EBook Gratuito

APPRENDIMENTO

Java Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#java

Sommario

Di

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.

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

Capitolo 1: Iniziare con Java Language

Osservazioni

Il linguaggio di programmazione Java è ...

- **Scopo generale** : è progettato per essere utilizzato per la scrittura di software in una vasta gamma di domini applicativi e privo di funzioni specializzate per qualsiasi dominio specifico.
- **Basato sulla classe** : la sua struttura dell'oggetto è definita in classi. Le istanze di classe hanno sempre quei campi e metodi specificati nelle loro definizioni di classe (vedi Classi e Oggetti). Questo è in contrasto con linguaggi non basati su classi come JavaScript.
- **Caratterizzato staticamente** : il compilatore verifica al momento della compilazione che i tipi di variabili siano rispettati. Ad esempio, se un metodo prevede un argomento di tipo `String` , tale argomento deve essere effettivamente una stringa quando viene chiamato il metodo.
- **Orientato agli oggetti** : la maggior parte delle cose in un programma Java sono istanze di classe, cioè bundle di stato (campi) e comportamento (metodi che operano su dati e formano l' *interfaccia* dell'oggetto con il mondo esterno).
- **Portatile** : può essere compilato su qualsiasi piattaforma con `javac` e i file di classe risultanti possono essere eseguiti su qualsiasi piattaforma dotata di JVM.

Java ha lo scopo di consentire agli sviluppatori di applicazioni di "scrivere una volta, eseguire ovunque" (WORA), il che significa che il codice Java compilato può essere eseguito su tutte le piattaforme che supportano Java senza necessità di ricompilazione.

Il codice Java è compilato in bytecode (i file `.class`) che a sua volta viene interpretato dalla Java Virtual Machine (JVM). In teoria, il bytecode creato da un compilatore Java dovrebbe funzionare allo stesso modo su qualsiasi JVM, anche su un diverso tipo di computer. La JVM potrebbe (e nei programmi del mondo reale) scegliere di compilare in comandi macchina nativi le parti del bytecode che vengono eseguite spesso. Questa è chiamata "compilazione Just-in-time (JIT)".

Edizioni Java e versioni

Esistono tre "edizioni" di Java definite da Sun / Oracle:

- *Java Standard Edition (SE)* è l'edizione progettata per uso generale.
- *Java Enterprise Edition (EE)* aggiunge una gamma di servizi per la creazione di servizi "di livello enterprise" in Java. Java EE è coperto separatamente .
- *Java Micro Edition (ME)* è basato su un sottoinsieme di *Java SE* ed è destinato all'uso su dispositivi di piccole dimensioni con risorse limitate.

Esiste un argomento separato sulle edizioni Java SE / EE / ME .

Ogni edizione ha più versioni. Le versioni di Java SE sono elencate di seguito.

Installazione di Java

C'è un argomento separato su Installazione di Java (Standard Edition) .

Compilazione ed esecuzione di programmi Java

Ci sono argomenti separati su:

- Compilazione del codice sorgente Java
- Implementazione di Java inclusa la creazione di file JAR
- Esecuzione di applicazioni Java
- Il percorso di classe

Qual'è il prossimo?

Ecco i collegamenti alle materie per continuare ad apprendere e comprendere il linguaggio di programmazione Java. Questi argomenti sono le basi della programmazione Java per iniziare.

- Tipi di dati primitivi in Java
- Operatori in Java
- Archi in Java
- Strutture di controllo di base in Java
- Classi e oggetti in Java
- Array in Java
- Standard di codice Java

analisi

Mentre Java non ha alcun supporto per i test nella libreria standard, esistono librerie di terze parti progettate per supportare il test. Le due librerie di test unità più popolari sono:

- JUnit (sito ufficiale)
- TestNG (sito ufficiale)

Altro

- I modelli di progettazione per Java sono trattati in Design Patterns .
- La programmazione per Android è coperta in Android .

- Le tecnologie Java Enterprise Edition sono trattate in Java EE .
- Le tecnologie Oracle JavaFX sono trattate in JavaFX .

1. Nella sezione **Versioni** la data di *fine vita (gratuita)* è quando Oracle interromperà la pubblicazione di ulteriori aggiornamenti di Java SE nei propri siti di download pubblici. I clienti che necessitano di un accesso continuo a correzioni di errori critici e correzioni di sicurezza, nonché una manutenzione generale per Java SE possono ottenere supporto a lungo termine tramite il supporto Oracle Java SE .

Versioni

Versione Java SE	Nome in codice	Fine vita (gratis [1])	Data di rilascio
Java SE 9 (accesso anticipato)	*Nessuna*	futuro	2017/07/27
Java SE 8	Ragno	futuro	2014/03/18
Java SE 7	Delfino	2015/04/14	2011-07-28
Java SE 6	Mustang	2013/04/16	2006-12-23
Java SE 5	Tigre	2009-11-04	2004-10-04
Java SE 1.4	smeriglio	prima del 2009-11-04	2002/02/06
Java SE 1.3	Gheppio	prima del 2009-11-04	2000/05/08
Java SE 1.2	Terreno di gioco	prima del 2009-11-04	1998/12/08
Java SE 1.1	*Nessuna*	prima del 2009-11-04	1997/02/19
Java SE 1.0	Quercia	prima del 2009-11-04	1996/01/21

Examples

Creare il tuo primo programma Java

Crea un nuovo file nell'editor di testo o IDE denominato `HelloWorld.java` . Quindi incolla questo blocco di codice nel file e salva:

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

Esegui live su Ideone

Nota: affinché Java riconosca questo come una `public class` (e non genera un errore di compilazione), il nome del file deve essere uguale al nome della classe (`HelloWorld` in questo

esempio) con un'estensione `.java` . Dovrebbe esserci anche un modificatore di accesso `public` prima di esso.

Le convenzioni di denominazione raccomandano che le classi Java inizino con un carattere maiuscolo e siano in formato cammello (in cui la prima lettera di ogni parola è in maiuscolo). Le convenzioni raccomandano rispetto ai caratteri di sottolineatura (`_`) e ai simboli del dollaro (`$`).

Per compilare, apri una finestra di terminale e vai alla directory di `HelloWorld.java` :

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

Nota: cd è il comando del terminale per cambiare directory.

Immettere `javac` seguito dal nome e dall'estensione del file come segue:

```
$ javac HelloWorld.java
```

È abbastanza comune ottenere l'errore `'javac' is not recognized as an internal or external command, operable program or batch file.` anche quando hai installato `JDK` e sei in grado di eseguire il programma da `IDE` ex. `eclipse` ecc. Poiché il percorso non viene aggiunto all'ambiente per impostazione predefinita.

Nel caso in cui si ottenga questo su Windows, per risolverlo, provare prima a navigare sul percorso di `javac.exe` , è molto probabilmente in `C:\Program Files\Java\jdk(version number)\bin` . Quindi prova a eseguirlo con sotto.

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

In precedenza, quando stavamo chiamando `javac` , era lo stesso del comando precedente. Solo in quel caso il tuo `OS` sapeva dove risiedeva `javac` . Quindi diciamolo ora, in questo modo non è necessario digitare l'intero percorso ogni volta. Dovremmo aggiungere questo al nostro `PATH`

Per modificare la variabile di ambiente `PATH` in Windows XP / Vista / 7/8/10:

- Pannello di controllo ⇒ Sistema ⇒ Impostazioni di sistema avanzate
- Passare alla scheda "Avanzate" ⇒ Variabili d'ambiente
- In "Variabili di sistema", scorrere verso il basso per selezionare "PERCORSO" ⇒ Modifica

Non puoi annullare questo, quindi fai attenzione. Prima copia il tuo percorso esistente sul blocco note. Quindi, per ottenere l'esatto PERCORSO sul tuo `javac` cerca manualmente nella cartella in cui risiede `javac` e fai clic sulla barra degli indirizzi e poi copialo. Dovrebbe sembrare qualcosa come `c:\Program Files\Java\jdk1.8.0_xx\bin`

Nel campo "Valore variabile", incolla questo **IN FRONTE** di tutte le directory esistenti, seguito da un punto e virgola (;). **NON CANCELLARE** le voci esistenti.

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

Ora questo dovrebbe risolversi.

Per i sistemi basati su Linux, prova qui .

***Nota:** il comando `javac` richiama il compilatore Java.*

Il compilatore genererà quindi un file bytecode chiamato `HelloWorld.class` che può essere eseguito nella Java Virtual Machine (JVM) . Il compilatore del linguaggio di programmazione Java, `javac` , legge i file sorgente scritti nel linguaggio di programmazione Java e li compila in file di classe `bytecode` . Facoltativamente, il compilatore può anche elaborare annotazioni trovate nei file di origine e di classe utilizzando l'API `Pluggable` Annotation Processing. Il compilatore è uno strumento da riga di comando, ma può anche essere richiamato utilizzando l'API Java Compiler.

Per eseguire il tuo programma, inserisci `java` seguito dal nome della classe che contiene il metodo `main` (`HelloWorld` nel nostro esempio). Nota come viene omessa la `.class` .

```
$ java HelloWorld
```

***Nota:** il comando `java` esegue un'applicazione Java.*

Questo verrà prodotto nella tua console:

> Ciao mondo!

Hai codificato e costruito con successo il tuo primo programma Java!

Nota: affinché i comandi Java (`java` , `javac` , ecc.) `javac` riconosciuti, è necessario accertarsi che:

- È installato un JDK (ad es. Oracle , OpenJDK e altre fonti)
- Le variabili di ambiente sono configurate correttamente

Dovrai utilizzare un compilatore (`javac`) e un executor (`java`) fornito da JVM. Per scoprire quali versioni sono state installate, immettere `java -version` e `javac -version` nella riga di comando. Il numero di versione del programma verrà stampato nel terminale (ad es. `1.8.0_73`).

Uno sguardo più da vicino al programma Hello World

Il programma "Hello World" contiene un singolo file, che consiste in una definizione di classe `HelloWorld` , un metodo `main` e un'istruzione all'interno del metodo `main` .

```
public class HelloWorld {
```

La `class` parola chiave inizia la definizione di classe per una classe denominata `HelloWorld` . Ogni applicazione Java contiene almeno una definizione di classe (Ulteriori informazioni sulle classi).

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

Questo è un metodo del punto di ingresso (definito dal nome e dalla firma del `public static void main(String[])`) da cui `JVM` può eseguire il programma. Ogni programma Java dovrebbe averne uno. È:

- `public` : il che significa che il metodo può essere chiamato da qualsiasi parte anche al di fuori del programma. Vedi Visibilità per maggiori informazioni su questo.
- `static` : significa che esiste e può essere eseguito da solo (a livello di classe senza creare un oggetto).
- `void` : il significato non restituisce alcun valore. ***Nota:*** *questo è diverso da C e C ++ in cui è previsto un codice di ritorno come* `int` *(il modo in cui Java è* `System.exit()` *).*

Questo metodo principale accetta:

- Una matrice (in genere chiamata `args`) di `String` s passata come argomenti alla funzione principale (ad esempio, dagli argomenti della riga di comando).

Quasi tutto ciò è richiesto per un metodo di punto di ingresso Java.

Parti non richieste:

- Il nome `args` è un nome variabile, quindi può essere chiamato qualsiasi cosa tu voglia, anche se in genere è chiamato `args` .
- Se il suo tipo di parametro è un array (`String[] args`) o Varargs (`String... args`) non ha importanza perché gli array possono essere passati in vararg.

Nota: una singola applicazione può avere più classi contenenti un metodo del punto di ingresso (`main`). Il punto di ingresso dell'applicazione è determinato dal nome della classe passato come argomento al comando `java` .

All'interno del metodo principale, vediamo la seguente dichiarazione:

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

Analizziamo questa affermazione elemento per elemento:

Elemento	Scopo
`System`	questo denota che l'espressione successiva invocherà la classe `System` , dal pacchetto `java.lang` .
`.`	questo è un "punto operatore". Gli operatori di punti forniscono l'accesso a membri di classi [1] ; cioè i suoi campi (variabili) e i suoi metodi. In questo caso, questo operatore punto consente di fare riferimento al `out` campo statico all'interno del `System` di classe.
`out`	questo è il nome del campo statico di tipo `PrintStream` all'interno della classe `System` contenente la funzionalità di output standard.

Elemento	Scopo
`.`	questo è un altro operatore punto. Questo operatore punto fornisce accesso al metodo `println` all'interno della variabile `out` .
`println`	questo è il nome di un metodo all'interno della classe PrintStream. Questo metodo in particolare stampa il contenuto dei parametri nella console e inserisce una nuova riga dopo.
`(`	questa parentesi indica che si sta accedendo a un metodo (e non a un campo) e inizia i parametri passati nel metodo `println` .
`"Hello, World!"`	questo è il letterale String che viene passato come parametro nel metodo `println` . Le doppie virgolette su ciascuna estremità delimitano il testo come una stringa.
`)`	questa parentesi significa la chiusura dei parametri passati nel metodo `println` .
`;`	questo punto e virgola segna la fine della dichiarazione.

Nota: *ogni istruzione in Java deve terminare con un punto e virgola (`;`).*

Il corpo del metodo e il corpo della classe vengono quindi chiusi.

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

Ecco un altro esempio che dimostra il paradigma OO. Modelliamo una squadra di calcio con un membro (sì, uno!). Ce ne possono essere di più, ma ne parleremo quando arriveremo agli array.

Per prima cosa, definiamo la nostra classe `Team` :

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

Ora, definiamo la nostra classe `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;
    }
}
```

Perché usiamo `private` qui? Beh, se qualcuno volesse sapere il tuo nome, dovrebbe chiederti direttamente, invece di prendere in tasca e tirare fuori la tua tessera di previdenza sociale. Questo `private` fa qualcosa del genere: impedisce alle entità esterne di accedere alle tue variabili. È possibile restituire solo membri `private` tramite le funzioni getter (mostrate sotto).

Dopo aver messo tutto insieme e aggiunto i getter e il metodo principale come discusso in precedenza, abbiamo:

```
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
    }
```

```
}
```

Produzione:

```
Aurieel
light
10
1
```

Corri su ideone

Ancora una volta, il metodo `main` all'interno della classe `Test` è il punto di ingresso al nostro programma. Senza il metodo `main` , non possiamo dire a Java Virtual Machine (JVM) da dove iniziare l'esecuzione del programma.

---

1 - Poiché la classe `HelloWorld` ha una scarsa relazione con la classe `System` , può solo accedere `public` dati `public` .

Leggi Iniziare con Java Language online: https://riptutorial.com/it/java/topic/84/iniziare-con-java-language

Capitolo 2: Accesso nativo Java

Examples

Introduzione a JNA

Cos'è JNA?

Java Native Access (JNA) è una libreria sviluppata dalla comunità che fornisce ai programmi Java un facile accesso alle librerie native native (file `.dll` su windows, file `.so` su Unix ...)

Come posso usarlo?

- In primo luogo, scarica l' ultima versione di JNA e fai riferimento a jna.jar nel CLASSPATH del tuo progetto.
- In secondo luogo, copia, compila ed esegui il codice Java qui sotto

  Ai fini di questa introduzione, supponiamo che la piattaforma nativa in uso sia Windows. Se stai `"msvcrt"` un'altra piattaforma, sostituisci semplicemente la stringa `"msvcrt"` con la stringa `"c"` nel codice qui sotto.

Il piccolo programma Java seguente stamperà un messaggio sulla console chiamando 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 !");
    }
}
```

Dove andare ora?

Salta su un altro argomento qui o vai al sito ufficiale .

Leggi Accesso nativo Java online: https://riptutorial.com/it/java/topic/5244/accesso-nativo-java

Capitolo 3: affermare

Sintassi

- asserire *espressione1* ;
- asser *expression1* : *expression2* ;

Parametri

Parametro	Dettagli
espressione1	L'affermazione assertion genera un `AssertionError` se questa espressione è `false` .
espressione2	Opzionale. Se utilizzato, `AssertionError` s generato `AssertionError` assert ha questo messaggio.

Osservazioni

Per impostazione predefinita, le asserzioni sono disabilitate in fase di runtime.

Per abilitare le asserzioni, è necessario eseguire java con flag `-ea` .

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

Le asserzioni sono affermazioni che generano un errore se la loro espressione è `false` . Le asserzioni dovrebbero essere utilizzate solo per *testare il* codice; non dovrebbero mai essere usati in produzione.

Examples

Controllo aritmetico con assert

```
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;
```

Leggi affermare online: https://riptutorial.com/it/java/topic/407/affermare

Capitolo 4: Agenti Java

Examples

Modifica delle classi con gli agenti

Innanzitutto, assicurati che l'agente in uso abbia i seguenti attributi in Manifest.mf:

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

L'avvio di un agente java consentirà all'agente di accedere alla Strumentazione della classe. Con Instrumentation è possibile chiamare *addTransformer (trasformatore ClassFileTransformer)* . ClassFileTransformers ti consente di riscrivere i byte delle classi. La classe ha un solo metodo che fornisce ClassLoader che carica la classe, il nome della classe, un'istanza java.lang.Class, è ProtectionDomain e infine i byte della classe stessa.

Sembra questo:

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

Modificare una classe puramente da byte può richiedere anni. Per ovviare a questo ci sono librerie che possono essere utilizzate per convertire i byte di classe in qualcosa di più utilizzabile.

In questo esempio userò ASM, ma altre alternative come Javassist e BCEL hanno caratteristiche simili.

```
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;
}
```

Da qui le modifiche possono essere apportate all'oggetto ClassNode. Ciò rende incredibilmente facile cambiare campo / metodo di accesso. Inoltre con ASM's Tree API modificare il bytecode dei metodi è un gioco da ragazzi.

Una volta terminate le modifiche, puoi convertire nuovamente il ClassNode in byte con il seguente metodo e restituirli nel metodo di *trasformazione* :

```
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;
}
```

Aggiunta di un agente in fase di runtime

Agenti possono essere aggiunti a una JVM in fase di runtime. Per caricare un agente è necessario utilizzare *VirtualMachine.attatch (ID stringa) di* Attach API. È quindi possibile caricare un jar compilato con il seguente metodo:

```
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);
    }
}
```

Questo non chiamerà *premain ((String agentArgs, Instrumentation inst)* nell'agente caricato, ma invece chiamerà *agentmain (String agentArgs, Instrumentation inst)* . Ciò richiede che *Agent-Class* sia impostata nell'agente Manifest.mf.

Impostazione di un agente di base

La classe Premain conterrà il metodo *"premain (String agentArgs Instrumentation inst)"*

Ecco un esempio:

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

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

Quando viene compilato in un file jar, apri il manifest e assicurati che abbia l'attributo Premain-Class.

Ecco un esempio:

```
Premain-Class: PremainExample
```

Per utilizzare l'agente con un altro programma java "myProgram", è necessario definire l'agente negli argomenti JVM:

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

Leggi Agenti Java online: https://riptutorial.com/it/java/topic/1265/agenti-java

Capitolo 5: Analisi XML usando le API JAXP

Osservazioni

L'analisi XML è l'interpretazione dei documenti XML al fine di manipolare il loro contenuto utilizzando costrutti sensibili, siano essi "nodi", "attributi", "documenti", "spazi dei nomi" o eventi correlati a questi costrutti.

Java ha un'API nativa per la gestione dei documenti XML, denominata JAXP o API Java per l'elaborazione XML . JAXP e un'implementazione di riferimento sono stati forniti in bundle con ogni versione Java da Java 1.4 (JAXP v1.1) e si sono evoluti da allora. Java 8 fornito con JAXP versione 1.6.

L'API offre diversi modi di interagire con i documenti XML, che sono:

- L'interfaccia DOM (Document Object Model)
- L'interfaccia SAX (Simple API for XML)
- L'interfaccia StAX (Streaming API for XML)

Principi dell'interfaccia DOM

L'interfaccia DOM mira a fornire un modo W3C DOM di interpretare XML. Varie versioni di JAXP hanno supportato vari Livelli DOM di specifica (fino al livello 3).

Sotto l'interfaccia Modello oggetto documento, un documento XML viene rappresentato come un albero, a partire da "Elemento documento". Il tipo di base dell'API è il tipo di `Node` , consente di navigare da un `Node` al relativo genitore, ai suoi figli o ai suoi fratelli (sebbene, non tutti i `Node` possano avere figli, ad esempio, i nodi di `Text` sono finali nell'albero, e non avere mai bambini). I tag XML sono rappresentati come `Element` s, che estendono notevolmente il `Node` con metodi relativi agli attributi.

L'interfaccia DOM è molto utile poiché consente l'analisi "a una riga" di documenti XML come alberi e consente di modificare facilmente l'albero costruito (aggiunta del nodo, soppressione, copia, ...) e infine la sua serializzazione (torna al disco) post modifiche. Questo ha un prezzo, però: l'albero risiede nella memoria, quindi gli alberi DOM non sono sempre pratici per enormi documenti XML. Inoltre, la costruzione dell'albero non è sempre il modo più rapido per gestire il contenuto XML, specialmente se non si è interessati a tutte le parti del documento XML.

Principi dell'interfaccia SAX

L'API SAX è un'API event-oriented per gestire documenti XML. Sotto questo modello, i componenti di un documento XML vengono interpretati come eventi (ad es. "Un tag è stato aperto", "un tag è stato chiuso", "è stato rilevato un nodo di testo", "è stato riscontrato un commento"). ..

L'API SAX utilizza un approccio "push parsing", in cui un `Parser` SAX è responsabile dell'interpretazione del documento XML e richiama i metodi su un delegato (un `ContentHandler`) per gestire qualsiasi evento venga trovato nel documento XML. Di solito, non si scrive mai un parser, ma si fornisce un gestore per raccogliere tutte le informazioni necessarie dal documento XML.

L'interfaccia SAX supera le limitazioni dell'interfaccia DOM mantenendo solo i dati minimi necessari a livello di parser (ad es. Contesti namespace, stato di validazione), quindi, solo le informazioni che sono tenute da `ContentHandler` - di cui tu, lo sviluppatore, è responsabile - sono tenuto in memoria. Il compromesso è che non c'è modo di "tornare indietro nel tempo / il documento XML" con un simile approccio: mentre il DOM consente a un `Node` di tornare ai suoi genitori, non esiste tale possibilità in SAX.

Principi dell'interfaccia StAX

L'API StAX adotta un approccio simile all'elaborazione di XML come API SAX (ovvero, basata su eventi), l'unica differenza molto significativa è che StAX è un parser di pull (dove SAX era un parser di push). In SAX, il `Parser` controllo e utilizza i callback su `ContentHandler` . In Stax, chiami il parser e controlli quando / se vuoi ottenere il successivo "evento" XML.

L'API inizia con XMLStreamReader (o XMLEventReader), che sono i gateway attraverso i quali lo sviluppatore può chiedere `nextEvent()` , in modo iteratore.

Examples

Analisi e navigazione di un documento utilizzando l'API DOM

Considerando il seguente documento:

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

Si può usare il seguente codice per costruire un albero DOM fuori da 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");
}
}
```

Il codice produce quanto segue:

```
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
```

Analisi di un documento utilizzando l'API StAX

Considerando il seguente 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>
```

Si può usare il seguente codice per analizzarlo e costruire una mappa di titoli di libri per id 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);
}
```

Questo 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}
```

In questo esempio, si deve essere carente di alcune cose:

1. L'uso di `xmlStreamReader.getAttributeValue` funziona perché abbiamo verificato prima che il parser si trovi nello stato `START_ELEMENT` . In tutti gli altri stati (eccetto gli `ATTRIBUTES`), il parser

è obbligato a lanciare `IllegalStateException` , perché gli attributi possono apparire solo all'inizio degli elementi.

2. lo stesso vale per `xmlStreamReader.getTextContent()` , funziona perché siamo a un `START_ELEMENT` e sappiamo in questo documento che l'elemento `<book>` non ha nodi figlio non di testo.

Per l'analisi di documenti più complessi (elementi più profondi, nidificati, ...), è buona norma "delegare" il parser a sottoprogrammi o altri oggetti, ad esempio avere una classe o un metodo `BookParser` e farlo trattare con ogni elemento da START_ELEMENT a END_ELEMENT del tag XML del libro.

Si può anche usare un oggetto `Stack` per mantenere intorno a dati importanti su e giù dall'albero.

Leggi Analisi XML usando le API JAXP online: https://riptutorial.com/it/java/topic/3943/analisi-xml-usando-le-api-jaxp

Capitolo 6: annotazioni

introduzione

In Java, un'annotazione è una forma di metadati sintattici che possono essere aggiunti al codice sorgente Java. Fornisce dati su un programma che non fa parte del programma stesso. Le annotazioni non hanno alcun effetto diretto sul funzionamento del codice annotato. Classi, metodi, variabili, parametri e pacchetti possono essere annotati.

Sintassi

- @AnnotationName // 'Annotazione marcatore' (nessun parametro)
- @AnnotationName (someValue) // imposta il parametro con il nome 'valore'
- @AnnotationName (param1 = value1) // parametro denominato
- @AnnotationName (param1 = value1, param2 = value2) // più parametri con nome
- @AnnotationName (param1 = {1, 2, 3}) // parametro con nome named
- @AnnotationName ({value1}) // array con singolo elemento come parametro con il nome 'valore'

Osservazioni

Tipi di parametri

Solo le espressioni costanti dei seguenti tipi sono consentite per i parametri, così come gli array di questi tipi:

- `String`
- `Class`
- tipi primitivi
- Tipi di Enum
- Tipi di annotazione

Examples

Annotazioni incorporate

La Standard Edition di Java viene fornita con alcune annotazioni predefinite. Non è necessario definirli da soli e puoi utilizzarli immediatamente. Permettono al compilatore di abilitare alcuni controlli fondamentali su metodi, classi e codici.

@Oltrepassare

Questa annotazione si applica a un metodo e afferma che questo metodo deve sovrascrivere il

metodo di una superclasse o implementare una definizione di metodo di una superclasse astratta. Se questa annotazione viene utilizzata con qualsiasi altro tipo di metodo, il compilatore genererà un errore.

Superclasse concreta

```
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");
    }
}
```

Classe astratta

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

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

Non funziona

```
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);
    }
}
```

Lo scopo principale è catturare l'errore di digitazione, in cui si pensa di ignorare un metodo, ma in realtà ne sta definendo uno nuovo.

```
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");
    }
}
```

Nota che il significato di `@Override` è cambiato nel tempo:

- In Java 5, significava che il metodo annotato doveva sovrascrivere un metodo non astratto dichiarato nella catena della superclasse.
- Da Java 6 in poi, è *anche* soddisfatto se il metodo annotato implementa un metodo astratto dichiarato nella gerarchia superclasse / interfaccia delle classi.

(Ciò può causare occasionalmente problemi quando si esegue il back-porting del codice su Java 5.)

@deprecated

Questo contrassegna il metodo come deprecato. Ci possono essere diverse ragioni per questo:

- l'API è difettosa e non è pratico da risolvere,
- l'utilizzo dell'API può causare errori,
- l'API è stata sostituita da un'altra API,
- l'API è obsoleta,
- l'API è sperimentale ed è soggetta a modifiche incompatibili,
- o qualsiasi combinazione di quanto sopra.

Il motivo specifico della deprecazione si trova di solito nella documentazione dell'API.

L'annotazione farà sì che il compilatore emetta un errore se lo si utilizza. Gli IDE possono anche evidenziare questo metodo in qualche modo come deprecato

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

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

```
}
```

@SuppressWarnings

In quasi tutti i casi, quando il compilatore emette un avviso, l'azione più appropriata è quella di correggere la causa. In alcuni casi (codice generics che utilizza un codice di pre-generica non sicuro, ad esempio) questo potrebbe non essere possibile ed è meglio sopprimere quegli avvertimenti che ci si aspetta e non è possibile correggere, in modo da poter vedere più chiaramente gli avvertimenti inattesi.

Questa annotazione può essere applicata a un'intera classe, metodo o linea. Prende la categoria di avvertimento come parametro.

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

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

È preferibile limitare il più possibile l'ambito dell'annotazione, per evitare che vengano eliminati anche gli avvisi imprevisti. Ad esempio, confinando l'ambito dell'annotazione su una singola riga:

```
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
```

Gli avvertimenti supportati da questa annotazione possono variare da compilatore a compilatore. Solo gli avvertimenti `unchecked` e di `deprecation` sono specificatamente menzionati nel JLS. I tipi di avvertimento non riconosciuti verranno ignorati.

@SafeVarargs

A causa della cancellazione dei tipi, il `void method(T... t)` verrà convertito in `void method(Object[] t)` il che significa che il compilatore non è sempre in grado di verificare che l'uso di varargs sia sicuro per il tipo. Per esempio:

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

Esistono casi in cui l'utilizzo è sicuro, nel qual caso è possibile annotare il metodo con l'annotazione di `SafeVarargs` per sopprimere l'avviso. Ovviamente questo nasconde l'avviso se il tuo uso non è sicuro.

@FunctionalInterface

Questa è un'annotazione opzionale utilizzata per contrassegnare una FunctionalInterface. Fa sì che il compilatore si lamenti se non è conforme alle specifiche di FunctionalInterface (ha un singolo metodo astratto)

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

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

Controlli delle annotazioni di runtime tramite riflessione

L'API Reflection di Java consente al programmatore di eseguire vari controlli e operazioni su campi di classi, metodi e annotazioni durante il runtime. Tuttavia, affinché un'annotazione sia visibile al momento dell'esecuzione, `RetentionPolicy` deve essere modificato in `RUNTIME` , come illustrato nell'esempio seguente:

```
@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));
    }
}
```

Definizione dei tipi di annotazione

I tipi di annotazione sono definiti con `@interface` . I parametri sono definiti come i metodi di un'interfaccia regolare.

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

Valori standard

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

Meta-Annotazioni

Le meta-annotazioni sono annotazioni che possono essere applicate ai tipi di annotazione. Una speciale meta-annotazione predefinita definisce come possono essere usati i tipi di annotazione.

@Bersaglio

La meta-annotazione `@Target` limita i tipi a cui può essere applicata l'annotazione.

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

È possibile aggiungere più valori utilizzando la notazione array, ad esempio
`@Target({ElementType.FIELD, ElementType.TYPE})`

Valori disponibili

ElementType	bersaglio	esempio di utilizzo sull'elemento target
ANNOTATION_TYPE	tipi di annotazione	`@Retention(RetentionPolicy.RUNTIME)` `@interface MyAnnotation`
COSTRUTTORE	costruttori	`@MyAnnotation` `public MyClass() {}`
CAMPO	campi, costanti enum	`@XmlAttribute`

ElementType	bersaglio	esempio di utilizzo sull'elemento target
		`private int count;`

LOCAL_VARIABLE	dichiarazioni variabili all'interno dei metodi	`for (@LoopVariable int i = 0; i < 100; i++) {`
		`  @Unused`
		`  String resultVariable;`
		`}`

PACCHETTO	pacchetto (in `package-info.java`)	`@Deprecated` `package very.old;`
METODO	metodi	`@XmlElement` `public int getCount() {...}`

PARAMETRO	parametri metodo / costruttore	`public Rectangle(`
		`  @NamedArg("width") double width,`
		`  @NamedArg("height") double height) {`
		`  ...`
		`}`

GENERE	classi, interfacce, enumerazioni	`@XmlRootElement` `public class Report {}`

Java SE 8

ElementType	bersaglio	esempio di utilizzo sull'elemento target
TYPE_PARAMETER	Digitare dichiarazioni dei parametri	`public <@MyAnnotation T> void f(T t) {}`
TYPE_USE	Uso di un tipo	`Object o = "42";` `String s = (@MyAnnotation String) o;`

@Ritenzione

La meta-annotazione `@Retention` definisce la visibilità dell'annotazione durante il processo di compilazione delle applicazioni o l'esecuzione. Per impostazione predefinita, le annotazioni sono incluse nei file `.class` , ma non sono visibili in fase di runtime. Per rendere accessibile un'annotazione in fase di esecuzione, `RetentionPolicy.RUNTIME` deve essere impostato su tale annotazione.

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

Valori disponibili

RetentionPolicy	Effetto
CLASSE	L'annotazione è disponibile nel file `.class` , ma non in fase di runtime
DURATA	L'annotazione è disponibile in runtime e si può accedere tramite riflessione
FONTE	L'annotazione è disponibile in fase di compilazione, ma non aggiunta ai file `.class` . L'annotazione può essere utilizzata ad esempio da un processore di annotazione.

@Documented

La meta-annotazione `@Documented` viene utilizzata per contrassegnare annotazioni il cui utilizzo dovrebbe essere documentato da generatori di documentazione API come javadoc . Non ha valori. Con `@Documented` , tutte le classi che utilizzano l'annotazione lo elencheranno nella pagina della documentazione generata. Senza `@Documented` , non è possibile vedere quali classi utilizzano l'annotazione nella documentazione.

@Ereditato

La meta-annotazione `@Inherited` è pertinente alle annotazioni applicate alle classi. Non ha valori. Contrassegnare un'annotazione come `@Inherited` altera il modo in cui funziona la query di annotazione.

- Per un'annotazione non ereditata, la query esamina solo la classe esaminata.
- Per un'annotazione ereditata, la query controlla anche la catena super-classe (in modo ricorsivo) finché non viene trovata un'istanza dell'annotazione.

Si noti che vengono interrogate solo le super classi: eventuali annotazioni associate alle interfacce nella gerarchia delle classi verranno ignorate.

@Ripetibile

`@Repeatable` La meta-annotazione `@Repeatable` stata aggiunta in Java 8. Indica che più istanze dell'annotazione possono essere associate alla destinazione dell'annotazione. Questa meta-annotazione non ha valori.

Ottenere i valori di annotazione in fase di esecuzione

È possibile recuperare le proprietà correnti dell'Annotazione utilizzando Reflection per recuperare il metodo o campo o classe a cui è stata applicata un'annotazione e quindi recuperare le proprietà desiderate.

```
@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);
        }
    }
}
```

L'output sarà

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

Ripetere le annotazioni

Fino a Java 8, due istanze della stessa annotazione non potevano essere applicate a un singolo elemento. La soluzione standard consisteva nell'utilizzare un'annotazione del contenitore contenente una serie di altre annotazioni:

```
// 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 fornisce un modo più pulito e più trasparente di utilizzare le annotazioni del contenitore, utilizzando l'annotazione `@Repeatable` . Per prima cosa aggiungiamo questo alla classe `Author` :

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

Ciò indica a Java di trattare più annotazioni `@Author` come se fossero circondate dal contenitore `@Authors` . Possiamo anche utilizzare `Class.getAnnotationsByType()` per accedere all'array `@Author` tramite la sua classe, anziché tramite il relativo contenitore:

```
@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
        }
    }
}
```

Annotazioni ereditate

Per impostazione predefinita, le annotazioni di classe non si applicano ai tipi che le estendono. Questo può essere modificato aggiungendo l'annotazione `@Inherited` alla definizione di annotazione

Esempio

Considera le seguenti 2 annotazioni:

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

e

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

Se tre classi sono annotate in questo modo:

```
@UninheritedAnnotationType
class A {
}

@InheritedAnnotationType
class B extends A {
}
```

```
class C extends B {
}
```

eseguendo questo codice

```
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));
```

stamperà un risultato simile a questo (a seconda dei pacchetti dell'annotazione):

```
null
@InheritedAnnotationType()
@InheritedAnnotationType()
_____
@UninheritedAnnotationType()
null
null
```

Nota che le annotazioni possono essere ereditate solo dalle classi, non dalle interfacce.

Compilare l'elaborazione del tempo usando il processore di annotazione

Questo esempio dimostra come eseguire il controllo del tempo di compilazione di un elemento annotato.

L'annotazione

L'annotazione `@Setter` è un indicatore che può essere applicato ai metodi. L'annotazione verrà scartata durante la compilazione e non sarà disponibile successivamente.

```
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 {
}
```

Il processore di annotazione

La classe `SetterProcessor` viene utilizzata dal compilatore per elaborare le annotazioni. Controlla, se i metodi annotati con l'annotazione `@Setter` sono metodi `public` , non `static` con un nome che inizia con `set` e che ha una lettera maiuscola come quarta lettera. Se una di queste condizioni non viene soddisfatta, viene inviato un errore al `Messager` . Il compilatore scrive questo su stderr, ma altri strumenti potrebbero usare queste informazioni in modo diverso. Ad esempio, l'IDE NetBeans consente all'utente di specificare i processori di annotazione utilizzati per visualizzare i messaggi di errore nell'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();
    }

}
```

Confezione

Per essere applicato dal compilatore, il processore di annotazione deve essere reso disponibile per l'SPI (vedere ServiceLoader).

Per fare ciò è necessario aggiungere un file di testo `META-INF/services/javax.annotation.processing.Processor` al file jar contenente il processore di annotazione e l'annotazione in aggiunta agli altri file. Il file deve includere il nome completo del processore di annotazione, ovvero dovrebbe apparire come questo

```
annotation.processor.SetterProcessor
```

Supponiamo che il file jar sia chiamato `AnnotationProcessor.jar` seguito.

Esempio di classe annotata

La classe seguente è una classe di esempio nel pacchetto predefinito con le annotazioni applicate agli elementi corretti in base al criterio di conservazione. Tuttavia, solo il processore di annotazione considera solo il secondo metodo un target di annotazione valido.

```
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) {}

}
```

Utilizzo del processore di annotazione con javac

Se il processore di annotazione viene rilevato utilizzando SPI, viene utilizzato automaticamente per elaborare gli elementi annotati. Ad esempio compilando la classe `AnnotationProcessorTest` utilizzando

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

produce il seguente output

```
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
```

invece di compilare normalmente. Nessun file `.class` creato.

Questo potrebbe essere evitato specificando l'opzione `-proc:none` per `javac` . Puoi anche rinunciare alla solita compilazione specificando `-proc:only` invece.

Integrazione IDE

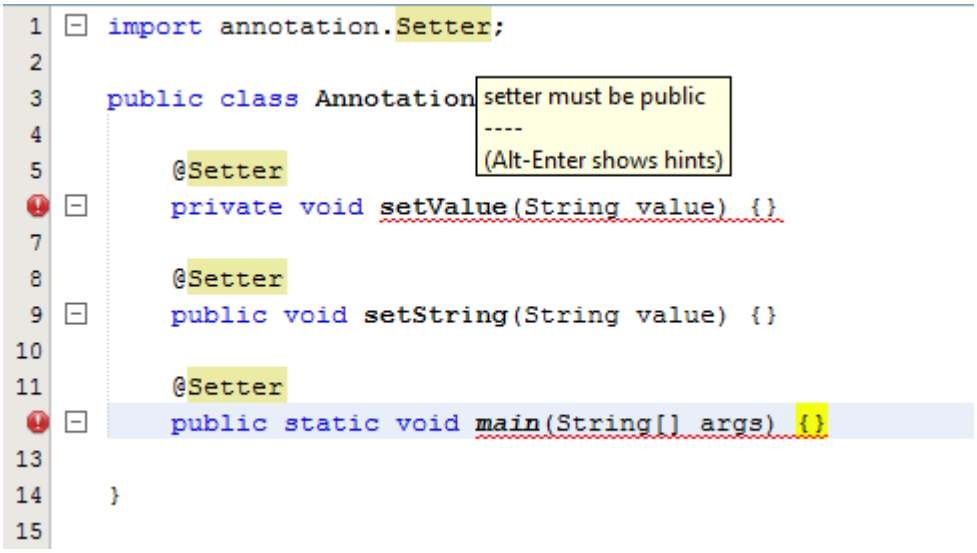
Netbeans

I processori di annotazione possono essere utilizzati nell'editor NetBeans. Per fare ciò è necessario specificare il processore di annotazione nelle impostazioni del progetto:

1. vai su `Project Properties` > `Build` > `Compiling`
2. aggiungere segni di spunta per `Enable Annotation Processing` e `Enable Annotation Processing in Editor`

3. fare clic su `Add` accanto all'elenco dei processori di annotazione
4. nel popup che appare inserisci il nome completo della classe del processore di annotazione e fai clic su `Ok` .

Risultato



L'idea alla base di Annotations

La specifica del linguaggio Java descrive le annotazioni come segue:

> Un'annotazione è un marcatore che associa informazioni con un costrutto di programma, ma non ha alcun effetto in fase di esecuzione.

Le annotazioni possono comparire prima dei tipi o delle dichiarazioni. È possibile che vengano visualizzati in un punto in cui potrebbero essere applicati sia a un tipo sia a una dichiarazione. A che cosa si applica esattamente un'annotazione è governato dalla "meta-annotazione" `@Target` . Vedere "Definizione dei tipi di annotazione" per ulteriori informazioni.

Le annotazioni sono utilizzate per una moltitudine di scopi. I framework come Spring e Spring-MVC fanno uso di annotazioni per definire dove devono essere iniettate le dipendenze o dove devono essere instradate le richieste.

Altri framework usano annotazioni per la generazione del codice. Lombok e JPA sono esempi primari, che usano annotazioni per generare codice Java (e SQL).

