérémie Bolduc, Johannes, Jon Ericson, k3b, Kenster, Lahiru Ashan, Maarten Bodewes, madx, Mark, Michael Myers, Mick Mnemonic, NageN, Nef10, Nolequen, OldCurmudgeon, OliPro007, OverCoder, P.J.Meisch, Panther, Paweł Albecki, Petter Friberg, Punika, Radouane ROUFID, RamenChef, rd22, Ronon Dex, Ryan Hilbert, S.K. Venkat, Samk, shmosel, Spina, Stephen Leppik, Tarun Maganti, Tim, Torsten, VGR, Victor G., Vinay , Wolf, Yury Fedorov, Zefick, ΦXocę Переúра ツ
66	Errores comunes de Java	akvyalkov, Anand Vaidya, Andy Thomas, Anton Hlinisty, anuvab1911, Conrad.Dean, Daniel Nugent, Dushko Jovanovski, Enwired, Gal Dreiman, Gerald Mücke, HTNW, james large, Jenny T-Type, John Starich, Lahiru Ashan, Makoto, Morgan Zhang, NamshubWriter, P.J.Meisch, Pirate_Jack, ppeterka, RamenChef, screab, Siva Sankar Rajendran, Squidward, Stephen C, Stephen

		Leppik, Steve Harris, tonirush, TuringTux, user3105453
67	Errores de Java - Nulls y NullPointerException	17slim, Andrii Abramov, Daniel Nugent, dorukayhan, fabian, François Cassin, Miles, Stephen C, Zircon
68	Errores de Java - Problemas de rendimiento	Dorian, GPI, John Starich, Jorn Vernee, Michał Rybak, mnoronha, ppeterka, Sharon Rozinsky, steffen, Stephen C, xTrollxDudex
69	Errores de Java - sintaxis de lenguaje	Alex T., Cody Gray, Enwired, Friederike, Gal Dreiman, hd84335, Hiren, Peter Rader, piyush_baderia, RamenChef, Ravindra HV, RudolphEst, Stephen C, Todd Sewell, user3105453
70	Errores de Java - Uso de excepciones	Bhoomika, bruno, dimo414, Gal Dreiman, hd84335, SachinSarawgi, scorpp, Stephen C, Stephen Leppik, user3105453
71	Escáner	Alek Mieczkowski, Chirag Parmar, Community, Jon Ericson, JonasCz, Ram, RamenChef, Redterd, Stephen C, sun-solar-arrow, Φ Xocę □ Переúра ツ
72	Escogiendo Colecciones	John DiFini
73	Escritor Buffered	Andrii Abramov, fabian, Jorn Vernee, Robin, VatsalSura
74	Estructuras de control básicas	Adrian Krebs, AJNeufeld, Andrew Brooke, AshanPerera, Buddy, Caleb Brinkman, Cas Eliëns, Coffeehouse Coder, CraftedCart, dedmass, ebo, fabian, intboolstring, Inzimam Tariq IT, Jens Schauder, JonasCz, Jorn Vernee, juergen d, Makoto, Matt Champion, philnate, Ram, Santhosh Ramanan, sevenforce, Stephen C, teek, Unihedron, Uri Agassi, xwoker
75	Evaluación XML XPath	17slim, manouti
76	Examen de la unidad	Ironcache
77	Excepciones y manejo de excepciones.	Adrian Krebs, agilob, akhilsk, Andrii Abramov, Bhavik Patel, Burkhard, Cache Staheli, Codebender, Dariusz, DarkV1, dimo414, Draken, EAX, Emil Sierżęga, enrico.bacis, fabian, FMC, Gal Dreiman, GreenGiant, Hernanibus, hexafraction, Ilya, intboolstring, Jabir, James Jensen, JavaHopper, Jens Schauder, John Nash, John Slegers, JonasCz, Kai, Kevin Thorne, Malt, Manish Kothari, Md. Nasir Uddin Bhuiyan, michaelbahr, Miljen Mikic, Mitch Talmadge, Mrunal Pagnis, Myridium, mzc, Nikita Kurtin, Oleg Sklyar, P.J.Meisch, Paweł Albecki, Peter Gordon, Petter Friberg, ppeterka, Radek Postołowicz, Radouane ROUFID, Raj, RamenChef, rdonuk, Renukaradhya, RobAu, sandbo00, Saša Šijak, sharif.io, Stephen C, Stephen Leppik, still_learning, Sudhir Singh, sv3k, tatoalo, Thomas Fritsch, Tripta Kiroula, vic-3, Vogel612, Wilson, yiwei