Questo argomento mira a fornire una panoramica completa di:

- Come definire le tue annotazioni?
- Quali annotazioni fornisce la lingua Java?
- Come vengono utilizzate le annotazioni nella pratica?

Annotazioni per "questo" e parametri del ricevitore

Quando sono state introdotte per la prima volta le annotazioni Java, non era prevista la possibilità di annotare la destinazione di un metodo di istanza o il parametro del costruttore nascosto per un costruttore di classi interne. Questo è stato risolto in Java 8 con l'aggiunta delle dichiarazioni dei *parametri* del *ricevitore* ; vedi JLS 8.4.1 .

> Il parametro receiver è un dispositivo sintattico opzionale per un metodo di istanza o un costruttore di una classe interna. Per un metodo di istanza, il parametro receiver rappresenta l'oggetto per cui viene invocato il metodo. Per il costruttore di una classe interna, il parametro ricevente rappresenta l'istanza immediatamente allegata dell'oggetto appena costruito. In entrambi i casi, il parametro ricevitore esiste unicamente per consentire al tipo dell'oggetto rappresentato di essere indicato nel codice sorgente, in modo che il tipo possa essere annotato. Il parametro del ricevitore non è un parametro formale; più precisamente, non è una dichiarazione di alcun tipo di variabile (§4.12.3), non è mai vincolata a nessun valore passato come argomento in un'espressione di chiamata di metodo o espressione di creazione di istanza di classe qualificata e non ha alcun effetto su tempo di esecuzione.

L'esempio seguente illustra la sintassi per entrambi i tipi di parametro del ricevitore:

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

L'unico scopo dei parametri del ricevitore è consentire l'aggiunta di annotazioni. Ad esempio, potresti avere un'annotazione personalizzata `@IsOpen` cui scopo è quello di affermare che un oggetto `Closeable` non è stato chiuso quando viene chiamato un metodo. Per esempio:

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

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

Ad un livello, l'annotazione `@IsOpen` su `this` potrebbe semplicemente servire come documentazione. Tuttavia, potremmo potenzialmente fare di più. Per esempio:

- Un processore di annotazione potrebbe inserire un runtime per verificare che `this` non sia in stato chiuso quando viene chiamato l' `update` .
- Un controllo del codice può eseguire un'analisi del codice statico per trovare casi in cui `this` *potrebbe essere* chiuso quando viene chiamato l' `update` .

Aggiungi più valori di annotazione

Un parametro Annotation può accettare più valori se è definito come array. Ad esempio, l'annotazione standard `@SuppressWarnings` è definita in questo modo:

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

Il parametro `value` è un array di stringhe. È possibile impostare più valori utilizzando una notazione simile agli inizializzatori di array:

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

Se hai solo bisogno di impostare un valore singolo, le parentesi possono essere omesse:

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

Leggi annotazioni online: https://riptutorial.com/it/java/topic/157/annotazioni

Capitolo 7: Apache Commons Lang

Examples

Implementa il metodo equals ()

Per implementare facilmente il metodo `equals` di un oggetto, è possibile utilizzare la classe `EqualsBuilder` .

Selezione dei campi:

```
@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 riflessione:

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

il parametro booleano indica se gli uguali devono controllare i campi transitori.

Usando la riflessione evitando alcuni campi:

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

Implementa il metodo hashCode ()

Per implementare facilmente il metodo `hashCode` di un oggetto, è possibile utilizzare la classe `HashCodeBuilder` .

Selezione dei campi:

```
@Override
public int hashCode() {

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

    return builder.hashCode();
}
```

Usando la riflessione:

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

il parametro booleano indica se deve utilizzare campi transienti.

Usando la riflessione evitando alcuni campi:

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

Implementa il metodo toString ()

Per implementare facilmente il metodo `toString` di un oggetto, è possibile utilizzare la classe `ToStringBuilder` .

Selezione dei campi:

```
@Override
public String toString() {

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

    return builder.toString();
}
```

Esempio di risultato:

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

Dare esplicitamente nomi ai campi:

```
@Override
public String toString() {
```

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

    return builder.toString();
}
```

Esempio di risultato:

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

Puoi cambiare lo stile tramite parametro:

```
@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();
}
```

Esempio di risultato:

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

Ci sono alcuni stili, ad esempio JSON, no Classname, short, ecc ...

Tramite riflessione:

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

Puoi anche indicare lo stile:

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

Leggi Apache Commons Lang online: https://riptutorial.com/it/java/topic/3338/apache-commons-lang

Capitolo 8: API di Reflection

introduzione

Reflection viene comunemente utilizzato dai programmi che richiedono la possibilità di esaminare o modificare il comportamento di runtime delle applicazioni in esecuzione nella JVM. L'API Java Reflection viene utilizzata a tale scopo dove consente di ispezionare classi, interfacce, campi e metodi in fase di runtime, senza conoscere i loro nomi in fase di compilazione. Inoltre, rende possibile l'istanziazione di nuovi oggetti e l'invocazione di metodi mediante la riflessione.

Osservazioni

Prestazione

Tenere presente che la riflessione potrebbe ridurre le prestazioni, ma usarla solo quando l'attività non può essere completata senza riflessione.

Dal tutorial Java L'API Reflection :

> Poiché la riflessione riguarda tipi risolti dinamicamente, non è possibile eseguire determinate ottimizzazioni della macchina virtuale Java. Di conseguenza, le operazioni di riflessione hanno prestazioni più lente rispetto alle controparti non riflettenti e dovrebbero essere evitate in sezioni di codice chiamate frequentemente in applicazioni sensibili alle prestazioni.

Examples

introduzione

Nozioni di base

L'API Reflection consente di controllare la struttura della classe del codice in fase di esecuzione e di richiamare il codice in modo dinamico. Questo è molto potente, ma è anche pericoloso poiché il compilatore non è in grado di determinare staticamente se le invocazioni dinamiche sono valide.

Un semplice esempio potrebbe essere quello di ottenere i costruttori e i metodi pubblici di una determinata classe:

```
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 questa informazione è possibile istanziare l'oggetto e chiamare dinamicamente diversi metodi.

Riflessione e tipi generici

Le informazioni di tipo generico sono disponibili per:

- parametri del metodo, usando `getGenericParameterTypes()` .
- metodo restituisce i tipi, usando `getGenericReturnType()` .
- campi **pubblici** , usando `getGenericType` .

L'esempio seguente mostra come estrarre le informazioni di tipo generico in tutti e tre i casi:

```
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;
    }
```

```
}
```

Ciò risulta nell'output seguente:

```
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
```

Invocare un metodo

Utilizzando reflection, un metodo di un oggetto può essere richiamato durante il runtime.

L'esempio mostra come richiamare i metodi di un oggetto `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!"
```

Recupero e impostazione dei campi

Utilizzando l'API di Reflection, è possibile modificare o ottenere il valore di un campo in fase di esecuzione. Ad esempio, è possibile utilizzarlo in un'API per recuperare campi diversi in base a un fattore, come il sistema operativo. Puoi anche rimuovere i modificatori come `final` per consentire i campi di modifica finali.

Per fare ciò, dovrai utilizzare il metodo Class # getField () in un modo come quello mostrato di seguito:

```
// 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);
```

Ottenere i campi è molto più facile. Possiamo usare Field # get () e le sue varianti per ottenere il suo valore:

```
// 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);
```

Prendi nota di questo:

Quando si utilizza Class # getDeclaredField , utilizzarlo per ottenere un campo nella classe stessa:

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

class Hacked {
    public String someState;
}
```

Qui, `HackMe#iAmDeclared` è dichiarato campo. Tuttavia, `HackMe#someState` non è un campo dichiarato

in quanto è ereditato dalla sua superclasse, Hacked.

Chiamare il costruttore

Ottenere l'oggetto Costruttore

È possibile ottenere la classe `Constructor` dall'oggetto `Class` questo modo:

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

Dove la variabile `constructors` avrà un'istanza `Constructor` per ogni costruttore pubblico dichiarato nella classe.

Se si conoscono i tipi di parametri precisi del costruttore a cui si desidera accedere, è possibile filtrare il costruttore specifico. Il prossimo esempio restituisce il costruttore pubblico della classe data che prende come parametro un `Integer` :

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

Se nessun costruttore corrisponde agli argomenti di costruzione forniti, viene generata una `NoSuchMethodException` .

Nuova istanza usando l'oggetto Costruttore

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

Ottenere le costanti di un'enumerazione

Dando questa enumerazione come esempio:

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

In Java una classe enum è come qualsiasi altra classe ma ha alcune costanti definite per i valori

enum. Inoltre ha un campo che è un array che contiene tutti i valori e due metodi statici con nome `values()` e `valueOf(String)` .
Possiamo vedere questo se usiamo Reflection per stampare tutti i campi in questa classe

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

l'output sarà:

NORD
EST
SUD
OVEST
grado
ENUM $ VALORI

Quindi potremmo esaminare le classi enum con Reflection come qualsiasi altra classe. Ma l'API Reflection offre tre metodi specifici per enum.

controllo enum

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

Restituisce true per le classi che rappresentano un tipo enum.

recupero di valori

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

Restituisce una matrice di tutti i valori enum come Compass.values () ma senza bisogno di un'istanza.

controllo costante enum

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

Elenca tutti i campi classe che sono valori enum.

Ottieni la classe dato il suo nome (completo)

Data una `String` contenente il nome di una classe, è possibile accedere all'oggetto `Class` utilizzando `Class.forName` :

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

```
} catch (ClassNotFoundException ex) {
    throw new IllegalStateException(ex);
}
```

Java SE 1.2

Può essere specificato, se la classe deve essere inizializzata (secondo parametro di `forName`) e quale `ClassLoader` deve essere utilizzato (terzo parametro):

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

Chiama costruttori sovraccaricati usando la riflessione

Esempio: richiama diversi costruttori passando parametri rilevanti

```
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"});

    }
}
```

produzione:

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

Spiegazione:

1. Crea istanza di classe usando `Class.forName` : chiama il costruttore predefinito
2. Richiamare `getDeclaredConstructor` della classe passando il tipo di parametri come `Class array`
3. Dopo aver ottenuto il costruttore, creare `newInstance` passando il valore del parametro come `Object array`

API Misuse of Reflection per modificare variabili private e finali

La riflessione è utile quando è usata correttamente per scopi giusti. Utilizzando la reflection, è possibile accedere a variabili private e reinizializzare le variabili finali.

Di seguito è riportato lo snippet di codice, che **non** è consigliato.

```
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++;
    }
}
```

Produzione:

```
A.name=StackOverFlow
A.age=20
```

```
A.rep=New Reputation
A.count=25
```

Spiegazione:

Nel normale scenario, non è possibile accedere alle variabili `private` al di fuori della classe dichiarata (senza metodi getter e setter). `final` variabili `final` non possono essere riassegnate dopo l'inizializzazione.

`Reflection` rompe entrambe le barriere che possono essere sfruttate per modificare sia le variabili private che quelle finali come spiegato sopra.

`field.setAccessible(true)` è la chiave per ottenere la funzionalità desiderata.

Costruttore di chiamate di classe nidificata

Se si desidera creare un'istanza di una classe nidificata interna, è necessario fornire un oggetto classe della classe di inclusione come parametro aggiuntivo con Class # 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"});
    }
}
```

Se la classe nidificata è statica, non avrai bisogno di questa istanza allegata.

Dynamic Proxies

I proxy dinamici non hanno molto a che fare con Reflection ma fanno parte dell'API. È fondamentalmente un modo per creare un'implementazione dinamica di un'interfaccia. Questo potrebbe essere utile quando si creano servizi di mockup.
Un proxy dinamico è un'istanza di un'interfaccia creata con un cosiddetto gestore di chiamata che intercetta tutte le chiamate di metodo e consente la gestione manuale della loro chiamata 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();
    }
}
```

Il risultato di questo codice è questo:

```
someMethod1 was invoked!
someMethod2 was invoked!
Parameter: stackoverflow
anotherMethod was invoked!
```

Evil Java hack con Reflection

L'API Reflection può essere utilizzata per modificare i valori dei campi privati e finali anche nella libreria predefinita JDK. Questo potrebbe essere usato per manipolare il comportamento di alcune classi ben note come vedremo.

Cosa non è possibile

Iniziamo prima con l'unica limitazione, l'unico campo che non possiamo cambiare con Reflection. Questo è Java `SecurityManager` . È dichiarato in java.lang.System come

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

Ma non verrà elencato nella classe System se eseguiamo questo codice

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

Questo è dovuto al `fieldFilterMap` in `sun.reflect.Reflection` che mantiene la mappa stessa e il campo di sicurezza in `System.class` e li protegge da qualsiasi accesso con Reflection. Quindi non abbiamo potuto disattivare `SecurityManager` .

Stringhe folli

Ogni stringa Java è rappresentata da JVM come istanza della classe `String` . Tuttavia, in alcune situazioni, la JVM salva lo spazio dell'heap utilizzando la stessa istanza per le stringhe che lo sono. Ciò accade per i letterali stringa e anche per le stringhe che sono state "internate" chiamando `String.intern()` . Quindi se hai `"hello"` nel tuo codice più volte è sempre la stessa istanza di oggetto.

Le stringhe dovrebbero essere immutabili, ma è possibile utilizzare il riflesso "cattivo" per cambiarle. L'esempio seguente mostra come possiamo cambiare i caratteri in una stringa sostituendo il suo campo `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");
    }
}
```

Quindi questo codice stamperà "tu puzzi!"

1 = 42

La stessa idea potrebbe essere utilizzata con la classe Integer

```
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));
    }
}
```

Tutto è vero

E secondo questo post StackOverflow possiamo usare la riflessione per fare qualcosa di veramente malvagio.

```
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);
    }
}
```

Nota che ciò che stiamo facendo qui farà sì che la JVM si comporti in modi inesplicabili. Questo è molto pericoloso.

Leggi API di Reflection online: https://riptutorial.com/it/java/topic/629/api-di-reflection

Capitolo 9: API Stack-Walking

introduzione

Prima di Java 9, l'accesso ai frame dello stack di thread era limitato a una classe interna `sun.reflect.Reflection` . In particolare il metodo `sun.reflect.Reflection::getCallerClass` . Alcune librerie si basano su questo metodo che è deprecato.

Un'API standard alternativa è ora disponibile in JDK 9 attraverso il `java.lang.StackWalker` classe, ed è progettato per essere efficiente, consentendo l'accesso pigro per stack frame. Alcune applicazioni possono utilizzare questa API per attraversare lo stack di esecuzione e filtrare sulle classi.

Examples

Stampa tutti i frame stack del thread corrente

Le seguenti stampe impilano tutti i frame del thread corrente:

```
1  package test;
2
3  import java.lang.StackWalker.StackFrame;
4  import java.lang.reflect.InvocationTargetException;
5  import java.lang.reflect.Method;
6  import java.util.List;
7  import java.util.stream.Collectors;
8
9  public class StackWalkerExample {
10
11    public static void main(String[] args) throws NoSuchMethodException, SecurityException,
IllegalAccessException, IllegalArgumentException, InvocationTargetException {
12        Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);
13        fooMethod.invoke(null, (Object[]) null);
14    }
15 }
16
17 class FooHelper {
18    protected static void foo() {
19        BarHelper.bar();
20    }
21 }
22
23 class BarHelper {
24    protected static void bar() {
25        List<StackFrame> stack = StackWalker.getInstance()
26                .walk((s) -> s.collect(Collectors.toList()));
27        for(StackFrame frame : stack) {
28            System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
frame.getMethodName());
29        }
30    }
31 }
```

Produzione:

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

Stampa la classe corrente dei chiamanti

Di seguito viene stampato l'attuale classe del chiamante. Nota che in questo caso, `StackWalker` deve essere creato con l'opzione `RETAIN_CLASS_REFERENCE` , in modo che le istanze della `Class` vengano mantenute negli oggetti `StackFrame` . Altrimenti si verificherebbe un'eccezione.

```
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());
    }
}
```

Produzione:

```
class test.FooHelper
```

Mostrando la riflessione e altri fotogrammi nascosti

Un paio di altre opzioni consentono alle tracce dello stack di includere cornici di implementazione e / o riflessione. Questo può essere utile per scopi di debug. Per esempio, possiamo aggiungere lo `SHOW_REFLECT_FRAMES` opzione per lo `StackWalker` esempio al momento della creazione, in modo che le cornici per i metodi riflettenti vengono stampati così:

```
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());
        }
    }
}
```

Produzione:

```
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
```

Nota che i numeri di linea per alcuni metodi di riflessione potrebbero non essere disponibili, quindi `StackFrame.getLineNumber()` può restituire valori negativi.

Leggi API Stack-Walking online: https://riptutorial.com/it/java/topic/9868/api-stack-walking

Capitolo 10: AppDynamics e TIBCO BusinessWorks Instrumentation per una facile integrazione

introduzione

Poiché AppDynamics mira a fornire un modo per misurare le prestazioni delle applicazioni, la velocità di sviluppo, la distribuzione (distribuzione) delle applicazioni è un fattore essenziale nel rendere gli sforzi DevOps un vero successo. Monitorare un'applicazione TIBCO BW con AppD è in genere semplice e non richiede molto tempo, ma quando si distribuiscono grandi serie di applicazioni è fondamentale la strumentazione rapida. Questa guida mostra come strumentare tutte le applicazioni BW in un unico passaggio senza modificare ciascuna applicazione prima della distribuzione.

Examples

Esempio di strumentazione di tutte le applicazioni BW in un unico passaggio per Appdynamics

1. Individua e apri tipicamente il tuo file bwengine.tra TIBCO BW in TIBCO_HOME / bw / 5.12 / bin / bwengine.tra (ambiente Linux)

2. Cerca la riga che indica:

*** Variabili comuni. Modifica solo questi. ***

3. Aggiungi la seguente riga subito dopo quella sezione tibco.deployment =% tibco.deployment%

4. Vai alla fine del file e aggiungi (sostituisci? Con i tuoi valori secondo necessità o rimuovi il flag che non si applica): 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. Salva file e ridistribuisci. Tutte le applicazioni dovrebbero ora essere strumentate automaticamente al momento dell'implementazione.

Leggi AppDynamics e TIBCO BusinessWorks Instrumentation per una facile integrazione online: https://riptutorial.com/it/java/topic/10602/appdynamics-e-tibco-businessworks-instrumentation-per-una-facile-integrazione

Capitolo 11: applet

introduzione

Le applet fanno parte di Java sin dalla sua versione ufficiale e sono state utilizzate per insegnare Java e la programmazione per diversi anni.

Gli ultimi anni hanno visto una spinta attiva per allontanarsi da Applet e altri plugin del browser, con alcuni browser che li bloccano o non li supportano attivamente.

Nel 2016, Oracle ha annunciato l'intenzione di deprecare il plug-in, passare a un Web senza plug-in

Sono ora disponibili API nuove e migliori

Osservazioni

Un'applet è un'applicazione Java che normalmente viene eseguita all'interno di un browser web. L'idea di base è quella di interagire con l'utente senza la necessità di interagire con il server e trasferire le informazioni. Questo concetto ha avuto molto successo intorno all'anno 2000, quando la comunicazione via internet era lenta e costosa.

Un'applet offre cinque metodi per controllare il loro ciclo di vita.

nome del metodo	descrizione
`init()`	viene chiamato una volta quando viene caricata l'applet
`destroy()`	viene chiamato una volta quando l'applet viene rimosso dalla memoria
`start()`	viene chiamato ogni volta che l'applet diventa visibile
`stop()`	viene chiamato ogni volta che l'applet viene sovrapposta ad altre finestre
`paint()`	viene chiamato quando necessario o attivato manualmente chiamando `repaint()`

Examples

Applet minima

Un'applet molto semplice disegna un rettangolo e stampa una stringa sullo schermo.

```
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 classe principale di un'applet si estende da `javax.swing.JApplet` .

Java SE 1.2

Prima che Java 1.2 e l'introduzione delle applet API swing fossero estese da `java.applet.Applet` .

Le applet non richiedono un metodo principale. Il punto di ingresso è controllato dal ciclo di vita. Per usarli, devono essere incorporati in un documento HTML. Questo è anche il punto in cui è definita la loro dimensione.

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

Creazione di una GUI

Le applet potrebbero essere facilmente utilizzate per creare una GUI. Si comportano come un `Container` e hanno un metodo `add()` che prende qualsiasi 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);
    }
}
```

Apri i collegamenti dall'applet

È possibile utilizzare il metodo `getAppletContext()` per ottenere un oggetto `AppletContext` che consente di richiedere al browser di aprire un collegamento. Per questo si usa il metodo `showDocument()` . Il suo secondo parametro dice al browser di usare una nuova finestra `_blank` o quella che mostra l'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);
    }
}
```

Caricamento di immagini, audio e altre risorse

Le applet Java sono in grado di caricare risorse diverse. Ma dal momento che sono in esecuzione nel browser Web del client, è necessario assicurarsi che tali risorse siano accessibili. Le applet non sono in grado di accedere alle risorse client come file system locale.

Se si desidera caricare risorse dallo stesso URL in cui è memorizzata l'applet, è possibile utilizzare il metodo `getCodeBase()` per recuperare l'URL di base. Per caricare risorse, le applet offrono i metodi `getImage()` e `getAudioClip()` per caricare immagini o file audio.

Carica e mostra un'immagine

```
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);
        }
}
```

Carica e riproduci un file 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();
    }
}
```

Carica e visualizza un file di testo

```
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 */ }
    }
}
```

Leggi applet online: https://riptutorial.com/it/java/topic/5503/applet

Capitolo 12: Array

introduzione

Le matrici consentono la memorizzazione e il recupero di una quantità arbitraria di valori. Sono analoghi ai vettori in matematica. Le matrici di array sono analoghe alle matrici e agiscono come array multidimensionali. Le matrici possono memorizzare qualsiasi tipo di dati: primitive come `int` o tipi di riferimento come `Object` .

Sintassi

- `ArrayType[] myArray;` // Dichiarare gli array
- `ArrayType myArray[];` // Un'altra sintassi valida (meno comunemente usata e scoraggiata)
- `ArrayType[][][] myArray;` // Dichiarazione di matrici frastagliate multidimensionali (ripeti [] s)
- `ArrayType myVar = myArray[index];` // Accesso (lettura) elemento all'indice
- `myArray[index] = value;` // valore Assign alla posizione `index` di matrice
- `ArrayType[] myArray = new ArrayType[arrayLength];` // Sintassi di inizializzazione dell'array
- `int[] ints = {1, 2, 3};` // Sintassi di inizializzazione dell'array con i valori forniti, la lunghezza viene dedotta dal numero di valori forniti: {[valore1 [, valore2] *]}
- `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};` // Inizializzazione array multidimensionale. int [] estende Object (e così anyType []) quindi null è un valore valido.

Parametri

Parametro	Dettagli
`ArrayType`	Tipo della matrice. Questo può essere primitivo (`int` , `long` , `byte`) o Objects (`String` , `MyObject` , ecc.).
indice	L'indice si riferisce alla posizione di un determinato oggetto in una matrice.
lunghezza	Ogni array, una volta creato, ha bisogno di una lunghezza specificata. Questo viene fatto quando si crea un array vuoto (`new int[3]`) o implicito quando si specificano valori (`{1, 2, 3}`).

Examples

Creazione e inizializzazione di matrici

Casi di base

```
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
```

Le matrici possono essere create usando qualsiasi tipo di primitivo o di riferimento.

```
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).
```

Per l'ultimo esempio, si noti che i sottotipi del tipo di array dichiarato sono consentiti nell'array.

Le matrici per i tipi definiti dall'utente possono anche essere costruite in modo simile ai tipi primitivi

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

Array, raccolte e flussi

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 );
```

Intro

Una *matrice* è una struttura di dati che contiene un numero fisso di valori primitivi **o** riferimenti a istanze di oggetti.

Ogni elemento in un array è chiamato un elemento e ogni elemento è accessibile tramite il suo indice numerico. La lunghezza di un array viene stabilita al momento della creazione dell'array:

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

La dimensione di un array è fissata al runtime quando inizializzata. Non può essere modificato dopo l'inizializzazione. Se la dimensione deve essere mutabile in fase di esecuzione, deve essere utilizzata una classe `Collection` come `ArrayList` . `ArrayList` archivia elementi in un array e supporta il ridimensionamento allocando un nuovo array e copiando gli elementi dal vecchio array.

Se l'array è di tipo primitivo, es

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

i valori sono memorizzati nella matrice stessa. In assenza di un inizializzatore (come in `array2` sopra), il valore predefinito assegnato a ciascun elemento è `0` (zero).

Se il tipo di matrice è un riferimento a un oggetto, come in

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

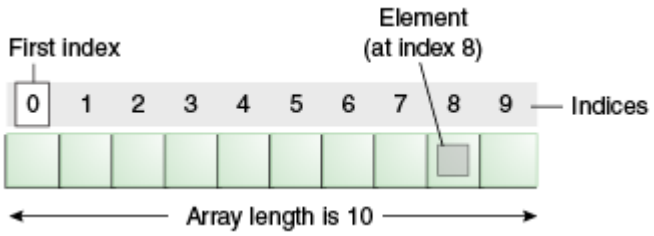
quindi la matrice contiene *riferimenti* a oggetti di tipo `SomeClassOrInterface` . Tali riferimenti possono fare riferimento a un'istanza di `SomeClassOrInterface` *o qualsiasi sottoclasse (per le classi) o alla classe di implementazione (per le interfacce) di* `SomeClassOrInterface` . Se la dichiarazione dell'array non ha inizializzatore, il valore predefinito di `null` viene assegnato a ciascun elemento.

Poiché tutti gli array sono `int` -indexed, la dimensione di un array deve essere specificata da un `int` . La dimensione dell'array non può essere specificata come una `long` :

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

Le matrici utilizzano un sistema di **indice a base zero** , il che significa che l'indicizzazione inizia da `0` e termina a `length - 1` .

Ad esempio, l'immagine seguente rappresenta una matrice con dimensione `10` . Qui, il primo elemento si trova nell'indice `0` e l'ultimo elemento è nell'indice `9` , invece del primo elemento nell'indice `1` e nell'ultimo elemento nell'indice `10` (vedi figura sotto).



Gli accessi agli elementi degli array vengono eseguiti in **tempo costante** . Ciò significa che l'accesso al primo elemento dell'array ha lo stesso costo (nel tempo) dell'accesso al secondo elemento, al terzo elemento e così via.

Java offre diversi modi per definire e inizializzare gli array, comprese le notazioni **letterali** e del **costruttore** . Quando si dichiarano gli array usando il `new Type[length]` costruttore `new Type[length]` , ogni elemento sarà inizializzato con i seguenti valori predefiniti:

- `0` per tipi numerici primitivi : `byte` , `short` , `int` , `long` , `float` e `double` .
- `'\u0000'` (carattere null) per il tipo di `char` .
- `false` per il tipo `boolean` .
- `null` per i tipi di riferimento .

Creazione e inizializzazione di matrici di tipi primitivi

```
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.
```

Quando si dichiara un array, `[]` apparirà come parte del tipo all'inizio della dichiarazione (dopo il nome del tipo), o come parte del dichiaratore per una particolare variabile (dopo il nome della variabile), o entrambi:

```
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() { ... }
```

Nell'esempio seguente, entrambe le dichiarazioni sono corrette e possono essere compilate ed eseguite senza problemi. Tuttavia, sia la convenzione di codifica Java che la Guida allo stile di Java di Google scoraggiano il modulo con parentesi dopo il nome della variabile: le parentesi identificano il tipo di matrice e dovrebbero apparire con la designazione del tipo . Lo stesso dovrebbe essere usato per le firme di ritorno del metodo.

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

Il tipo scoraggiato è pensato per accogliere gli utenti in transizione C , che hanno familiarità con la sintassi per C che ha le parentesi dopo il nome della variabile.

In Java, è possibile avere array di dimensione `0` :

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

Tuttavia, poiché si tratta di un array vuoto, non è possibile leggere da esso né assegnargli elementi:

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

Tali array vuoti sono in genere utili come valori di ritorno, in modo che il codice chiamante debba preoccuparsi solo di gestire un array, piuttosto che un potenziale valore `null` che può portare a una `NullPointerException` .

La lunghezza di un array deve essere un numero intero non negativo:

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

La dimensione dell'array può essere determinata utilizzando un campo finale pubblico chiamato `length` :

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

Nota : `array.length` restituisce la dimensione effettiva dell'array e non il numero di elementi dell'array a cui è stato assegnato un valore, diversamente da `ArrayList.size()` che restituisce il numero di elementi dell'array a cui è stato assegnato un valore.

Creazione e inizializzazione di array multidimensionali

Il modo più semplice per creare un array multidimensionale è il seguente:

```
int[][] a = new int[2][3];
```

Creerà due `int` arriys `a[0]` tre lunghezze: `a[0]` e `a[1]` . Questo è molto simile alla classica inizializzazione in stile C degli array rettangolari multidimensionali.

Puoi creare e inizializzare allo stesso tempo:

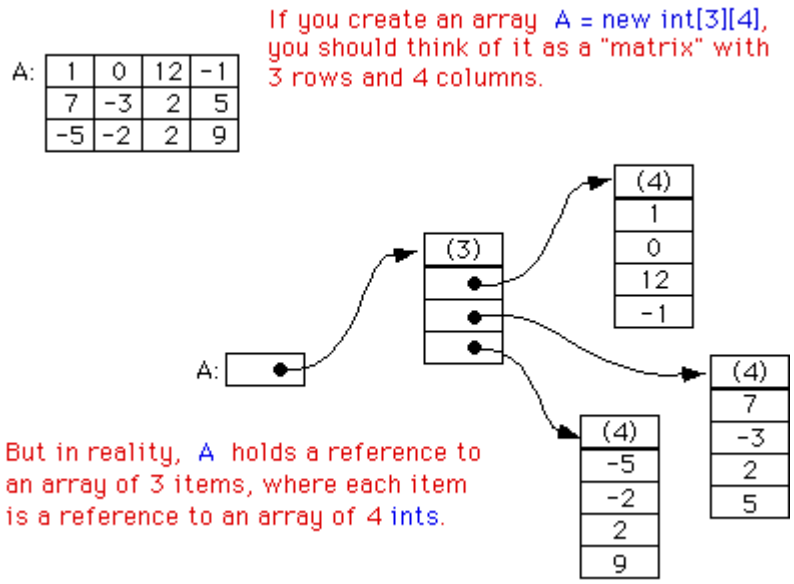
```
int[][] a = { {1, 2}, {3, 4}, {5, 6} };
```

A differenza di C , in cui sono supportati solo array rettangolari multidimensionali, gli array interni non devono necessariamente avere la stessa lunghezza o anche essere definiti:

```
int[][] a = { {1}, {2, 3}, null };
```

Qui, `a[0]` è un array `int` lunghezza singola, mentre `a[1]` è un array `int` due lunghezze e `a[2]` è `null` . Array come questo sono chiamati array frastagliati o array frastagliati , cioè sono matrici di array. Gli array multidimensionali in Java sono implementati come array di matrici, cioè `array[i][j][k]` è equivalente a `((array[i])[j])[k]` . A differenza di C # , l' `array[i,j]` sintassi `array[i,j]` non è supportato in Java.

Rappresentazione di array multidimensionali in Java



Source - Live on Ideone

Creazione e inizializzazione di matrici di tipi di riferimento

```
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
```

Vivi su Ideone

Oltre ai letterali e alle primitive `String` mostrati sopra, la sintassi di scelta rapida per l'inizializzazione dell'array funziona anche con tipi di `Object` canonici:

```
Object[] array10 = { new Object(), new Object() };
```

Poiché gli array sono covarianti, un array di tipi di riferimento può essere inizializzato come array di una sottoclasse, sebbene venga generata `ArrayStoreException` se si tenta di impostare un elemento su qualcosa di diverso da una `String` :

```
Object[] array11 = new String[] { "foo", "bar", "baz" };
array11[1] = "qux"; // fine
array11[1] = new StringBuilder(); // throws ArrayStoreException
```

La sintassi di scelta rapida non può essere utilizzata per questo perché la sintassi di scelta rapida avrebbe un tipo implicito di `Object[]` .

Un array può essere inizializzato con zero elementi utilizzando `String[] emptyArray = new String[0]` . Ad esempio, una matrice con lunghezza zero come questa viene utilizzata per la creazione di una `Array` da una `Collection` quando il metodo richiede il tipo di esecuzione di un oggetto.

In entrambi i tipi primitivi e di riferimento, un'inizializzazione di array vuota (ad esempio `String[] array8 = new String[3]`) inizializzerà la matrice con il valore predefinito per ogni tipo di dati .

Creazione e inizializzazione di array di tipi generici

Nelle classi generiche, le matrici di tipi generici **non possono** essere inizializzate in questo modo a causa della cancellazione del tipo :

```
public class MyGenericClass<T> {
    private T[] a;

    public MyGenericClass() {
        a = new T[5]; // Compile time error: generic array creation
    }
}
```

Invece, possono essere creati usando uno dei seguenti metodi: (nota che questi genereranno avvisi non controllati)

1. Creando una matrice `Object` e convertendola nel tipo generico:

   ```
   a = (T[]) new Object[5];
   ```

   Questo è il metodo più semplice, ma poiché l'array sottostante è ancora di tipo `Object[]` , questo metodo non fornisce sicurezza di tipo. Pertanto, questo metodo di creazione di un

array viene utilizzato al meglio solo all'interno della classe generica, non è esposto pubblicamente.

2. Usando `Array.newInstance` con un parametro di classe:

```
public MyGenericClass(Class<T> clazz) {
    a = (T[]) Array.newInstance(clazz, 5);
}
```

Qui la classe di `T` deve essere esplicitamente passata al costruttore. Il tipo di ritorno di `Array.newInstance` è sempre `Object` . Tuttavia, questo metodo è più sicuro perché l'array appena creato è sempre di tipo `T[]` , e quindi può essere tranquillamente esternalizzato.

Riempimento di un array dopo l'inizializzazione

Java SE 1.2

`Arrays.fill()` può essere utilizzato per riempire una matrice con **lo stesso valore** dopo l'inizializzazione:

```
Arrays.fill(array8, "abc");         // { "abc", "abc", "abc" }
```

Vivi su Ideone

`fill()` può anche assegnare un valore a ciascun elemento dell'intervallo specificato dell'array:

```
Arrays.fill(array8, 1, 2, "aaa");  // Placing "aaa" from index 1 to 2.
```

Vivi su Ideone

Java SE 8

Dalla versione 8 di Java, è possibile utilizzare il metodo `setAll` e il relativo `parallelSetAll` `Concurrent` per impostare ogni elemento di un array su valori generati. Questi metodi sono passati a una funzione generatore che accetta un indice e restituisce il valore desiderato per quella posizione.

L'esempio seguente crea un array intero e imposta tutti i suoi elementi sul rispettivo valore di indice:

```
int[] array = new int[5];
Arrays.setAll(array, i -> i); // The array becomes { 0, 1, 2, 3, 4 }.
```

Vivi su Ideone

Dichiarazione separata e inizializzazione degli array

Il valore di un indice per un elemento dell'array deve essere un numero intero (0, 1, 2, 3, 4, ...) e inferiore alla lunghezza dell'array (gli indici sono a base zero). Altrimenti verrà lanciata 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 }
```

Gli array non possono essere reinizializzati con la sintassi di scelta rapida di array initializer

Non è possibile inizializzare nuovamente un array tramite una sintassi di scelta rapida con un inizializzatore di array poiché un inizializzatore di array può essere specificato solo in una dichiarazione di campo o in una dichiarazione di variabile locale o come parte di un'espressione di creazione di matrice.

Tuttavia, è possibile creare un nuovo array e assegnarlo alla variabile utilizzata per fare riferimento al vecchio array. Mentre ciò comporta che l'array a cui fa riferimento tale variabile venga reinizializzato, il contenuto della variabile è un array completamente nuovo. Per fare ciò, il `new` operatore può essere utilizzato con un inizializzatore di array e assegnato alla variabile array:

```
// 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.
```

Vivi su Ideone

Creazione di una matrice da una raccolta

Due metodi in `java.util.Collection` creano una matrice da una raccolta:

- `Object[] toArray()`
- `<T> T[] toArray(T[] a)`

`Object[] toArray()` può essere utilizzato come segue:

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)` può essere utilizzato come segue:

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 differenza tra loro è molto più che avere risultati non tipizzati rispetto a quelli digitati. Anche le loro prestazioni possono differire (per ulteriori dettagli, leggi questa sezione sull'analisi delle prestazioni):

- `Object[] toArray()` usa `arraycopy` vettorizzato, che è molto più veloce rispetto al `arraycopy` controllo del `arraycopy` utilizzato in `T[] toArray(T[] a)` .
- `T[] toArray(new T[non-zero-size])` bisogno di azzerare l'array durante il runtime, mentre `T[] toArray(new T[0])` no. Tale evitamento fa chiamare quest'ultima più veloce della prima. Analisi dettagliata qui: Arrays of Wisdom of the Ancients .

Java SE 8

A partire da Java SE 8+, in cui è stato introdotto il concetto di `Stream` , è possibile utilizzare il `Stream` prodotto dalla raccolta per creare un nuovo array utilizzando il metodo `Stream.toArray` .

```
String[] strings = list.stream().toArray(String[]::new);
```

Esempi presi da due risposte (1 , 2) alla conversione di 'ArrayList in' String [] 'in Java su Stack Overflow.

Matrici a una stringa

Java SE 5

Da Java 1.5 è possibile ottenere una rappresentazione `String` dei contenuti della matrice specificata senza iterare su ogni elemento. Basta usare `Arrays.toString(Object[])` o `Arrays.deepToString(Object[])` per gli array multidimental:

```
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()` metodo `Arrays.toString()` utilizza il metodo `Object.toString()` per produrre valori di `String` di ogni elemento dell'array, oltre all'array di tipo primitivo, può essere utilizzato per tutti i tipi di matrici. Per esempio:

```
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!]
```

Se non esiste alcun `toString()` override per la classe, verrà utilizzato il `toString()` ereditato da `Object` . Di solito l'output non è molto utile, ad esempio:

```
public class Dog {
    /* implicitly extends Object */
}

Dog[] arr = { new Dog() };
System.out.println(Arrays.toString(arr));        // [Dog@17ed40e0]
```

Creare una lista da una matrice

Il metodo `Arrays.asList()` può essere usato per restituire una `List` dimensione fissa contenente gli elementi dell'array dato. L' `List` risultante avrà lo stesso tipo di parametro del tipo base dell'array.

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = Arrays.asList(stringArray);
```

Nota : questo elenco è supportato da *una vista* della matrice originale, il che significa che qualsiasi modifica alla lista cambierà la matrice e viceversa. Tuttavia, le modifiche alla lista che cambierebbero le sue dimensioni (e quindi la lunghezza dell'array) genereranno un'eccezione.

Per creare una copia dell'elenco, utilizzare il costruttore di `java.util.ArrayList` che `java.util.ArrayList` una `Collection` come argomento:

Java SE 5

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<String>(Arrays.asList(stringArray));
```

Java SE 7

In Java SE 7 e versioni successive, è possibile utilizzare una coppia di parentesi angolari `<>` (set vuoto di argomenti tipo), che si chiama Diamond . Il compilatore può determinare gli argomenti di tipo dal contesto. Ciò significa che le informazioni sul tipo possono essere omesse quando si chiama il costruttore di `ArrayList` e verranno dedotte automaticamente durante la compilazione. Questo è chiamato tipo Inferenza che fa parte di 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 degno di nota sul diamante è che non può essere utilizzato con le classi anonime .

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());
```

Note importanti relative all'uso del metodo Arrays.asList ()

- Questo metodo restituisce `List` , che è un'istanza di `Arrays$ArrayList` (classe interna statica di `Arrays`) e non `java.util.ArrayList` . L' `List` risultante è di dimensioni fisse. Ciò significa che l'aggiunta o la rimozione di elementi non è supportata e genererà un `UnsupportedOperationException` :

```
stringList.add("something"); // throws java.lang.UnsupportedOperationException
```

- Un nuovo `List` può essere creato passando un `List` supportato da array al costruttore di una nuova `List` . Ciò crea una nuova copia dei dati, che ha dimensioni variabili e che non è supportata dall'array originale:

```
List<String> modifiableList = new ArrayList<>(Arrays.asList("foo", "bar"));
```

- Chiamando `<T> List<T> asList(T... a)` su un array primitivo, ad esempio un `int[]` , verrà generato un `List<int[]>` cui unico elemento è l'array primitivo di origine anziché gli elementi effettivi dell'array sorgente.

  La ragione di questo comportamento è che i tipi primitivi non possono essere usati al posto dei parametri di tipo generico, quindi l'intero array primitivo sostituisce il parametro di tipo generico in questo caso. Per convertire una matrice primitiva in una `List` , convertire innanzitutto la matrice primitiva in una matrice del tipo di wrapper corrispondente (ad esempio, chiamare `Arrays.asList` su un numero `Integer[]` anziché su un `int[]`).

  Pertanto, questo verrà stampato `false` :

```
int[] arr = {1, 2, 3};      // primitive array of int
System.out.println(Arrays.asList(arr).contains(1));
```

  Visualizza la demo

  D'altra parte, questo verrà stampato `true` :

```
Integer[] arr = {1, 2, 3};  // object array of Integer (wrapper for int)
System.out.println(Arrays.asList(arr).contains(1));
```

  Visualizza la demo

  Anche questo verrà stampato `true` , poiché l'array verrà interpretato come un `Integer[]`):

```
System.out.println(Arrays.asList(1,2,3).contains(1));
```

  Visualizza la demo

Matrici multidimensionali e frastagliate

È possibile definire una matrice con più di una dimensione. Invece di accedere con un singolo indice, è possibile accedere a un array multidimensionale specificando un indice per ogni dimensione.

La dichiarazione dell'array multidimensionale può essere eseguita aggiungendo `[]` per ogni dimensione a una normale declinazione dell'array. Ad esempio, per creare un array `int` bidimensionale, aggiungi un altro insieme di parentesi alla dichiarazione, ad esempio `int[][]` . Questo continua per gli array 3-dimensionale (`int[][][]`) e così via.

Per definire una matrice bidimensionale con tre righe e tre colonne:

```
int rows = 3;
int columns = 3;
int[][] table = new int[rows][columns];
```

L'array può essere indicizzato e assegnare valori ad esso con questo costrutto. Si noti che i valori non assegnati sono i valori predefiniti per il tipo di un array, in questo caso `0` per `int` .

```
table[0][0] = 0;
table[0][1] = 1;
table[0][2] = 2;
```

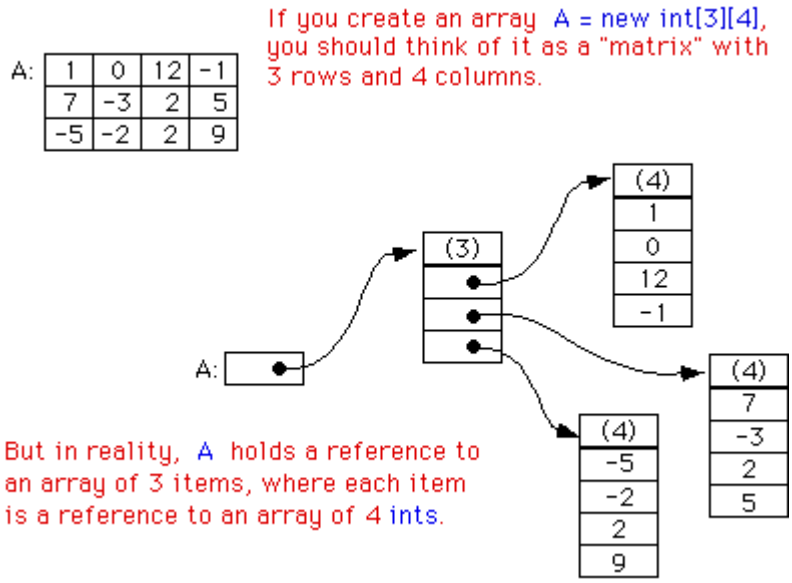
È anche possibile creare un'istanza di una dimensione alla volta e persino creare array non rettangolari. Questi sono più comunemente indicati come **matrici frastagliate** .

```
int[][] nonRect = new int[4][];
```

È importante notare che, sebbene sia possibile definire qualsiasi dimensione della matrice seghettata, è ***necessario*** definire il livello precedente.

```
// valid
String[][] employeeGraph = new String[30][];

// invalid
int[][] unshapenMatrix = new int[][10];

// also invalid
int[][][] misshapenGrid = new int[100][][10];
```

Come gli array multidimensionali sono rappresentati in Java



Fonte immagine: http://math.hws.edu/eck/cs124/javanotes3/c8/s5.html

---

Intializzazione letterale a matrice seghettata

Array multidimensionali e matrici frastagliate possono anche essere inizializzati con un'espressione letterale. Quanto segue dichiara e popola un array `int` 2x3:

```
int[][] table = {
    {1, 2, 3},
    {4, 5, 6}
};
```

Nota : i sottotitoli seghettati potrebbero anche essere `null` . Ad esempio, il seguente codice dichiara e popola un array `int` bidimensionale il cui primo sottoarray è `null` , il secondo sottoarray è di lunghezza zero, il terzo sottoarray è di una lunghezza e l'ultimo sottoarray è un array di due lunghezze:

```
int[][] table = {
    null,
    {},
    {1},
    {1,2}
};
```

Per array multidimensionali è possibile estrarre array di dimensioni di livello inferiore dai loro indici:

```
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[]
```

Indice della Matrice Fuori Dai Limiti d'Eccezione

L' `ArrayIndexOutOfBoundsException` viene generata quando si accede a un indice non esistente di un

array.

Le matrici sono indicizzate su base zero, quindi l'indice del primo elemento è `0` e l'indice dell'ultimo elemento è la capacità dell'array meno `1` (cioè `array.length - 1`).

Pertanto, qualsiasi richiesta per un elemento dell'array dall'indice `i` deve soddisfare la condizione `0 <= i < array.length` , altrimenti verrà `ArrayIndexOutOfBoundsException` l' `ArrayIndexOutOfBoundsException` .

Il seguente codice è un semplice esempio in cui viene lanciata 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.
```

Produzione:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
    at your.package.path.method(YourClass.java:15)
```

Si noti che l'indice illegale a cui si accede è anche incluso nell'eccezione (`2` nell'esempio); questa informazione potrebbe essere utile per trovare la causa dell'eccezione.

Per evitare ciò, è sufficiente verificare che l'indice sia entro i limiti dell'array:

```
int index = 2;
if (index >= 0 && index < people.length) {
    System.out.println(people[index]);
}
```

Ottenere la lunghezza di una matrice

Le matrici sono oggetti che forniscono spazio per memorizzare fino alla sua dimensione di elementi di tipo specificato. La dimensione di un array non può essere modificata dopo la creazione dell'array.

```
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
```

Il campo `length` in un array memorizza la dimensione di un array. È un campo `final` e non può essere modificato.

Questo codice mostra la differenza tra la `length` di un array e la quantità di oggetti che un array memorizza.

```
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");
}
```

Risultato:

```
Array 'arr' has a length of 13
and it contains 7 non-empty values
```

Confronto tra array per l'uguaglianza

I tipi di matrice ereditano le loro implementazioni `equals()` (e `hashCode()`) da java.lang.Object , quindi `equals()` restituisce true solo quando si confronta con lo stesso identico oggetto array. Per confrontare gli array per l'uguaglianza in base ai loro valori, utilizzare `java.util.Arrays.equals` , che è sovraccarico per tutti i tipi di array.

```
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
```

Quando il tipo di elemento è un tipo di riferimento, `Arrays.equals()` chiama `equals()` sugli elementi dell'array per determinare l'uguaglianza. In particolare, se il tipo di elemento è esso stesso un tipo di matrice, verrà utilizzato il confronto dell'identità. Per confrontare gli array multidimensionali per l'uguaglianza, utilizzare invece `Arrays.deepEquals()` come di seguito:

```
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
```

Poiché gli insiemi e le mappe utilizzano `equals()` e `hashCode()` , gli array generalmente non sono utili come elementi impostati o chiavi della mappa. Indirizzali in una classe helper che implementa `equals()` e `hashCode()` in termini di elementi dell'array, o convertili in istanze di `List` e memorizzi gli elenchi.

Array per lo streaming

Java SE 8

Conversione di una matrice di oggetti su `Stream` :

```
String[] arr = new String[] {"str1", "str2", "str3"};
Stream<String> stream = Arrays.stream(arr);
```

La conversione di una matrice di primitivi in `Stream` utilizzando `Arrays.stream()` trasformerà l'array in una specializzazione primitiva di Stream:

```
int[] intArr = {1, 2, 3};
IntStream intStream = Arrays.stream(intArr);
```

È inoltre possibile limitare il `Stream` a un intervallo di elementi nell'array. L'indice iniziale è inclusivo e l'indice finale è esclusivo:

```
int[] values = {1, 2, 3, 4};
IntStream intStream = Arrays.stream(values, 2, 4);
```

Un metodo simile a `Arrays.stream()` appare nella classe `Stream` : `Stream.of()` . La differenza è che `Stream.of()` usa un parametro varargs, quindi puoi scrivere qualcosa come:

```
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});
```

Iterare su array

È possibile eseguire iterazioni su array utilizzando il ciclo for enhanced (aka foreach) o utilizzando gli indici di array:

```
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 notare che non esiste un modo diretto per usare un Iterator su una matrice, ma attraverso la libreria di Array può essere facilmente convertito in una lista per ottenere un oggetto `Iterable` .

Per gli array in box usa Arrays.asList :

```
Integer[] boxed = {1, 2, 3};
Iterable<Integer> boxedIt = Arrays.asList(boxed); // list-backed iterable
Iterator<Integer> fromBoxed1 = boxedIt.iterator();
```

Per gli array primitivi (usando java 8) utilizzare gli stream (in particolare in questo esempio: Arrays.stream -> IntStream):

```
int[] primitives = {1, 2, 3};
IntStream primitiveStream = Arrays.stream(primitives); // list-backed iterable
PrimitiveIterator.OfInt fromPrimitive1 = primitiveStream.iterator();
```

Se non puoi utilizzare gli stream (senza java 8), puoi scegliere di utilizzare la libreria guava di google:

```
Iterable<Integer> fromPrimitive2 = Ints.asList(primitives);
```

In matrici bidimensionali o più, entrambe le tecniche possono essere utilizzate in un modo leggermente più complesso.

Esempio:

```
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);
    }
}
```

È impossibile impostare una matrice su qualsiasi valore non uniforme senza utilizzare un ciclo basato su indice.

Ovviamente puoi anche usare loop `while` o `do-while` quando si itera usando gli indici.

Una nota di cautela: quando si usano gli indici di matrice, assicurarsi che l'indice sia compreso tra `0` e `array.length - 1` (entrambi inclusi). Non fare supposizioni hard coded sulla lunghezza dell'array altrimenti potresti rompere il tuo codice se la lunghezza dell'array cambia ma i tuoi valori hard coded no.

Esempio:

```
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;
    }
}
```

È anche la soluzione migliore se non si utilizzano calcoli elaborati per ottenere l'indice ma si utilizza l'indice per ripetere e se è necessario calcolare valori diversi.

Esempio:

```
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;
    }
}
```

Accedere agli array in ordine 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]);
}
```

Utilizzare matrici temporanee per ridurre la ripetizione del codice

L'iterazione su una matrice temporanea anziché la ripetizione del codice può rendere il tuo codice più pulito. Può essere utilizzato dove viene eseguita la stessa operazione su più variabili.

```
// 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));
```

Tenere presente che questo codice non deve essere utilizzato nelle sezioni critiche per le prestazioni, poiché ogni volta che viene immesso il loop viene creato un array e tali variabili primitive vengono copiate nell'array e non possono quindi essere modificate.

Copia di array

Java offre diversi modi per copiare un array.

per ciclo

```
int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

Si noti che l'uso di questa opzione con una matrice Oggetto invece di una matrice primitiva riempirà la copia con riferimento al contenuto originale anziché alla sua copia.

Object.clone ()

Poiché gli array sono `Object` s in Java, puoi utilizzare `Object.clone()` .

```
int[] a = { 4, 1, 3, 2 };
int[] b = a.clone(); // [4, 1, 3, 2]
```

Si noti che il metodo `Object.clone` per un array esegue una **copia superficiale** , ovvero restituisce

un riferimento a un nuovo array che fa riferimento agli **stessi** elementi dell'array sorgente.

Arrays.copyOf ()

`java.util.Arrays` fornisce un modo semplice per eseguire la copia di un array su un altro. Ecco l'utilizzo di base:

```
int[] a = {4, 1, 3, 2};
int[] b = Arrays.copyOf(a, a.length); // [4, 1, 3, 2]
```

Nota che `Arrays.copyOf` fornisce anche un overload che ti permette di cambiare il tipo di array:

```
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 matrice dall'array di origine specificato, iniziando dalla posizione specificata, nella posizione specificata della matrice di destinazione.

Di seguito un esempio di utilizzo

```
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 ()

Utilizzato principalmente per copiare una parte di una matrice, è anche possibile utilizzarlo per copiare l'intera matrice su un'altra come di seguito:

```
int[] a = { 4, 1, 3, 2 };
int[] b = Arrays.copyOfRange(a, 0, a.length); // [4, 1, 3, 2]
```

Casting Arrays

Le matrici sono oggetti, ma il loro tipo è definito dal tipo degli oggetti contenuti. Pertanto, non si può semplicemente trasmettere `A[]` a `T[]` , ma ogni membro A dello specifico `A[]` deve essere lanciato su un oggetto `T` Esempio generico:

```
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;
}
```

Pertanto, dato un array `A[]` :

```
T[] target = new T[array.Length];
target = castArray(target, array);
```

Java SE fornisce il metodo `Arrays.copyOf(original, newLength, newType)` per questo scopo:

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```

Rimuovi un elemento da una matrice

Java non fornisce un metodo diretto in `java.util.Arrays` per rimuovere un elemento da una matrice. Per eseguirlo, è possibile copiare l'array originale in uno nuovo senza l'elemento per rimuovere o convertire l'array in un'altra struttura consentendo la rimozione.

Utilizzando ArrayList

È possibile convertire l'array in un `java.util.List` , rimuovere l'elemento e riconvertire l'elenco in un array come segue:

```
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]
```

Utilizzando System.arraycopy

`System.arraycopy()` può essere utilizzato per creare una copia dell'array originale e rimuovere l'elemento desiderato. Di seguito un esempio:

```
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]
```

Utilizzando Apache Commons Lang

Per rimuovere facilmente un elemento, è possibile utilizzare la libreria di Apache Commons Lang e in particolare il metodo statico `removeElement()` della classe `ArrayUtils` . Di seguito un esempio:

```
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

Gli array di oggetti sono covarianti, il che significa che proprio come `Integer` è una sottoclasse di `Number` , `Integer[]` è una sottoclasse di `Number[]` . Questo può sembrare intuitivo, ma può comportare comportamenti sorprendenti:

```
Integer[] integerArray = {1, 2, 3};
Number[] numberArray = integerArray;  // valid
Number firstElement = numberArray[0]; // valid
numberArray[0] = 4L;                  // throws ArrayStoreException at runtime
```

Sebbene `Integer[]` sia una sottoclasse di `Number[]` , può contenere solo `Integer` , e il tentativo di assegnare un elemento `Long` genera un'eccezione di runtime.

Si noti che questo comportamento è unico per gli array e può essere evitato utilizzando invece un `List` generico:

```
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
```

Non è necessario che tutti gli elementi dell'array condividano lo stesso tipo, purché siano una sottoclasse del tipo dell'array:

```
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.
```

Come si modifica la dimensione di un array?

La semplice risposta è che non puoi farlo. Una volta creata una matrice, la sua dimensione non può essere modificata. Invece, un array può essere "ridimensionato" solo creando una nuova matrice con le dimensioni appropriate e copiando gli elementi dall'array esistente a quello nuovo.

```
String[] listOfCities = new String[3];   // array created with size 3.
listOfCities[0] = "New York";
listOfCities[1] = "London";
listOfCities[2] = "Berlin";
```

Supponiamo (per esempio) che un nuovo elemento debba essere aggiunto alla matrice `listOfCities` definita come sopra. Per fare ciò, dovrai:

1. creare un nuovo array con dimensione 4,
2. copia i 3 elementi esistenti del vecchio array sul nuovo array con gli offset 0, 1 e 2 e
3. aggiungi il nuovo elemento al nuovo array all'offset 3.

Ci sono vari modi per fare quanto sopra. Prima di Java 6, il modo più conciso era:

```
String[] newArray = new String[listOfCities.length + 1];
System.arraycopy(listOfCities, 0, newArray, 0, listOfCities.length);
newArray[listOfCities.length] = "Sydney";
```

Da Java 6 in poi, i metodi `Arrays.copyOf` e `Arrays.copyOfRange` possono farlo più semplicemente:

```
String[] newArray = Arrays.copyOf(listOfCities, listOfCities.length + 1);
newArray[listOfCities.length] = "Sydney";
```

Per altri modi di copiare un array, fare riferimento al seguente esempio. Tieni presente che per il ridimensionamento hai bisogno di una copia dell'array con una lunghezza diversa rispetto all'originale.

- Copia di array

Una migliore alternativa al ridimensionamento degli array

Esistono due principali svantaggi con il ridimensionamento di un array come descritto sopra:

- È inefficiente Rendere un array più grande (o più piccolo) implica copiare molti o tutti gli elementi dell'array esistenti e allocare un nuovo oggetto array. Più grande è l'array, più costoso diventa.
- Devi essere in grado di aggiornare qualsiasi variabile "live" che contiene riferimenti al vecchio array.

Un'alternativa è creare l'array con una dimensione abbastanza grande per iniziare. Questo è possibile solo se è possibile determinare esattamente la dimensione *prima di allocare la matrice* . Se non è possibile farlo, il problema del ridimensionamento della matrice si ripresenta.

L'altra alternativa è usare una classe di struttura dati fornita dalla libreria di classi Java SE o da una libreria di terze parti. Ad esempio, il framework "collections" di Java SE fornisce una serie di implementazioni delle API `List` , `Set` e `Map` con diverse proprietà di runtime. La classe `ArrayList` è la più vicina alle caratteristiche prestazionali di un array semplice (ad es. O (N) lookup, O (1) get e set, O (N) inserimento casuale e cancellazione) fornendo un ridimensionamento più efficiente senza il problema dell'aggiornamento di riferimento.

(L'efficienza di ridimensionamento per `ArrayList` deriva dalla sua strategia di raddoppiamento della dimensione dell'array di supporto su ogni ridimensionamento.Per un tipico caso d'uso, questo significa che si ridimensiona solo occasionalmente. Quando si ammortizza per tutta la durata della lista, il costo di ridimensionamento per insert è `O(1)` . Potrebbe essere possibile utilizzare la stessa strategia per ridimensionare un array semplice.)

Trovare un elemento in una matrice

Esistono molti modi per trovare la posizione di un valore in una matrice. I seguenti frammenti di esempio presuppongono che l'array sia uno dei seguenti:

```
String[] strings = new String[] { "A", "B", "C" };
int[] ints = new int[] { 1, 2, 3, 4 };
```

Inoltre, ognuno imposta `index` o `index2` sull'indice dell'elemento richiesto o `-1` se l'elemento non è presente.

Utilizzo di `Arrays.binarySearch` (solo per gli array ordinati)

```
int index = Arrays.binarySearch(strings, "A");
int index2 = Arrays.binarySearch(ints, 1);
```

Uso di `Arrays.asList` (solo per gli array non primitivi)

```
int index = Arrays.asList(strings).indexOf("A");
int index2 = Arrays.asList(ints).indexOf(1);  // compilation error
```

Utilizzando un `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
```

Ricerca lineare usando un ciclo

```
  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
```

Ricerca lineare utilizzando librerie di terze parti come org.apache.commons

```
  int index = org.apache.commons.lang3.ArrayUtils.contains(strings, "A");
  int index2 = org.apache.commons.lang3.ArrayUtils.contains(ints, 1);
```

Nota: l'utilizzo di una ricerca lineare diretta è più efficiente rispetto al wrapping in un elenco.

Verificare se una matrice contiene un elemento

Gli esempi sopra possono essere adattati per verificare se la matrice contiene un elemento semplicemente testando per vedere se l'indice calcolato è maggiore o uguale a zero.

In alternativa, ci sono anche alcune varianti più concise:

```
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);
```

Ordinamento di matrici

Ordinare gli array può essere fatto facilmente con l'API Array .

```
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));
```

Ordinamento degli array di stringhe:

`String` non è un dato numerico, definisce il proprio ordine che è chiamato ordine lessicografico, noto anche come ordine alfabetico. Quando si ordina una matrice di String usando il metodo `sort()` , ordina l'array in un ordine naturale definito dall'interfaccia Comparabile, come mostrato di seguito:

Ordine crescente

```
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));
```

Produzione:

```
String array before sorting : [John, Steve, Shane, Adam, Ben]
String array after sorting in ascending order : [Adam, Ben, John, Shane, Steve]
```

Decrescente ordine

```
Arrays.sort(names, 0, names.length, Collections.reverseOrder());
System.out.println("String array after sorting in descending order : " +
Arrays.toString(names));
```

Produzione:

```
String array after sorting in descending order : [Steve, Shane, John, Ben, Adam]
```

Ordinamento di una matrice di oggetti

Per ordinare un array di oggetti, tutti gli elementi devono implementare l'interfaccia `Comparable` o `Comparator` per definire l'ordine dell'ordinamento.

Possiamo usare il metodo `sort(Object[])` per ordinare un array di oggetti nel suo ordine naturale,

ma è necessario assicurarsi che tutti gli elementi dell'array implementino `Comparable` .

Inoltre, devono essere reciprocamente comparabili, ad esempio `e1.compareTo(e2)` non deve lanciare *`ClassCastException`* per qualsiasi elemento e1 ed e2 nella matrice. In alternativa è possibile ordinare una matrice di oggetti su ordine personalizzato usando il metodo `sort(T[], Comparator)` come mostrato nell'esempio seguente.

```
// 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));
```

Produzione:

```
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 ]
```

Conversione di matrici tra primitive e tipi scatolati

A volte è necessaria la conversione di tipi primitivi in tipi in scatola .

Per convertire l'array, è possibile utilizzare gli stream (in Java 8 e versioni successive):

Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray =
    Arrays.stream(primitiveArray).boxed().toArray(Integer[]::new);
```

Con versioni inferiori può essere iterando l'array primitivo e copiandolo esplicitamente all'array scatolato:

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
}
```

Allo stesso modo, una matrice in scatola può essere convertita in una matrice della sua controparte 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
}
```

Leggi Array online: https://riptutorial.com/it/java/topic/99/array

Capitolo 13: Audio

Osservazioni

Invece di usare javax.sound.sampled `Clip` , puoi usare anche `AudioClip` che proviene dall'API dell'applet. Si consiglia comunque di utilizzare `Clip` poiché `AudioClip` è solo più vecchio e presenta funzionalità limitate.

Examples

Riproduce un file audio in loop

Importazioni necessarie:

```
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
```

Questo codice creerà una clip e la riprodurrà continuamente una volta avviata:

```
Clip clip = AudioSystem.getClip();
clip.open(AudioSystem.getAudioInputStream(new URL(filename)));
clip.start();
clip.loop(Clip.LOOP_CONTINUOUSLY);
```

Ottieni una matrice con tutti i tipi di file supportati:

```
AudioFileFormat.Type [] audioFileTypes = AudioSystem.getAudioFileTypes();
```

Riproduci un file MIDI

I file MIDI possono essere riprodotti utilizzando diverse classi dal pacchetto `javax.sound.midi` . Un `Sequencer` esegue la riproduzione del file MIDI e molti dei suoi metodi possono essere utilizzati per impostare i controlli di riproduzione come il conteggio dei loop, il tempo, il silenziamento della traccia e altri.

La riproduzione generale dei dati MIDI può essere eseguita in questo modo:

```
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();
        }
    }
}
```

Per interrompere la riproduzione, utilizzare:

```
sequencer.stop(); // Stop the playback
```

Un sequencer può essere impostato per silenziare una o più tracce della sequenza durante la riproduzione in modo che nessuno degli strumenti in quelli specificati suoni. L'esempio seguente imposta la prima traccia nella sequenza da disattivare:

```
import javax.sound.midi.Track;
// ...

Track[] track = sequence.getTracks();
sequencer.setTrackMute(track[0]);
```

Un sequencer può riprodurre ripetutamente una sequenza se viene dato il numero di cicli. Quanto segue imposta il sequencer per riprodurre una sequenza quattro volte e indefinitamente:

```
sequencer.setLoopCount(3);
sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
```

Il sequencer non deve sempre suonare la sequenza dall'inizio, né deve suonare la sequenza fino alla fine. Può iniziare e terminare in qualsiasi momento specificando il *segno* di *spunta* nella sequenza da cui iniziare e terminare. È anche possibile specificare manualmente quale tick nella sequenza che il sequencer dovrebbe riprodurre da:

```
sequencer.setLoopStartPoint(512);
sequencer.setLoopEndPoint(32768);
sequencer.setTickPosition(8192);
```

I sequencer possono anche riprodurre un file MIDI ad un determinato tempo, che può essere controllato specificando il tempo in battiti al minuto (BPM) o microsecondi per nota da un quarto (MPQ). Anche il fattore con cui viene riprodotta la sequenza può essere regolato.

```
sequencer.setTempoInBPM(1250f);
sequencer.setTempoInMPQ(4750f);
sequencer.setTempoFactor(1.5f);
```

Quando hai finito di usare il `Sequencer` , ricorda di chiuderlo

```
sequencer.close();
```

Suono di metallo nudo

Puoi anche fare quasi bare metal quando produci suoni con java. Questo codice scriverà dati binari grezzi nel buffer audio del sistema operativo per generare suoni. È estremamente importante capire le limitazioni e i calcoli necessari per generare un suono come questo. Poiché la riproduzione è fondamentalmente istantanea, i calcoli devono essere eseguiti quasi in tempo reale.

In quanto tale, questo metodo è inutilizzabile per un campionamento del suono più complicato. Per tali scopi utilizzando strumenti specializzati è l'approccio migliore.

Il seguente metodo genera e genera direttamente un'onda di rettangolo di una determinata frequenza in un dato volume per una data durata.

```
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
}
```

Per un modo più differenziato di generare differenti calcoli del seno sonoro e possibilmente dimensioni del campione più grandi sono necessarie. Ciò si traduce in codice significativamente più complesso ed è quindi omesso qui.

Uscita audio di base

Hello Audio! di Java che riproduce un file audio dall'archivio locale o su Internet, appare come segue. Funziona con file wav non compressi e non deve essere utilizzato per riprodurre file mp3 o compressi.

```
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();
   }
}
```

Leggi Audio online: https://riptutorial.com/it/java/topic/160/audio

Capitolo 14: autoboxing

introduzione

Autoboxing è la conversione automatica effettuata dal compilatore Java tra i tipi primitivi e le corrispondenti classi wrapper degli oggetti. Esempio, convertendo int -> Integer, double -> Double ... Se la conversione va diversamente, viene chiamato unboxing. In genere, questo viene utilizzato in raccolte che non possono contenere oggetti diversi da Oggetti, in cui sono necessari i tipi di boxing primitivi prima di impostarli nella raccolta.

Osservazioni

Autoboxing può avere problemi di prestazioni se usato frequentemente nel tuo codice.

- http://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html
- L'auto-unboxing intero e il boxing automatico offrono problemi di prestazioni?

Examples

Utilizzando int e Integer in modo intercambiabile

Poiché si utilizzano tipi generici con classi di utilità, è possibile che i tipi di numeri non siano molto utili quando vengono specificati come tipi di oggetto, in quanto non sono uguali alle loro controparti primitive.

```
List<Integer> ints = new ArrayList<Integer>();
```

Java SE 7

```
List<Integer> ints = new ArrayList<>();
```

Fortunatamente, le espressioni che valutano in `int` possono essere utilizzate al posto di un `Integer` quando è necessario.

```
for (int i = 0; i < 10; i++)
    ints.add(i);
```

Il `ints.add(i);` la dichiarazione è equivalente a:

```
ints.add(Integer.valueOf(i));
```

E conserva le proprietà dal valore `Integer#valueOf` come se avesse gli stessi oggetti `Integer` memorizzati nella cache da JVM quando si trova nell'intervallo di memorizzazione nella cache dei numeri.

Questo vale anche per:

- `byte` e `Byte`
- `short` e `Short`
- `float` e `Float`
- `double` e `Double`
- `long` e `Long`
- `char` e `Character`
- `boolean` e `Boolean`

Bisogna fare attenzione, tuttavia, in situazioni ambigue. Considera il seguente codice:

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
ints.add(3);
ints.remove(1); // ints is now [1, 3]
```

L'interfaccia `java.util.List` contiene sia un `remove(int index)` (metodo di interfaccia `List`) che un `remove(Object o)` (metodo ereditato da `java.util.Collection`). In questo caso non avviene la boxe e si `remove(int index)` .

Un altro esempio di strano comportamento del codice Java causato da autoboxing Interi con valori nell'intervallo da `-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
```

Ciò accade perché `>=` operatore chiama implicitamente `intValue()` che restituisce `int` mentre `==` confronta i **riferimenti** , non i valori `int` .

Per impostazione predefinita, Java memorizza nella cache i valori nell'intervallo `[-128, 127]` , quindi l'operatore `==` funziona perché i `Integers` di questo intervallo fanno riferimento agli stessi oggetti se i loro valori sono uguali. Il valore massimo dell'intervallo memorizzabile nella cache può essere definito con `-XX:AutoBoxCacheMax` opzione `-XX:AutoBoxCacheMax` JVM. Quindi, se esegui il programma con `-XX:AutoBoxCacheMax=1000` , il seguente codice verrà stampato `true` :

```
Integer a = 1000;
Integer b = 1000;
System.out.println(a == b); // true
```

Usando l'istruzione booleana in if

A causa dell'auto-unboxing, si può usare un `Boolean` in un'istruzione `if` :

```
Boolean a = Boolean.TRUE;
if (a) { // a gets converted to boolean
    System.out.println("It works!");
}
```

Questo funziona per `while` , `do while` e la condizione nelle istruzioni `for` pure.

Nota che, se il `Boolean` è `null` , verrà `NullPointerException` una `NullPointerException` nella conversione.

L'unbox automatico può portare a NullPointerException

Questo codice compila:

```
Integer arg = null;
int x = arg;
```

Ma si romperà in fase di esecuzione con una `java.lang.NullPointerException` sulla seconda riga.

Il problema è che un `int` primitivo non può avere un valore `null` .

Questo è un esempio minimalista, ma nella pratica si manifesta spesso in forme più sofisticate. La `NullPointerException` non è molto intuitiva e spesso è di scarso aiuto nell'individuazione di tali bug.

Affidati alla funzione di autoboxing e auto-unboxing con attenzione, assicurati che i valori non condivisi non abbiano valori `null` in fase di runtime.

Memoria e overhead computazionale di Autoboxing

Autoboxing può arrivare a un sovraccarico di memoria sostanziale. Per esempio:

```
Map<Integer, Integer> square = new HashMap<Integer, Integer>();
for(int i = 256; i < 1024; i++) {
    square.put(i, i * i); // Autoboxing of large integers
}
```

in genere consuma una notevole quantità di memoria (circa 60kb per 6k di dati effettivi).

Inoltre, gli interi in scatola di solito richiedono ulteriori round trip nella memoria, e quindi rendono meno efficaci le cache della CPU. Nell'esempio sopra, la memoria a cui si accede è distribuita in cinque posizioni diverse che possono trovarsi in regioni completamente diverse della memoria: 1. l'oggetto `HashMap` , 2. l'oggetto della `Entry[] table` della mappa, 3. l'oggetto `Entry` , 4. il affida l'oggetto `key` (inscatola la chiave primitiva), 5. l'oggetto `value` entrys (che inscatola il valore primitivo).

```
class Example {
  int primitive; // Stored directly in the class `Example`
  Integer boxed; // Reference to another memory location
}
```

La lettura in `boxed` richiede due accessi alla memoria, l'accesso alla `primitive` solo uno.

Quando si ricevono dati da questa mappa, il codice apparentemente innocente

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(i);
}
```

è equivalente a:

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(Integer.valueOf(i)).intValue();
}
```

In genere, il codice precedente causa la *creazione e la garbage collection* di un oggetto `Integer` per ogni operazione `Map#get(Integer)` . (Vedi la nota sotto per maggiori dettagli.)

Per ridurre questo overhead, diverse librerie offrono raccolte ottimizzate per tipi primitivi che *non* richiedono il pugilato. Oltre a evitare l'overhead di boxe, questa raccolta richiederà circa 4 volte meno memoria per voce. Mentre Java Hotspot *può* essere in grado di ottimizzare l'autoboxing lavorando con gli oggetti nello stack anziché con l'heap, non è possibile ottimizzare il sovraccarico della memoria e la conseguente deduzione della memoria.

Gli stream Java 8 hanno anche interfacce ottimizzate per tipi di dati primitivi, come `IntStream` che non richiede il pugilato.

Nota: un tipico runtime Java mantiene una semplice cache di `Integer` e di un altro oggetto wrapper primitivo utilizzato dai metodi `valueOf` factory e autoboxing. Per `Integer` , l'intervallo predefinito di questa cache è compreso tra -128 e +127. Alcune JVM forniscono un'opzione della riga di comando JVM per modificare la dimensione / intervallo della cache.

Casi diversi Quando Integer e int possono essere utilizzati in modo intercambiabile

Caso 1: durante l'utilizzo al posto degli argomenti del metodo.

Se un metodo richiede un oggetto di classe wrapper come argomento. Quindi l'argomento può essere passato una variabile del rispettivo tipo primitivo e viceversa.

Esempio:

```
int i;
Integer j;
void ex_method(Integer i)//Is a valid statement
void ex_method1(int j)//Is a valid statement
```

Caso 2: durante il passaggio dei valori di ritorno:

Quando un metodo restituisce una variabile di tipo primitivo, un oggetto della corrispondente classe wrapper può essere passato come valore di ritorno in modo intercambiabile e viceversa.

Esempio:

```
int i;
Integer j;
int ex_method()
{...
return j;}//Is a valid statement
Integer ex_method1()
{...
return i;//Is a valid statement
}
```

Caso 3: durante l'esecuzione delle operazioni.

Ogni volta che si eseguono operazioni su numeri, la variabile del tipo primitivo e l'oggetto della rispettiva classe wrapper possono essere usati in modo intercambiabile.

```
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
```

Trappola : ricorda di inizializzare o assegnare un valore a un oggetto della classe wrapper.

Mentre si usa l'oggetto classe wrapper e la variabile primitiva intercambiabilmente non si dimentica o manca di inizializzare o assegnare un valore all'oggetto classe wrapper altrimenti può portare all'eccezione del puntatore nullo in fase di runtime.

Esempio:

```
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
    }
```

Nell'esempio sopra, il valore dell'oggetto non è assegnato e non è inizializzato e quindi in fase di esecuzione il programma verrà eseguito in un'eccezione di puntatore nullo. Pertanto, come illustrato nell'esempio precedente, il valore dell'oggetto non deve mai essere non inizializzato e non assegnato.

Leggi autoboxing online: https://riptutorial.com/it/java/topic/138/autoboxing

Capitolo 15: Bandiere JVM

Osservazioni

Si consiglia vivamente di utilizzare solo queste opzioni:

- Se hai una conoscenza approfondita del tuo sistema.
- Sono consapevoli che, se usate in modo improprio, queste opzioni possono avere un effetto negativo sulla stabilità o sulle prestazioni del sistema.

Informazioni raccolte dalla documentazione ufficiale di Java .

Examples

-XXaggressive

`-XXaggressive` è una raccolta di configurazioni che rendono la JVM performante ad alta velocità e raggiunge uno stato stabile il prima possibile. Per raggiungere questo obiettivo, la JVM utilizza più risorse interne all'avvio; tuttavia, richiede una ottimizzazione meno adattiva una volta raggiunto l'obiettivo. Ti consigliamo di utilizzare questa opzione per applicazioni a uso intensivo e con memoria intensiva che funzionano da sole.

Uso:

```
-XXaggressive:<param>
```

<Param>	Descrizione
`opt`	Pianifica le ottimizzazioni adattive in anticipo e consente nuove ottimizzazioni, che dovrebbero essere predefinite nelle versioni future.
`memory`	Configura il sistema di memoria per i carichi di lavoro a uso intensivo di memoria e imposta l'aspettativa di abilitare grandi quantità di risorse di memoria per garantire un throughput elevato. JRockit JVM utilizzerà anche pagine di grandi dimensioni, se disponibili.

-XXallocClearChunks

Questa opzione ti consente di cancellare un TLA per riferimenti e valori al momento dell'allocazione TLA e di prelavare il blocco successivo. Quando viene dichiarato un numero intero, un riferimento o qualsiasi altra cosa, ha un valore predefinito di 0 o null (a seconda del tipo). Al momento opportuno, sarà necessario cancellare questi riferimenti e valori per liberare la memoria sull'heap in modo che Java possa usarlo o riutilizzarlo. Puoi farlo quando l'oggetto è allocato o, usando questa opzione, quando richiedi un nuovo TLA.

Uso:

```
-XXallocClearChunks
```

```
-XXallocClearChunks=<true | false>
```

Quanto sopra è un'opzione booleana ed è generalmente raccomandato sui sistemi IA64; in definitiva, il suo utilizzo dipende dall'applicazione. Se si desidera impostare la dimensione dei blocchi deselezionati, combinare questa opzione con `-XXallocClearChunkSize` . Se si utilizza questo flag ma non si specifica un valore booleano, il valore predefinito è `true` .

-XXallocClearChunkSize

Se utilizzato con `-XXallocClearChunkSize` , questa opzione imposta la dimensione dei blocchi da cancellare. Se questo flag viene utilizzato ma non viene specificato alcun valore, il valore predefinito è 512 byte.

Uso:

```
-XXallocClearChunks -XXallocClearChunkSize=<size>[k|K][m|M][g|G]
```

-XXcallProfiling

Questa opzione abilita l'uso del profilo di chiamata per l'ottimizzazione del codice. La creazione di profili registra utili statistiche di runtime specifiche per l'applicazione e, in molti casi, può aumentare le prestazioni perché JVM può quindi agire su tali statistiche.

> Nota: questa opzione è supportata con JRockit JVM R27.3.0 e versioni successive. Potrebbe diventare predefinito nelle versioni future.

Uso:

```
java -XXcallProfiling myApp
```

Questa opzione è disabilitata di default. Devi abilitarlo per usarlo.

-XXdisableFatSpin

Questa opzione disabilita il codice di rotazione del blocco fat in Java, consentendo ai thread che bloccano il tentativo di acquisire un blocco fat andare direttamente in sospensione.

Gli oggetti in Java diventano un blocco non appena un thread entra in un blocco sincronizzato su quell'oggetto. Tutte le serrature vengono mantenute (ovvero bloccate) finché non vengono rilasciate dal filo di bloccaggio. Se il blocco non verrà rilasciato molto velocemente, può essere gonfiato in un "blocco di grasso". "Spinning" si verifica quando un thread che desidera un blocco specifico controlla continuamente tale blocco per vedere se è ancora preso, ruotando in un ciclo stretto come rende il controllo. Spinning contro un blocco del grasso è generalmente vantaggioso

anche se, in alcuni casi, può essere costoso e potrebbe influire sulle prestazioni. `-XXdisableFatSpin` consente di disattivare la rotazione contro un blocco di grasso ed eliminare il potenziale `-XXdisableFatSpin` sulle prestazioni.

Uso:

```
-XXdisableFatSpin
```

-XXdisableGCHeuristics

Questa opzione disabilita le modifiche alla strategia di Garbage Collector. L'euristica della compattazione e l'euristica delle dimensioni del vivaio non sono influenzate da questa opzione. Per impostazione predefinita, l'euristica della garbage collection è abilitata.

Uso:

```
-XXdisableFatSpin
```

-XXdumpSize

Questa opzione provoca la generazione di un file di dump e consente di specificare la dimensione relativa di quel file (cioè piccolo, medio o grande).

Uso:

```
-XXdumpsize:<size>
```

<Dimensioni>	Descrizione
`none`	Non genera un file di dettagli.
`small`	Su Windows, viene generato un piccolo file di dump (su Linux viene generato un dump core completo). Una piccola discarica include solo le pile di thread incluse le loro tracce e molto poco altro. Questo era l'impostazione predefinita in JRockit JVM 8.1 con i service pack 1 e 2, nonché 7.0 con Service Pack 3 e versioni successive).
`normal`	Fa sì che venga generato un normale dump su tutte le piattaforme. Questo file di dettagli include tutta la memoria tranne l'heap java. Questo è il valore predefinito per JRockit JVM 1.4.2 e versioni successive.
`large`	Include tutto ciò che è in memoria, incluso l'heap Java. Questa opzione rende `-XXdumpSize` equivalente a `-XXdumpFullState` .

-XXexitOnOutOfMemory

Questa opzione fa sì che JRockit JVM esca al primo verificarsi di un errore di memoria

insufficiente. Può essere usato se si preferisce riavviare un'istanza di Jock di JRockit piuttosto che gestire gli errori di memoria insufficiente. Immettere questo comando all'avvio per forzare JRockit JVM a uscire alla prima occorrenza di un errore di memoria insufficiente.

Uso:

```
-XXexitOnOutOfMemory
```

Leggi Bandiere JVM online: https://riptutorial.com/it/java/topic/2500/bandiere-jvm

Capitolo 16: benchmark

introduzione

Scrivere benchmark delle prestazioni in java non è così semplice come ottenere `System.currentTimeMillis()` all'inizio e alla fine e calcolare la differenza. Per scrivere benchmark delle prestazioni validi, si dovrebbero usare strumenti adeguati.