78	Expresiones	1d0m3n30, Andreas, EJP, Li357, RamenChef, shmosel, Stephen C, Stephen Leppik
79	Expresiones lambda	Abhishek Jain, Ad Infinitum, Adam, aioobe, Amit Gupta, Andrei Maieras, Andrew Tobilko, Andrii Abramov, Ankit Katiyar, Anony-Mousse, assylias, Brian Goetz, Burkhard, Conrad.Dean, cringe, Daniel M., David Soroko, dimitrisli, Draken, DVarga, Emre Bolat , enrico.bacis, fabian, fgb, Gal Dreiman, gar, GPI, Hank D, hexafraction, Ivan Vergiliev, J Atkin, Jean-François Savard, Jeroen Vandevelde, John Slegers, JonasCz, Jorn Vernee, Jude

		Niroshan, JudgingNotJudging, Kevin Raoofi, Malt, Mark Green, Matt, Matthew Trout, Matthias Braun, ncmathsadist, nobeh, Ortomala Lokni, Paŭlo Ebermann, Paweł Albecki, Petter Friberg, philnate, Pujan Srivastava, Radouane ROUFID, RamenChef, rolve, Saclyr Barlonium, Sergii Bishyr, Skylar Sutton, solomonope, Stephen C, Stephen Leppik, timbooo, Tunaki, Unihedron, vincentvanjoe, Vlasec, Vogel612, webo80, William Ritson, Wolfgang, Xaerxess, xploreraj, Yogi, Ze Rubeus
80	Expresiones regulares	Amani Kilumanga, Andy Thomas, Asaph, ced-b, Daniel M., fabian, hd84335, intboolstring, kaotikmynd, Laurel, Makoto, nhahtdh, ppeterka, Ram, RamenChef, Saif, Tot Zam, Unihedron, Vogel612
81	Fechas y hora (java.time. *)	Bilbo Baggins, bowmore, Michael Piefel, Miles, mnoronha, Simon, Squidward, Tarun Maganti, Vogel612, ΦXocę □ Переúра ツ
82	FileUpload a AWS	Amit Gujarathi
83	Formato numérico	arpit pandey, John Nash, RamenChef, ΦXocę □ Переúра ツ
84	FTP (Protocolo de transferencia de archivos)	Kelvin Kellner
85	Generación de números aleatorios	Arthur, David Grant, David Soroko, dorukayhan, F. Stephen Q, Kichiin, MasterBlaster, michaelbahr, rokonoid, Stephen C, Thodgnir
86	Generando Código Java	Tony
87	Genéricos	1d0m3n30, 4444, Aaron Digulla, Abhishek Jain, Alex Meiburg, alex s, Andrei Maieras, Andrii Abramov, Anony-Mousse, Bart Enkelaar, bitek, Blubberguy22, Bob Brinks, Burkhard, Cache Staheli, Cannon, Ce7, Chriss, code11, Codebender, Daniel Figueroa, daphshez, DVarga, Emil Sierżęga, enrico.bacis, Eran, faraa, hd84335, hexafraction, Jan Vladimir Mostert, Jens Schauder, Jorn Vernee, Jude Niroshan, kcoppock, Kevin Montrose, Lahiru Ashan, Lii, manfcas, Mani Muthusamy, Marc, Matt, Mistalis, Mshnik, mvd, Mzzzzzz, NatNgs, nishizawa23, Oleg Sklyar, Onur, Ortomala Lokni, paisanco, Paul Bellora, Paweł Albecki, PcAF, Petter Friberg, phant0m, philnate, Radouane ROUFID, RamenChef, rap-2-h, rd22, Rogério, rolve, RutledgePaulV , S.K. Venkat, Siguza, Stephen C, Stephen Leppik, suj1th, tainy , ThePhantomGamer, Thomas, TNT, ʇolɐəz əɥʇ qoq, Unihedron, Vlad-HC, Wesley, Wilson, yiwei, Yury Fedorov