Examples

Semplice esempio JMH

JMH è uno degli strumenti per scrivere i test di benchmark appropriati . Diciamo che vogliamo confrontare le prestazioni della ricerca di un elemento in `HashSet` vs `TreeSet` .

Il modo più semplice per ottenere JHM nel tuo progetto è utilizzare il plugin maven e shade . Inoltre puoi vedere `pom.xml` dagli esempi 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>
```

Dopo questo è necessario scrivere la classe di benchmark stessa:

```
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));
    }
}
```

Si prega di tenere presente questo `blackhole.consume()` , ci torneremo più tardi. Inoltre abbiamo bisogno della classe principale per il benchmark in esecuzione:

```
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();
    }
}
```

E siamo a posto. Abbiamo solo bisogno di eseguire il `mvn package` (creerà `benchmarks.jar` nella cartella `/target`) ed eseguire il test di benchmark:

```
java -cp target/benchmarks.jar benchmark.BenchmarkMain
```

E dopo alcune iterazioni di riscaldamento e calcolo, avremo i nostri risultati:

```
# 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
```

A proposito di `blackhole.consume()` . Se i tuoi calcoli non cambiano lo stato della tua applicazione, molto probabilmente lo ignorerai. Quindi, per evitarlo, puoi rendere i tuoi metodi di riferimento restituire un valore, o usare l'oggetto `Blackhole` per consumarlo.

Puoi trovare maggiori informazioni sulla scrittura di benchmark appropriati nel blog di Aleksey Shipilëv , nel blog di Jacob Jenkov e nel blog di java-performance: 1 , 2 .

Leggi benchmark online: https://riptutorial.com/it/java/topic/9514/benchmark

Capitolo 17: BigDecimal

introduzione

La classe BigDecimal fornisce operazioni per l'aritmetica (addizione, sottrazione, moltiplicazione, divisione), manipolazione della scala, arrotondamento, confronto, hashing e conversione del formato. Il BigDecimal rappresenta i numeri decimali firmati immutabili, con precisione arbitraria. Questa classe deve essere utilizzata in una necessità di calcolo ad alta precisione.

Examples

Gli oggetti BigDecimal sono immutabili

Se vuoi calcolare con BigDecimal devi usare il valore restituito perché gli oggetti BigDecimal sono immutabili:

```
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
```

Confronto di BigDecimals

Il metodo `compareTo` dovrebbe essere usato per confrontare `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
```

Comunemente **non** si dovrebbe usare il `equals` metodo in quanto considera due `BigDecimals` uguali solo se sono uguali in valore e anche **scala:**

```
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
```

Operazioni matematiche con BigDecimal

Questo esempio mostra come eseguire operazioni matematiche di base usando BigDecimals.

1.Addition

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a + b
BigDecimal result = a.add(b);
System.out.println(result);
```

Risultato: 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);
```

Risultato: -2

3.Multiplication

Quando si moltiplicano due `BigDecimal` , il risultato avrà una scala uguale alla somma delle scale degli operandi.

```
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);
```

Risultato: 36.89931

Per modificare la scala del risultato utilizzare il metodo di moltiplicazione sovraccarico che consente di passare `MathContext` , un oggetto che descrive le regole per gli operatori, in particolare la precisione e la modalità di arrotondamento del risultato. Per ulteriori informazioni sulle modalità di arrotondamento disponibili, consultare la documentazione 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);
```

Risultato: 36.90

4.Division

La divisione è un po 'più complicata delle altre operazioni aritmetiche, ad esempio si consideri l'esempio seguente:

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

BigDecimal result = a.divide(b);
System.out.println(result);
```

Ci aspettiamo che questo dia qualcosa di simile a: 0.7142857142857143, ma otterremmo:

> Risultato: **java.lang.ArithmeticException: espansione decimale senza terminazione; nessun risultato decimale rappresentabile esatto.**

Ciò funzionerebbe perfettamente quando il risultato sarebbe un decimale terminante dire se volessi dividere 5 per 2, ma per quei numeri che al momento della divisione darebbero un risultato non terminante otterremmo una `ArithmeticException` . Nello scenario del mondo reale, non è possibile prevedere i valori che verrebbero rilevati durante la divisione, quindi è necessario specificare la **scala** e la **modalità di arrotondamento** per la divisione BigDecimal. Per ulteriori informazioni sulla modalità di scala e arrotondamento, consultare la documentazione Oracle .

Ad esempio, potrei fare:

```
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);
```

> Risultato: 0,7142857143

5. Remainder o modulo

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

//Equivalent to result = a % b
BigDecimal result = a.remainder(b);
System.out.println(result);
```

> Risultato: 5

6.Power

```
BigDecimal a = new BigDecimal("5");

//Equivalent to result = a^10
BigDecimal result = a.pow(10);
System.out.println(result);
```

Risultato: 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);
```

Risultato: 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);
```

Risultato: 5

9.Move Point To Left

```
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);
```

Risultato: 52.3449843776

10. Spostare il punto verso destra

```
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);
```

Risultato: 5234498.43776

Ci sono molte più opzioni e combinazioni di parametri per gli esempi sopra citati (ad esempio, ci sono 6 varianti del metodo di divisione), questo insieme è un elenco non esaustivo e copre alcuni esempi di base.

Utilizzo di BigDecimal anziché float

A causa del modo in cui il tipo float è rappresentato nella memoria del computer, i risultati delle operazioni che utilizzano questo tipo possono essere inaccurati - alcuni valori sono memorizzati come approssimazioni. Buoni esempi di questo sono i calcoli monetari. Se è necessaria un'elevata precisione, è necessario utilizzare altri tipi. ad es. Java 7 fornisce 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));
}
```

L'output di questo programma è:

```
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
```

Per un saldo iniziale di 10000,00, dopo 1000 operazioni per 1,99, ci aspettiamo che il saldo sia 8010,00. L'uso del tipo float ci fornisce una risposta intorno a 8009,77, che è inaccettabilmente imprecisa nel caso di calcoli monetari. Usando BigDecimal ci dà il risultato corretto.

BigDecimal.valueOf ()

La classe BigDecimal contiene una cache interna di numeri utilizzati frequentemente, ad esempio da 0 a 10. I metodi BigDecimal.valueOf () vengono forniti preferibilmente a costruttori con parametri di tipo simili, ovvero nell'esempio sotto a è preferito 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
```

Inizializzazione di BigDecimals con valore zero, uno o dieci

BigDecimal fornisce proprietà statiche per i numeri zero, uno e dieci. È buona norma utilizzare questi invece dei numeri effettivi:

- `BigDecimal.ZERO`
- `BigDecimal.ONE`
- `BigDecimal.TEN`

Usando le proprietà statiche, si evita un'istanza non necessaria, inoltre si ha un letterale nel codice anziché un "numero magico".

```
//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;
```

Leggi BigDecimal online: https://riptutorial.com/it/java/topic/1667/bigdecimal

Capitolo 18: BigInteger

introduzione

La classe `BigInteger` viene utilizzata per operazioni matematiche che coinvolgono numeri interi grandi con grandezze troppo grandi per tipi di dati primitivi. Ad esempio 100-factorial è composto da 158 cifre, molto più grandi di quelle `long` . `BigInteger` fornisce analoghi a tutti gli operatori di interi primitivi di Java e tutti i metodi pertinenti di `java.lang.Math` e alcune altre operazioni.

Sintassi

- BigInteger variable_name = new BigInteger ("12345678901234567890"); // un intero decimale come stringa
- BigInteger variable_name = new BigInteger ("10101011010101001010100110001100111010110001111100001010110100 10", 2) // un numero intero binario come stringa
- BigInteger variable_name = new BigInteger ("ab54a98ceb1f0800", 16) // un numero intero esadecimale come stringa
- BigInteger variable_name = new BigInteger (64, new Random ()); // un generatore di numeri pseudocasuali che fornisce 64 bit per costruire un numero intero
- BigInteger variable_name = new BigInteger (nuovo byte [] {0, -85, 84, -87, -116, -21, 31, 10, -46}); // ha firmato la rappresentazione a complemento di due di un intero (big endian)
- BigInteger variable_name = new BigInteger (1, new byte [] {- 85, 84, -87, -116, -21, 31, 10, -46}); // rappresentazione di complementi a due senza segno di un numero intero positivo (big endian)

Osservazioni

`BigInteger` è immutabile. Quindi non puoi cambiare il suo stato. Ad esempio, quanto segue non funzionerà in quanto la `sum` non verrà aggiornata a causa dell'immutabilità.

```
BigInteger sum = BigInteger.ZERO;
for(int i = 1; i < 5000; i++) {
   sum.add(BigInteger.valueOf(i));
}
```

Assegna il risultato alla variabile `sum` per farlo funzionare.

```
sum = sum.add(BigInteger.valueOf(i));
```

Java SE 8

La documentazione ufficiale di `BigInteger` afferma che `BigInteger` implementazioni di `BigInteger` dovrebbero supportare tutti i numeri interi compresi tra -2 ^{2147483647} e 2 ^{2147483647} (esclusivo).

Ciò significa che `BigInteger` può avere più di 2 *miliardi di* bit!

Examples

Inizializzazione

La classe `java.math.BigInteger` fornisce operazioni analoghe a tutti gli operatori di numeri interi primitivi di Java e per tutti i metodi rilevanti da `java.lang.Math` . Poiché il pacchetto `java.math` non viene reso automaticamente disponibile, potrebbe essere necessario importare `java.math.BigInteger` prima di poter utilizzare il nome semplice della classe.

Per convertire valori `long` o `int` in `BigInteger` utilizzare:

```
long longValue = Long.MAX_VALUE;
BigInteger valueFromLong = BigInteger.valueOf(longValue);
```

o, per i numeri interi:

```
int intValue = Integer.MIN_VALUE; // negative
BigInteger valueFromInt = BigInteger.valueOf(intValue);
```

che si *amplia* `intValue` intero a lungo, con segno estensione bit per i valori negativi, in modo che i valori negativi rimarranno negativo.

Per convertire una `String` numerica in `BigInteger` utilizzare:

```
String decimalString = "-1";
BigInteger valueFromDecimalString = new BigInteger(decimalString);
```

Il costruttore seguente viene utilizzato per tradurre la rappresentazione String di un `BigInteger` nella radice specificata in un `BigInteger` .

```
String binaryString = "10";
int binaryRadix = 2;
BigInteger valueFromBinaryString = new BigInteger(binaryString , binaryRadix);
```

Java supporta anche la conversione diretta di byte in un'istanza di `BigInteger` . Attualmente è possibile utilizzare solo la codifica big endian con segno e senza segno:

```
byte[] bytes = new byte[] { (byte) 0x80 };
BigInteger valueFromBytes = new BigInteger(bytes);
```

Questo genererà un'istanza `BigInteger` con valore -128 poiché il primo bit viene interpretato come il bit di segno.

```
byte[] unsignedBytes = new byte[] { (byte) 0x80 };
int sign = 1; // positive
```

```
BigInteger valueFromUnsignedBytes = new BigInteger(sign, unsignedBytes);
```

Ciò genererà un'istanza `BigInteger` con il valore 128 poiché i byte vengono interpretati come numero senza segno e il segno è impostato esplicitamente su 1, un numero positivo.

Esistono costanti predefinite per valori comuni:

- `BigInteger.ZERO` - valore di "0".
- `BigInteger.ONE` - valore di "1".
- `BigInteger.TEN` - valore di "10".

C'è anche `BigInteger.TWO` (valore di "2"), ma non puoi usarlo nel tuo codice perché è `private` .

Confronto tra i BigInteger

Puoi confrontare `BigIntegers` come se confrontassi `String` o altri oggetti in Java.

Per esempio:

```
BigInteger one = BigInteger.valueOf(1);
BigInteger two = BigInteger.valueOf(2);

if(one.equals(two)){
    System.out.println("Equal");
}
else{
    System.out.println("Not Equal");
}
```

Produzione:

```
Not Equal
```

Nota:

In generale, **non** utilizzare l'operatore `==` per confrontare BigIntegers

- `==` operatore: confronta i riferimenti; vale a dire se due valori si riferiscono allo stesso oggetto
- metodo `equals()` : confronta il contenuto di due BigIntegers.

Ad esempio, BigIntegers **non** dovrebbe essere confrontato nel modo seguente:

```
if (firstBigInteger == secondBigInteger) {
  // Only checks for reference equality, not content equality!
}
```

Ciò potrebbe portare a comportamenti imprevisti, in quanto l'operatore `==` controlla solo l'uguaglianza di riferimento. Se entrambi i BigInteger contengono lo stesso contenuto, ma non si riferiscono allo stesso oggetto, **ciò fallirà.** Invece, confronta BigIntegers usando i metodi `equals` , come spiegato sopra.

Puoi anche confrontare il tuo `BigInteger` con valori costanti come 0,1,10.

per esempio:

```
BigInteger reallyBig = BigInteger.valueOf(1);
if(BigInteger.ONE.equals(reallyBig)){
    //code when they are equal.
}
```

Puoi anche confrontare due BigInteger usando il metodo `compareTo()` , come segue: `compareTo()` restituisce 3 valori.

- **0:** quando entrambi sono **uguali** .
- **1:** Quando il primo è **maggiore del** secondo (quello tra parentesi).
- **-1:** quando il primo è **meno del** secondo

```
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.
}
```

Esempi di operazioni matematiche di BigInteger

BigInteger si trova in un oggetto immutabile, quindi è necessario assegnare i risultati di qualsiasi operazione matematica a una nuova istanza di BigInteger.

Aggiunta: *10 + 10 = 20*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("10");

BigInteger sum = value1.add(value2);
System.out.println(sum);
```

uscita: 20

Sottrazione: *10 - 9 = 1*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("9");

BigInteger sub = value1.subtract(value2);
System.out.println(sub);
```

uscita: 1

Divisione: *10/5 = 2*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

uscita: 2

Divisione: *17/4 = 4*

```
BigInteger value1 = new BigInteger("17");
BigInteger value2 = new BigInteger("4");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

uscita: 4

Moltiplicazione: *10 * 5 = 50*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger mul = value1.multiply(value2);
System.out.println(mul);
```

uscita: 50

Potenza: *10 ^ 3 = 1000*

```
BigInteger value1 = new BigInteger("10");
BigInteger power = value1.pow(3);
System.out.println(power);
```

uscita: 1000

Resto: *10% 6 = 4*

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("6");

BigInteger power = value1.remainder(value2);
System.out.println(power);
```

uscita: 4

GCD: Greatest Common Divisor (GCD) per `12` e `18` è `6` .

```
BigInteger value1 = new BigInteger("12");
BigInteger value2 = new BigInteger("18");
```

```
System.out.println(value1.gcd(value2));
```

Uscita: 6

Massimo di due BigInteger:

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.max(value2));
```

Uscita: 11

Minimo di due BigInteger:

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.min(value2));
```

Uscita: 10

Operazioni di logica binaria su BigInteger

BigInteger supporta anche le operazioni di logica binaria disponibili per i tipi `Number` . Come con tutte le operazioni, vengono implementate chiamando un metodo.

Binario o:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.or(val2);
```

Uscita: 11 (che equivale a `10 | 9`)

Binario E:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.and(val2);
```

Uscita: 8 (che equivale a `10 & 9`)

Binario Xor:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");
```

```
val1.xor(val2);
```

Uscita: 3 (che equivale a `10 ^ 9`)

RightShift:

```
BigInteger val1 = new BigInteger("10");

val1.shiftRight(1);   // the argument be an Integer
```

Uscita: 5 (equivalente a `10 >> 1`)

Tasto maiuscolo di sinistra:

```
BigInteger val1 = new BigInteger("10");

val1.shiftLeft(1);   // here parameter should be Integer
```

Uscita: 20 (equivalente a `10 << 1`)

Inversione binaria (non):

```
BigInteger val1 = new BigInteger("10");

val1.not();
```

Uscita: 5

*NAND (And-Not): **

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.andNot(val2);
```

Uscita: 7

Generazione di BigInteger casuali

La classe `BigInteger` ha un costruttore dedicato alla generazione di `BigIntegers` casuali, data un'istanza di `java.util.Random` e un `int` che specifica quanti bit avrà il `BigInteger` . Il suo utilizzo è abbastanza semplice: quando chiamate il costruttore `BigInteger(int, Random)` questo modo:

```
BigInteger randomBigInt = new BigInteger(bitCount, sourceOfRandomness);
```

quindi finirai con un `BigInteger` cui valore è compreso tra 0 (incluso) e 2 `bitCount` (esclusivo).

Ciò significa anche che il `new BigInteger(2147483647, sourceOfRandomness)` può restituire tutti i `BigInteger` positivi con un tempo sufficiente.

Che cosa sarà `sourceOfRandomness` dipende da te. Ad esempio, un `new Random()` è abbastanza buono nella maggior parte dei casi:

```
new BigInteger(32, new Random());
```

Se sei disposto a rinunciare alla velocità per numeri casuali di alta qualità, puoi invece utilizzare un `new SecureRandom ()` :

```
import java.security.SecureRandom;

// somewhere in the code...
new BigInteger(32, new SecureRandom());
```

Puoi addirittura implementare un algoritmo al volo con una classe anonima! Nota che il **lancio del tuo algoritmo RNG *ti farà* finire con una casualità di bassa qualità** , quindi assicurati sempre di usare un algoritmo che sia risultato accettabile a meno che tu non voglia che i `BigInteger` risultanti siano prevedibili.

```
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;
    }
});
```

Leggi BigInteger online: https://riptutorial.com/it/java/topic/1514/biginteger

Capitolo 19: BufferedWriter

Sintassi

- nuovo BufferedWriter (Writer); // Il costruttore predefinito
- BufferedWriter.write (int c); // Scrive un singolo carattere
- BufferedWriter.write (String str); // Scrive una stringa
- BufferedWriter.newLine (); // Scrive un separatore di riga
- BufferedWriter.close (); // Chiude BufferedWriter

Osservazioni

- Se si tenta di scrivere da un `BufferedWriter` (usando `BufferedWriter.write()`) dopo aver chiuso il `BufferedWriter` (usando `BufferedWriter.close()`), verrà `IOException` .
- Il costruttore di `BufferedWriter(Writer)` NON lancia `IOException` . Tuttavia, il costruttore `FileWriter(File)` genera un oggetto `FileNotFoundException` , che estende `IOException` . Quindi catturare `IOException` catturerà anche `FileNotFoundException` , non c'è mai bisogno di una seconda istruzione catch a meno che non si intenda fare qualcosa di diverso con `FileNotFoundException` .

Examples

Scrivi una riga di testo su File

Questo codice scrive la stringa in un file. E 'importante chiudere lo scrittore, quindi questo è fatto in un `finally` blocco.

```
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();
    }
  }
}
```

Si noti inoltre che `write(String s)` non inserisce caratteri di nuova riga dopo che la stringa è stata scritta. Per dirla usa il metodo `newLine()` .

Java SE 7

Java 7 aggiunge il pacchetto `java.nio.file` e 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);
    }
}
```

Leggi BufferedWriter online: https://riptutorial.com/it/java/topic/3063/bufferedwriter

Capitolo 20: ByteBuffer

introduzione

La classe `ByteBuffer` è stata introdotta in java 1.4 per facilitare il lavoro sui dati binari. È particolarmente adatto per l'uso con dati di tipo primitivi. Permette la creazione, ma anche la successiva manipolazione di un `byte[]` s su un livello di astrazione più alto

Sintassi

- 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);
- short s = buffer.getShort (10);
- buffer.put ((byte) 120);
- buffer.putChar ('a');

Examples

Utilizzo di base - Creazione di un ByteBuffer

Esistono due modi per creare un `ByteBuffer` , in cui è possibile suddividerlo nuovamente.

Se hai un `byte[]` già esistente `byte[]` , puoi *"racchiuderlo"* in un `ByteBuffer` per semplificare l'elaborazione:

```
byte[] reqBuffer = new byte[BUFFER_SIZE];
int readBytes = socketInputStream.read(reqBuffer);
final ByteBuffer reqBufferWrapper = ByteBuffer.wrap(reqBuffer);
```

Questa sarebbe una possibilità per il codice che gestisce le interazioni di rete di basso livello

---

Se non si dispone di un `byte[]` già esistente `byte[]` , è possibile creare un `ByteBuffer` su un array specificamente assegnato per il buffer in questo modo:

```
final ByteBuffer respBuffer = ByteBuffer.allocate(RESPONSE_BUFFER_SIZE);
putResponseData(respBuffer);
socketOutputStream.write(respBuffer.array());
```

Se il percorso del codice è estremamente critico per le prestazioni ed è necessario **l'accesso diretto alla memoria di sistema** , `ByteBuffer` può anche allocare buffer *diretti* usando `#allocateDirect()`

Utilizzo di base: scrittura dei dati nel buffer

Data un'istanza `ByteBuffer` possibile scrivere dati di tipo primitivo usando *la* `put` *relativa* e *assoluta* . La sorprendente differenza è che mettendo i dati usando il metodo *relativo si* tiene traccia dell'indice in cui i dati sono inseriti, mentre il metodo assoluto richiede sempre un indice per `put` i dati.

Entrambi i metodi consentono chiamate *"concatenate"* . Dato un buffer sufficientemente grande, si può fare quanto segue:

```
buffer.putInt(0xCAFEBABE).putChar('c').putFloat(0.25).putLong(0xDEADBEEFCAFEBABE);
```

che è equivalente a:

```
buffer.putInt(0xCAFEBABE);
buffer.putChar('c');
buffer.putFloat(0.25);
buffer.putLong(0xDEADBEEFCAFEBABE);
```

Si noti che il metodo operativo su `byte` s non è denominato appositamente. Nota inoltre che è anche valido per passare sia un `ByteBuffer` sia un `byte[]` da `put` . Oltre a questo, tutti i tipi primitivi hanno metodi di `put` specializzati.

Una nota aggiuntiva: l'indice dato quando si usa la `put*` assoluta `put*` viene sempre conteggiato in `byte` s.

Utilizzo di base: utilizzo di DirectByteBuffer

`DirectByteBuffer` è un'implementazione speciale di `ByteBuffer` che non ha `byte[]` posto sotto.

Possiamo assegnare tale ByteBuffer chiamando:

```
ByteBuffer directBuffer = ByteBuffer.allocateDirect(16);
```

Questa operazione alloca 16 byte di memoria. Il contenuto dei buffer diretti *può* risiedere al di fuori del normale heap spazzato.

Possiamo verificare se ByteBuffer è diretto chiamando:

```
directBuffer.isDirect(); // true
```

Le caratteristiche principali di `DirectByteBuffer` è che JVM proverà a lavorare in modo nativo sulla memoria allocata senza alcun buffer aggiuntivo, in modo che le operazioni eseguite su di esso possano essere più veloci di quelle eseguite su ByteBuffer con array sottostanti.

Si consiglia di utilizzare `DirectByteBuffer` con pesanti operazioni IO che si basano sulla velocità di esecuzione, come la comunicazione in tempo reale.

Dobbiamo essere consapevoli che se proviamo ad usare il metodo `array()` otterremo `UnsupportedOperationException` . Quindi è una buona pratica per controllare se il nostro ByteBuffer lo ha (array di byte) prima di provare ad accedervi:

```
byte[] arrayOfBytes;
if(buffer.hasArray()) {
    arrayOfBytes = buffer.array();
}
```

Un altro uso del buffer di byte diretto è l'interoperabilità con JNI. Poiché un buffer di byte diretto non usa un `byte[]` , ma un effettivo blocco di memoria, è possibile accedere a quella memoria direttamente tramite un puntatore nel codice nativo. Ciò può far risparmiare un po 'di problemi e sovraccarico sul marshalling tra la rappresentazione Java e nativa dei dati.

L'interfaccia JNI definisce diverse funzioni per gestire i buffer di byte diretti: supporto NIO .

Leggi ByteBuffer online: https://riptutorial.com/it/java/topic/702/bytebuffer

Capitolo 21: Calendario e le sue sottoclassi

Osservazioni

A partire da Java 8, `Calendar` e le sue sottoclassi sono state sostituite dal pacchetto java.time e dai relativi pacchetti secondari . Dovrebbero essere preferiti, a meno che un'API legacy non richieda Calendar.

Examples

Creazione di oggetti del calendario

`Calendar` oggetti del `Calendar` possono essere creati usando `getInstance()` o usando il costruttore `GregorianCalendar` .

È importante notare che i mesi in `Calendar` sono a base zero, il che significa che JANUARY è rappresentato da un valore `int` 0. Per fornire un codice migliore, utilizzare sempre le costanti del `Calendar` , ad esempio `Calendar.JANUARY` per evitare equivoci.

```
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 : usa sempre le costanti del mese: la rappresentazione numerica è fuorviante , ad esempio `Calendar.JANUARY` ha il valore `0`

Aumentare / diminuire i campi del calendario

`add()` e `roll()` possono essere usati per aumentare / diminuire i campi del `Calendar` .

```
Calendar calendar = new GregorianCalendar(2016, Calendar.MARCH, 31); // 31 March 2016
```

Il metodo `add()` effetto su tutti i campi e si comporta in modo efficace se uno dovesse aggiungere o sottrarre date effettive dal calendario

```
calendar.add(Calendar.MONTH, -6);
```

L'operazione sopra riportata rimuove sei mesi dal calendario, riportandoci al 30 settembre 2015.

Per cambiare un particolare campo senza influenzare gli altri campi, usa `roll()` .

```
calendar.roll(Calendar.MONTH, -6);
```

L'operazione sopra riportata rimuove sei mesi dal *mese* corrente, quindi il mese viene identificato

come settembre. Nessun altro campo è stato modificato; l'anno non è cambiato con questa operazione.

Trovare AM / PM

Con la classe Calendar è facile trovare 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");
```

Sottrai i calendari

Per ottenere una differenza tra due `Calendar` , usa il metodo `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)
```

Leggi Calendario e le sue sottoclassi online: https://riptutorial.com/it/java/topic/165/calendario-e-le-sue-sottoclassi

Capitolo 22: Classe - Java Reflection

introduzione

La classe java.lang.Class fornisce molti metodi che possono essere utilizzati per ottenere metadati, esaminare e modificare il comportamento del tempo di esecuzione di una classe.

I pacchetti java.lang e java.lang.reflect forniscono classi per la riflessione java.

Dove è usato

L'API Reflection è utilizzata principalmente in:

IDE (Integrated Development Environment) ad es. Eclipse, MyEclipse, NetBeans ecc. Strumenti di test del debugger ecc.

Examples

metodo getClass () della classe 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);
    }
}
```

Leggi Classe - Java Reflection online: https://riptutorial.com/it/java/topic/10151/classe---java-reflection

Capitolo 23: Classe EnumSet

introduzione

La classe Java EnumSet è l'implementazione dell'insieme specializzato da utilizzare con i tipi enum. Eredita la classe AbstractSet e implementa l'interfaccia Set.

Examples

Enum Set Example

```
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());
  }
}
```

Leggi Classe EnumSet online: https://riptutorial.com/it/java/topic/10159/classe-enumset

Capitolo 24: Classe immutabile

introduzione

Gli oggetti immutabili sono istanze il cui stato non cambia dopo che è stato inizializzato. Ad esempio, String è una classe immutabile e una volta istanziata il suo valore non cambia mai.

Osservazioni

Alcune classi immutabili in Java:

1. java.lang.String
2. Le classi wrapper per i tipi primitivi: 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 maggior parte delle classi di enume è immutabile, ma in realtà dipende dal caso concreto.
4. java.math.BigInteger e java.math.BigDecimal (almeno oggetti di quelle stesse classi)
5. java.io.File. Si noti che questo rappresenta un oggetto esterno alla VM (un file sul sistema locale), che può o non può esistere e che ha alcuni metodi di modifica e interrogazione dello stato di questo oggetto esterno. Ma l'oggetto File stesso rimane immutabile.

Examples

Regole per definire classi immutabili

Le seguenti regole definiscono una strategia semplice per la creazione di oggetti immutabili.

1. Non fornire metodi "setter" - metodi che modificano campi o oggetti cui fanno riferimento i campi.
2. Rendi tutti i campi definitivi e privati.
3. Non consentire alle sottoclassi di sovrascrivere i metodi. Il modo più semplice per farlo è dichiarare la classe come definitiva. Un approccio più sofisticato è rendere privato il costruttore e costruire istanze in metodi di fabbrica.
4. Se i campi dell'istanza includono riferimenti a oggetti mutabili, non consentire la modifica di tali oggetti:
5. Non fornire metodi che modificano gli oggetti mutabili.
6. Non condividere riferimenti agli oggetti mutabili. Non memorizzare mai riferimenti a oggetti esterni mutabili trasmessi al costruttore; se necessario, creare copie e memorizzare i riferimenti alle copie. Allo stesso modo, crea copie dei tuoi oggetti interni mutabili quando necessario per evitare di restituire gli originali nei tuoi metodi.

Esempio senza riferimenti mutabili

```
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);
    }
}
```

Esempio con riferimenti mutabili

In questo caso la classe Point è mutabile e alcuni utenti possono modificare lo stato dell'oggetto di questa classe.

```
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;
    }
```

Qual è il vantaggio dell'immutabilità?

Il vantaggio dell'immutabilità viene fornito con la concorrenza. È difficile mantenere la correttezza in oggetti mutabili, poiché più thread potrebbero tentare di cambiare lo stato dello stesso oggetto, portando a qualche thread che vede uno stato diverso dello stesso oggetto, a seconda della sincronizzazione delle letture e delle scritture su detto oggetto.

Avendo un oggetto immutabile, è possibile garantire che tutti i thread che guardano l'oggetto vedranno lo stesso stato, poiché lo stato di un oggetto immutabile non cambierà.

Leggi Classe immutabile online: https://riptutorial.com/it/java/topic/10561/classe-immutabile

Capitolo 25: Classe interna locale

introduzione

Una classe cioè creata all'interno di un metodo è chiamata local inner class in java. Se vuoi invocare i metodi della classe interna locale, devi istanziare questa classe all'interno del metodo.

Examples

Classe interna locale

```
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();
 }
}
```

Leggi Classe interna locale online: https://riptutorial.com/it/java/topic/10160/classe-interna-locale

Capitolo 26: Classi e oggetti

introduzione

Gli oggetti hanno stati e comportamenti. Esempio: un cane ha degli stati - colore, nome, razza e comportamenti - scodinzolando, abbaiando, mangiando. Un oggetto è un'istanza di una classe.

Classe - Una classe può essere definita come un modello / modello che descrive il comportamento / stato che l'oggetto del suo tipo supporta.

Sintassi

- Esempio di classe {} // class parola chiave, nome, corpo

Examples

La più semplice possibile

```
class TrivialClass {}
```

Una classe comprende almeno la parola chiave `class` , un nome e un corpo, che potrebbe essere vuoto.

Si istanzia una classe con il `new` operatore.

```
TrivialClass tc = new TrivialClass();
```

Membro oggetto vs membro statico

Con questa classe:

```
class ObjectMemberVsStaticMember {

    static int staticCounter = 0;
    int memberCounter = 0;

    void increment() {
        staticCounter ++;
        memberCounter++;
    }
}
```

il seguente frammento di codice:

```
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 questo risultato:

```
o1 static counter 3
o1 member counter 1

o2 static counter 3
o2 member counter 2

ObjectMemberVsStaticMember.staticCounter = 3
```

Nota: non chiamare `static` membri `static` sugli oggetti, ma sulle classi. Mentre non fa la differenza per la JVM, i lettori umani lo apprezzeranno.

`static` membri `static` fanno parte della classe ed esistono solo una volta per classe. I membri non `static` esistono sulle istanze, esiste una copia indipendente per ogni istanza. Ciò significa anche che è necessario accedere a un oggetto di quella classe per accedere ai suoi membri.

Metodi di sovraccarico

A volte è necessario scrivere la stessa funzionalità per diversi tipi di input. A quel tempo, si può usare lo stesso nome di metodo con un diverso set di parametri. Ogni diverso insieme di parametri è noto come firma del metodo. Come visto nell'esempio, un singolo metodo può avere più firme.

```
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"
    }
}
```

Il vantaggio è che la stessa funzionalità viene chiamata con due diversi numeri di input. Durante il richiamo del metodo in base all'input che stiamo passando, (in questo caso un valore di stringa o due valori di stringa) viene eseguito il metodo corrispondente.

I metodi possono essere sovraccaricati:

1. In base al **numero di parametri** passati.

   Esempio: `method(String s)` e `method(String s1, String s2)` .

2. Basato **sull'ordine dei parametri** .

   Esempio: `method(int i, float f)` e `method(float f, int i))` .

***Nota: i** metodi non possono essere sovraccaricati modificando solo il tipo restituito (il `int method()` è considerato uguale al `String method()` e genera una `RuntimeException` se tentata). Se si modifica il tipo di reso, è necessario modificare anche i parametri per sovraccaricare.*

Costruzione e uso di oggetti di base

Gli oggetti arrivano nella loro classe, quindi un semplice esempio potrebbe essere una macchina (spiegazioni dettagliate di seguito):

```
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;
    }

}
```

Gli oggetti sono **esempi della** loro classe. Quindi, il modo in cui **creeresti un oggetto** sarebbe chiamando la classe Car in **due modi** nella tua classe principale (metodo principale in Java o onCreate in Android).

opzione 1

```
`Car newCar = new Car(30, 10, "Ferrari", "Red");
```

L'opzione 1 è quella in cui essenzialmente dici al programma tutto ciò che riguarda l'auto alla creazione dell'oggetto. La modifica di qualsiasi proprietà della macchina richiederebbe di chiamare

uno dei metodi come il metodo `repaintCar` . Esempio:

```
 newCar.repaintCar("Blue");
```

Nota: assicurati di passare il tipo di dati corretto al metodo. Nell'esempio precedente, è anche possibile passare una variabile al metodo `repaintCar` **a condizione che il tipo di dati sia corretto** .

Quello era un esempio di modifica delle proprietà di un oggetto, la ricezione di proprietà di un oggetto richiederebbe l'utilizzo di un metodo dalla classe Car che ha un valore di ritorno (ovvero un metodo che non è `void`). Esempio:

```
String myCarName = newCar.getName();  //returns string "Ferrari"
```

L'opzione 1 è l'opzione **migliore** quando si hanno **tutti i dati dell'oggetto** al momento della creazione.

opzione 2

```
`Car newCar = new Car();
```

L'opzione 2 ottiene lo stesso effetto ma richiede più lavoro per creare correttamente un oggetto. Voglio ricordare questo Costruttore nella classe Car:

```
public void Car(){
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }
```

Si noti che non è necessario passare effettivamente alcun parametro nell'oggetto per crearlo. Questo è molto utile quando non si hanno tutti gli aspetti dell'oggetto, ma è necessario utilizzare le parti che si hanno. Questo imposta i dati generici in ciascuna delle variabili di istanza dell'oggetto in modo che, se si richiama un dato che non esiste, non vengono generati errori.

Nota: non dimenticare che è necessario impostare le parti dell'oggetto in un secondo momento per il quale non è stato inizializzato. Per esempio,

```
Car myCar = new Car();
String color = Car.getColor(); //returns empty string
```

Questo è un errore comune tra gli oggetti che non sono inizializzati con tutti i loro dati. Gli errori sono stati evitati perché c'è un Costruttore che consente di creare un oggetto Car vuoto con **variabili stand-in** (`public Car(){}`), ma nessuna parte della myCar è stata effettivamente personalizzata. **Esempio corretto di creazione di oggetto auto:**

```
Car myCar = new Car();
myCar.nameCar("Ferrari");
```

```
myCar.paintCar("Purple");
myCar.setGallonsInTank(10);
myCar.setMPG(30);
```

E, come promemoria, ottieni le proprietà di un oggetto chiamando un metodo nella tua classe principale. Esempio:

```
String myCarName = myCar.getName(); //returns string "Ferrari"
```

Costruttori

I costruttori sono metodi speciali che prendono il nome dalla classe e senza un tipo di ritorno e sono usati per costruire oggetti. I costruttori, come i metodi, possono prendere i parametri di input. I costruttori sono utilizzati per inizializzare gli oggetti. Le classi astratte possono avere anche costruttori.

```
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
```

È importante capire che i costruttori sono diversi dai metodi in diversi modi:

1. I costruttori possono solo rendere i modificatori `public` , `private` e `protected` e non possono essere dichiarati `abstract` , `final` , `static` o `synchronized` .
2. I costruttori non hanno un tipo di ritorno.
3. I costruttori DEVONO essere nominati come il nome della classe. Nell'esempio `Hello` , il nome del costruttore dell'oggetto `Hello` è uguale al nome della classe.
4. `this` parola chiave ha un utilizzo aggiuntivo all'interno dei costruttori. `this.method(...)` chiama un metodo `this.method(...)` corrente, mentre `this(...)` riferisce a un altro costruttore nella classe corrente con diverse firme.

I costruttori possono anche essere richiamati attraverso l'ereditarietà usando la parola chiave `super` .

```
public class SuperManClass{

    public SuperManClass(){
        // some implementation
    }

    // ... methods
```

```
}


public class BatmanClass extends SupermanClass{
    public BatmanClass(){
        super();
    }
    //... methods...
}
```

Vedere le specifiche della lingua Java # 8.8 e # 15.9

Inizializzazione di campi finali statici mediante un inizializzatore statico

Per inizializzare `static final` campi `static final` che richiedono l'utilizzo di più di una singola espressione, è possibile utilizzare un inizializzatore `static` per assegnare il valore. L'esempio seguente inizializza un insieme non modificabile di `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);
    }
}
```

Spiegare qual è il metodo di sovraccarico e di sovrascrittura.

Il metodo Overriding e Overloading sono due forme di polimorfismo supportate da Java.

Sovraccarico del metodo

L'overloading del metodo (noto anche come Polymorphism statico) è un modo in cui è possibile avere due (o più) metodi (funzioni) con lo stesso nome in una singola classe. Sì, è così semplice.

```
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;
    }
```

```
}
```

In questo modo l'utente può chiamare lo stesso metodo per l'area a seconda del tipo di forma che ha.

Ma la vera domanda ora è, in che modo il compilatore java distinguerà quale metodo deve essere eseguito?

Bene, Java ha chiarito che anche se i **nomi dei metodi** (`area()` nel nostro caso) **possono essere uguali, ma il metodo degli argomenti sta prendendo dovrebbe essere diverso.**

> I metodi sovraccaricati devono avere elenchi di argomenti diversi (quantità e tipi).

Detto questo non possiamo aggiungere un altro metodo per calcolare l'area di un quadrato come questo: `public Double area(Long side)` perché in questo caso, entrerà in conflitto con il metodo area del cerchio e causerà **ambiguità** per il compilatore java.

Grazie a dio, ci sono alcuni rilassamenti durante la scrittura di metodi sovraccarichi come

> Può avere diversi tipi di rendimento.
>
> Può avere diversi modificatori di accesso.
>
> Può lanciare diverse eccezioni.

Perché questo è chiamato polimorfismo statico?

Beh, questo è il motivo per cui i metodi di overload che devono essere invocati vengono decisi al momento della compilazione, in base al numero effettivo di argomenti e ai tipi di argomenti in fase di compilazione.

> Uno dei motivi comuni dell'utilizzo dell'overload dei metodi è la semplicità del codice fornito. Ad esempio, ricorda `String.valueOf()` che accetta quasi ogni tipo di argomento? Quello che è scritto dietro la scena è probabilmente qualcosa del genere: -

```
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)
```

Metodo Overriding

Bene, il metodo che sovrascrive (sì, lo indovini, è anche conosciuto come polimorfismo dinamico) è un argomento un po 'più interessante e complesso.

Nel metodo che sovrascrive, sovrascriviamo il corpo del metodo fornito dalla classe genitore.

Fatto? No? Facciamo un esempio.

```
public abstract class Shape{

    public abstract Double area(){
        return 0.0;
    }
}
```

Quindi abbiamo una classe chiamata Shape e ha un metodo chiamato area che probabilmente restituirà l'area della forma.

Diciamo che ora abbiamo due classi chiamate Circle e Rectangle.

```
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;
    }
}
```

Allo stesso modo, classe rettangolo:

```
 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;
    }
}
```

Quindi, ora entrambe le classi dei bambini hanno un corpo del metodo aggiornato fornito dalla classe padre (`Shape`). Ora la domanda è come vedere il risultato? Bene, lascia fare al vecchio modo di `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());

    }
}
```

Wow! non è fantastico? Due oggetti dello stesso tipo chiamano gli stessi metodi e restituiscono valori diversi. Amico mio, questo è il potere del polimorfismo dinamico.

Ecco una tabella per confrontare meglio le differenze tra questi due: -

Sovraccarico del metodo	Metodo Overriding
Il sovraccarico del metodo viene utilizzato per aumentare la leggibilità del programma.	L'override del metodo viene utilizzato per fornire l'implementazione specifica del metodo che è già fornito dalla sua super classe.
L'overloading del metodo viene eseguito all'interno della classe.	L'override del metodo si verifica in due classi che hanno una relazione IS-A (ereditarietà).
In caso di sovraccarico del metodo, il parametro deve essere diverso.	In caso di override del metodo, i parametri devono essere uguali.
L'overloading del metodo è l'esempio del polimorfismo del tempo di compilazione.	L'override del metodo è l'esempio del polimorfismo del tempo di esecuzione.
In Java, l'overloading dei metodi non può essere eseguito cambiando solo il tipo di ritorno del metodo. Il tipo di reso può essere uguale o diverso nell'overload del metodo. Ma è necessario modificare il parametro.	Il tipo di restituzione deve essere uguale o covariante nel metodo prioritario.

Leggi Classi e oggetti online: https://riptutorial.com/it/java/topic/114/classi-e-oggetti

Capitolo 27: Classi nidificate e interiori

introduzione

Usando Java, gli sviluppatori hanno la possibilità di definire una classe all'interno di un'altra classe. Tale classe è chiamata classe annidata . Le classi nidificate sono chiamate classi interne se sono state dichiarate non statiche, in caso contrario vengono semplicemente denominate classi nidificate statiche. Questa pagina è per documentare e fornire dettagli con esempi su come utilizzare le classi nidificate e interne di Java.

Sintassi

- public class OuterClass {public class InnerClass {}} // Le classi interne possono anche essere private
- public class OuterClass {public static class StaticNestedClass {}} // Le classi nidificate statiche possono anche essere private
- public void method () {classe privata LocalClass {}} // Le classi locali sono sempre private
- SomeClass anonymousClassInstance = new SomeClass () {}; // Le classi interne anonime non possono essere nominate, quindi l'accesso è discutibile. Se 'SomeClass ()' è astratto, il corpo deve implementare tutti i metodi astratti.
- SomeInterface anonymousClassInstance = new SomeInterface () {}; // Il corpo deve implementare tutti i metodi di interfaccia.

Osservazioni

Terminologia e classificazione

La JLS (Java Language Specification) classifica i diversi tipi di classe Java come segue:

> Una *classe di primo livello* è una classe che non è una classe nidificata.
>
> Una *classe annidata* è una classe la cui dichiarazione si verifica all'interno del corpo di un'altra classe o interfaccia.
>
> Una *classe interna* è una classe nidificata che non è dichiarata in modo esplicito o implicito statico.
>
> Una classe interna può essere una *classe membro non statica* , una *classe locale* o una *classe anonima* . Una classe membro di un'interfaccia è implicitamente statica, quindi non è mai considerata una classe interiore.

In pratica i programmatori si riferiscono a una classe di livello superiore che contiene una classe interna come "classe esterna". Inoltre, c'è una tendenza ad usare "classe nidificata" per riferirsi solo a classi nidificate statiche (esplicitamente o implicitamente).

Si noti che esiste una stretta relazione tra le classi interne anonime e le lambda, ma le lambda sono classi.

Differenze semantiche

- Le classi di livello superiore sono il "caso base". Sono visibili ad altre parti di un programma soggette a normali regole di visibilità basate sulla semantica del modificatore di accesso. Se non astratti, possono essere istanziati da qualsiasi codice che mostra dove i costruttori rilevanti sono visibili in base ai modificatori di accesso.

- Le classi nidificate statiche seguono le stesse regole di accesso e istanziazione delle classi di primo livello, con due eccezioni:

    - Una classe nidificata può essere dichiarata come `private` , il che la rende inaccessibile al di fuori della classe di livello superiore che la racchiude.
    - Una classe nidificata ha accesso ai membri `private` della classe di livello superiore che la include e di tutta la sua classe testata.

    Ciò rende le classi nidificate statiche utili quando è necessario rappresentare più "tipi di entità" all'interno di uno stretto limite di astrazione; ad esempio quando le classi nidificate vengono utilizzate per nascondere "dettagli di implementazione".

- Le classi interne aggiungono la possibilità di accedere a variabili non statiche dichiarate negli ambiti che racchiudono:

    - Una classe membro non statica può fare riferimento a variabili di istanza.
    - Una classe locale (dichiarata all'interno di un metodo) può anche fare riferimento alle variabili locali del metodo, a condizione che siano `final` . (Per Java 8 e versioni successive, possono essere *effettivamente definitive* .)
    - Una classe interna anonima può essere dichiarata all'interno di una classe o di un metodo e può accedere alle variabili in base alle stesse regole.

    Il fatto che un'istanza di classe interna possa fare riferimento a variabili in un'istanza di classe che include ha implicazioni per l'istanziazione. In particolare, deve essere fornita un'istanza di inclusione, implicitamente o esplicitamente, quando viene creata un'istanza di una classe interna.

Examples

Una pila semplice che utilizza una classe annidata

```
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;
    }
}
```

E il suo uso, che (in particolare) non riconosce affatto l'esistenza della classe annidata.

```
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() + ", ");
        }
    }
}
```

Classi nidificate statiche e non statiche

Quando si crea una classe nidificata, si affronta una scelta di avere quella statica della classe annidata:

```
public class OuterClass1 {

    private static class StaticNestedClass {

    }

}
```

O non statico:

```
public class OuterClass2 {

    private class NestedClass {
```

```
    }

}
```

Al suo interno, le classi nidificate statiche *non hanno **un'istanza** circostante* della classe esterna, mentre le classi nidificate non statiche lo fanno. Ciò influenza sia quando / quando è consentito creare un'istanza di una classe nidificata, sia quali istanze di quelle classi nidificate sono autorizzate ad accedere. Aggiungendo all'esempio precedente:

```
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
    }
}
```

Pertanto, la decisione di statico o non statico dipende principalmente dal fatto che sia necessario o meno poter accedere direttamente ai campi e ai metodi della classe esterna, sebbene abbia anche conseguenze su quando e dove è possibile costruire la classe nidificata.

Come regola generale, rendere statiche le classi nidificate, a meno che non sia necessario accedere ai campi e ai metodi della classe esterna. Come rendere privati i propri campi a meno che non siano necessari per la pubblica, ciò riduce la visibilità disponibile per la classe nidificata (non consentendo l'accesso a un'istanza esterna), riducendo la probabilità di errore.

Modificatori di accesso per le classi interne

Una spiegazione completa dei Modificatori di accesso in Java può essere trovata qui . Ma come interagiscono con le classi interne?

`public` , come al solito, dà accesso illimitato a qualsiasi ambito in grado di accedere 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
    }
}
```

entrambi `protected` e il modificatore di default (del nulla) si comportano come previsto, come fanno per le classi non annidate.

`private` , abbastanza interessante, non limita la classe a cui appartiene. Piuttosto, limita l'unità di compilazione - il file .java. Ciò significa che le classi Outer hanno pieno accesso ai campi e ai metodi della classe Inner, anche se sono contrassegnati come `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 stessa classe interna può avere una visibilità diversa dal `public` . Contrassegnandolo come `private` o con un altro modificatore di accesso limitato, altre classi (esterne) non potranno importare e assegnare il tipo. Possono comunque ottenere riferimenti a oggetti di quel tipo, comunque.

```
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
    }
}
```

Classi interiori anonime

Una classe interiore anonima è una forma di classe interiore che viene dichiarata e istanziata con una singola affermazione. Di conseguenza, non esiste un nome per la classe che possa essere usato altrove nel programma; cioè è anonimo.

Le classi anonime vengono in genere utilizzate in situazioni in cui è necessario essere in grado di creare una classe leggera da passare come parametro. Questo è in genere fatto con un'interfaccia. Per esempio:

```
public static Comparator<String> CASE_INSENSITIVE =
        new Comparator<String>() {
            @Override
            public int compare(String string1, String string2) {
                return string1.toUpperCase().compareTo(string2.toUpperCase());
            }
        };
```

Questa classe anonima definisce un oggetto `Comparator<String>` (`CASE_INSENSITIVE`) che confronta due stringhe ignorando le differenze nel caso.

Altre interfacce che vengono spesso implementate e istanziate utilizzando classi anonime sono `Runnable` e `Callable` . Per esempio:

```
// 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"
```

Anche le classi interne anonime possono essere basate su classi. In questo caso, la classe anonima `extends` implicitamente la classe esistente. Se la classe estesa è astratta, la classe anonima deve implementare tutti i metodi astratti. Può anche sovrascrivere i metodi non astratti.

Costruttori

Una classe anonima non può avere un costruttore esplicito. Invece, viene definito un costruttore implicito che utilizza `super(...)` per passare qualsiasi parametro a un costruttore nella classe che viene estesa. Per esempio:

```
SomeClass anon = new SomeClass(1, "happiness") {
            @Override
            public int someMethod(int arg) {
                // do something
            }
        };
```

Il costruttore implicito per la nostra sottoclasse anonima di `SomeClass` chiamerà un costruttore di `SomeClass` che corrisponde alla firma di chiamata `SomeClass(int, String)` . Se nessun costruttore è disponibile, si otterrà un errore di compilazione. Eventuali eccezioni lanciate dal costruttore abbinato vengono anche lanciate dal costruttore implicito.

Naturalmente, questo non funziona quando si estende un'interfaccia. Quando si crea una classe anonima da un'interfaccia, la superclasse delle classi è `java.lang.Object` che ha solo un costruttore no-args.

Metodo Class Inner Classes

Una classe scritta all'interno di un metodo chiamato **metodo inner class locale** . In tal caso l'ambito della classe interna è limitato all'interno del metodo.

Una classe interna del metodo locale può essere istanziata solo all'interno del metodo in cui è definita la classe interna.

L'esempio dell'uso del metodo inner class locale:

```
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();
    }
}
```

L'esecuzione darà un risultato: `Method local inner class 1` .

Accedere alla classe esterna da una classe interiore non statica

Il riferimento alla classe esterna usa il nome della classe e `this`

```
public class OuterClass {
    public class InnerClass {
        public void method() {
            System.out.println("I can access my enclosing class: " + OuterClass.this);
        }
    }
}
```

È possibile accedere direttamente ai campi e ai metodi della classe esterna.

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        public void method() {
            System.out.println("I can access " + counter);
        }
    }
}
```

Ma in caso di collisione di nomi è possibile utilizzare il riferimento di classe esterno.

```
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;
        }
    }
}
```

Crea un'istanza di classe interna non statica dall'esterno

Una classe interna che è visibile a qualsiasi classe esterna può essere creata anche da questa classe.

La classe interiore dipende dalla classe esterna e richiede un riferimento a un'istanza di essa. Per creare un'istanza della classe interna, il `new` operatore deve solo essere richiamato su un'istanza della classe esterna.

```
class OuterClass {

    class InnerClass {
    }
}

class OutsideClass {

    OuterClass outer = new OuterClass();

    OuterClass.InnerClass createInner() {
        return outer.new InnerClass();
    }
}
```

Si noti l'utilizzo come `outer.new` .

Leggi Classi nidificate e interiori online: https://riptutorial.com/it/java/topic/3317/classi-nidificate-e-interiori

Capitolo 28: classloader

Osservazioni

Un classloader è una classe il cui scopo principale è mediare la posizione e il caricamento delle classi utilizzate da un'applicazione. Un programma di caricamento classe può anche trovare e caricare *risorse* .

Le classi classloader standard possono caricare classi e risorse dalle directory nel file system e dai file JAR e ZIP. Possono anche scaricare e memorizzare in cache i file JAR e ZIP da un server remoto.

I programmi di caricamento di classe sono normalmente concatenati, in modo che la JVM proverà a caricare le classi dalle librerie di classi standard preferibilmente alle origini fornite dall'applicazione. I classloader personalizzati consentono al programmatore di modificare questo. Inoltre può fare cose come decodificare i file bytecode e la modifica bytecode.

Examples

Istanziare e usare un classloader

Questo esempio di base mostra come un'applicazione può creare un'istanza di un classloader e utilizzarla per caricare dinamicamente una classe.

```
URL[] urls = new URL[] {new URL("file:/home/me/extras.jar")};
Classloader loader = new URLClassLoader(urls);
Class<?> myObjectClass = loader.findClass("com.example.MyObject");
```

Il programma di caricamento classi creato in questo esempio avrà come predefinito il classloader predefinito e tenterà prima di cercare qualsiasi classe nel classloader padre prima di cercare in "extra.jar". Se la classe richiesta è già stata caricata, la chiamata `findClass` restituirà il riferimento alla classe caricata in precedenza.

La chiamata `findClass` può fallire in molti modi. I più comuni sono:

- Se la classe indicata non può essere trovata, la chiamata con lancio `ClassNotFoundException` .
- Se la classe denominata dipende da un'altra classe che non può essere trovata, la chiamata genererà `NoClassDefFoundError` .

Implementazione di un classLoader personalizzato

Ogni caricatore personalizzato deve estendere direttamente o indirettamente la classe `java.lang.ClassLoader` . I *punti di estensione* principali sono i seguenti metodi:

- `findClass(String)` - sovraccaricare questo metodo se il classloader segue il modello di delega standard per il caricamento della classe.

- `loadClass(String, boolean)` - sovraccaricare questo metodo per implementare un modello di delega alternativo.
- `findResource` e `findResources` : sovraccaricare questi metodi per personalizzare il caricamento delle risorse.

I `defineClass` metodi che sono responsabili per caricare effettivamente la classe da una matrice di byte sono `final` per evitare sovraccarichi. Qualsiasi comportamento personalizzato deve essere eseguito prima di chiamare `defineClass` .

Ecco un semplice che carica una classe specifica da un array di byte:

```
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);
        }
    }
}
```

Poiché abbiamo solo sovrascritto il metodo `findClass` , questo caricatore di classi personalizzato si comporterà come segue quando viene chiamato `loadClass` .

1. Il metodo `loadClass` del classloader chiama `findLoadedClass` per vedere se una classe con questo nome è già stata caricata da questo classloader. Se ciò riesce, l'oggetto `Class` risultante viene restituito al richiedente.
2. Il metodo `loadClass` quindi delega al classloader genitore chiamando la sua chiamata `loadClass` . Se il genitore può gestire la richiesta, restituirà un oggetto di `Class` che viene quindi restituito al richiedente.
3. Se il classloader genitore non può caricare la classe, `findClass` chiama quindi il nostro metodo `findClass` override, passando il nome della classe da caricare.
4. Se il nome richiesto corrisponde a `this.classname` , chiamiamo `defineClass` per caricare la classe effettiva `this.classfile` byte `this.classfile` . L'oggetto `Class` risultante viene quindi restituito.
5. Se il nome non coincide, gettiamo `ClassNotFoundException` .

Caricamento di un file .class esterno

Per caricare una classe, dobbiamo prima definirla. La classe è definita da `ClassLoader` . C'è solo un problema, Oracle non ha scritto il codice di `ClassLoader` con questa funzione disponibile. Per definire la classe avremo bisogno di accedere a un metodo denominato `defineClass()` che è un

metodo privato di `ClassLoader` .

Per accedervi, ciò che faremo è creare una nuova classe, `ByteClassLoader` , ed estenderla a `ClassLoader` . Ora che abbiamo esteso la nostra classe a `ClassLoader` , possiamo accedere ai metodi privati di `ClassLoader` . Per rendere disponibile `defineClass()` , creeremo un nuovo metodo che funzionerà come un mirror per il metodo private `defineClass()` . Per chiamare il metodo privato avremo bisogno il nome della classe, `name` , i byte di classe, `classBytes` , offset del primo byte, che sarà `0` perché `classBytes` dati 'inizia alle `classBytes[0]` , e compensare dell'ultimo byte, che sarà `classBytes.lenght` perché rappresenta la dimensione dei dati, che sarà l'ultimo offset.

```
public class ByteClassLoader extends ClassLoader {

    public Class<?> defineClass(String name, byte[] classBytes) {
        return defineClass(name, classBytes, 0, classBytes.length);
    }

}
```

Ora abbiamo un metodo pubblico `defineClass()` . Può essere chiamato passando il nome della classe e i byte della classe come argomenti.

Diciamo che abbiamo classe denominata `MyClass` nel pacchetto `stackoverflow` ...

Per chiamare il metodo abbiamo bisogno dei byte di classe, quindi creiamo un oggetto `Path` che rappresenta il percorso della nostra classe utilizzando il metodo `Paths.get()` e passando il percorso della classe binary come argomento. Ora, possiamo ottenere i byte di classe con `Files.readAllBytes(path)` . Quindi creiamo un'istanza `ByteClassLoader` e usiamo il metodo che abbiamo creato, `defineClass()` . Abbiamo già i byte di classe ma per chiamare il nostro metodo abbiamo anche bisogno del nome della classe che è dato dal nome del pacchetto (punto) il nome canonico della classe, in questo caso `stackoverflow.MyClass` .

```
Path path = Paths.get("MyClass.class");

ByteClassLoader loader = new ByteClassLoader();
loader.defineClass("stackoverflow.MyClass", Files.readAllBytes(path);
```

Nota : il metodo `defineClass()` restituisce un oggetto di `Class<?>` . Puoi salvarlo se vuoi.

Per caricare la classe, chiamiamo `loadClass()` e passiamo il nome della classe. Questo metodo può generare `ClassNotFoundException` quindi è necessario utilizzare un blocco cath try

```
try{
    loader.loadClass("stackoverflow.MyClass");
} catch(ClassNotFoundException e){
    e.printStackTrace();
}
```

Leggi classloader online: https://riptutorial.com/it/java/topic/5443/classloader

Capitolo 29: Clonazione dell'oggetto

Osservazioni

La clonazione può essere complicata, specialmente quando i campi dell'oggetto contengono altri oggetti. Vi sono situazioni in cui si desidera eseguire una copia profonda , invece di copiare solo i valori del campo (cioè i riferimenti agli altri oggetti).

La linea di fondo è clone è rotto , e si dovrebbe pensare due volte prima di implementare l'interfaccia `Cloneable` e sovrascrivere il metodo `clone` . Il metodo `clone` è dichiarato nella classe `Object` e non nell'interfaccia `Cloneable` , quindi `Cloneable` non funziona come interfaccia perché manca un metodo `clone` pubblico. Il risultato è che il contratto per l'uso del `clone` è sottilmente documentato e debolmente applicato. Ad esempio, una classe che sovrascrive il `clone` volte si basa su tutte le sue classi genitore che sovrascrivono anche il `clone` . Non sono forzati a farlo, e se non lo fanno, il tuo codice può generare eccezioni.

Una soluzione molto migliore per fornire funzionalità di clonazione è quella di fornire un *costruttore di copia* o una *fabbrica di copie* . Fai riferimento a Joshua Bloch's Effective Java Item 11: Override clone in modo giudizioso.

Examples

Clonazione usando un costruttore di copie

Un modo semplice per clonare un oggetto è implementare un costruttore di copie.

```
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
```

Clonazione implementando l'interfaccia Clonabile

Clonazione di un oggetto implementando l'interfaccia 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
```

Clonazione eseguendo una copia superficiale

Il comportamento predefinito durante la clonazione di un oggetto consiste nell'eseguire una copia superficiale dei campi dell'oggetto. In tal caso, sia l'oggetto originale che l'oggetto clonato, contengono riferimenti agli stessi oggetti.

Questo esempio mostra questo comportamento.

```
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));
}
```

Clonazione eseguendo una copia profonda

Per copiare oggetti nidificati, è necessario eseguire una copia profonda , come mostrato in questo esempio.

```
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));
}
```

Clonazione utilizzando una fotocopiatrice

```
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)
    }

}
```

Leggi Clonazione dell'oggetto online: https://riptutorial.com/it/java/topic/2830/clonazione-dell-oggetto

Capitolo 30: Code e Deques

Examples

L'uso di PriorityQueue

`PriorityQueue` è una struttura dati. Come `SortedSet` , `PriorityQueue` ordina anche i suoi elementi in base alle loro priorità. Gli elementi, che hanno una priorità più alta, vengono prima di tutto. Il tipo di `PriorityQueue` dovrebbe implementare un'interfaccia `comparable` o di `comparator` , i cui metodi decidono le priorità degli elementi della struttura dati.

```
//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 come coda FIFO

La classe `java.util.LinkedList` , mentre si implementa `java.util.List` è un'implementazione generica dell'interfaccia `java.util.Queue` che opera anche su un principio FIFO (First In, First Out) .

Nell'esempio seguente, con il metodo `offer()` , gli elementi vengono inseriti in `LinkedList` . Questa operazione di inserimento si chiama `enqueue` . Nel ciclo `while` seguito, gli elementi vengono rimossi dalla `Queue` basata su FIFO. Questa operazione si chiama `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() );
```

```
}
```

L'output di questo codice è

```
first element
second element
third element
fourth element
fifth element
```

Come visto nell'output, il primo elemento inserito "first element" viene rimosso per primo, "second element" viene rimosso in secondo luogo, ecc.

Stacks

Cos'è una pila?

In Java, gli stack sono una struttura di dati LIFO (Last In, First Out) per oggetti.

Stack API

Java contiene un'API dello stack con i seguenti metodi

```
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
```

Esempio

```
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");
      }
   }
}
```

Questo ritorna:

```
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 è un'interfaccia, che è una coda che si blocca quando si tenta di disconnettersi da esso e la coda è vuota, o se si tenta di accodare gli elementi a esso e la coda è già piena. Un thread che tenta di disconnettere da una coda vuota viene bloccato fino a quando un altro thread inserisce un elemento nella coda. Un thread che tenta di accodare un elemento in una coda completa viene bloccato fino a quando un altro thread non fa spazio nella coda, eliminando uno o più elementi o eliminando completamente la coda.

I metodi BlockingQueue sono disponibili in quattro forme, con diversi modi di gestire le operazioni che non possono essere soddisfatti immediatamente, ma potrebbero essere soddisfatti in futuro: uno lancia un'eccezione, il secondo restituisce un valore speciale (nullo o falso, a seconda della operazione), il terzo blocca il thread corrente per un tempo indefinito fino a quando l'operazione può avere successo, e il quarto blocco solo per un dato limite di tempo massimo prima di rinunciare.

operazione	Genera l'eccezione	Valore speciale	blocchi	Il tempo è scaduto
Inserire	Inserisci()	offerta (e)	mettere (e)	offerta (e, tempo, unità)
Rimuovere	rimuovere()	sondaggio()	prendere()	sondaggio (tempo, unità)
Esaminare	elemento()	sbirciare()	N / A	N / A

Un `BlockingQueue` può essere **limitato** o **illimitato** . Un BlockingQueue limitato è uno che viene inizializzato con la capacità iniziale.

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);
```

Qualsiasi chiamata a un metodo put () verrà bloccata se la dimensione della coda è uguale alla capacità iniziale definita.

Una coda illimitata è una che è inizializzata senza capacità, in realtà per impostazione predefinita è inizializzata con Integer.MAX_VALUE.

Alcune implementazioni comuni di `BlockingQueue` sono:

1. ArrayBlockingQueue
2. LinkedBlockingQueue
3. PriorityBlockingQueue

Ora diamo un'occhiata a un esempio di `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");
```

Questo stamperà:

```
Entry one done
Entry two done
```

E il thread verrà bloccato dopo la seconda uscita.

Interfaccia della coda

Nozioni di base

Una `Queue` è una raccolta per contenere elementi prima dell'elaborazione. Le code tipicamente, ma non necessariamente, ordinano gli elementi in un modo FIFO (first-in-first-out).

Capo della coda è l'elemento che verrebbe rimosso da una chiamata per rimuovere o eseguire il polling. In una coda FIFO, tutti i nuovi elementi sono inseriti nella coda della coda.

L'interfaccia della coda

```
public interface Queue<E> extends Collection<E> {
    boolean add(E e);

    boolean offer(E e);

    E remove();

    E poll();

    E element();

    E peek();
}
```

Ogni metodo di `Queue` esiste in due forme:

- si genera un'eccezione se l'operazione fallisce;
- altro restituisce un valore speciale se l'operazione fallisce (`null` o `false` seconda dell'operazione.

Tipo di operazione	Genera eccezione	Restituisce un valore speciale
Inserire	`add(e)`	`offer(e)`
Rimuovere	`remove()`	`poll()`
Esaminare	`element()`	`peek()`

deque

Un `Deque` è una "coda doppia", il che significa che è possibile aggiungere elementi nella parte anteriore o coda della coda. Solo la coda può aggiungere elementi alla coda di una coda.

Il `Deque` eredita l'interfaccia `Queue` , il che significa che i metodi regolari rimangono, tuttavia l'interfaccia di Deque offre metodi aggiuntivi per essere più flessibili con una coda. I metodi aggiuntivi parlano davvero da soli se si sa come funziona una coda, poiché questi metodi sono pensati per aggiungere più flessibilità:

Metodo	Breve descrizione
`getFirst()`	Ottiene il primo elemento del **capo** della coda senza rimuoverlo.
`getLast()`	Ottiene il primo elemento della **coda** della coda senza rimuoverlo.

Metodo	Breve descrizione
`addFirst(E e)`	Aggiunge un articolo alla **testa** della coda
`addLast(E e)`	Aggiunge un articolo alla **coda** della coda
`removeFirst()`	Rimuove il primo elemento in **testa** alla coda
`removeLast()`	Rimuove il primo elemento alla **coda** della coda

Ovviamente sono disponibili le stesse opzioni per `offer` , `poll` e `peek` , tuttavia non funzionano con eccezioni ma piuttosto con valori speciali. Non ha senso mostrare ciò che fanno qui.

Aggiunta e accesso agli elementi

Per aggiungere elementi alla coda di un Deque si chiama il suo metodo `add()` . Puoi anche usare i `addFirst()` e `addLast()` , che aggiungono elementi alla testa e alla coda 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
```

Puoi dare un'occhiata all'elemento in testa alla coda senza togliere l'elemento dalla coda. Questo viene fatto tramite il metodo `element()` . È anche possibile utilizzare i metodi `getLast()` `getFirst()` e `getLast()` , che restituiscono il primo e l'ultimo elemento nel `Deque` . Ecco come appare:

```
String firstElement0 = dequeA.element();
String firstElement1 = dequeA.getFirst();
String lastElement = dequeA.getLast();
```

Rimozione di elementi

Per rimuovere elementi da un deque, si chiamano i metodi `remove()` , `removeFirst()` e `removeLast()` . Ecco alcuni esempi:

```
String firstElement = dequeA.remove();
String firstElement = dequeA.removeFirst();
String lastElement  = dequeA.removeLast();
```

Leggi Code e Deques online: https://riptutorial.com/it/java/topic/7196/code-e-deques

Capitolo 31: Codifica dei caratteri

Examples

Leggere il testo da un file codificato in 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);
    }
}
```

Scrittura di un testo in un file in 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");
        }
    }
}
```

Ottenere la rappresentazione in byte di una stringa in 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));
    }
}
```

Leggi Codifica dei caratteri online: https://riptutorial.com/it/java/topic/2735/codifica-dei-caratteri

Capitolo 32: collezioni

introduzione

Il framework delle collezioni in `java.util` fornisce un numero di classi generiche per insiemi di dati con funzionalità che non possono essere fornite da array regolari.

Il framework Collections contiene interfacce per `Collection<O>` , con le sottointerfacce principali `List<O>` e `Set<O>` e la mappatura della collezione `Map<K,V>` . Le raccolte sono l'interfaccia di root e vengono implementate da molti altri framework di raccolta.

Osservazioni

Le raccolte sono oggetti che possono contenere raccolte di altri oggetti all'interno di esse. È possibile specificare il tipo di dati memorizzati in una raccolta utilizzando Generics .

Le raccolte generalmente utilizzano gli spazi dei nomi `java.util` o `java.util.concurrent` .

Java SE 1.4

Java 1.4.2 e versioni successive non supportano i generici. Pertanto, non è possibile specificare i parametri di tipo contenuti in una raccolta. Oltre a non avere la sicurezza del tipo, è necessario utilizzare anche i cast per ottenere il tipo corretto da una raccolta.

Oltre alla `Collection<E>` , esistono diversi tipi principali di raccolte, alcune delle quali hanno sottotipi.

- `List<E>` è una raccolta ordinata di oggetti. È simile a un array, ma non definisce un limite di dimensioni. Le implementazioni di solito aumentano di dimensioni internamente per accogliere nuovi elementi.
- `Set<E>` è una raccolta di oggetti che non consente duplicati.
    - `SortedSet<E>` è un `Set<E>` che specifica anche l'ordinamento degli elementi.
- `Map<K,V>` è una raccolta di coppie chiave / valore.
    - `SortedMap<K,V>` è una `Map<K,V>` che specifica anche l'ordinamento degli elementi.

Java SE 5

Java 5 aggiunge un nuovo tipo di raccolta:

- `Queue<E>` è una raccolta di elementi che devono essere elaborati in un ordine specifico. L'implementazione specifica se si tratta di FIFO o LIFO. Questo obsoleta la classe dello `Stack` .

Java SE 6

Java 6 aggiunge alcuni nuovi sottotipi di raccolte.

- `NavigableSet<E>` è un `Set<E>` con metodi di navigazione speciali integrati.
- `NavigableMap<K,V>` è una `Map<K,V>` con metodi di navigazione speciali integrati.
- `Deque<E>` è una `Queue<E>` che può essere letta da entrambe le estremità.

Si noti che gli elementi di cui sopra sono tutte le interfacce. Per poterli utilizzare, è necessario trovare le classi di implementazione appropriate, come `ArrayList` , `HashSet` , `HashMap` o `PriorityQueue` .

Ogni tipo di raccolta ha più implementazioni che hanno parametri di prestazioni e casi d'uso diversi.

Si noti che il Principio di sostituzione di Liskov si applica ai sottotipi di raccolta. Cioè, un `SortedSet<E>` può essere passato ad una funzione che si aspetta un `Set<E>` . È inoltre utile leggere i Parametri limitati nella sezione Generici per ulteriori informazioni su come utilizzare le raccolte con ereditarietà delle classi.

Se si desidera creare le proprie raccolte, potrebbe essere più semplice ereditare una delle classi astratte (come `AbstractList`) anziché implementare l'interfaccia.

Java SE 1.2

Prima della versione 1.2, dovevi utilizzare le seguenti classi / interfacce:

- `Vector` invece di `ArrayList`
- `Dictionary` invece di `Map` . Nota che Dictionary è anche una classe astratta piuttosto che un'interfaccia.
- `Hashtable` invece di `HashMap`

Queste classi sono obsolete e non dovrebbero essere utilizzate nel codice moderno.

Examples

Dichiarazione di una lista di array e aggiunta di oggetti

Possiamo creare un `ArrayList` (seguendo l'interfaccia `List`):

```
List aListOfFruits = new ArrayList();
```

Java SE 5

```
List<String> aListOfFruits = new ArrayList<String>();
```

Java SE 7

```
List<String> aListOfFruits = new ArrayList<>();
```

Ora, usa il metodo `add` per aggiungere una `String` :

```
aListOfFruits.add("Melon");
```

```
aListOfFruits.add("Strawberry");
```

Nell'esempio precedente, `ArrayList` conterrà la `String` "Melon" nell'indice 0 e la `String` "Strawberry" nell'indice 1.

Inoltre possiamo aggiungere più elementi con il `addAll(Collection<? extends E> c)`

```
List<String> aListOfFruitsAndVeggies = new ArrayList<String>();
aListOfFruitsAndVeggies.add("Onion");
aListOfFruitsAndVeggies.addAll(aListOfFruits);
```

Ora "Onion" è posto a 0 indice in `aListOfFruitsAndVeggies` , "Melon" è nell'indice 1 e "Strawberry" è nell'indice 2.

Costruire collezioni da dati esistenti

Collezioni standard

Quadro di collezioni Java

Un modo semplice per costruire un `List` dai singoli valori di dati è utilizzare il metodo `java.utils.Arrays` `Arrays.asList` :

```
List<String> data = Arrays.asList("ab", "bc", "cd", "ab", "bc", "cd");
```

Tutte le implementazioni di raccolta standard forniscono costruttori che accettano un'altra raccolta come argomento aggiungendo tutti gli elementi alla nuova raccolta al momento della costruzione:

```
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
```

Framework Google Guava Collections

Un altro grande quadro è `Google Guava` che è straordinaria classe di utilità (fornisce metodi statici di convenienza) per la costruzione di diversi tipi di collezioni standard di `Lists` e `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");
```

Raccolta di mappe

Quadro di collezioni Java

Allo stesso modo per le mappe, data una `Map<String, Object> map` una nuova mappa può essere costruita con tutti gli elementi come segue:

```
Map<String, Object> map1 = new HashMap<>(map);
SortedMap<String, Object> map2 = new TreeMap<>(map);
```

Quadro delle collezioni Apache Commons

Usando `Apache Commons` puoi creare una mappa usando una matrice in `ArrayUtils.toMap` e `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"}});
```

Ogni elemento dell'array deve essere un Map.Entry o una Array, contenente almeno due elementi, in cui il primo elemento viene utilizzato come chiave e il secondo come valore.

Framework Google Guava Collections

La classe di utilità del framework `Google Guava` si chiama `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

Utilizzando `Stream` ,

```
Stream.of("xyz", "abc").collect(Collectors.toList());
```

o

```
Arrays.stream("xyz", "abc").collect(Collectors.toList());
```

Iscriviti alle liste

I seguenti modi possono essere utilizzati per unire gli elenchi senza modificare la / e lista / i di origine.

Primo approccio Ha più linee ma è facile da capire

```
List<String> newList = new ArrayList<String>();
newList.addAll(listOne);
newList.addAll(listTwo);
```

Secondo approccio Ha una linea in meno ma meno leggibile.

```
List<String> newList = new ArrayList<String>(listOne);
newList.addAll(listTwo);
```

Terzo approccio. Richiede libreria di raccolta di risorse di Apache di terze parti.

```
ListUtils.union(listOne,listTwo);
```

Java SE 8

Usando i flussi si può ottenere lo stesso da

```
List<String> newList = Stream.concat(listOne.stream(),
listTwo.stream()).collect(Collectors.toList());
```

Riferimenti. Elenco delle interfacce

Rimozione di elementi da un elenco all'interno di un ciclo

È difficile rimuovere gli elementi da un elenco mentre si è all'interno di un ciclo, ciò è dovuto al fatto che l'indice e la lunghezza dell'elenco vengono modificati.

Dato il seguente elenco, ecco alcuni esempi che daranno un risultato inaspettato e alcuni che daranno il risultato corretto.

```
List<String> fruits = new ArrayList<String>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Strawberry");
```

NON CORRETTO

La rimozione di iterazione `for` dichiarazione *Salta "Banana":*

L'esempio di codice stampa solo `Apple` e `Strawberry` . `Banana` viene saltato perché si muove all'indice `0` , una volta `Apple` viene eliminato, ma allo stesso tempo `i` viene incrementato di `1` .

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

Rimozione nella maggiore `for` dichiarazione *genera un'eccezione:*

A causa dell'iterazione della raccolta e della modifica allo stesso tempo.

> Lanci: java.util.ConcurrentModificationException

```
for (String fruit : fruits) {
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruits.remove(fruit);
    }
}
```

CORRETTA

Rimozione durante il ciclo 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();
    }
}
```

L'interfaccia `Iterator` ha un metodo `remove()` costruito solo per questo caso. Tuttavia, questo metodo è contrassegnato come "facoltativo" nella documentazione e potrebbe generare un `UnsupportedOperationException` .

> Genera: UnsupportedOperationException - se l'operazione di rimozione non è supportata da questo iteratore

Pertanto, è consigliabile controllare la documentazione per assicurarsi che questa operazione sia

supportata (in pratica, a meno che la raccolta non sia immutabile ottenuta attraverso una libreria di terze parti o l'uso di uno dei metodi `Collections.unmodifiable...()` , l'operazione è quasi sempre supportata).

Durante l'utilizzo di `Iterator` viene generata `modCount ConcurrentModificationException` quando viene modificato il `modCount List` da quando è stato creato `Iterator` . Questo potrebbe essere accaduto nella stessa discussione o in un'applicazione multithreading che condividesse la stessa lista.

Un `modCount` è una variabile `int` che conta il numero di volte in cui questo elenco è stato modificato strutturalmente. Una modifica strutturale significa essenzialmente un'operazione `add()` o `remove()` invocata sull'oggetto `Collection` (le modifiche apportate da `Iterator` non vengono conteggiate). Quando viene creato l' `Iterator` , memorizza questo `modCount` e ad ogni iterazione `List` controlla se il `modCount` corrente è uguale a quando è stato creato l' `Iterator` . Se c'è una modifica nel valore `modCount` , lancia una `ConcurrentModificationException` .

Quindi per la lista sopra dichiarata, un'operazione come sotto non genererà alcuna eccezione:

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.set(0, "Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());
}
```

Ma aggiungendo un nuovo elemento alla `List` dopo aver inizializzato un `Iterator` verrà `Iterator` una `ConcurrentModificationException` :

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.add("Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());    //ConcurrentModificationException here
}
```

Iterazione all'indietro

```
for (int i = (fruits.size() - 1); i >=0; i--) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

Questo non salta nulla. Lo svantaggio di questo approccio è che l'output è invertito. Tuttavia, nella maggior parte dei casi, rimuovi gli articoli che non contano. Non dovresti mai farlo con `LinkedList` .

Iterazione in avanti, regolazione dell'indice del loop

```
for (int i = 0; i < fruits.size(); i++) {
```

```
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
         i--;
    }
}
```

Questo non salta nulla. Quando l' `i` -esimo elemento viene rimosso dalla `List` , l'elemento originariamente posizionato in corrispondenza dell'indice `i+1` diventa il nuovo `i` -esimo elemento. Pertanto, il ciclo può decrementare `i` in modo che l'iterazione successiva elabori l'elemento successivo, senza saltare.

Utilizzando una lista "dovrebbe-essere-rimosso"

```
ArrayList shouldBeRemoved = new ArrayList();
for (String str : currentArrayList) {
    if (condition) {
        shouldBeRemoved.add(str);
    }
}
currentArrayList.removeAll(shouldBeRemoved);
```

Questa soluzione consente allo sviluppatore di verificare se gli elementi corretti vengono rimossi in modo più pulito.

Java SE 8

In Java 8 sono possibili le seguenti alternative. Questi sono più puliti e più diretti se la rimozione non deve avvenire in un ciclo.

Filtrare un flusso

Un `List` può essere trasmesso in streaming e filtrato. Un filtro appropriato può essere utilizzato per rimuovere tutti gli elementi indesiderati.

```
List<String> filteredList =
    fruits.stream().filter(p -> !"Apple".equals(p)).collect(Collectors.toList());
```

Si noti che, a differenza di tutti gli altri esempi qui, questo esempio produce una nuova istanza di `List` e mantiene inalterato l' `List` originale.

Utilizzando `removeIf`

Salva il sovraccarico di costruire un flusso se tutto ciò che è necessario è rimuovere un insieme di elementi.

```
fruits.removeIf(p -> "Apple".equals(p));
```

Collezione non modificabile

A volte non è una buona pratica esporre una raccolta interna in quanto può portare a una vulnerabilità di codice dannoso a causa della sua caratteristica mutevole. Per fornire raccolte "di sola lettura", java fornisce le sue versioni non modificabili.

Una collezione non modificabile è spesso una copia di una collezione modificabile che garantisce che la collezione stessa non può essere alterata. I tentativi di modificarlo generano un'eccezione UnsupportedOperationException.

È importante notare che gli oggetti presenti all'interno della collezione possono ancora essere modificati.

```
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);
    }
}
```

Il seguente tentativo di modificare una collezione non modificabile genererà un'eccezione:

```
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);
    }
}
```

produzione:

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
    at App.main(App.java:12)
```

Iterating over Collections

Iterare sulla lista

```
List<String> names  = new ArrayList<>(Arrays.asList("Clementine", "Duran", "Mike"));
```

Java SE 8

```
names.forEach(System.out::println);
```

Se abbiamo bisogno di usare il parallelismo

```
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());
}
```

Iterating over 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());
}
```

Iterazione sulla mappa

```
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());
}
```

Collezioni immutabili vuote

A volte è opportuno utilizzare una collezione vuota immutabile. La classe `Collections` fornisce metodi per ottenere tali raccolte in modo efficiente:

```
List<String> anEmptyList = Collections.emptyList();
Map<Integer, Date> anEmptyMap = Collections.emptyMap();
Set<Number> anEmptySet = Collections.emptySet();
```

Questi metodi sono generici e convertiranno automaticamente la raccolta restituita nel tipo a cui è assegnata. Cioè, una `emptyList()` ad esempio `emptyList()` può essere assegnata a qualsiasi tipo di `List` e allo stesso modo per `emptySet()` e `emptyMap()` .

Le raccolte restituite da questi metodi sono immutabili in quanto generano `UnsupportedOperationException` se si tenta di chiamare metodi che potrebbero cambiare il loro contenuto (`add` , `put` , ecc.). Queste raccolte sono principalmente utili come sostituti per i risultati del metodo vuoto o altri valori predefiniti, invece di utilizzare `null` o creare oggetti con `new` .