88	Gerente de seguridad	alphaloop, hexafraction, Uux
89	Gestión de memoria de Java	Daniel M., engineercoding, fgb, John Nash, jwd630, mnoronha, OverCoder, padippist, RamenChef, Squidward, Stephen C
90	Gráficos 2D en Java	17slim, ABDUL KHALIQ
91	Hechiceros y Setters	Fildor, Ironcache, Kröw, martin, Petter Friberg, Stephen C, Sujith Niraikulathan, Thisaru Guruge, uzaif
92	Herencia	Ad Infinitum, Adam, Adrian Krebs, agoeb, Ali Dehghani, Andrii Abramov, ar4ers, Arkadiy, Blubberguy22, Bohemian, Brad Larson, Burkhard, CodeCore, coder-croc, Dariusz, David Grinberg, devnull69, DonyorM, DVarga, Emre Bolat, explv, fabian, gattsbr, geniushkg, GhostCat, Gubbel, hirosht, HON95, J Atkin, Jason V, JavaHopper, Jeffrey Bosboom, Jens Schauder, Jonathan, Jorn

		Vernee, Kai, Kevin DiTraglia, kiuby_88, Lahiru Ashan, Luan Nico , maheshkumar, Mshnik, Muhammed Refaat, OldMcDonald, Oleg Sklyar, Ortomala Lokni, PM 77-1, Prateek Agarwal, QoP, Radouane ROUFID, RamenChef, Ravindra babu, Shog9, Simulant, SjB, Slava Babin, Stephen C, Stephen Leppik, still_learning, Sudhir Singh, Theo, ToTheMaximum, uhrm, Unihedron, Vasiliy Vlasov, Vucko
93	Hora local	100rabh, A_Arnold, Alex, Andrii Abramov, Bob Rivers, Cache Staheli, DimaSan, Jasper, Kakarot, Kuroda, Manuel Vieda, Michael Piefel, phatfingers, RamenChef, Skylar Sutton, Vivek Anoop
94	HttpURLConnection	Community, Datagrammar, EJP, Inzimam Tariq IT, JonasCz, kiedysktos, Mureinik, NageN, Stephen C, still_learning
95	Implementaciones del sistema de plugin Java	Alexiy
96	InputStreams y OutputStreams	akgren_soar, EJP, Gubbel, J Atkin, Jens Schauder, John Nash, Kip, KIRAN KUMAR MATAM, Matt Clark, Michael, RamenChef, Stephen C, Vogel612
97	Instalando Java (Edición Estándar)	4444, Adeel Ansari, ajablonski, akhilsk, Alex A, altomnr, Ani Menon, Anthony Raymond, anuvab1911, Confiqure, CraftedCart, Emil Sierżęga, Gautam Jose, hd84335, ipsi, Jeffrey Brett Coleman, Lambda Ninja, Nithanim, Radouane ROUFID, Rakitić, ronnyfm, Sanandrea, Sandeep Chatterjee, sohnryang, Stephen C, S нађошfa, tonirush, Walery Strauch, Ze Rubeus
98	Instrumentos de cuerda	17slim, A.J. Brown, A_Arnold, Abhishek Jain, Abubakkar, Adam Ratzman, Adrian Krebs, agilob, Aiden Deom, Alex Meiburg, Alex Shesterov, altomnr, Amani Kilumanga, Andrew Tobilko, Andrii Abramov, Andy Thomas, Anony-Mousse, Asaph, Ataeraxia, Austin, Austin Day, ben75, bfd, Bob Brinks, bpoiss, Burkhard, Cache Staheli, Caner Balım, Chris Midgley, Christian, Christophe Weis , coder-croc, Community, cyberscientist, Daniel Käfer, Daniel Stradowski, DarkV1, dedmass, DeepCoder, dnup1092, dorukayhan, drov, DVarga, ekeith, Emil Sierżęga, emotionlessbananas, enrico.bacis, Enwired, fabian, FlyingPiMonster, Gabriele Mariotti, Gal Dreiman, Gergely