Collezioni e valori primitivi

Le raccolte in Java funzionano solo per gli oggetti. Cioè non c'è `Map<int, int>` in Java. Invece, i valori primitivi devono essere *racchiusi* negli oggetti, come in `Map<Integer, Integer>` . L'auto-boxing Java abiliterà l'uso trasparente di queste raccolte:

```
Map<Integer, Integer> map = new HashMap<>();
map.put(1, 17); // Automatic boxing of int to Integer objects
int a = map.get(1); // Automatic unboxing.
```

Sfortunatamente, il sovraccarico di questo è *sostanziale* . Una `HashMap<Integer, Integer>` richiederà circa 72 byte per voce (es. Su JVM a 64 bit con puntatori compressi e assumendo numeri interi maggiori di 256 e assumendo il 50% di carico della mappa). Poiché i dati effettivi sono solo 8 byte, questo genera un sovraccarico enorme. Inoltre, richiede due livelli di riferimento indiretto (Mappa -> Voce -> Valore) è inutilmente lento.

Esistono diverse librerie con raccolte ottimizzate per tipi di dati primitivi (che richiedono solo ~ 16 byte per voce con il 50% di carico, ovvero 4x meno memoria, e un livello di riferimento indiretto inferiore), che possono produrre notevoli vantaggi in termini di prestazioni quando si utilizzano grandi collezioni di primitive valori in Java.

Rimozione degli elementi corrispondenti dagli elenchi utilizzando Iterator.

Sopra ho notato un esempio per rimuovere elementi da una lista all'interno di un loop e ho pensato ad un altro esempio che potrebbe tornare utile questa volta usando l'interfaccia di `Iterator` .
Questa è una dimostrazione di un trucco che potrebbe rivelarsi utile quando si ha a che fare con articoli duplicati negli elenchi di cui si vuole sbarazzarsi.

Nota: questa operazione si aggiunge solo alla **rimozione di elementi da un elenco all'interno di un** esempio di **ciclo** :

Quindi definiamo i nostri elenchi come al solito

```
    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));
```

Il seguente metodo prende in due oggetti Collection ed esegue la magia di rimuovere gli elementi nel nostro `removeNameList` che corrispondono agli elementi in `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
    }
}
```

Chiamando il metodo e passando il `nameList` e il `removeNameList` come segue
`removeNames(nameList,removeNameList);`
Produrrà il seguente risultato:

> Elenco di matrici prima di rimuovere i nomi: **James Smith Sonny Huckle Berry Finn Allan**
> Elenco di matrici dopo aver rimosso i nomi: **James Smith Finn Allan**

Un uso semplice e intuitivo per le raccolte che può rivelarsi utile per rimuovere elementi ripetitivi all'interno degli elenchi.

Creazione della propria struttura Iterable da utilizzare con Iterator o for-each loop.

Per garantire che la nostra raccolta possa essere iterata utilizzando iteratore o ciclo per-one, dobbiamo occuparci dei seguenti passaggi:

1. Le cose che vogliamo iterare devono essere `Iterable` ed esporre `iterator()` .
2. Progettare un `java.util.Iterator` `hasNext()` override di `hasNext()` , `next()` e `remove()` .

Ho aggiunto una semplice implementazione di liste collegate generiche sotto che utilizza le entità sopra per rendere iterabile l'elenco collegato.

```
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 + " ");
        }
    }
}
```

Produzione

```
using Iterator:
1 2 4 3
using for-each:
1 2 4 3
```

Questo verrà eseguito in Java 7+. Puoi farlo girare su Java 5 e Java 6 anche sostituendo:

```
LinkedList<Integer> list = new LinkedList<>(1);
```

con

```
LinkedList<Integer> list = new LinkedList<Integer>(1);
```

o semplicemente qualsiasi altra versione incorporando le modifiche compatibili.

Trappola: eccezioni di modifica simultanea

Questa eccezione si verifica quando una raccolta viene modificata mentre viene iterata su metodi diversi da quelli forniti dall'oggetto iteratore. Ad esempio, abbiamo un elenco di cappelli e vogliamo rimuovere tutti quelli che hanno i padiglioni auricolari:

```
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);
    }
```

```
}
```

Se eseguiamo questo codice, ***ConcurrentModificationException*** verrà generato poiché il codice modifica la raccolta durante l'iterazione. La stessa eccezione può verificarsi se uno dei thread multipli che lavorano con lo stesso elenco sta provando a modificare la raccolta mentre altri lo iterano su di esso. La modifica simultanea di raccolte in più thread è una cosa naturale, ma dovrebbe essere trattata con gli strumenti usuali della toolbox di programmazione simultanea come i blocchi di sincronizzazione, le raccolte speciali adottate per la modifica simultanea, la modifica della raccolta clonata dall'iniziale ecc.

Collezioni secondarie

List subList (int fromIndex, int toIndex)

Qui fromIndex è inclusivo e toIndex è esclusivo.

```
List list = new ArrayList();
List list1 = list.subList(fromIndex,toIndex);
```

1. Se l'elenco non esiste nell'intervallo give, genera IndexOutofBoundException.
2. Qualsiasi modifica apportata alla lista1 avrà conseguenze sulle stesse modifiche nella lista. Si tratta di raccolte supportate.
3. Se fromnIndex è maggiore di toIndex (fromIndex> toIndex) genera IllegalArgumentException.

Esempio:

```
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);
```

Produzione:
Prima della sottolista [Hello1, Hello2]
Dopo le modifiche di sottolista [Hello1, Hello3, Hello2]

Imposta sottoset (daIndex, aIndex)

Qui fromIndex è inclusivo e toIndex è esclusivo.

```
Set set = new TreeSet();
Set set1 = set.subSet(fromIndex,toIndex);
```

Il set restituito genererà un IllegalArgumentException nel tentativo di inserire un elemento al di fuori del suo intervallo.

Mappa sottocappa (daKey, toKey)

fromKey è inclusivo e toKey è esclusivo

```
Map map = new TreeMap();
Map map1 = map.get(fromKey,toKey);
```

Se fromKey è maggiore di toKey o se questa mappa ha un intervallo limitato e fromKey o toKey si trova al di fuori dei limiti dell'intervallo, genera IllegalArgumentException.

Tutte le raccolte supportano le raccolte di spalle, le modifiche apportate alla sotto collezione avranno lo stesso cambiamento nella raccolta principale.

Leggi collezioni online: https://riptutorial.com/it/java/topic/90/collezioni

Capitolo 33: Collezioni alternative

Osservazioni

Questo argomento sulle raccolte Java da guava, apache, eclipse: Multiset, Bag, Multimap, utilizza la funzione da questa libreria e così via.

Examples

Apache HashBag, Guava HashMultiset ed Eclipse HashBag

Un sacchetto / ultiset memorizza ogni oggetto nella collezione insieme a un conteggio di occorrenze. Metodi aggiuntivi sull'interfaccia consentono di aggiungere o rimuovere più copie contemporaneamente di un oggetto. L'analogo JDK è HashMap <T, Integer>, quando i valori sono il conteggio delle copie di questa chiave.

genere	Guaiava	Collezioni Apache Commons	Collezioni GS	JDK
Ordine non definito	HashMultiset	HashBag	HashBag	HashM
smistato	TreeMultiset	TreeBag	TreeBag	TreeMa
Inserimento-ordine	LinkedHashMultiset	-	-	LinkedH
Variante concomitante	ConcurrentHashMultiset	SynchronizedBag	SynchronizedBag	`Collect`
Concorrente e ordinato	-	SynchronizedSortedBag	SynchronizedSortedBag	`Collect`
Collezione immutabile	ImmutableMultiset	UnmodifiableBag	UnmodifiableBag	`Collect`
Immutabile e ordinato	ImmutableSortedMultiset	UnmodifiableSortedBag	UnmodifiableSortedBag	`Collect`)

Esempi :

1. Utilizzo di SynchronizedSortedBag da 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. Utilizzo di TreeBag da 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. Utilizzo di LinkedHashMultiset da 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
```

Altri esempi:

I. Collezione Apache:

1. HashBag : ordine non definito
2. SynchronizedBag - concurrent e ordine non definito
3. SynchronizedSortedBag - - ordine concorrente e ordinato
4. TreeBag - ordine ordinato

II. Collezione GS / Eclipse

5. MutableBag - ordine non definito
6. MutableSortedBag - ordine ordinato

III. Guaiava

7. HashMultiset - ordine non definito
8. TreeMultiset - ordine ordinato
9. LinkedHashMultiset - ordine di inserzione
10. ConcurrentHashMultiset - concurrent e ordine non definito

Multimap nelle raccolte Guava, Apache ed Eclipse

Questo multimap consente coppie di valori-chiave duplicati. Gli analoghi JDK sono HashMap <K, Elenco>, HashMap <K, Set> e così via.

L'ordine di chiave	L'ordine del valore	Duplicare	Chiave analogica	Valore analogico	Guaiava	A
non definito	Inserimento-ordine	sì	HashMap	Lista di array	ArrayListMultimap	Mu
non definito	non definito	no	HashMap	HashSet	HashMultimap	Mu mu Ha Ha
non definito	smistato	no	HashMap	TreeSet	Multimaps. newMultimap(HashMap, Supplier <TreeSet>)	Mu ne Tr
Inserimento-ordine	Inserimento-ordine	sì	LinkedHashMap	Lista di array	LinkedListMultimap	M m Li ()
Inserimento-ordine	Inserimento-ordine	no	LinkedHashMap	LinkedHashSet	LinkedHashMultimap	Mu mu Li Li
smistato	smistato	no	TreeMap	TreeSet	TreeMultimap	Mu mu Tr Se

Esempi che utilizzano Multimap

Attività : analisi "Hello World! Hello All! Hi World!" stringa per separare le parole e stampare tutti gli indici di ogni parola usando MultiMap (ad esempio, Hello = [0, 2], World! = [1, 5] e così via)

1. MultiValueMap di 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 dalla collezione GS / Eclipse

```
    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 di Guava

```
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
```

Esempi esemplificativi:

I. Collezione Apache:

1. MultiValueMap
2. MultiValueMapLinked
3. MultiValueMapTree

II. Collezione GS / Eclipse

1. FastListMultimap
2. HashBagMultimap
3. TreeSortedSetMultimap
4. UnifiedSetMultimap

III. Guaiava

1. HashMultiMap
2. LinkedHashMultimap
3. LinkedListMultimap
4. TreeMultimap
5. ArrayListMultimap

Confronta le operazioni con le raccolte - Crea raccolte

Confronta le operazioni con le raccolte - Crea raccolte

1. Crea lista

Descrizione	JDK	guaiava	gs-collezioni
Crea una	`new ArrayList<> ()`	`Lists.newArrayList()`	`FastList.newList()`

Descrizione	JDK	guaiava	gs-collezioni
lista vuota			
Crea una lista da valori	`Arrays.asList("1", "2", "3")`	`Lists.newArrayList("1", "2", "3")`	`FastList.newListWith("1", "2", "3")`
Crea lista con capacità = 100	`new ArrayList<>(100)`	`Lists.newArrayListWithCapacity(100)`	`FastList.newList(100)`
Crea una lista da qualsiasi raccolta	`new ArrayList<>(collection)`	`Lists.newArrayList(collection)`	`FastList.newList(collecti`
Crea una lista da qualsiasi Iterable	-	`Lists.newArrayList(iterable)`	`FastList.newList(iterable`
Crea una lista da Iterator	-	`Lists.newArrayList(iterator)`	-
Crea una lista dalla matrice	`Arrays.asList(array)`	`Lists.newArrayList(array)`	`FastList.newListWith(arra`
Crea una lista usando la fabbrica	-	-	`FastList.newWithNValues(1 () -> "1")`

Esempi:

```
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 Crea Set

Descrizione	JDK	guaiava	gs-collezioni
Crea set vuoto	`new HashSet<>()`	`Sets.newHashSet()`	`UnifiedSet.newSet()`
Creatre impostato dai valori	`new HashSet<>(Arrays.asList("alpha", "beta", "gamma"))`)	`Sets.newHashSet("alpha", "beta", "gamma")`	`UnifiedSet.newSetWith("a "beta", "gamma")`

Descrizione	JDK	guaiava	gs-collezioni
Crea set da qualsiasi collezione	`new HashSet<>(collection)`	`Sets.newHashSet(collection)`	`UnifiedSet.newSet(collec`
Crea set da qualsiasi Iterable	-	`Sets.newHashSet(iterable)`	`UnifiedSet.newSet(iterab`
Crea set da qualsiasi Iterator	-	`Sets.newHashSet(iterator)`	-
Crea set dalla matrice	`new HashSet<>(Arrays.asList(array))`	`Sets.newHashSet(array)`	`UnifiedSet.newSetWith(ar`

Esempi:

```
    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 Crea mappa

Descrizione	JDK	guaiava	gs-collezioni
Crea una mappa vuota	`new HashMap<>()`	`Maps.newHashMap()`	`UnifiedMap.newMap()`
Crea una mappa con capacità = 130	`new HashMap<>(130)`	`Maps.newHashMapWithExpectedSize(100)`	`UnifiedMap.newMap(130)`
Crea una mappa da un'altra mappa	`new HashMap<>(map)`	`Maps.newHashMap(map)`	`UnifiedMap.newMap(map)`
Crea una mappa con le chiavi	-	-	`UnifiedMap.newWithKeysValues("1", "a", "2", "b")`

Esempi:

```
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");
```

Altri esempi: CreateCollectionTest

1. CollectionCompare
2. CollectionSearch
3. JavaTransform

Leggi Collezioni alternative online: https://riptutorial.com/it/java/topic/2958/collezioni-alternative

Capitolo 34: Collezioni simultanee

introduzione

Una *raccolta simultanea* è una [raccolta] [1] che consente l'accesso da più di un thread contemporaneamente. Filetti diversi possono tipicamente scorrere i contenuti della raccolta e aggiungere o rimuovere elementi. La raccolta è responsabile per garantire che la raccolta non venga corrotta. [1]: http://stackoverflow.com/documentation/java/90/collections#t=20161222193649729848 4

Examples

Collezioni thread-safe

Per impostazione predefinita, i vari tipi di raccolta non sono thread-safe.

Tuttavia, è abbastanza facile creare una raccolta sicura per i thread.

```
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>());
```

Quando crei una raccolta thread-safe, non devi mai accedervi tramite la raccolta originale, solo attraverso il wrapper thread-safe.

Java SE 5

A partire da Java 5, `java.util.collections` ha diverse nuove raccolte thread-safe che non richiedono i vari metodi `Collections.synchronized` .

```
List<String> threadSafeList = new CopyOnWriteArrayList<String>();
Set<String> threadSafeSet = new ConcurrentHashSet<String>();
Map<String, String> threadSafeMap = new ConcurrentHashMap<String, String>();
```

Collezioni simultanee

Le raccolte concorrenti sono una generalizzazione di raccolte thread-safe che consentono un utilizzo più ampio in un ambiente concorrente.

Sebbene le raccolte thread-safe abbiano un'aggiunta o rimozione sicura degli elementi da più thread, non necessariamente hanno un'iterazione sicura nello stesso contesto (potrebbe non essere possibile iterare in modo sicuro attraverso la raccolta in un thread, mentre un altro lo modifica aggiungendo / rimozione di elementi).

È qui che vengono utilizzate le raccolte simultanee.

Poiché l'iterazione è spesso l'implementazione di base di diversi metodi di massa nelle raccolte, come `addAll` , `removeAll` o anche la raccolta di copie (tramite un costruttore o altri mezzi), l'ordinamento, ... il caso d'uso per le raccolte concorrenti è in realtà piuttosto ampio.

Ad esempio, Java SE 5 `java.util.concurrent.CopyOnWriteArrayList` è un'implementazione di `Lis` t thread-safe e simultanea, i suoi stati javadoc :

> Il metodo di iterazione di stile "snapshot" utilizza un riferimento allo stato dell'array nel punto in cui è stato creato l'iteratore. Questo array non cambia mai durante il ciclo di vita dell'iteratore, quindi l'interferenza è impossibile e l'iteratore è garantito non lanciare ConcurrentModificationException.

Pertanto, il seguente codice è sicuro:

```
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();
        }
    }
}
}
```

Un'altra raccolta concomitante relativa all'iterazione è `ConcurrentLinkedQueue` , che afferma:

> Gli iteratori sono debolmente coerenti, elementi di ritorno che riflettono lo stato della coda a un certo punto o dalla creazione dell'iteratore. Non gettano java.util.ConcurrentModificationException e possono procedere in concomitanza con altre operazioni. Gli elementi contenuti nella coda dalla creazione dell'iteratore verranno restituiti esattamente una volta.

Si dovrebbe controllare il javadocs per vedere se una raccolta è concomitante, oppure no. Gli attributi dell'iteratore restituiti dal metodo `iterator()` ("fail fast", "weakly consistent", ...) è l'attributo più importante da cercare.

Filetto di esempi sicuri ma non simultanei

Nel codice sopra, cambiando la dichiarazione `LIST` su

```
public static final List<Integer> LIST = Collections.synchronizedList(new ArrayList<>());
```

Potrebbe (e statisticamente sulla maggior parte delle architetture CPU / core moderne) portare a delle eccezioni.

Le raccolte sincronizzate dai metodi di utilità `Collections` sono thread-safe per l'aggiunta / rimozione di elementi, ma non l'iterazione (a meno che la raccolta sottostante non sia già passata ad essa).

Inserimento in ConcurrentHashMap

```
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.
       }
    }
```

```
}
```

Leggi Collezioni simultanee online: https://riptutorial.com/it/java/topic/8363/collezioni-simultanee

Capitolo 35: Comandi di runtime

Examples

Aggiungere ganci di arresto

A volte hai bisogno di un pezzo di codice da eseguire quando il programma si ferma, ad esempio rilasciando le risorse di sistema che apri. È possibile eseguire un thread quando il programma si arresta con il metodo `addShutdownHook` :

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    ImportantStuff.someImportantIOStream.close();
}));
```

Leggi Comandi di runtime online: https://riptutorial.com/it/java/topic/7304/comandi-di-runtime

Capitolo 36: Comparabile e comparatore

Sintassi

- La classe pubblica MyClass implementa `<MyClass` > comparabile
- MyComparator di classe pubblica implementa il comparatore `<SomeOtherClass` >
- public int compareTo (MyClass altro)
- int pubblico (SomeOtherClass o1, SomeOtherClass o2)

Osservazioni

Quando si implementa un `compareTo(..)` che dipende da un `double` , **non** effettuare le seguenti operazioni:

```
public int comareTo(MyClass other) {
    return (int)(doubleField - other.doubleField); //THIS IS BAD
}
```

Il troncamento causato dal cast `(int)` farà sì che il metodo restituisca a volte in modo errato `0` anziché un numero positivo o negativo e possa quindi portare a confronti e errori di ordinamento.

Invece, l'implementazione corretta più semplice è utilizzare Double.compare , in quanto tale:

```
public int comareTo(MyClass other) {
    return Double.compare(doubleField,other.doubleField); //THIS IS GOOD
}
```

Una versione non generica di `Comparable<T>` , semplicemente `Comparable` , esiste da Java 1.2 . A parte l'interfaccia con il codice legacy, è sempre meglio implementare la versione generica `Comparable<T>` , poiché non richiede il casting al confronto.

È molto normale che una classe sia paragonabile a se stessa, come in:

```
public class A implements Comparable<A>
```

Mentre è possibile uscire da questo paradigma, sii cauto quando lo fai.

Un `Comparator<T>` può ancora essere utilizzato su istanze di una classe se tale classe implementa `Comparable<T>` . In questo caso, verrà utilizzata la logica del `Comparator` ; l'ordine naturale specificato dall'implementazione `Comparable` sarà ignorato.

Examples

Ordinamento di una lista usando Paragonabile o un comparatore

Diciamo che stiamo lavorando su una classe che rappresenta una persona con il loro nome e cognome. Abbiamo creato una classe base per farlo e implementato i metodi `equals` e `hashCode` appropriati.

```
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);
    }
}
```

Ora vorremmo ordinare una lista di oggetti `Person` per nome, come nel seguente scenario:

```
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.
}
```

Sfortunatamente, come indicato, quanto sopra non verrà compilato. `Collections.sort(..)` sa solo come ordinare un elenco se gli elementi in quell'elenco sono comparabili o se viene fornito un metodo di confronto personalizzato.

Se ti è stato chiesto di ordinare il seguente elenco: `1,3,5,4,2` , non avresti problemi a dire che la risposta è `1,2,3,4,5` . Questo perché gli Integer (sia in Java che matematicamente) hanno un *ordinamento naturale* , un ordinamento di base di confronto standard di default. Per dare alla nostra classe Person un ordinamento naturale, implementiamo `Comparable<Person>` , che richiede l'implementazione del metodo `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);
        }
    }
}
```

Ora, il metodo principale fornito funzionerà correttamente

```
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
}
```

Se, tuttavia, non si desidera o non è possibile modificare la classe `Person` , è possibile fornire un `Comparator<T>` personalizzato `Comparator<T>` che gestisce il confronto di due oggetti `Person` . Se ti è stato chiesto di ordinare il seguente elenco: `circle, square, rectangle, triangle, hexagon` non puoi, ma se ti venisse chiesto di ordinare quell'elenco in *base al numero di angoli* , potresti farlo. Proprio così, fornire un comparatore indica a Java come confrontare due oggetti normalmente non confrontabili.

```
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
}
```

I comparatori possono anche essere creati / usati come una classe interiore anonima

```
//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

Comparatori basati sull'espressione Lambda

A partire da Java 8, i comparatori possono anche essere espressi come espressioni lambda

```
    //Lambda
    Collections.sort(people, (p1, p2) -> { //Legal
        //Method code....
    });
```

Metodi predefiniti del comparatore

Inoltre, esistono interessanti metodi predefiniti nell'interfaccia di Comparatore per la costruzione di comparatori: il seguente costruisce un comparatore confrontandolo con `lastName` e quindi `firstName` .

```
Collections.sort(people, Comparator.comparing(Person::getLastName)
                                .thenComparing(Person::getFirstName));
```

Inversione dell'ordine di un comparatore

Qualsiasi comparatore può anche essere facilmente invertito usando il `reversedMethod` che cambierà l'ordine crescente in discendente.

Il confronto e confrontare i metodi

L'interfaccia `Comparable<T>` richiede un metodo:

```
public interface Comparable<T> {

    public int compareTo(T other);

}
```

E l'interfaccia `Comparator<T>` richiede un metodo:

```
public interface Comparator<T> {

    public int compare(T t1, T t2);

}
```

Questi due metodi fanno essenzialmente la stessa cosa, con una differenza minore: `compareTo` confronta `this` con gli `other` , mentre `compare` confronta `t1` con `t2` , senza preoccuparsi affatto di `this` .

A parte questa differenza, i due metodi hanno requisiti simili. Specificamente (per compareTo), confronta questo oggetto con l'oggetto specificato per l'ordine. Restituisce un numero intero negativo, zero o un numero intero positivo poiché questo oggetto è minore, uguale o maggiore dell'oggetto specificato. Così, per il confronto di `a` e `b` :

- Se `a < b` , `a.compareTo(b)` e `compare(a,b)` dovrebbe restituire un intero negativo, e `b.compareTo(a)` e `compare(b,a)` dovrebbe restituire un intero positivo
- Se `a > b` , `a.compareTo(b)` e `compare(a,b)` devono restituire un intero positivo e `b.compareTo(a)` e `compare(b,a)` devono restituire un numero intero negativo
- Se `a` uguale `b` per il confronto, tutti i confronti dovrebbero restituire `0` .

Ordinamento naturale (comparabile) vs esplicito (comparatore)

Esistono due metodi `Collections.sort()` :

- Uno che accetta un `List<T>` come parametro in cui `T` deve implementare Comparable e sovrascrive il metodo `compareTo()` che determina l'ordinamento.
- Uno che accetta un elenco e un comparatore come argomenti, in cui il comparatore determina l'ordinamento.

Innanzitutto, ecco una classe Person che 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();
    }

}
```

Ecco come `compareTo()` la classe precedente per ordinare una lista nell'ordinamento naturale dei suoi elementi, definito dal metodo di confronto `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);
```

Ecco come utilizzare un comparatore in linea anonimo per ordinare un elenco che non implementa Comparable o, in questo caso, per ordinare un elenco in un ordine diverso dall'ordinamento naturale:

```
            //-- 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);
```

Ordinamento delle voci della mappa

A partire da Java 8, ci sono metodi predefiniti sull'interfaccia `Map.Entry` per consentire l'ordinamento delle iterazioni della mappa.

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
```

Ovviamente, questi possono anche essere usati al di fuori della stream stream:

Java SE 8

```
List<Map.Entry<String, Integer>> entries = new ArrayList<>(numberOfEmployees.entrySet());
Collections.sort(entries, Map.Entry.comparingByValue());
```

Creazione di un comparatore utilizzando il metodo di confronto

```
Comparator.comparing(Person::getName)
```

Questo crea un comparatore per la classe `Person` che usa questo nome di persona come origine di confronto. Inoltre è possibile utilizzare la versione del metodo per confrontare long, int e double. Per esempio:

```
Comparator.comparingInt(Person::getAge)
```

Ordine inverso

Per creare un comparatore che impone l'ordine inverso usa il metodo `reversed()` :

```
Comparator.comparing(Person::getName).reversed()
```

Catena di comparatori

```
Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstName)
```

Questo creerà un comparatore che raffigura il cognome con il cognome e poi lo confronta con il nome. Puoi concatenare tutti i comparatori che vuoi.

Leggi Comparabile e comparatore online: https://riptutorial.com/it/java/topic/3137/comparabile-e-comparatore

Capitolo 37: Compilatore Java - 'javac'

Osservazioni

Il comando `javac` viene utilizzato per compilare i file di origine Java in file bytecode. I file Bytecode sono indipendenti dalla piattaforma. Ciò significa che è possibile compilare il codice su un tipo di hardware e sistema operativo e quindi eseguire il codice su qualsiasi altra piattaforma che supporti Java.

Il comando `javac` è incluso nelle distribuzioni Java Development Kit (JDK).

Il compilatore Java e il resto della toolchain Java standard mettono le seguenti restrizioni sul codice:

- Il codice sorgente è contenuto in file con il suffisso ".java"
- I bytecode sono contenuti in file con il suffisso ".class"
- Per i file sorgente e bytecode nel file system, il percorso dei file deve riflettere la denominazione del pacchetto e della classe.

Nota: il compilatore `javac` non deve essere confuso con il compilatore JIT (Just in Time) che compila bytecode su codice nativo.

Examples

Il comando 'javac': per iniziare

Semplice esempio

Supponendo che "HelloWorld.java" contenga la seguente sorgente Java:

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

(Per una spiegazione del codice precedente, fai riferimento a Come iniziare con Java Language .)

Possiamo compilare il file sopra usando questo comando:

```
$ javac HelloWorld.java
```

Questo produce un file chiamato "HelloWorld.class", che possiamo quindi eseguire come segue:

```
$ java HelloWorld
Hello world!
```

I punti chiave da notare da questo esempio sono:

1. Il nome file di origine "HelloWorld.java" deve corrispondere al nome della classe nel file di origine ... che è `HelloWorld` . Se non corrispondono, si otterrà un errore di compilazione.
2. Il nome file bytec "HelloWorld.class" corrisponde al nome della classe. Se si dovesse rinominare "HelloWorld.class", si otterrebbe un errore quando si tenta di eseguirlo.
3. Quando si esegue un'applicazione Java usando `java` , si fornisce il nome di classe NOT al nome file bytecode.

Esempio con i pacchetti

Il codice Java più pratico utilizza pacchetti per organizzare lo spazio dei nomi per le classi e ridurre il rischio di collisioni accidentali di nomi di classi.

Se volessimo dichiarare la classe `HelloWorld` in una chiamata di pacchetto `com.example` , "HelloWorld.java" conterrà la seguente sorgente Java:

```
package com.example;

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

Questo file del codice sorgente deve essere memorizzato in un albero di directory la cui struttura corrisponde alla denominazione del pacchetto.

```
.    # the current directory (for this example)
|
 ----com
     |
      ----example
          |
           ----HelloWorld.java
```

Possiamo compilare il file sopra usando questo comando:

```
$ javac com/example/HelloWorld.java
```

Questo produce un file chiamato "com / example / HelloWorld.class"; cioè dopo la compilazione, la struttura del file dovrebbe assomigliare a questa:

```
.    # the current directory (for this example)
|
 ----com
     |
      ----example
          |
           ----HelloWorld.java
           ----HelloWorld.class
```

Possiamo quindi eseguire l'applicazione come segue:

```
$ java com.example.HelloWorld
Hello world!
```

Ulteriori punti da notare da questo esempio sono:

1. La struttura della directory deve corrispondere alla struttura del nome del pacchetto.
2. Quando si esegue la classe, è necessario fornire il nome completo della classe; cioè "com.example.HelloWorld" non "HelloWorld".
3. Non è necessario compilare ed eseguire il codice Java fuori dalla directory corrente. Lo stiamo facendo solo qui per l'illustrazione.

Compilare più file contemporaneamente con 'javac'.

Se la tua applicazione è composta da più file di codice sorgente (e quasi tutti!) Puoi compilarli uno alla volta. In alternativa, puoi compilare più file contemporaneamente elencando i nomi dei percorsi:

```
$ javac Foo.java Bar.java
```

o usando la funzionalità jolly del nome del comando shell shell ....

```
$ javac *.java
$ javac com/example/*.java
$ javac */**/*.java #Only works on Zsh or with globstar enabled on your shell
```

In questo modo verranno compilati tutti i file di origine Java nella directory corrente, nella directory "com / example" e, rispettivamente, in modo ricorsivo nelle directory secondarie. Una terza alternativa consiste nel fornire un elenco di nomi di file di origine (e opzioni del compilatore) come un file. Per esempio:

```
$ javac @sourcefiles
```

dove il file `sourcefiles` contiene:

```
Foo.java
Bar.java
com/example/HelloWorld.java
```

Nota: il codice di compilazione come questo è appropriato per piccoli progetti di una sola persona e per programmi una tantum. Oltre a questo, è consigliabile selezionare e utilizzare uno strumento di compilazione Java. In alternativa, la maggior parte dei programmatori utilizza un IDE Java (ad esempio NetBeans , eclipse , IntelliJ IDEA) che offre un compilatore incorporato e la creazione incrementale di "progetti".

Opzioni "javac" comunemente usate

Ecco alcune opzioni per il comando `javac` che potrebbero esserti utili

- L'opzione `-d` imposta una directory di destinazione per scrivere i file ".class".
- L'opzione `-sourcepath` imposta un percorso di ricerca del codice sorgente.
- L'opzione `-cp` o `-classpath` imposta il percorso di ricerca per trovare classi esterne e compilate in precedenza. Per ulteriori informazioni sul classpath e su come specificarlo, fare riferimento a The Classpath Topic.
- L'opzione `-version` stampa le informazioni sulla versione del compilatore.

Un elenco più completo di opzioni del compilatore verrà descritto in un esempio separato.

Riferimenti

Il riferimento definitivo per il comando `javac` è la pagina di manuale di Oracle per `javac` .

Compilare per una versione diversa di Java

Il linguaggio di programmazione Java (e il suo runtime) ha subito numerosi cambiamenti dal suo rilascio dalla sua prima pubblicazione pubblica. Queste modifiche includono:

- Cambiamenti nella sintassi e nella semantica del linguaggio di programmazione Java
- Cambiamenti nelle API fornite dalle librerie di classi standard Java.
- Modifiche nel set di istruzioni Java (bytecode) e nel file di classi.

Con pochissime eccezioni (ad esempio la parola chiave `enum` , le modifiche ad alcune classi "interne", ecc.), Queste modifiche sono retrocompatibili.

- Un programma Java compilato utilizzando una versione precedente della toolchain Java verrà eseguito su una piattaforma Java versione più recente senza ricompilazione.
- Un programma Java che è stato scritto in una versione precedente di Java verrà compilato correttamente con un nuovo compilatore Java.

Compilazione di Java vecchio con un compilatore più recente

Se è necessario (ri) compilare il codice Java precedente su una piattaforma Java più recente per l'esecuzione sulla piattaforma più recente, in genere non è necessario fornire alcun flag di compilazione speciale. In alcuni casi (ad es. Se hai usato `enum` come identificatore) potresti usare l'opzione `-source` per disabilitare la nuova sintassi. Ad esempio, data la seguente classe:

```
public class OldSyntax {
    private static int enum;  // invalid in Java 5 or later
}
```

quanto segue è necessario per compilare la classe usando un compilatore Java 5 (o successivo):

```
$ javac -source 1.4 OldSyntax.java
```

Compilare per una piattaforma di esecuzione precedente

Se è necessario compilare Java per eseguire su piattaforme Java precedenti, l'approccio più semplice è installare un JDK per la versione più vecchia che è necessario supportare e utilizzare il compilatore di JDK nei propri build.

Puoi anche compilare con un compilatore Java più recente, ma ci sono complicati. Prima di tutto, ci sono alcune condizioni preliminari importanti che devono essere soddisfatte:

- Il codice che stai compilando non deve utilizzare costrutti del linguaggio Java che non erano disponibili nella versione di Java che hai scelto come target.
- Il codice non deve dipendere da classi, campi, metodi e standard Java standard che non erano disponibili nelle piattaforme precedenti.
- Le librerie di terze parti che il codice dipende devono essere costruite per la piattaforma precedente e disponibili in fase di compilazione e in fase di esecuzione.

Date le condizioni preliminari soddisfatte, è possibile ricompilare il codice per una piattaforma precedente utilizzando l'opzione `-target` . Per esempio,

```
$ javac -target 1.4 SomeClass.java
```

compilerà la classe precedente per produrre bytecode compatibili con Java 1.4 o successive JVM. (In effetti, l'opzione `-source` implica un `-target` compatibile, quindi `javac -source 1.4 ...` avrebbe lo stesso `-source` relazione tra `-source` e `-target` è descritta nella documentazione di Oracle.)

Detto questo, se si usa semplicemente `-target` o `-source` , si procederà comunque alla compilazione delle librerie di classi standard fornite dal JDK del compilatore. Se non si presta attenzione, è possibile ritrovarsi con classi con la versione di bytecode corretta, ma con dipendenze da API non disponibili. La soluzione è usare l'opzione `-bootclasspath` . Per esempio:

```
$ javac -target 1.4 --bootclasspath path/to/java1.4/rt.jar SomeClass.java
```

si compilerà con un insieme alternativo di librerie di runtime. Se la classe che si sta compilando ha dipendenze (accidentali) sulle nuove librerie, questo ti darà errori di compilazione.

Leggi Compilatore Java - 'javac' online: https://riptutorial.com/it/java/topic/4478/compilatore-java---javac-

Capitolo 38: Compilatore Just in Time (JIT)

Osservazioni

Storia

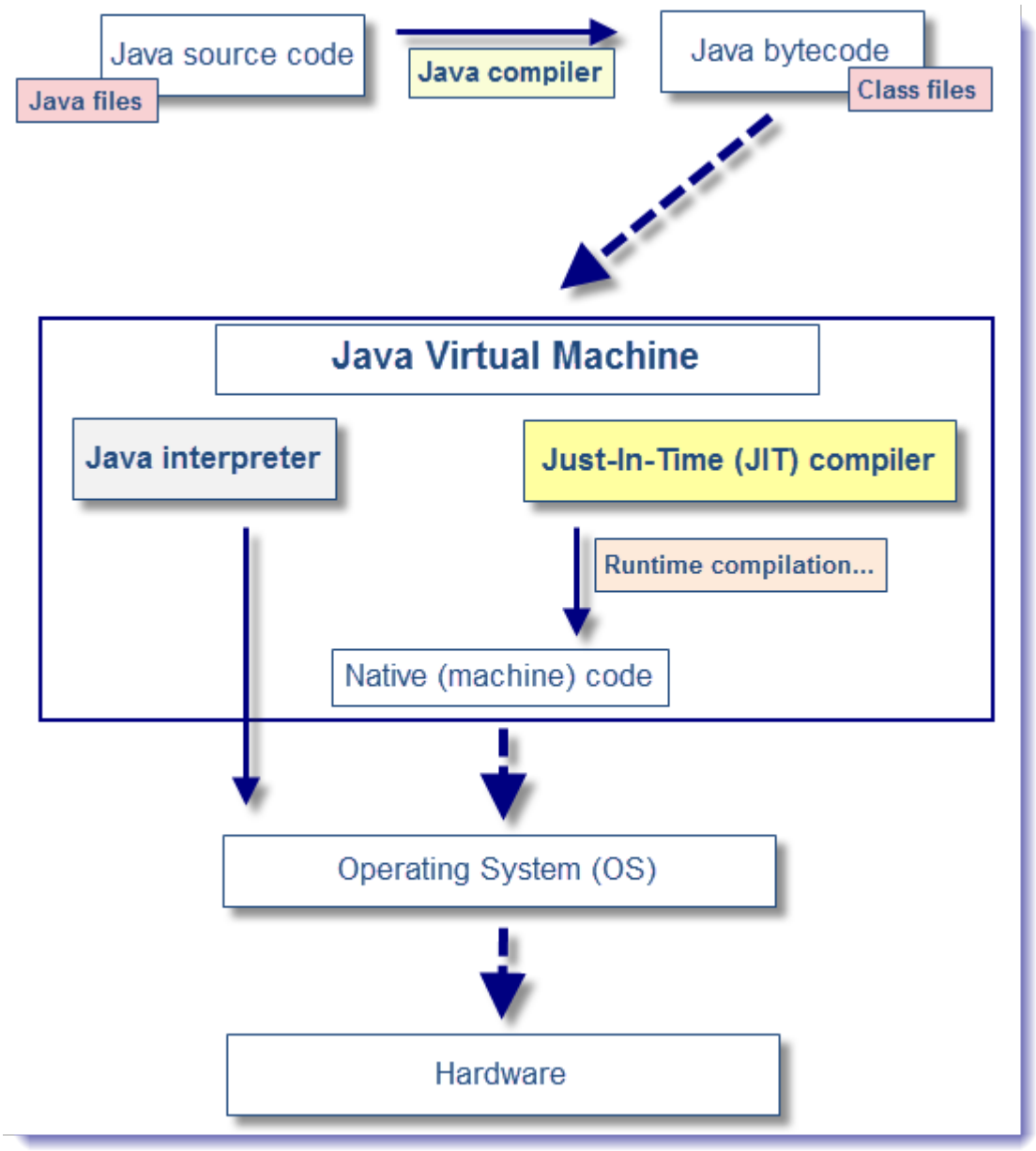
Il compilatore Symantec JIT era disponibile in Sun Java dalla 1.1.5 in poi, ma aveva problemi.

Il compilatore Hotspot JIT è stato aggiunto a Sun Java in 1.2.2 come plug-in. In Java 1.3, JIT era abilitato per impostazione predefinita.

(Fonte: quando Java ha ottenuto un compilatore JIT?

Examples

Panoramica



Il compilatore JIT (Just-In-Time) è un componente di Java ™ Runtime Environment che migliora le prestazioni delle applicazioni Java in fase di esecuzione.

- I programmi Java sono costituiti da classi, che contengono bytecode neutrali alla piattaforma che possono essere interpretati da una JVM su molte architetture di computer differenti.
- In fase di esecuzione, la JVM carica i file di classe, determina la semantica di ogni singolo bytecode ed esegue il calcolo appropriato.

  L'utilizzo aggiuntivo del processore e della memoria durante l'interpretazione significa che un'applicazione Java ha prestazioni più lente rispetto a un'applicazione nativa.

  Il compilatore JIT aiuta a migliorare le prestazioni dei programmi Java compilando bytecode nel codice macchina nativo in fase di esecuzione.

Il compilatore JIT è abilitato per impostazione predefinita e viene attivato quando viene chiamato un metodo Java. Il compilatore JIT compila i bytecode di quel metodo nel codice macchina nativo, compilandolo `"just in time"` per l'esecuzione.

Quando un metodo è stato compilato, la JVM chiama direttamente il codice compilato di quel metodo invece di interpretarlo. Teoricamente, se la compilazione non richiedesse tempo di processore e utilizzo della memoria, la compilazione di ogni metodo potrebbe consentire alla velocità del programma Java di avvicinarsi a quella di un'applicazione nativa.

La compilazione JIT richiede tempo di processore e utilizzo della memoria. Quando la JVM si avvia per la prima volta, vengono chiamati migliaia di metodi. La compilazione di tutti questi metodi può influire in modo significativo sui tempi di avvio, anche se il programma raggiunge prestazioni di picco molto buone.

- In pratica, i metodi non vengono compilati la prima volta che vengono chiamati. Per ogni metodo, la JVM mantiene un `call count` che viene incrementato ogni volta che viene chiamato il metodo.
- La JVM interpreta un metodo fino a quando il conteggio delle chiamate supera una soglia di compilazione JIT.
- Pertanto, i metodi utilizzati più spesso vengono compilati subito dopo l'avvio della JVM e i metodi meno utilizzati vengono compilati molto più tardi, o per niente.
- La soglia di compilazione JIT aiuta JVM ad avviarsi rapidamente e a migliorare le prestazioni.
- La soglia è stata accuratamente selezionata per ottenere un equilibrio ottimale tra i tempi di avvio e le prestazioni a lungo termine.
- Dopo che un metodo è stato compilato, il conteggio delle chiamate viene azzerato e le chiamate successive al metodo continuano ad incrementare il conteggio.
- Quando il conteggio chiamate di un metodo raggiunge una soglia di ricompilazione JIT, il compilatore JIT lo compila una seconda volta, applicando una selezione più ampia di ottimizzazioni rispetto alla compilazione precedente.
- Questo processo viene ripetuto fino al raggiungimento del livello massimo di ottimizzazione.

  I metodi più impegnativi di un programma Java sono sempre ottimizzati in modo più aggressivo, massimizzando i vantaggi prestazionali dell'uso del compilatore JIT.

Il compilatore JIT può anche misurare i `operational data at run time` e utilizzare tali dati per migliorare la qualità di ulteriori ricompilazioni.

> Il compilatore JIT può essere disabilitato, nel qual caso verrà interpretato l'intero programma Java. La disabilitazione del compilatore JIT non è consigliata eccetto per diagnosticare o aggirare i problemi di compilazione JIT.

Leggi Compilatore Just in Time (JIT) online: https://riptutorial.com/it/java/topic/5152/compilatore-just-in-time--jit-

Capitolo 39: CompletableFuture

introduzione

CompletableFuture è una classe aggiunta a Java SE 8 che implementa l'interfaccia Future di Java SE 5. Oltre a supportare l'interfaccia Future, vengono aggiunti molti metodi che consentono la richiamata asincrona quando il futuro è completato.

Examples

Convertire il metodo di blocco in modo asincrono

Il seguente metodo impiegherà un secondo o due a seconda della connessione per recuperare una pagina Web e contare la lunghezza del testo. Qualunque thread lo chiamerà, bloccherà per quel periodo di tempo. Inoltre ripropone un'eccezione che è utile in seguito.

```
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);
    }
}
```

Questo lo converte in un metodo che ritornerà immediatamente spostando la chiamata del metodo di blocco su un altro thread. Per impostazione predefinita, il metodo supplyAsync eseguirà il fornitore nel pool comune. Per un metodo di blocco, probabilmente non è una buona scelta dato che si potrebbero esaurire i thread in quel pool ed è per questo che ho aggiunto il parametro di servizio opzionale.

```
static private ExecutorService service = Executors.newCachedThreadPool();

static public CompletableFuture<Long> asyncGetWebPageLength(String url) {
    return CompletableFuture.supplyAsync(() -> blockingGetWebPageLength(url), service);
}
```

Per usare la funzione in modo asincrono si dovrebbe usare uno dei metodi che accettano una lamda da chiamare con il risultato del fornitore quando completa come ad esempio Accetta. È inoltre importante utilizzare eccezionalmente o gestire il metodo per registrare eventuali eccezioni che potrebbero essere accadute.

```
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;
            });

}
```

Semplice esempio di CompletableFuture

Nell'esempio qui sotto, il `calculateShippingPrice` metodo calcola il costo di trasporto, che richiede un certo tempo di elaborazione. In un esempio reale, questo sarebbe per esempio contattare un altro server che restituisce il prezzo in base al peso del prodotto e al metodo di spedizione.

Modellando questo in modo asincrono tramite `CompletableFuture` , possiamo continuare il lavoro differente nel metodo (cioè calcolare i costi di imballaggio).

```
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!
    });
}
```

Leggi CompletableFuture online: https://riptutorial.com/it/java/topic/10935/completablefuture

Capitolo 40: Confronto C ++

introduzione

Java e C ++ sono lingue simili. Questo argomento funge da guida di riferimento rapido per gli ingegneri Java e C ++.

Osservazioni

Classi definite all'interno di altri costrutti

Definito all'interno di un'altra classe

C ++

Classe annidata [ref] (richiede un riferimento per includere la classe)

```
class Outer {
   class Inner {
      public:
         Inner(Outer* o) :outer(o) {}

      private:
         Outer*  outer;
   };
};
```

Giava

[non statico] Classe annidata (alias anche Inner Class o Member Class)

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Definito staticamente in un'altra classe

C ++

Classe annidata statica

```
class Outer {
   class Inner {
      ...
   };
};
```

Giava

Classe annidata statica (aka Classe membro statico) [ref]

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

Definito all'interno di un metodo

(es. gestione degli eventi)

C ++

Classe locale [ref]

```
void fun() {
   class Test {
      /* members of Test class */
   };
}
```

Vedi anche espressioni Lambda

Giava

Classe locale [ref]

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
            }
        }).start();
    }
}
```

Override vs Sovraccarico

I seguenti punti Overriding e Overloading si applicano sia a C ++ che a Java:

- Un metodo sottoposto a override ha lo stesso nome e gli stessi argomenti del suo metodo base.
- Un metodo sovraccarico ha lo stesso nome ma argomenti diversi e non si basa sull'ereditarietà.
- Due metodi con lo stesso nome e argomenti ma tipi di ritorno diversi sono illegali. Vedi le domande relative a StackOverflow relative a "overloading con diverso tipo di ritorno in Java" - Domanda 1 ; Domanda 2

Polimorfismo

Il polimorfismo è la capacità di oggetti di classi diverse legate dall'ereditarietà di rispondere in modo diverso alla stessa chiamata di metodo. Ecco un esempio:

- classe base Forma con area come metodo astratto
- due classi derivate, Square e Circle, implementano i metodi dell'area
- Forma i punti di riferimento su Square e l'area viene invocata

In C ++, il polimorfismo è abilitato con metodi virtuali. In Java, i metodi sono virtuali per impostazione predefinita.

Ordine di costruzione / distruzione

Pulizia degli oggetti

In C ++, è una buona idea dichiarare un distruttore come virtuale per garantire che il distruttore della sottoclasse venga chiamato se il puntatore della classe base viene cancellato.

In Java, un metodo finalizzato è simile a un distruttore in C ++; tuttavia, i finalizzatori sono imprevedibili (si basano su GC). Best practice: utilizzare un metodo "close" per eseguire una pulizia esplicita.

```
protected void close() {
    try {
       // do subclass cleanup
    }
    finally {
       isClosed = true;
       super.close();
    }
}
```

```
protected void finalize() {
    try {
       if(!isClosed) close();
    }
    finally {
       super.finalize();
    }
}
```

Metodi e classi astratte

Concetto	C ++	Giava
Metodo astratto dichiarato senza implementazione	metodo virtuale puro `virtual void eat(void) = 0;`	metodo astratto `abstract void draw();`
Classe astratta non può essere istanziato	non può essere istanziato; ha almeno un metodo virtuale puro `class AB {public: virtual void f() = 0;};`	non può essere istanziato; può avere metodi non astratti `abstract class GraphicObject {}`
Interfaccia nessun campo istanza	nessuna parola chiave "interfaccia", ma può simulare un'interfaccia Java con funzioni di una classe astratta	molto simile alla classe astratta, ma 1) supporta l'ereditarietà multipla; 2) nessun campo di istanza `interface TestInterface {}`

Modificatori di accessibilità

Modificatore	C ++	Giava
Pubblico - accessibile a tutti	*senza note speciali*	*senza note speciali*
Protetto - accessibile da sottoclassi	accessibile anche dagli amici	accessibile anche all'interno dello stesso pacchetto
Privato - accessibile dai membri	accessibile anche dagli amici	*senza note speciali*
predefinito	l'impostazione predefinita della classe è privata; struct default è public	accessibile da tutte le classi all'interno dello stesso pacchetto

Modificatore	C ++	Giava
altro	Amico: un modo per concedere l'accesso a membri privati e protetti senza ereditarietà (vedi sotto)	

Esempio di amico C ++

```
class Node {
  private:
    int key;  Node *next;
    // LinkedList::search() can access "key" & "next"
    friend int LinkedList::search();
};
```

Il temuto problema del diamante

> Il problema dei diamanti è un'ambiguità che sorge quando due classi B e C ereditano da A, e la classe D eredita sia da B che da C. Se c'è un metodo in A che B e C hanno scavalcato, e D non lo sovrascrive, quindi quale versione del metodo eredita D: quella di B, o quella di C? (da Wikipedia)

Sebbene C ++ sia sempre stato sensibile al problema dei diamanti, Java era suscettibile fino a Java 8. Originariamente, Java non supportava l'ereditarietà multipla, ma con l'avvento dei metodi di interfaccia predefiniti, le classi Java non possono ereditare "implementazione" da più di una classe .

java.lang.Object Class

In Java tutte le classi ereditano, in modo implicito o esplicito, dalla classe Object. Qualsiasi riferimento Java può essere convertito nel tipo Object.

C ++ non ha una classe "Object" comparabile.

Collezioni Java e contenitori C ++

Le raccolte Java sono sinonimi dei contenitori C ++.

Diagramma di flusso delle raccolte Java

Diagramma di flusso dei contenitori C ++

Tipi interi

bits	min	Max	Tipo C ++ (su LLP64 o LP64)	Tipo Java
8	-2 (8-1) = -128	2 (8-1) -1 = 127	carbonizzare	byte
8	0	2 (8) -1 = 255	char unsigned	-
16	-2 (16-1) = -32.768	2 (16-1) -1 = 32.767	corto	corto
16	0 (\ u0000)	2 (16) -1 = 65.535 (\ uFFFF)	corto senza firma	char (non firmato)
32	-2 (32-1) = -2.147 miliardi	2 (32-1) -1 = 2,147 miliardi	int	int
32	0	2 (32) -1 = 4,295 miliardi	int non firmato	-
64	-2 (64-1)	2 (16-1) -1	lungo*	lungo lungo
64	0	2 (16) -1	lungo non firmato * non firmato lungo	-

* L'API Win64 è solo a 32 bit

Molti altri tipi di C ++

Examples

Membri della classe statica

I membri statici hanno un ambito di classe rispetto all'ambito dell'oggetto

Esempio di C ++

```
// define in header
class Singleton {
   public:
      static Singleton *getInstance();

   private:
      Singleton() {}
      static Singleton *instance;
```

```
};

// initialize in .cpp
Singleton* Singleton::instance = 0;
```

Esempio di Java

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Classi definite all'interno di altri costrutti

Definito all'interno di un'altra classe

C ++

Classe annidata [ref] (richiede un riferimento per includere la classe)

```
class Outer {
   class Inner {
      public:
         Inner(Outer* o) :outer(o) {}

      private:
         Outer*  outer;
   };
};
```

Giava

[non statico] Classe annidata (alias anche Inner Class o Member Class)

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Definito staticamente in un'altra classe

C ++

Classe annidata statica

```
class Outer {
   class Inner {
      ...
   };
};
```

Giava

Classe annidata statica (aka Classe membro statico) [ref]

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

Definito all'interno di un metodo

(es. gestione degli eventi)

C ++

Classe locale [ref]

```
void fun() {
   class Test {
      /* members of Test class */
   };
}
```

Giava

Classe locale [ref]

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
```

```
            }
        }).start();
    }
}
```

Pass-by-value e Pass-by-riferimento

Molti sostengono che Java sia SOLO un valore pass-by, ma è più sfumato di così. Confrontate i seguenti esempi C ++ e Java per vedere i molti sapori del pass-by-value (ovvero la copia) e il pass-by-reference (alias alias).

Esempio C ++ (codice 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;
}
```

Esempio Java (codice 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);
   }
```

Ereditarietà rispetto alla composizione

C ++ e Java sono entrambi linguaggi orientati agli oggetti, quindi il seguente diagramma si applica a entrambi.

Downcasting dell'outcast

Attenzione all'utilizzo di "downcasting" - Downcasting sta gettando giù la gerarchia dell'eredità da una classe base a una sottoclasse (cioè opposta al polimorfismo). In generale, usa il polimorfismo e l'override invece di instanceof & downcasting.

Esempio di C ++

```
// explicit type case required
Child *pChild =  (Child *) &parent;
```

Esempio di Java

```
if(mySubClass instanceof SubClass) {
   SubClass mySubClass = (SubClass)someBaseClass;
   mySubClass.nonInheritedMethod();
}
```

Metodi e classi astratte

Metodo astratto

dichiarato senza implementazione

C ++

metodo virtuale puro

```
virtual void eat(void) = 0;
```

Giava

metodo astratto

```
abstract void draw();
```

Classe astratta

non può essere istanziato

C ++

non può essere istanziato; ha almeno un metodo virtuale puro

```
class AB {public: virtual void f() = 0;};
```

Giava

non può essere istanziato; può avere metodi non astratti

```
abstract class GraphicObject {}
```

Interfaccia

nessun campo istanza

C ++

niente paragonabile a Java

Giava

molto simile alla classe astratta, ma 1) supporta l'ereditarietà multipla; 2) nessun campo di istanza

```
interface TestInterface {}
```

Leggi Confronto C ++ online: https://riptutorial.com/it/java/topic/10849/confronto-c-plusplus

Capitolo 41: Console I / O

Examples

Leggere l'input dell'utente dalla console

Utilizzando `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());
}
```

Le seguenti importazioni sono necessarie per questo codice:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

Utilizzo dello `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 seguente importazione è necessaria per questo esempio:

```
import java.util.Scanner;
```

Per leggere più di una riga, richiamare `scanner.nextLine()` ripetutamente:

```
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 + "!");
```

Esistono due metodi per ottenere `Strings` , `next()` e `nextLine()` . `next()` restituisce il testo fino al primo spazio (noto anche come "token"), e `nextLine()` restituisce tutto il testo immesso dall'utente fino a quando non si preme enter.

`Scanner` fornisce anche metodi di utilità per la lettura di tipi di dati diversi da `String` . Questi includono:

```
scanner.nextByte();
scanner.nextShort();
scanner.nextInt();
scanner.nextLong();
scanner.nextFloat();
scanner.nextDouble();
scanner.nextBigInteger();
scanner.nextBigDecimal();
```

Prefixing di uno qualsiasi di questi metodi con `has` (come in `hasNextLine()` , `hasNextInt()`) restituisce `true` se lo stream ha più del tipo di richiesta. Nota: questi metodi causeranno il crash del programma se l'input non è del tipo richiesto (ad esempio, digitando "a" per `nextInt()`). Puoi usare `try {} catch() {}` per impedirlo (vedi: Eccezioni)

```
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");
}
```

Utilizzando `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();
```

Vantaggi :

- I metodi di lettura sono sincronizzati
- È possibile utilizzare la sintassi della stringa del formato

Nota : funziona solo se il programma viene eseguito da una riga di comando reale senza reindirizzare gli stream di input e output standard. Non funziona quando il programma viene eseguito da determinati IDE, come ad esempio Eclipse. Per il codice che funziona all'interno degli

IDE e con il reindirizzamento del flusso, vedere gli altri esempi.

Implementazione del comportamento di base della riga di comando

Per i prototipi di base o il comportamento di base della riga di comando, il seguente ciclo è utile.

```
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");
    }
```

```
}
```

Allineare le stringhe in console

Il metodo `PrintWriter.format` (chiamato tramite `System.out.format`) può essere utilizzato per stampare stringhe allineate nella console. Il metodo riceve una `String` con le informazioni sul formato e una serie di oggetti da formattare:

```
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();
}
```

Produzione:

```
1   1          1
1234 1234    1234
1234567 1234567 123456
123456789 12345678 123456
```

L'utilizzo di stringhe di formato con dimensioni fisse consente di stampare le stringhe in un aspetto simile a una tabella con colonne a dimensione fissa:

```
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();
}
```

Produzione:

```
1   1             1
123 1234       1234
123 1234567  123456
```

```
123 12345678 123456
```

Formattare gli esempi di stringhe

- `%s` : solo una stringa senza formattazione
- `%5s` : formatta la stringa con un **minimo** di 5 caratteri; se la stringa è più corta sarà **riempita** a 5 caratteri e allineata a **destra**
- `%-5s` : formatta la stringa con un **minimo** di 5 caratteri; se la stringa è più corta sarà **riempita** a 5 caratteri e allineata a **sinistra**
- `%5.10s` : formatta la stringa con un **minimo** di 5 caratteri e un **massimo** di 10 caratteri; se la stringa è più corta di 5 sarà **riempita** a 5 caratteri e allineata a **destra** ; se la stringa è più lunga di 10, verrà **troncata** a 10 caratteri e allineata a **destra**
- `%-5.5s` : formatta la stringa con una dimensione **fissa** di 5 caratteri (il minimo e il massimo sono uguali); se la stringa è più corta di 5 sarà **riempita** a 5 caratteri e allineata a **sinistra** ; se la stringa è più lunga di 5 verrà **troncata** a 5 caratteri e allineata a **sinistra**

Leggi Console I / O online: https://riptutorial.com/it/java/topic/126/console-i---o

Capitolo 42: Conversione da e verso le stringhe

Examples

Conversione di altri tipi di dati in String

- È possibile ottenere il valore di altri tipi di dati primitivi come String utilizzando uno dei metodi `valueOf` della classe String.

  Per esempio:

  ```
  int i = 42;
  String string = String.valueOf(i);
  //string now equals "42”.
  ```

  Questo metodo è anche sovraccaricato per altri tipi di dati, come `float` , `double` , `boolean` e anche `Object` .

- Puoi anche ottenere qualsiasi altro oggetto (qualsiasi istanza di qualsiasi classe) come stringa chiamando su `.toString` . Affinché questo fornisca un output utile, la classe deve eseguire l'override di `toString()` . La maggior parte delle classi di librerie Java standard, come `Date` e altri.

  Per esempio:

  ```
  Foo foo = new Foo(); //Any class.
  String stringifiedFoo = foo.toString().
  ```

  Qui `stringifiedFoo` contiene una rappresentazione di `foo` come una stringa.

Puoi anche convertire qualsiasi tipo di numero in String con notazione breve come sotto.

```
int i = 10;
String str = i + "";
```

O semplicemente il modo semplice è

```
String str = 10 + "";
```

Conversione da / a byte

Per codificare una stringa in un array di byte, puoi semplicemente utilizzare il metodo `String#getBytes()` , con uno dei set di caratteri standard disponibili su qualsiasi runtime Java:

```
byte[] bytes = "test".getBytes(StandardCharsets.UTF_8);
```

e per decodificare:

```
String testString = new String(bytes, StandardCharsets.UTF_8);
```

puoi semplificare ulteriormente la chiamata utilizzando un'importazione statica:

```
import static java.nio.charset.StandardCharsets.UTF_8;
...
byte[] bytes = "test".getBytes(UTF_8);
```

Per set di caratteri meno comuni puoi indicare il set di caratteri con una stringa:

```
byte[] bytes = "test".getBytes("UTF-8");
```

e il contrario:

```
String testString = new String (bytes, "UTF-8");
```

questo tuttavia significa che devi gestire l' `UnsupportedCharsetException` controllato.

La seguente chiamata utilizzerà il set di caratteri predefinito. Il set di caratteri predefinito è specifico per piattaforma e generalmente differisce tra piattaforme Windows, Mac e Linux.

```
byte[] bytes = "test".getBytes();
```

e il contrario:

```
String testString = new String(bytes);
```

Si noti che caratteri e byte non validi possono essere sostituiti o saltati con questi metodi. Per un maggiore controllo, ad esempio per la convalida dell'input, sei incoraggiato a utilizzare le classi `CharsetEncoder` e `CharsetDecoder` .

Codifica / decodifica Base64

Occasionalmente troverai la necessità di codificare i dati binari come una stringa con codifica *base64* .

Per questo siamo in grado di utilizzare la `DatatypeConverter` classe dal `javax.xml.bind` pacchetto:

```
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

In alternativa, possiamo usare `Base64` da 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)));
```

Se si ispeziona questo programma wile, si vedrà che `someBinaryData` codificano in `c29tZUJpbmFyeURhdGE=` , un oggetto String *UTF-8* molto `c29tZUJpbmFyeURhdGE=` .

Java SE 8

I dettagli per lo stesso possono essere trovati su 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);
```

Riferimento

Analisi delle stringhe su un valore numerico

Stringa con un tipo numerico primitivo o un tipo di wrapper numerico:

Ogni classe di wrapper numerico fornisce un metodo `parseXxx` che converte una `String` nel tipo

primitivo corrispondente. Il codice seguente converte una `String` in un `int` utilizzando il metodo `Integer.parseInt` :

```
String string = "59";
int primitive = Integer.parseInteger(string);
```

Per convertire una `String` in un'istanza di una classe wrapper numerica è possibile utilizzare un overload del metodo `valueOf` classi wrapper:

```
String string = "59";
Integer wrapper = Integer.valueOf(string);
```

o fare affidamento sul box automatico (Java 5 e versioni successive):

```
String string = "59";
Integer wrapper = Integer.parseInteger(string);  // 'int' result is autoboxed
```

Il modello precedente funziona per `byte` , `short` , `int` , `long` , `float` e `double` e le corrispondenti classi wrapper (`Byte` , `Short` , `Integer` , `Long` , `Float` e `Double`).

Da String a Integer 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
```

eccezioni

L'eccezione NumberFormatException deselezionata verrà generata se viene chiamato un metodo numerico `valueOf(String)` o `parseXxx(...)` per una stringa che non è una rappresentazione numerica accettabile o che rappresenta un valore non compreso nell'intervallo.

Ottenere un `String` da un` InputStream`

Una `String` può essere letta da un `InputStream` utilizzando il costruttore di array di byte.

```
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);
}
```

Questo utilizza il set di caratteri predefinito del sistema, sebbene possa essere specificato un set di caratteri alternativo:

```
return new String(bytes, Charset.forName("UTF-8"));
```

Conversione di stringhe in altri tipi di dati.

È possibile convertire una stringa **numerica** in vari tipi numerici Java come segue:

String to int:

```
String number = "12";
int num = Integer.parseInt(number);
```

String to float:

```
String number = "12.0";
float num = Float.parseFloat(number);
```

Stringa da raddoppiare:

```
String double = "1.47";
double num = Double.parseDouble(double);
```

Da stringa a booleana:

```
String falseString = "False";
boolean falseBool = Boolean.parseBoolean(falseString);   // falseBool = false

String trueString = "True";
boolean trueBool = Boolean.parseBoolean(trueString);     // trueBool = true
```

Stringa a lungo:

```
String number = "47";
long num = Long.parseLong(number);
```

Stringa a BigInteger:

```
String bigNumber = "21";
BigInteger reallyBig = new BigInteger(bigNumber);
```

String to BigDecimal:

```
String bigFraction = "17.21455";
BigDecimal reallyBig = new BigDecimal(bigFraction);
```

Eccezioni di conversione:

Le conversioni numerico sopra saranno tutti lanciare un (selezionata) `NumberFormatException` se si tenta di analizzare una stringa che non è un numero opportunamente formattato, o è fuori portata per il tipo di bersaglio. L'argomento Eccezioni illustra come gestire tali eccezioni.

Se vuoi testare che puoi analizzare una stringa, puoi implementare un metodo `tryParse...` questo

modo:

```
boolean tryParseInt (String value) {
    try {
        String somechar = Integer.parseInt(value);
        return true;
     } catch (NumberFormatException e) {
        return false;
     }
}
```

Tuttavia, chiamare questo metodo `tryParse...` immediatamente prima dell'analisi è (discutibilmente) una cattiva pratica. Sarebbe meglio chiamare il metodo `parse...` e gestire l'eccezione.

Leggi Conversione da e verso le stringhe online:
https://riptutorial.com/it/java/topic/6678/conversione-da-e-verso-le-stringhe

Capitolo 43: Costruttori

introduzione

Sebbene non richiesti, i costruttori in Java sono metodi riconosciuti dal compilatore per creare istanze di valori specifici per la classe che possono essere essenziali per il ruolo dell'oggetto. Questo argomento dimostra l'uso corretto dei costruttori di classi Java.

Osservazioni

La specifica del linguaggio Java parla a lungo della natura esatta della semantica del costruttore. Possono essere trovati in JLS §8.8

Examples

Costruttore predefinito

Il "default" per i costruttori è che non hanno argomenti. Nel caso in cui non si specifica **alcun** costruttore, il compilatore genererà per te un costruttore predefinito.
Ciò significa che i seguenti due snippet sono semanticamente equivalenti:

```
public class TestClass {
    private String test;
}
```

```
public class TestClass {
    private String test;
    public TestClass() {

    }
}
```

La visibilità del costruttore predefinito è la stessa della visibilità della classe. Quindi un pacchetto definito dalla classe ha un costruttore predefinito privato del pacchetto

Tuttavia, se hai un costruttore non predefinito, il compilatore non genererà per te un costruttore predefinito. Quindi questi non sono equivalenti:

```
public class TestClass {
    private String test;
    public TestClass(String arg) {
    }
}
```

```
public class TestClass {
    private String test;
    public TestClass() {
```

```
    }
    public TestClass(String arg) {
    }
}
```

Fare attenzione che il costruttore generato non esegue l'inizializzazione non standard. Ciò significa che tutti i campi della classe avranno il loro valore predefinito, a meno che non abbiano un inizializzatore.

```
public class TestClass {

    private String testData;

    public TestClass() {
        testData = "Test"
    }
}
```

I costruttori sono chiamati così:

```
TestClass testClass = new TestClass();
```

Costruttore con argomenti

I costruttori possono essere creati con qualsiasi tipo di argomento.

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
        this.testData = testData;
    }
}
```

Chiamato in questo modo:

```
TestClass testClass = new TestClass("Test Data");
```

Una classe può avere più costruttori con diverse firme. Per concatenare le chiamate del costruttore (chiamare un costruttore diverso della stessa classe durante l'istanziazione) usare `this()` .

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
        this.testData = testData;
    }

    public TestClass() {
```

```
        this("Test"); // testData defaults to "Test"
    }
}
```

Chiamato in questo modo:

```
TestClass testClass1 = new TestClass("Test Data");
TestClass testClass2 = new TestClass();
```

Chiama il costruttore genitore

Supponiamo che tu abbia una classe Parent e una classe Child. Per costruire un'istanza Child, è necessario sempre un costruttore Parent da eseguire nella gebinning del costruttore Child. Possiamo selezionare il costruttore principale che vogliamo chiamando esplicitamente `super(...)` con gli argomenti appropriati come la nostra prima istruzione del costruttore Child. Questo ci fa risparmiare tempo riutilizzando il costruttore delle classi Parent invece di riscrivere lo stesso codice nel costruttore delle classi Child.

Senza metodo `super(...)` :

(implicitamente, la versione no-args `super()` è chiamata invisibilmente)

```
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 il metodo `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: le chiamate a un altro costruttore (concatenamento) o al super costruttore **DEVONO** essere la prima istruzione all'interno del costruttore.

Se si chiama esplicitamente il costruttore `super(...)` , deve esistere un costruttore genitore corrispondente (è semplice, non è vero?).

Se non si richiama esplicitamente alcun costruttore `super(...)` , la classe genitore deve avere un costruttore no-args e questo può essere scritto esplicitamente o creato come predefinito dal compilatore se la classe genitore non fornisce qualsiasi costruttore.

```
class Parent{
    public Parent(String tName, int tAge) {}
}

class Child extends Parent{
    public Child(){}
}
```

La classe Parent non ha un costruttore predefinito, quindi il compilatore non può aggiungere `super` nel costruttore Child. Questo codice non verrà compilato. Devi cambiare i costruttori per adattarli a entrambi i lati, o scrivere la tua `super` chiamata, in questo modo:

```
class Child extends Parent{
    public Child(){
          super("",0);
    }
}
```

Leggi Costruttori online: https://riptutorial.com/it/java/topic/682/costruttori

Capitolo 44: Creazione di immagini a livello di codice

Osservazioni

`BufferedImage.getGraphics()` restituisce sempre `Graphics2D` .

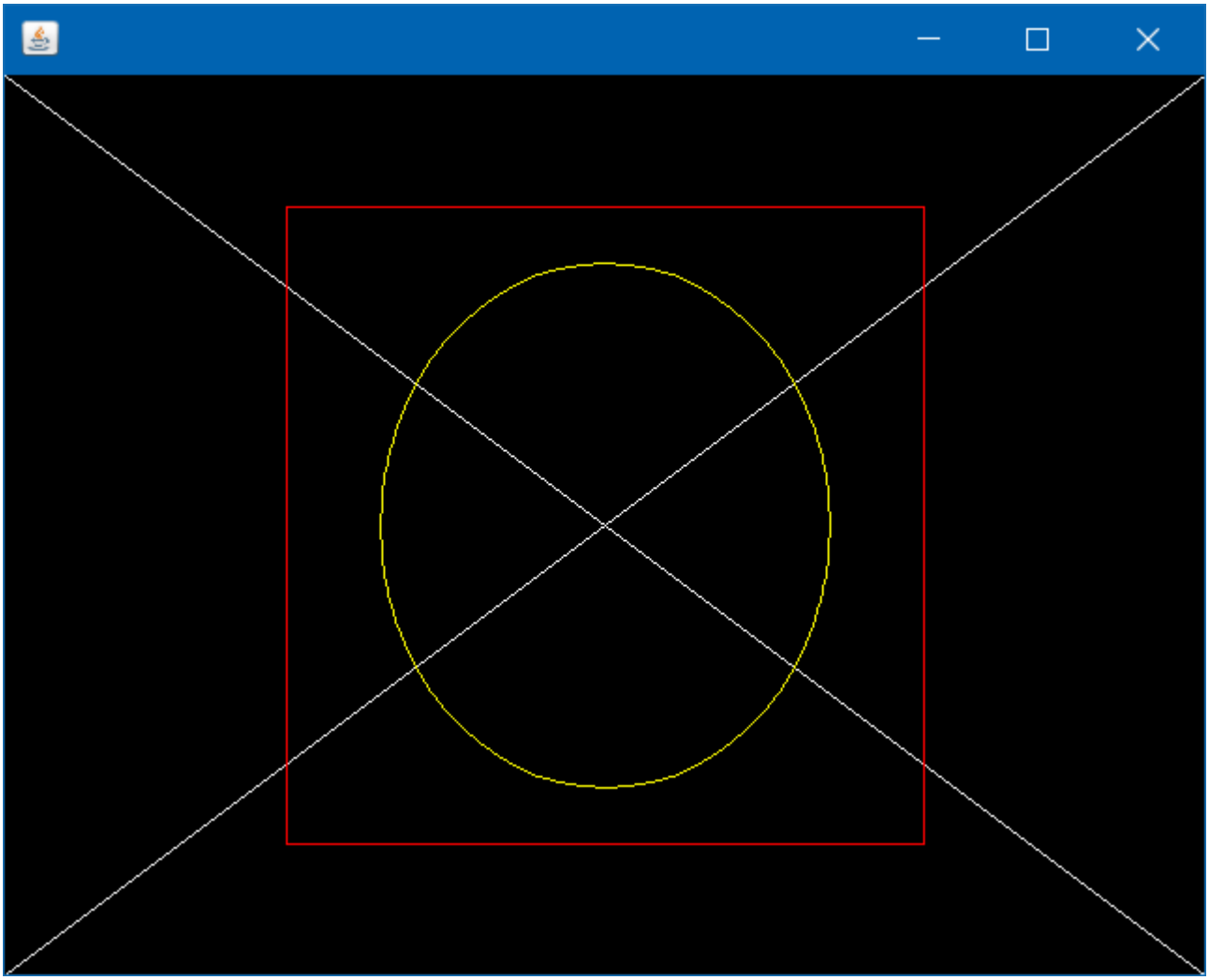
L'utilizzo di `VolatileImage` può migliorare significativamente la velocità delle operazioni di disegno, ma ha anche i suoi svantaggi: il suo contenuto può essere perso in qualsiasi momento e potrebbe essere necessario ridisegnarlo da zero.

Examples

Creare una semplice immagine a livello di programmazione e visualizzarla

```
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);
    }
}
```



Salva un'immagine su disco

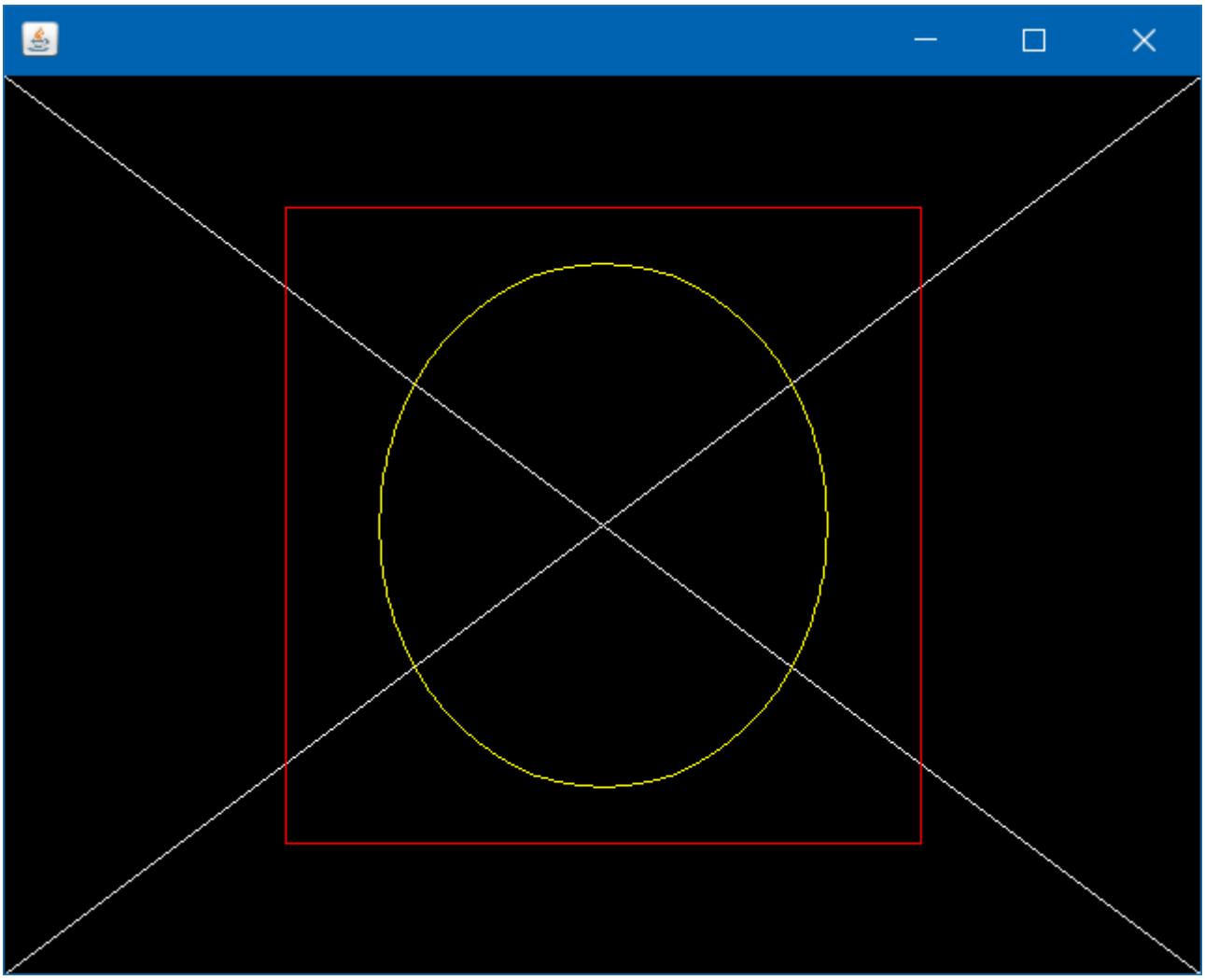
```
public static void saveImage(String destination) throws IOException {
    // method implemented in "Creating a simple image Programmatically and displaying it"
example
    BufferedImage img = createSampleImage();

    // ImageIO provides several write methods with different outputs
    ImageIO.write(img, "png", new File(destination));
}
```

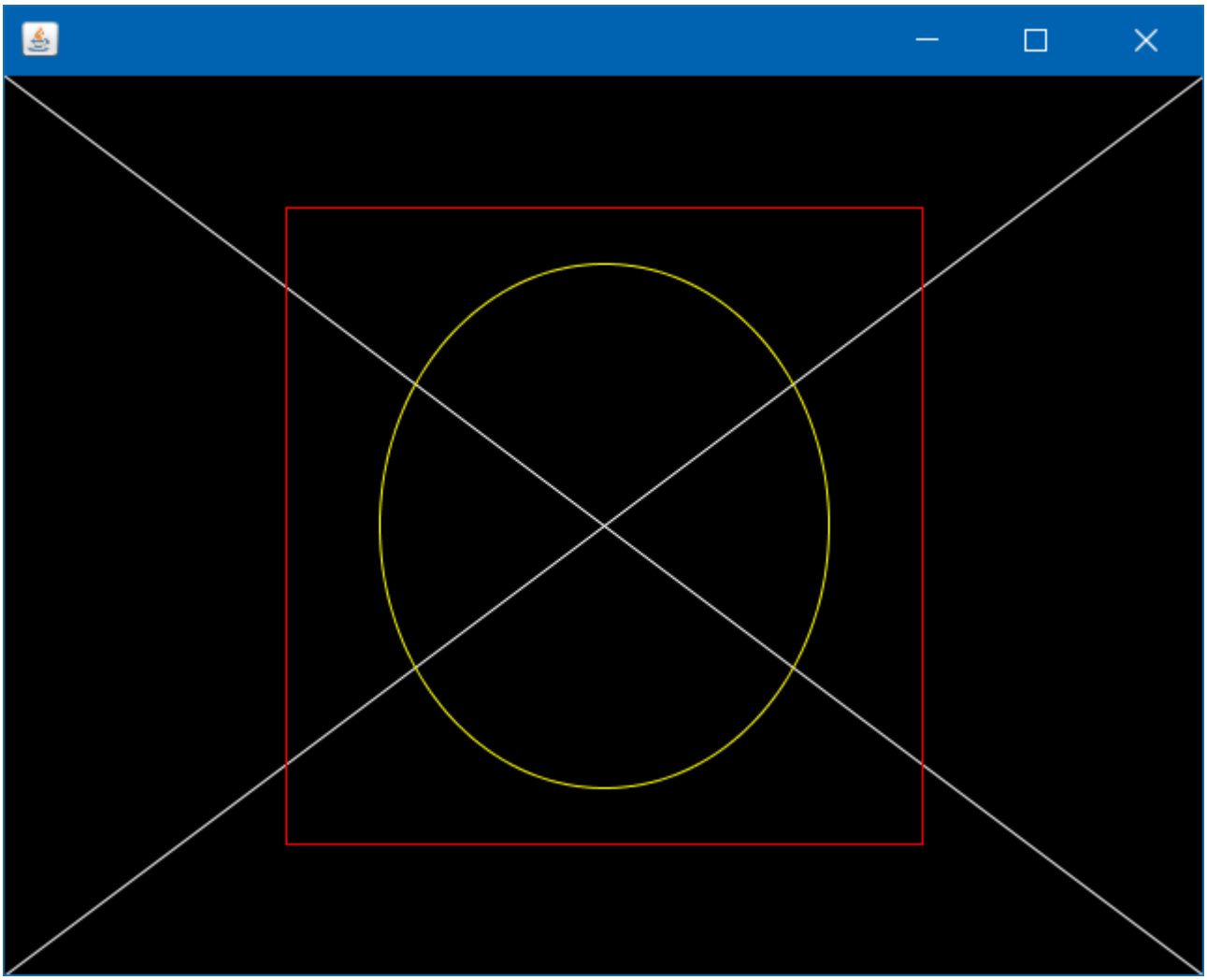
Specifica della qualità del rendering dell'immagine

```
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);
}
```

Confronto tra il rendering QUALITÀ e VELOCITÀ dell'immagine di esempio:





Creazione di un'immagine con la classe 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"));
```

Produzione:

Modifica e riutilizzo dell'immagine 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"));
```

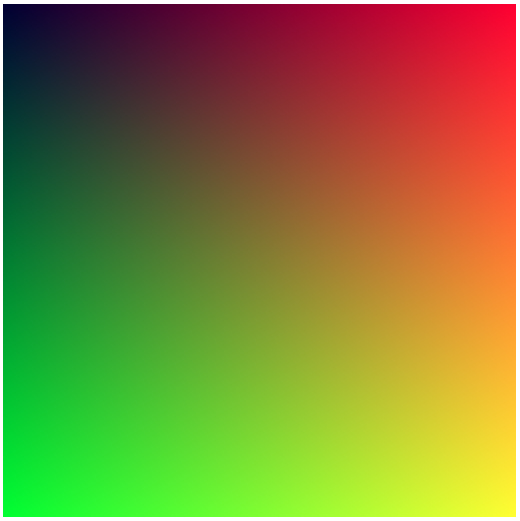
File cat originale:

File prodotto:

Impostazione del colore dei singoli pixel in 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"));
```

Produzione:



Come ridimensionare una BufferedImage

```
/**
* 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;
}
```

Leggi Creazione di immagini a livello di codice online:
https://riptutorial.com/it/java/topic/5482/creazione-di-immagini-a-livello-di-codice

Capitolo 45: Crittografia RSA

Examples

Un esempio che utilizza un crittosistema ibrido costituito da OAEP e GCM

L'esempio seguente crittografa i dati utilizzando un crittosistema ibrido costituito da AES GCM e OAEP, utilizzando le dimensioni dei parametri predefinite e una dimensione della chiave AES di 128 bit.