Toth, Gihan Chathuranga, GingerHead, giucal, Gray, GreenGiant, hamena314, Harish Gyanani , HON95, iliketocode, Ilya, Infuzed guy, intboolstring, J Atkin , Jabir, javac, JavaHopper, Jeffrey Lin, Jens Schauder, Jérémie Bolduc, John Slegers, Jojodmo, Jon Ericson, JonasCz, Jordi Castilla, Jorn Vernee, JSON C11, Jude Niroshan, Kamil Akhuseyinoglu, Kapep, Kaushal28, Kaushik NP, Kehinde Adedamola Shittu, Kenster, kstandell, Lachlan Dowding, Lahiru Ashan, Laurel, Leo Aso, Liju Thomas, LisaMM, M.Sianaki, Maarten Bodewes, Makoto, Malav, Malt, Manoj, Manuel Spigolon, Mark Stewart, Marvin, Matej Kormuth, Matt Clark, Matthias Braun, maxdev, Maxim Plevako, mayha, Michael, MikeW, Miles, Miljen Mikic, Misa Lazovic, mr5, Myridium, NikolaB, Nufail, Nuri Tasdemir, OldMcDonald, OliPro007, Onur, Optimiser, ozOli, P.J.Meisch, Paolo Forgia, Paweł Albecki, Petter Friberg, phant0m, piyush_baderia, ppeterka, Přemysl Šťastný, PSo, QoP, Radouane ROUFID, Raj, RamenChef, RAnders00, Rocherlee, Ronnie Wang, Ryan Hilbert, ryanyuyu, Sayakiss, SeeuD1, sevenforce, Shaan, ShivBuyya, Shoe, Sky, SmS, solidcell, Squidward, Stefan Isele - prefabware.com, stefanobaghino, Stephen C, Stephen Leppik, Steven Benitez, still_learning, Sudhir Singh, Swanand Pathak, Sнађошfa, TDG, TheLostMind, ThePhantomGamer, Tony

		BenBrahim, Unihedron, VGR, Vishal Biyani, Vogel612, vsminkov, vvtx, Wilson, winseybash, xwoker, yuku, Yury Fedorov, Zachary David Saunders, Zack Teater, Ze Rubeus, ΦXocę Переúра ツ
99	Interfaces	100rabh, A Boschman, Abhishek Jain, Adowrath, Alex Shesterov, Andrew Tobilko, Andrii Abramov, Cà phê đen, Chirag Parmar, Conrad.Dean, Daniel Käfer, devguy, DVarga, Hilikus, inovaovao, intboolstring, James Oswald, Jan Vladimir Mostert, JavaHopper, Johannes, Jojodmo, Jonathan, Jorn Vernee, Kai, kstandell, Laurel, Marvin, MikeW, Paul Nelson Baker, Peter Rader, ppovoski , Prateek Agarwal, Radouane ROUFID, RamenChef, Robin, Simulant, someoneigna, Stephen C, Stephen Leppik, Sujith Niraikulathan, Thomas Gerot, user187470, Vasiliy Vlasov, Vince Emigh, xwoker, Zircon
100	Interfaces funcionales	Andreas
101	Interfaz de herramientas JVM	desilijic
102	Interfaz de salida de cola	Suketu Patel
103	Interfaz fluida	bn., noscreenname, P.J.Meisch, RamenChef, TuringTux
104	Interfaz nativa de Java	Coffee Ninja, Fjoni Yzeiri, Jorn Vernee, RamenChef, Stephen C, user1803551
105	Invocación de método remoto (RMI)	RamenChef, smichel, Stephen C, user1803551, Vasiliy Vlasov
106	Iterador e iterable	Abubakkar, Comic Sans, Dariusz, Hulk, Lukas Knuth, RamenChef, Stephen C, user1121883, WillShackleford
107	JavaBean	foxt7ot, J. Pichardo, James Fry, SaWo, Stephen C
108	JAXB	Dariusz, Drunix, fabian, hd84335, Jabir, ppeterka, Ram, Stephan , Thomas Fritsch, vallismortis, Walery Strauch
109	JAX-WS	ext1812, Jonathan Barbero, Stephen Leppik
110	JMX	esin88
111	JNDI	EJP, neohope, RamenChef
112	JShell	ostrichofevil, Sudip Bhandari
113	JSON en Java	Asaph, Bogdan Korinnyi, Burkhard, Cache Staheli, hd84335, ipsi, Jared Hooper, Kurzalead, MikaelF, Mrunal Pagnis, Nicholas J Panella, Nikita Kurtin, ppeterka, Prem Singh Bist, RamenChef, Ray Kiddy, SirKometa, still_learning, Stoyan Dekov, systemfreund, Tim, Vikas Gupta, vsminkov, Yury Fedorov
114	Just in Time (JIT) compilador	Liju Thomas, Stephen C
115	La clase java.util.Objects	mnoronha, RamenChef, Stephen C
116	Las trampas de Java - Hilos y concurrencia	dorukayhan, james large, Stephen C

117	Lectores y escritores	JD9999, KIRAN KUMAR MATAM, Mureinik, Stephen C, VatsalSura
118	LinkedHashMap	Amit Gujarathi, KIRAN KUMAR MATAM
119	Lista vs SET	KIRAN KUMAR MATAM
120	Literales	1d0m3n30, EJP, ParkerHalo, Stephen C, ThePhantomGamer
121	Liza	17slim, A Boschman, Arthur, Avinash Kumar Yadav, Blubberguy22, ced-b, Daniel Nugent, granmirupa, Ilya, Jan Vladimir Mostert, janos, JD9999, jopasserat, Karthikeyan Vaithilingam, Kenster, Krzysztof Krasoń, Oleg Sklyar, RamenChef, Sheshnath, Stephen C, sudo, Thisaru Guruge, Vasilis Vasilatos, ΦXocę Перейра ツ
122	Localización e internacionalización.	Code.IT, dimo414, Eduard Wirch, emotionlessbananas, Squidward, sun-solar-arrow
123	log4j / log4j2	Daniel Wild, Fildor, HCarrasko, hd84335, Mrunal Pagnis, Rens van der Heijden
124	Los operadores	17slim, 1d0m3n30, A Boschman, acdcjunior, afuc func, AJ Jwair, Amani Kilumanga, Andreas, Andrew, Andrii Abramov, Blake Yarbrough, Blubberguy22, Bobas_Pett, c.uent, Cache Staheli, Chris Midgley, Claudia, clinomaniac, Dariusz, Darth Shadow, Davis, EJP, Emil Sierżęga, Eran, fabian, FedeWar, FlyingPiMonster, futureelite7, Harsh Vakharia, hd84335, J Atkin , JavaHopper, Jérémie Bolduc, jimrm, Jojodmo, Jorn Vernee, kanhaiya agarwal, Kevin Thorne, Li357, Loris Securo, Lynx Brutal, Maarten Bodewes, Mac70, Makoto, Marvin, Michael Anderson, Mshnik, NageN, Nuri Tasdemir, Ortomala Lokni, OverCoder, ParkerHalo, Peter Gordon, ppeterka, qxz, rahul tyagi , RamenChef, Ravan, Reut Sharabani, Rubén, sargue, Sean Owen, ShivBuyya, shmosel, SnoringFrog, Stephen C, tonirush, user3105453, Vogel612, Winter
125	Manipulación de bits	Aimee Borda, Blubberguy22, dosdebug, esin88, Gerald Mücke, Jorn Vernee, Kineolyan, mnoronha, Nayuki, Rednivrug, Ryan Hilbert, Stephen C, thatguy
126	Mapa débil	Amit Gujarathi, KIRAN KUMAR MATAM
127	Mapa Enum	KIRAN KUMAR MATAM
128	Mapas	17slim, agilob, alain.janinm, ata, Binary Nerd, Burkhard, coobird, Dmitriy Kotov, Durgpal Singh, Emil Sierżęga, Emily Mabrey, Enigo, fabian, GPI, hd84335, J Atkin, Jabir, Javant, Javier Diaz, Jeffrey Bosboom, johnnyaug, Jonathan, Kakarot, KartikKannapur, Kenster, michaelbahr, Mo.Ashfaq, Nathaniel Ford , phatfingers, Ram, RamenChef, ravthiru, sebkur, Stephen C, Stephen Leppik, Viacheslav Vedenin, VISHWANATH N P, Vogel612
129	Máquina virtual de Java (JVM)	Dushman, RamenChef, Rory McCrossan, Stephen Leppik
130	Método dinámico de envío	Jeet
131	Métodos de la fábrica de recolección	Jacob G.
132	Métodos predeterminados	ar4ers, hd84335, intboolstring, javac, Jeffrey Bosboom, Jens Schauder, Kai, matt freake, o_nix, philnate, Ravindra HV,

		richersoon, Ruslan Bes, Stephen C, Stephen Leppik, Vasiliy Vlasov
133	Modelo de memoria de Java	Shree, Stephen C, Suminda Sirinath S. Dharmasena
134	Modificación Bytecode	bloo, Display Name, rakwaht, Squidward
135	Modificadores sin acceso	Ankit Katiyar, Arash, fabian, Florian Weimer, FlyingPiMonster, Grzegorz Górkiewicz, J-Alex, JavaHopper, Ken Y-N, KIRAN KUMAR MATAM, Miljen Mikic, NageN, Nuri Tasdemir, Onur, ppeterka, Prateek Agarwal
136	Módulos	Jonathan, user140547
137	Moneda y dinero	Alexey Lagunov
138	Motor de JavaScript Nashorn	ben75, ekaerovets, Francesco Menzani, hd84335, Ilya, InitializeSahib, kasperjj, VatsalSura
139	NIO - Redes	Matthieu, mnoronha
140	Nuevo archivo I / O	dorukayhan, niheno, TuringTux
141	Objetos inmutables	1d0m3n30, Bohemian, Holger, Idcmp, Jon Ericson, kristyna, Michael Piefel, Stephen C, Vogel612
142	Objetos seguros	Ankit Katiyar
143	Opcional	A Boschman, Abubakkar, Andrey Rubtsov, Andrii Abramov, assylias , bowmore, Charlie H, Chris H., Christophe Weis, compuhosny, Dair, Emil Sierżęga, enrico.bacis, fikovnik, Grzegorz Górkiewicz, gwintrob, Hadson, hd84335, hzpz, J Atkin, Jean-François Savard, John Slegers, Jude Niroshan, Maroun Maroun, Michael Wiles, OldMcDonald, shmosel, Squidward, Stefan Dollase, Stephen C, ultimate_guy, Unihedron, user140547, Vince, vsminkov , xwoker
144	Operaciones de punto flotante de Java	Dariusz, hd84335, HTNW, Ilya, Mr. P, Petter Friberg, ravthiru, Stephen C, Stephen Leppik, Vogel612
145	Paquetes	JamesENL, KIRAN KUMAR MATAM
146	Polimorfismo	Adrian Krebs, Amani Kilumanga, Daniel LIn, Dushman, Kakarot, Lernkurve, Markus L, NageN, Pawan, Ravindra babu, Saiful Azad, Stephen C
147	Preferencias	RAnders00
148	Procesamiento de argumentos de línea de comando	Burkhard, Michael von Wenckstern, Stephen C
149	Proceso	Andy Thomas, Bob Rivers, ppeterka, vorburger, yitzih
150	Programación Concurrente (Hilos)	adino, Alex, assylias, bfd, Bhagyashree Jog, bowmore, Burkhard, Chetya, corsiKa, Dariusz, Diane Chastain, DimaSan, dimo414, Fildor, Freddie Coleman, GPI, Grzegorz Górkiewicz, hd84335, hellrocker, hexafraction, Ilya, james large, Jens Schauder, Johannes, Jorn Vernee, Kakarot, Lance Clark, Malt, Matěj Kripner, Md. Nasir Uddin Bhuiyan, Michael Piefel, michaelbahr, Mitchell Tracy, MSB, Murat K., Mureinik, mvd, NatNgs,

		nickguletskii, Olivier Durin, OlivierTheOlive, Panda, parakmiakos, Paweł Albecki, ppeterka, RamenChef, Ravindra babu, rd22, RudolphEst, snowe2010, Squidward, Stephen C, Sudhir Singh , Tobias Friedinger, Unihedron, Vasiliy Vlasov, Vlad-HC, Vogel612, wolfcastle, xTrollxDudex, YCF_L, Yury Fedorov, ZX9
151	Programación paralela con el framework Fork / Join.	Community, Joe C
152	Publicación por entregas	akhilsk, Batty, Bilesh Ganguly, Burkhard, EJP, emotionlessbananas, faraa, GradAsso, KIRAN KUMAR MATAM, noscreenname, Onur, rokonoid, Siva Sainath Reddy Bandi, Vasilis Vasilatos, Vasiliy Vlasov
153	Puntos de referencia	esin88
154	Recursion	Andy Thomas, atom, Bobas_Pett, Ce7, charlesreid1, Confiqure, David Soroko, fabian, hamena314, hd84335, JavaHopper, Javant, Matej Kormuth, mayojava, Nicktar, Peter Gordon, RamenChef, Raviteja, Ruslan Bes, Stephen C, sumit
155	Recursos (en classpath)	Androbin, Christian, Emily Mabrey, Enwired, fabian, Gerald Mücke, Jesse van Bekkum, Kenster, Stephen C, timbooo, VGR, vorburger
156	Redes	Arthur, Burkhard, devnull69, DonyorM, glee8e, Grayson Croom, Ilya, Malt, Matej Kormuth, Matthieu, Mine_Stone, ppeterka, RamenChef, Stephen C, Tot Zam, vsav
157	Referencias de objetos	Andrii Abramov, arcy, Vasiliy Vlasov
158	Registro (java.util.logging)	bn., Christophe Weis, Emil Sierżęga, P.J.Meisch, vallismortis
159	Seguridad y criptografía	John Nash, shibli049
160	ServiceLoader	fabian, Florian Genser, Gerald Mücke
161	Servicio de impresión de Java	Danilo Guimaraes, Leonardo Pina
162	Singletons	aasu, Andrew Antipov, Daniel Käfer, Dave Ranjan, David Soroko, Emil Sierżęga, Enigo, fabian, Filip Smola, GreenGiant, Gubbel, Hulk, Jabir, Jens Schauder, JonasCz, Jonathan, JonK, Malt, Matsemann, Michael Lloyd Lee mlk, Mifeet, Miroslav Bradic, NamshubWriter, Pablo, Peter Rader, RamenChef, riyaz-ali, sanastasiadis, shmosel, Stefan Dollase, stefanobaghino, Stephen C, Stephen Leppik, still_learning, Uri Agassi, user3105453, Vasiliy Vlasov, Vlad-HC, Vogel612, xploreraj
163	Sockets de Java	Nikhil R
164	SortedMap	Amit Gujarathi
165	StringBuffer	Amit Gujarathi
166	StringBuilder	Andrii Abramov, Cache Staheli, David Soroko, Enigo, fabian, fgb , JudgingNotJudging, KIRAN KUMAR MATAM, Nicktar, P.J.Meisch, Stephen C

167	sun.misc.Unsafe	4444, Daniel Nugent, Grexis, Stephen C, Suminda Sirinath S. Dharmasena
168	súper palabra clave	Abhijeet
169	Tabla de picadillo	KIRAN KUMAR MATAM
170	ThreadLocal	Dariusz, Liju Thomas, Manish Kothari, Nithanim, taer
171	Tipo de conversión	4castle, Filip Smola, Joshua Carmody, Nick Donnelly, RamenChef, Squidward
172	Tipos atómicos	Daniel Nugent, Stephen C, Suminda Sirinath S. Dharmasena, xTrollxDudex
173	Tipos de datos de referencia	Do Nhu Vy, giucal, Jorn Vernee, Lord Farquaad, Yohanes Khosiawan □□□
174	Tipos de datos primitivos	17slim, 1d0m3n30, Amani Kilumanga, Ani Menon, Anony-Mousse, Bilesh Ganguly, Bob Rivers, Burkhard, Conrad.Dean, Daniel, Dariusz, DimaSan, dnup1092, Do Nhu Vy, enrico.bacis, fabian, Francesco Menzani, Francisco Guimaraes, gar, Ilya, IncrediApp, ipsi, J Atkin, JakeD, javac, Jean-François Savard, Jojodmo, Kapep, KdgDev, Lahiru Ashan, Master Azazel, Matt, mayojava, MBorsch, nimrod, Pang, Panther, ParkerHalo, Petter Friberg, Radek Postołowicz, Radouane ROUFID, RAnders00, RobAu, Robert Columbia, Simulant, Squidward, Stephen C, Stephen Leppik, Sundeep, SuperStormer, ThePhantomGamer, TMN, user1803551, user2314737, Veedrac, Vogel612
175	Tipos de referencia	EJP, NageN, Thisaru Guruge
176	Tokenizador de cuerdas	M M
177	TreeMap y TreeSet	Malt, Stephen C
178	Usando la palabra clave estática	17slim, Amir Rachum, Andrew Brooke, Arthur, ben75, CarManuel, Daniel Nugent, EJP, Hi I'm Frogatto, Mark Yisri, Sadiq Ali, Skepter, Squidward
179	Usando otros lenguajes de scripting en Java	Nikhil R
180	Usando ThreadPoolExecutor en aplicaciones MultiThreaded.	Brendon Dugan
181	Varargs (Argumento Variable)	Daniel Nugent, Dushman, Omar Ayala, Rafael Pacheco, RamenChef, VGR, xsami
182	Visibilidad (control de acceso a los miembros de una clase)	Aasmund Eldhuset, Abhishek Balaji R, Catalina Island, Daniel M. , intboolstring, Jonathan, Mark Yisri, Mureinik, NageN, ParkerHalo, Stephen C, Vogel612
183	XJC	Danilo Guimaraes, fabian
184	XOM - Modelo de objetos XML	Arthur, Makoto