OAEP è meno vulnerabile agli attacchi oracle di riempimento rispetto al padding PKCS # 1 v1.5. GCM è anche protetto dagli attacchi di oracle.

La decrittografia può essere eseguita recuperando prima la lunghezza della chiave incapsulata e quindi recuperando la chiave incapsulata. La chiave incapsulata può quindi essere decodificata utilizzando la chiave privata RSA che forma una coppia di chiavi con la chiave pubblica. Successivamente, il testo cifrato crittografato AES / GCM può essere decodificato sul testo in chiaro originale.

Il protocollo consiste di:

1. un campo lunghezza per la chiave spostata (`RSAPrivateKey` manca un metodo `getKeySize()`);
2. la chiave incapsulata / incapsulata, della stessa dimensione della dimensione della chiave RSA in byte;
3. il testo cifrato di GCM e il tag di autenticazione a 128 bit (aggiunto automaticamente da Java).

Gli appunti:

- Per usare correttamente questo codice devi fornire una chiave RSA di almeno 2048 bit, più grande è meglio (ma più lento, specialmente durante la decodifica);
- Per utilizzare AES-256 è necessario installare prima i file di criteri di crittografia illimitati ;
- Invece di creare il tuo protocollo potresti voler usare un formato contenitore come la Sintassi dei messaggi crittografici (CMS / PKCS # 7) o PGP.

Quindi, ecco l'esempio:

```
/**
 * 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();
}
```

Certo, la crittografia non è molto utile senza decifrare. Si noti che questo restituirà informazioni minime se la decifrazione fallisce.

```
/**
 * 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();
}
```

Leggi Crittografia RSA online: https://riptutorial.com/it/java/topic/1889/crittografia-rsa

Capitolo 46: Data di lezione

Sintassi

- `Date object = new Date();`
- `Date object = new Date(long date);`

Parametri

Parametro	Spiegazione
Nessun parametro	Crea un nuovo oggetto Date usando il tempo di allocazione (al millisecondo più vicino)
lunga data	Crea un nuovo oggetto Date con il tempo impostato sul numero di millisecondi da "the epoch" (1 gennaio 1970, 00:00:00 GMT)

Osservazioni

Rappresentazione

Internamente, un oggetto Data Java è rappresentato come un lungo; è il numero di millisecondi da un momento specifico (indicato come l' *epoca*). La classe Data originale di Java aveva metodi per gestire i fusi orari, ecc., Ma questi erano deprecati a favore della nuova classe Calendar.

Quindi se tutto ciò che si vuole fare nel proprio codice è rappresentato da un momento specifico, è possibile creare una classe Date e memorizzarla, ecc. Se si desidera stampare una versione leggibile da quella data, tuttavia, si crea una classe Calendar e usa la sua formattazione per produrre ore, minuti, secondi, giorni, fusi orari, ecc. Ricorda che un millisecondo specifico viene visualizzato come ore diverse in diversi fusi orari; normalmente si desidera visualizzarne uno nel fuso orario "locale", ma i metodi di formattazione devono tenere conto che è possibile visualizzarlo per altri.

Sappi anche che gli orologi usati dalle JVM non hanno solitamente una precisione millisecondo; l'orologio potrebbe "spuntare" solo ogni 10 millisecondi, e quindi, se cronometrano le cose, non puoi fare affidamento sulla misurazione accurata di quel livello.

Importazione

```
import java.util.Date;
```

La classe `Date` può essere importata dal pacchetto `java.util` .

Attenzione

`Date` istanze di `Date` sono mutabili, quindi usarle può rendere difficile scrivere codice thread-safe o può accidentalmente fornire accesso in scrittura allo stato interno. Ad esempio, nella classe sottostante, il metodo `getDate()` consente al chiamante di modificare la data della transazione:

```
public final class Transaction {
  private final Date date;

  public Date getTransactionDate() {
    return date;
  }
}
```

La soluzione è restituire una copia del campo `date` o utilizzare le nuove API in `java.time` introdotte in Java 8.

La maggior parte dei metodi di costruzione nella classe `Date` è stata deprecata e non dovrebbe essere utilizzata. In quasi tutti i casi, è consigliabile utilizzare la classe `Calendar` per le operazioni sulla data.

Java 8

Java 8 introduce nuove API di data e ora nel pacchetto `java.time` , compresi LocalDate e LocalTime . Le classi nel pacchetto `java.time` forniscono un'API revisionata che è più facile da usare. Se stai scrivendo su Java 8, ti consigliamo vivamente di utilizzare questa nuova API. Vedi date e orari (java.time. *) .

Examples

Creazione di oggetti Date

```
Date date = new Date();
System.out.println(date); // Thu Feb 25 05:03:59 IST 2016
```

Qui questo oggetto `Date` contiene la data e l'ora corrente in cui questo oggetto è stato creato.

```
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` oggetti `Date` sono creati meglio attraverso un'istanza di `Calendar` poiché l'uso dei costruttori di dati è deprecato e scoraggiato. Per fare ciò abbiamo bisogno di ottenere un'istanza della classe `Calendar` dal metodo factory. Quindi possiamo impostare l'anno, il mese e il giorno del mese utilizzando i numeri o, in caso di mesi, le costanti fornite dalla classe Calendar per migliorare la leggibilità e ridurre gli errori.

```
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
```

Insieme alla data, possiamo anche passare il tempo nell'ordine di ore, minuti e secondi.

Confronto degli oggetti Data

Calendario, data e data locale

Java SE 8

prima, dopo, confronta e uguaglia i metodi

```
//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 e uguaglia i metodi

```
//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)));
```

Confronto delle date prima di Java 8

Prima di Java 8, le date potevano essere confrontate usando le classi java.util.Calendar e java.util.Date . La classe Date offre 4 metodi per confrontare le date:

- dopo (Data in cui)
- prima (Data in cui)
- compareTo (Date anotherDate)
- uguale (oggetto obj)

`after` , `before` , i metodi `compareTo` ed `equals` confrontano i valori restituiti dal metodo getTime () per ogni data.

`compareTo` metodo `compareTo` restituisce un numero intero positivo.

- Valore maggiore di 0: quando la data è successiva all'argomento data
- Valore maggiore di 0: quando la data è precedente all'argomento Date
- Il valore è uguale a 0: quando la data è uguale all'argomento della data

`equals` risultati `equals` possono essere sorprendenti, come mostrato nell'esempio, poiché i valori, come millisecondi, non vengono inizializzati con lo stesso valore se non esplicitamente indicati.

Da quando Java 8

Con Java 8 è disponibile java.time.LocalDate, un nuovo oggetto con cui lavorare con Date. `LocalDate` implementa ChronoLocalDate , la rappresentazione astratta di una data in cui la cronologia, o sistema di calendario, è inseribile.

Per avere la precisione temporale della data è necessario utilizzare l'oggetto java.time.LocalDateTime . `LocalDate` e `LocalDateTime` utilizzano lo stesso nome di metodi per il confronto.

Confrontare le date utilizzando un `LocalDate` è diverso dall'uso di `ChronoLocalDate` perché la cronologia o il sistema di calendario non vengono presi in considerazione nel primo.

Poiché la maggior parte delle applicazioni deve utilizzare `LocalDate` , `ChronoLocalDate` non è incluso negli esempi. Ulteriore lettura qui .

> La maggior parte delle applicazioni dovrebbe dichiarare firme, campi e variabili del metodo come LocalDate, non questa interfaccia [ChronoLocalDate].

`LocalDate` ha 5 metodi per confrontare le date:

- isAfter (ChronoLocalDate altro)

- isBefore (ChronoLocalDate altro)
- isEqual (ChronoLocalDate altro)
- compareTo (ChronoLocalDate altro)
- uguale (oggetto obj)

In caso di parametro `LocalDate` , `isAfter` , `isBefore` , `isEqual` , `equals` e `compareTo` ora usa questo metodo:

```
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` controllo del metodo se il riferimento al parametro è uguale alla data, mentre `isEqual` chiama direttamente `compareTo0` .

In caso di un'altra istanza di classe di `ChronoLocalDate` le date vengono confrontate utilizzando `Epoch Day` . Il conteggio Epoch Day è un semplice conteggio incrementale di giorni in cui il giorno 0 è 1970-01-01 (ISO).

Convertire la data in un determinato formato di stringa

`format()` della classe `SimpleDateFormat` consente di convertire un oggetto `Date` in un determinato oggetto `String` format utilizzando la *stringa di pattern* fornita.

```
Date today = new Date();

SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MMM-yy"); //pattern is specified here
System.out.println(dateFormat.format(today)); //25-Feb-16
```

I pattern possono essere applicati nuovamente usando `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: Qui `mm` (la lettera minuscola m) indica i minuti e `MM` (la lettera maiuscola M) indica il mese. Prestare particolare attenzione durante la formattazione degli anni: la lettera maiuscola "Y" (`Y`) indica la "settimana dell'anno" mentre la lettera minuscola "y" (`y`) indica l'anno.

Conversione di stringa in data

`parse()` dalla classe `SimpleDateFormat` aiuta a convertire un pattern `String` in un oggetto `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
```

Esistono 4 stili diversi per il formato di testo, `SHORT` , `MEDIUM` (questo è il valore predefinito), `LONG` e `FULL` , che dipendono tutti dalle impostazioni locali. Se non viene specificata alcuna locale, viene utilizzata la locale predefinita del sistema.

Stile	Locale.US	Locale.France
CORTO	6/30/09	30/06/09
MEDIO	30 giugno 2009	30 giugno 2009
LUNGO	30 giugno 2009	30 giugno 2009
PIENO	Martedì 30 giugno 2009	martedì 30 giugno 2009

Una data di uscita di base

Usando il seguente codice con la stringa di formato `yyyy/MM/dd hh:mm.ss` , riceveremo il seguente output

    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()));
```

Converti la rappresentazione di stringa formattata di data in oggetto Date

Questo metodo può essere utilizzato per convertire una rappresentazione di stringa formattata di una data in un oggetto `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;
    }
```

Creazione di una data specifica

Mentre la classe Date di Java ha diversi costruttori, noterai che molti sono deprecati. L'unico modo accettabile di creare direttamente un'istanza di Date consiste nell'utilizzare il costruttore vuoto o nel passare un lungo (numero di millisecondi dall'ora di base standard). Né sono a portata di mano se non stai cercando la data corrente o hai già un'altra istanza di Data in mano.

Per creare una nuova data, avrai bisogno di un'istanza di Calendar. Da lì puoi impostare l'istanza di Calendar sulla data di cui hai bisogno.

```
Calendar c = Calendar.getInstance();
```

Ciò restituisce una nuova istanza di Calendar impostata sull'ora corrente. Calendar ha molti metodi per modificare la data e l'ora o impostarlo a titolo definitivo. In questo caso, lo imposteremo su una data specifica.

```
c.set(1974, 6, 2, 8, 0, 0);
Date d = c.getTime();
```

Il metodo `getTime` restituisce l'istanza Date di cui abbiamo bisogno. Tieni presente che i metodi di impostazione del calendario impostano solo uno o più campi, ma non li impostano tutti. Cioè, se si imposta l'anno, gli altri campi rimangono invariati.

trappola

In molti casi, questo snippet di codice soddisfa il suo scopo, ma tieni presente che due parti importanti della data / ora non sono definite.

- i parametri `(1974, 6, 2, 8, 0, 0)` sono interpretati all'interno del fuso orario predefinito, definito da qualche altra parte,
- i millisecondi non sono impostati su zero, ma vengono riempiti dall'orologio di sistema nel momento in cui viene creata l'istanza di Calendar.

Oggetti Java 8 LocalDate e LocalDateTime

Gli oggetti Date e LocalDate **non possono** essere convertiti *esattamente* tra loro in quanto un oggetto Date rappresenta sia un giorno che un'ora specifici, mentre un oggetto LocalDate non contiene informazioni sul fuso orario o sul fuso orario. Tuttavia, può essere utile convertire tra i due se ti interessano solo le informazioni sulla data effettiva e non le informazioni sull'ora.

Crea un LocalDate

```
// 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());
```

LocalDate to Date e 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 e 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());
```

Fusi orari e java.util.Date

Un oggetto `java.util.Date` *non* ha un concetto di fuso orario.

- Non c'è modo di **impostare** un fuso orario per una data
- Non c'è modo di **cambiare** il fuso orario di un oggetto Date
- Un oggetto Date creato con il `new Date()` costruttore predefinito `new Date()` verrà inizializzato con l'ora corrente nel fuso orario predefinito del sistema

Tuttavia, è possibile visualizzare la data rappresentata dal punto nel tempo descritto dall'oggetto Date in un fuso orario diverso, ad es. `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));
```

Produzione:

```
Central European Time
2016-07-21 22:50:56
2016-07-21 21:50:56
```

Convertire java.util.Date in java.sql.Date

`java.util.Date` conversione da `java.util.Date` a `java.sql.Date` è solitamente necessaria quando un oggetto Date deve essere scritto in un database.

`java.sql.Date` è un wrapper attorno al valore millisecondo ed è usato da `JDBC` per identificare un tipo `SQL DATE`

Nell'esempio seguente, usiamo il costruttore `java.util.Date()` , che crea un oggetto Date e lo inizializza per rappresentare il tempo al millisecondo più vicino. Questa data viene utilizzata nel metodo `convert(java.util.Date utilDate)` per restituire un oggetto `java.sql.Date`

Esempio

```
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;
    }

}
```

Produzione

```
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` ha informazioni sia sulla data che sull'ora, mentre `java.sql.Date` ha solo informazioni sulla data

Ora locale

Per utilizzare solo la parte temporale di una data, utilizzare LocalTime. Puoi creare un'istanza di un oggetto LocalTime in un paio di modi

1. `LocalTime time = LocalTime.now();`
2. `time = LocalTime.MIDNIGHT;`
3. `time = LocalTime.NOON;`
4. `time = LocalTime.of(12, 12, 45);`

`LocalTime` ha anche un built-in metodo toString che visualizza il formato molto bene.

```
System.out.println(time);
```

puoi anche ottenere, aggiungere e sottrarre ore, minuti, secondi e nanosecondi dall'oggetto LocalTime, ad es

```
time.plusMinutes(1);
time.getMinutes();
time.minusMinutes(1);
```

Puoi trasformarlo in un oggetto Date con il seguente codice:

```
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);
```

questa classe funziona molto bene all'interno di una classe di timer per simulare una sveglia.

Leggi Data di lezione online: https://riptutorial.com/it/java/topic/164/data-di-lezione

Capitolo 47: Date e ora (java.time. *)

Examples

Manipolazioni di date semplici

Ottieni la data corrente.

```
LocalDate.now()
```

Ricevi la data di ieri.

```
LocalDate y = LocalDate.now().minusDays(1);
```

Ricevi la data di domani

```
LocalDate t = LocalDate.now().plusDays(1);
```

Ottieni una data specifica.

```
LocalDate t = LocalDate.of(1974, 6, 2, 8, 30, 0, 0);
```

Oltre ai metodi `plus` e `minus` , esiste un insieme di metodi "con" che possono essere utilizzati per impostare un campo particolare in un'istanza di `LocalDate` .

```
LocalDate.now().withMonth(6);
```

L'esempio sopra restituisce una nuova istanza con il mese impostato su June (questo differisce da `java.util.Date` dove `setMonth` stato indicizzato a 0 facendo il 5 giugno).

Poiché le manipolazioni di LocalDate restituiscono istanze LocalDate immutabili, questi metodi possono anche essere concatenati.

```
LocalDate ld = LocalDate.now().plusDays(1).plusYears(1);
```

Questo ci darebbe la data di domani tra un anno.

Data e ora

Data e ora senza informazioni sul fuso orario

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 27, 8, 0);
LocalDateTime now = LocalDateTime.now();
LocalDateTime parsed = LocalDateTime.parse("2016-07-27T07:00:00");
```

Data e ora con informazioni sul fuso orario

```
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]");
```

Data e ora con informazioni sull'offset (ovvero nessuna modifica dell'ora legale presa in considerazione)

```
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");
```

Operazioni su date e orari

```
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]"))
```

Immediato

Rappresenta un istante nel tempo. Può essere pensato come un wrapper attorno a un timestamp 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
```

Utilizzo di varie classi di API Date Time

L'esempio seguente ha anche una spiegazione necessaria per comprendere l'esempio al suo interno.

```
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);

}
}
```

Formattazione della data e ora

Prima di Java 8, esistevano classi `DateFormat` e `SimpleDateFormat` nel pacchetto `java.text` e questo codice legacy continuerà ad essere usato per qualche tempo.

Ma, Java 8 offre un approccio moderno alla gestione della formattazione e dell'analisi.

In fase di formattazione e analisi, si passa prima un oggetto `String` a `DateTimeFormatter` e, a sua volta, viene utilizzato per la formattazione o l'analisi.

```
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 avviso importante, invece di usare i pattern personalizzati, è una buona pratica usare i formattatori predefiniti. Il tuo codice appare più chiaro e l'utilizzo di ISO8061 ti aiuterà sicuramente a lungo termine.

Calcola la differenza tra 2 date locali

Usa `LocalDate` e `ChronoUnit` :

```
LocalDate d1 = LocalDate.of(2017, 5, 1);
LocalDate d2 = LocalDate.of(2017, 5, 18);
```

ora, poiché il metodo `between` l'enumeratore `ChronoUnit` prende 2 `Temporal` come parametri in modo da poter passare senza problemi le istanze di `LocalDate`

```
long days = ChronoUnit.DAYS.between(d1, d2);
System.out.println( days );
```

Leggi Date e ora (java.time. *) online: https://riptutorial.com/it/java/topic/4813/date-e-ora--java-time----

Capitolo 48: Disassemblare e decompilare

Sintassi

- javap [opzioni] <classi>

Parametri

Nome	Descrizione
`<classes>`	Elenco delle classi da smontare. Può essere in formato `package1.package2.Classname` *o in* `package1/package2/Classname` . **Non** includere il `.class` estensione.
`-help` , `--help` , `-?`	Stampa questo messaggio d'uso
`-version`	Informazioni sulla versione
`-v` , `-verbose`	Stampa informazioni aggiuntive
`-l`	Stampa il numero di riga e le tabelle delle variabili locali
`-public`	Mostra solo classi pubbliche e membri
`-protected`	Mostra classi e membri protetti / pubblici
`-package`	Mostra pacchetto / protetto / classi pubbliche e membri (predefinito)
`-p` , `-private`	Mostra tutte le classi e i membri
`-c`	Smonta il codice
`-s`	Stampa le firme dei tipi interni
`-sysinfo`	Mostra informazioni di sistema (percorso, dimensioni, data, hash MD5) della classe in elaborazione
`-constants`	Mostra le costanti finali
`-classpath <path>`	Specificare dove trovare i file della classe utente
`-cp <path>`	Specificare dove trovare i file della classe utente
`-bootclasspath <path>`	Sostituire la posizione dei file di classe bootstrap

Examples

Visualizzazione di bytecode con javap

Se si desidera visualizzare il codice byte generato per un programma Java, è possibile utilizzare il comando `javap` fornito per visualizzarlo.

Supponendo che abbiamo il seguente file sorgente 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");
    }
}
```

Dopo aver compilato il file sorgente, l'utilizzo più semplice è:

```
cd <directory containing classes> (e.g. target/classes)
javap com/stackoverflow/documentation/SpringExample
```

Che produce l'output

```
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();
}
```

Elenca tutti i metodi non privati nella classe, ma ciò non è particolarmente utile per la maggior parte degli scopi. Il seguente comando è molto più utile:

```
javap -p -c -s -constants -l -v com/stackoverflow/documentation/HelloWorldService
```

Che produce l'output:

```
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()
```

Leggi Disassemblare e decompilare online: https://riptutorial.com/it/java/topic/2318/disassemblare-e-decompilare

Capitolo 49: Divisione di una stringa in parti di lunghezza fissa

Osservazioni

L'obiettivo qui è non perdere il contenuto, quindi la regex non deve consumare (corrispondere) a qualsiasi input. Piuttosto, deve corrispondere *tra* l'ultimo carattere dell'ingresso target precedente e il primo carattere del prossimo input target. ad es. per le sottostringhe di 8 caratteri, abbiamo bisogno di interrompere l'input up (cioè la corrispondenza) nei punti indicati di seguito:

```
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
             ^               ^               ^
```

Ignora gli spazi nell'input che dovevano mostrare *tra le* posizioni dei personaggi.

Examples

Rompere una stringa in sottostringhe di una lunghezza nota

Il trucco è usare un look-behind con regex `\G` , che significa "fine della precedente corrispondenza":

```
String[] parts = str.split("(?<=\\G.{8})");
```

La regex corrisponde a 8 caratteri dopo la fine dell'ultima partita. Dato che in questo caso la corrispondenza è a larghezza zero, potremmo semplicemente dire "8 caratteri dopo l'ultima partita".

Convenientemente, `\G` viene inizializzato per iniziare l'input, quindi funziona anche per la prima parte dell'ingresso.

Rompere una stringa in sottostringhe di lunghezza variabile

Come l'esempio di lunghezza noto, ma inserisci la lunghezza in regex:

```
int length = 5;
String[] parts = str.split("(?<=\\G.{" + length + "})");
```

Leggi Divisione di una stringa in parti di lunghezza fissa online:
https://riptutorial.com/it/java/topic/5613/divisione-di-una-stringa-in-parti-di-lunghezza-fissa

Capitolo 50: Documentazione del codice Java

introduzione

La documentazione per il codice java viene spesso generata usando javadoc . Javadoc è stato creato da Sun Microsystems allo scopo di generare documentazione API in formato HTML dal codice sorgente java. L'utilizzo del formato HTML offre la comodità di essere in grado di collegare insieme documenti correlati.

Sintassi

- / ** - avvio di JavaDoc su una classe, un campo, un metodo o un pacchetto
- @author // Per nominare l'autore della classe, interfaccia o enum. È richiesto.
- @version // La versione di quella classe, interfaccia o enum. È richiesto. È possibile utilizzare macro come% I% o% G% affinché il software di controllo del codice sorgente compili il pagamento.
- @param // Per mostrare gli argomenti (parametri) di un metodo o di un costruttore. Specifica un tag @param per ogni parametro.
- @return // Per mostrare i tipi di ritorno per i metodi non void.
- @exception // Mostra quali eccezioni possono essere generate dal metodo o dal costruttore. Le eccezioni che **DEVONO** essere catturate dovrebbero essere elencate qui. Se lo desideri, puoi anche includere quelli che non devono essere catturati, come ArrayIndexOutOfBoundsException. Specifica una @ eccezione per ogni eccezione che può essere generata.
- @throws // Uguale a @exception.
- @see // Collegamenti a un metodo, campo, classe o pacchetto. Utilizzare sotto forma di package.Class # qualcosa.
- @since // Quando questo metodo, campo o classe è stato aggiunto. Ad esempio, JDK-8 per una classe come java.util.Optional <T> .
- @serial, @serialField, @serialData // Utilizzato per mostrare serialVersionUID.
- @deprecated // Per contrassegnare una classe, un metodo o un campo come deprecato. Ad esempio, uno sarebbe java.io.StringBufferInputStream . Vedi un elenco completo delle classi deprecate esistenti qui .
- {@link} // Simile a @see, ma può essere utilizzato con testo personalizzato: {@link #setDefaultCloseOperation (int closeOperation) vedi JFrame # setDefaultCloseOperation per maggiori informazioni}.
- {@linkplain} // Simile a {@link}, ma senza il carattere del codice.
- {@code} // per codice letterale, ad esempio tag HTML. Ad esempio: {@code <html> </ html>}. Tuttavia, questo utilizzerà un carattere a spaziatura fissa. Per ottenere lo stesso risultato senza il carattere monospaziale, utilizzare {@literal}.
- {@literal} // Uguale a {@code}, ma senza il carattere a spaziatura fissa.
- {@value} // Mostra il valore di un campo statico: il valore di JFrame # EXIT_ON_CLOSE è {@value}. In alternativa, puoi collegare a un determinato campo: utilizza il nome dell'app {@value AppConstants # APP_NAME}.

- {@docRoot} // La cartella radice del codice HTML JavaDoc relativa al file corrente. Esempio: Crediti .
- L'HTML è permesso: <code> "Ciao cookie" .substring (3) </ code>.
- * / - fine della dichiarazione JavaDoc

Osservazioni

Javadoc è uno strumento incluso con JDK che consente di convertire i commenti nel codice in una documentazione HTML. La specifica dell'API Java è stata generata utilizzando Javadoc. Lo stesso vale per gran parte della documentazione delle librerie di terze parti.

Examples

Documentazione di classe

Tutti i commenti Javadoc iniziano con un commento di blocco seguito da un asterisco (`/**`) e terminano quando fa il commento del blocco (`*/`). Facoltativamente, ogni riga può iniziare con spazi bianchi arbitrari e un singolo asterisco; questi vengono ignorati quando vengono generati i file di documentazione.

```
/**
 * 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 {
}
```

Gli stessi tag e formati usati per le `Classes` possono essere usati anche per `Enums` e `Interfaces` .

Documentazione del metodo

Tutti i commenti Javadoc iniziano con un commento di blocco seguito da un asterisco (`/**`) e terminano quando fa il commento del blocco (`*/`). Facoltativamente, ogni riga può iniziare con spazi bianchi arbitrari e un singolo asterisco; questi vengono ignorati quando vengono generati i file di documentazione.

```
/**
 * 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;
}
```

Documentazione sul campo

Tutti i commenti Javadoc iniziano con un commento di blocco seguito da un asterisco (`/**`) e terminano quando fa il commento del blocco (`*/`). Facoltativamente, ogni riga può iniziare con spazi bianchi arbitrari e un singolo asterisco; questi vengono ignorati quando vengono generati i file di documentazione.

```
/**
 * 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";
```

Documentazione del pacchetto

Java SE 5

È possibile creare documentazione a livello di pacchetto in Javadoc utilizzando un file chiamato `package-info.java` . Questo file deve essere formattato come di seguito. Spazi bianchi iniziali e asterischi facoltativi, tipicamente presenti in ogni riga per motivi di formattazione

```
/**
 * 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.
```

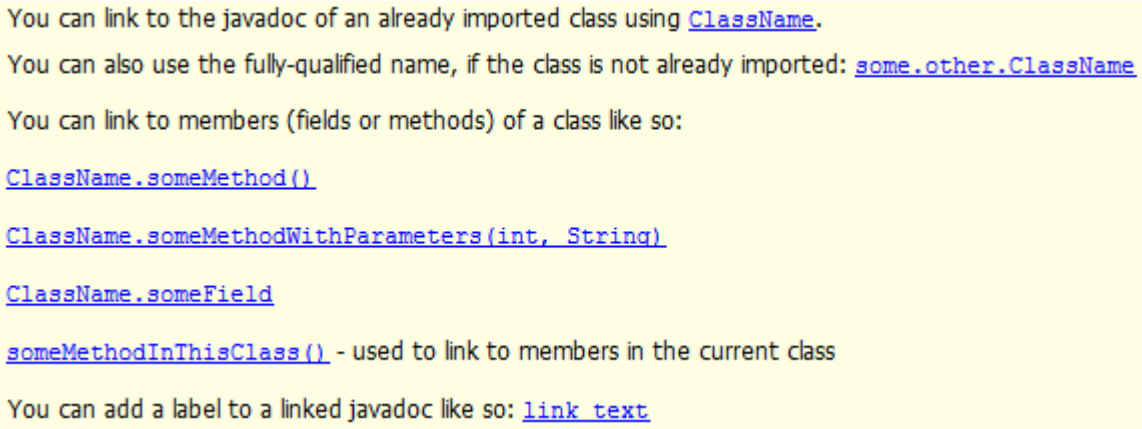
Nel caso precedente, devi inserire questo file `package-info.java` nella cartella del pacchetto Java `com.example.foo` .

link

Il collegamento ad altri Javadoc è fatto con il tag `@link` :

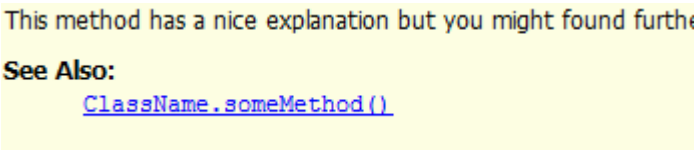
```
/**
 * You can link to the javadoc of an already imported class using {@link ClassName}.
 *
 * You can also use the fully-qualified name, if the class is not already imported:
 *  {@link some.other.ClassName}
```

```
 *
 * 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}
 */
```



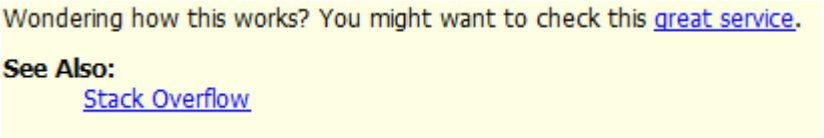
Con il tag `@see` puoi aggiungere elementi alla sezione *Vedi anche* . Come `@param` o `@return` il luogo in cui appaiono non è rilevante. Le specifiche dicono che dovresti scriverlo dopo `@return` .

```
/**
 * This method has a nice explanation but you might found further
 * information at the bottom.
 *
 * @see ClassName#someMethod()
 */
```



Se si desidera aggiungere **collegamenti a risorse esterne,** è sufficiente utilizzare il tag HTML `<a>` . Puoi usarlo in linea ovunque o all'interno di entrambi i tag `@link` e `@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>
 */
```



Costruire Javadocs dalla riga di comando

Molti IDE forniscono supporto per generare automaticamente HTML da Javadocs; alcuni strumenti di compilazione (Maven e Gradle , ad esempio) hanno anche plugin in grado di gestire la creazione HTML.

Tuttavia, questi strumenti non sono necessari per generare l'HTML Javadoc; questo può essere fatto usando lo strumento `javadoc` riga di comando.

L'uso più basilare dello strumento è:

```
javadoc JavaFile.java
```

Che genererà HTML dai commenti Javadoc in `JavaFile.java` .

Un uso più pratico dello strumento della riga di comando, che leggerà in modo ricorsivo tutti i file java in `[source-directory]` , creerà la documentazione per `[package.name]` e tutti i sotto-pacchetti, e posizionerà l'HTML generato nella `[docs-directory]` è:

```
javadoc -d [docs-directory] -subpackages -sourcepath [source-directory] [package.name]
```

Documentazione del codice inline

Oltre al codice di documentazione Javadoc può essere documentato in linea.

I commenti di una singola riga vengono avviati da `//` e possono essere posizionati dopo un'istruzione sulla stessa riga, ma non prima.

```
public void method() {

  //single line comment
  someMethodCall(); //single line comment after statement

}
```

I commenti su più righe sono definiti tra `/*` e `*/` . Possono estendersi su più righe e possono anche essere posizionati tra le dichiarazioni.

```
public void method(Object object) {

  /*
    multi
    line
    comment
  */
  object/*inner-line-comment*/.method();
}
```

I JavaDocs sono una forma speciale di commenti su più righe, a partire da `/**` .

Poiché troppi commenti in linea possono ridurre la leggibilità del codice, dovrebbero essere usati scarsamente nel caso in cui il codice non sia sufficientemente esplicativo o la decisione di progettazione non sia ovvia.

Un ulteriore caso d'uso per i commenti a riga singola è l'uso di TAG, che sono brevi parole chiave basate sulla convenzione. Alcuni ambienti di sviluppo riconoscono alcune convenzioni per tali commenti singoli. Esempi comuni sono

- `//TODO`
- `//FIXME`

O rilasciare riferimenti, cioè per Jira

- `//PRJ-1234`

Frammenti di codice all'interno della documentazione

Il modo canonico di scrivere il codice all'interno della documentazione è con il costrutto `{@code }` . Se disponi di un code code multiplo all'interno di `<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 volte potrebbe essere necessario inserire un codice complesso all'interno del commento javadoc. Il segno `@` è particolarmente problematico. L'uso del vecchio tag `<code>` insieme al `{@literal }` risolve il 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 { }
```

Leggi Documentazione del codice Java online:
https://riptutorial.com/it/java/topic/140/documentazione-del-codice-java

Capitolo 51: Eccezioni e gestione delle eccezioni

introduzione

Oggetti di tipo `Throwable` e i suoi sottotipi possono essere inviati allo stack con la parola chiave `throw` e catturati con `try…catch` statements.

Sintassi

- void someMethod () genera la dichiarazione del metodo SomeException {} //, forza l'intercettazione dei chiamanti del metodo se SomeException è un tipo di eccezione verificata
- provare {

```
someMethod(); //code that might throw an exception
```

  }
- catch (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

Cattura un'eccezione con try-catch

Un'eccezione può essere catturata e gestita usando la `try...catch` . (In effetti le dichiarazioni di `try` prendono altre forme, come descritto in altri esempi su `try...catch...finally` e `try-with-resources` .)

Prova a prendere con un blocco di cattura

La forma più semplice ha questo aspetto:

```
try {
    doSomething();
} catch (SomeException e) {
    handle(e);
}
// next statement
```

Il comportamento di un semplice `try...catch` è il seguente:

- Le istruzioni nel blocco `try` vengono eseguite.
- Se nessuna eccezione è generata dalle dichiarazioni nel blocco `try` , il controllo passa alla successiva dichiarazione dopo il `try...catch` .
- Se viene lanciata un'eccezione all'interno del blocco `try` .
    - L'oggetto eccezione viene verificato per verificare se si tratta di un'istanza di `SomeException` o di un sottotipo.
    - Se lo è, il blocco `catch` *attira* l'eccezione:
        - La variabile `e` è associata all'oggetto eccezione.
        - Il codice all'interno del blocco `catch` viene eseguito.
        - Se quel codice genera un'eccezione, l'eccezione appena generata viene propagata al posto di quella originale.
        - Altrimenti, il controllo passa alla dichiarazione successiva dopo il `try...catch` .
    - Se non lo è, l'eccezione originale continua a propagarsi.

Prova a catturare con più catture

Un `try...catch` può anche avere più blocchi `catch` . Per esempio:

```
try {
    doSomething();
} catch (SomeException e) {
    handleOneWay(e)
} catch (SomeOtherException e) {
    handleAnotherWay(e);
}
// next statement
```

Se ci sono più blocchi `catch` , vengono provati uno alla volta iniziando dal primo, finché non viene trovata una corrispondenza per l'eccezione. Il gestore corrispondente viene eseguito (come sopra) e quindi il controllo viene passato all'istruzione successiva dopo l'istruzione `try...catch` . I blocchi `catch` dopo quello che corrisponde vengono sempre saltati, *anche se il codice del gestore genera un'eccezione* .

La strategia di abbinamento "top down" ha conseguenze per i casi in cui le eccezioni nei blocchi `catch` non sono disgiunte. Per esempio:

```
try {
    throw new RuntimeException("test");
} catch (Exception e) {
```

```
    System.out.println("Exception");
} catch (RuntimeException e) {
    System.out.println("RuntimeException");
}
```

Questo snippet di codice genererà "Eccezione" anziché "RuntimeException". Poiché `RuntimeException` è un sottotipo di `Exception` , il primo (più generale) `catch` verrà confrontato. La seconda `catch` (più specifica) non verrà mai eseguita.

La lezione da imparare da questo è che i blocchi di `catch` più specifici (in termini di tipi di eccezione) dovrebbero apparire per primi, e quelli più generali dovrebbero essere gli ultimi. (Alcuni compilatori Java ti avviseranno se una `catch` non può mai essere eseguita, ma questo non è un errore di compilazione.)

Blocchi di cattura multi-eccezione

Java SE 7

A partire da Java SE 7, un singolo blocco `catch` può gestire un elenco di eccezioni non correlate. Il tipo di eccezione è elencato, separato da un simbolo di barra verticale (`|`). Per esempio:

```
try {
    doSomething();
} catch (SomeException | SomeOtherException e) {
    handleSomeException(e);
}
```

Il comportamento di un'eccezione a più eccezioni è un'estensione semplice per il caso a eccezione singola. Il `catch` corrisponde se l'eccezione generata soddisfa (almeno) una delle eccezioni elencate.

C'è qualche sottigliezza aggiuntiva nelle specifiche. Il tipo di `e` è *un'unione* sintetica dei tipi di eccezione nell'elenco. Quando viene utilizzato il valore di `e` , il suo tipo statico è il supertipo meno comune dell'unione di tipo. Tuttavia, se `e` viene ripubblicato all'interno del blocco `catch` , i tipi di eccezioni lanciati sono i tipi nell'unione. Per esempio:

```
public void method() throws IOException, SQLException
    try {
        doSomething();
    } catch (IOException | SQLException e) {
        report(e);
        throw e;
    }
```

In quanto sopra, `IOException` e `SQLException` sono eccezioni controllate il cui supertipo minimo comune è `Exception` . Ciò significa che il metodo del `report` deve corrispondere al `report(Exception)` . Tuttavia, il compilatore sa che il `throw` può generare solo `IOException` o `SQLException` . Pertanto, il `method` può essere dichiarato come `throws IOException, SQLException` piuttosto che `throws Exception` . (Che è una buona cosa: vedi Pitfall - Throwable Throwable, Exception, Error o RuntimeException .)

Lanciare un'eccezione

Il seguente esempio mostra le basi del lancio di un'eccezione:

```
public void checkNumber(int number) throws IllegalArgumentException {
    if (number < 0) {
        throw new IllegalArgumentException("Number must be positive: " + number);
    }
}
```

L'eccezione è lanciata sulla terza linea. Questa affermazione può essere suddivisa in due parti:

- `new IllegalArgumentException(...)` sta creando un'istanza della classe `IllegalArgumentException` , con un messaggio che descrive l'errore segnalato da tale eccezione.
- `throw ...` lancia quindi l'oggetto eccezione.

Quando viene generata l'eccezione, le istruzioni di *chiusura vengono terminate in modo anomalo* finché non viene *gestita* l'eccezione. Questo è descritto in altri esempi.

È buona norma creare e lanciare l'oggetto eccezione in una singola istruzione, come mostrato sopra. È inoltre buona norma includere un messaggio di errore significativo nell'eccezione per aiutare il programmatore a comprendere la causa del problema. Tuttavia, questo non è necessariamente il messaggio che dovresti mostrare all'utente finale. (Per cominciare, Java non ha supporto diretto per l'internazionalizzazione dei messaggi di eccezione.)

Ci sono un paio di altri punti da fare:

- Abbiamo dichiarato il `checkNumber` come `throws IllegalArgumentException` . Questo non era strettamente necessario, poiché `IllegalArgumentException` è un'eccezione controllata; vedere La gerarchia delle eccezioni Java - Eccezioni non selezionate e controllate . Tuttavia, è buona pratica farlo e includere anche le eccezioni generate dai commenti javadoc di un metodo.
- Codice immediatamente dopo che una dichiarazione di `throw` è *irraggiungibile* . Quindi se abbiamo scritto questo:

  ```
  throw new IllegalArgumentException("it is bad");
  return;
  ```

  il compilatore avrebbe segnalato un errore di compilazione per la dichiarazione di `return` .

Eccezione concatenata

Molte eccezioni standard hanno un costruttore con un secondo argomento di `cause` oltre all'argomento di `message` convenzionale. La `cause` consente di concatenare eccezioni. Ecco un esempio.

Per prima cosa definiamo un'eccezione non controllata che la nostra applicazione sta per gettare quando incontra un errore non recuperabile. Si noti che abbiamo incluso un costruttore che accetta un argomento di `cause` .

```
public class AppErrorException extends RuntimeException {
    public AppErrorException() {
        super();
    }

    public AppErrorException(String message) {
        super(message);
    }

    public AppErrorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Successivamente, ecco un codice che illustra il concatenamento delle eccezioni.

```
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);
    }
}
```

Il `throw` all'interno del blocco `try` rileva un problema e lo segnala tramite un'eccezione con un semplice messaggio. Al contrario, il `throw` all'interno del blocco `catch` sta gestendo `IOException` avvolgendolo in una nuova (controllata) eccezione. Tuttavia, non sta gettando via l'eccezione originale. Passando la `IOException` come `cause` , la registriamo in modo che possa essere stampata nello stacktrace, come spiegato in Creazione e lettura di stacktraces .

Eccezioni personalizzate

Nella maggior parte dei casi, è più semplice da un punto di vista della progettazione del codice utilizzare le classi di `Exception` generiche esistenti quando si generano eccezioni. Questo è particolarmente vero se hai solo bisogno dell'eccezione per portare un semplice messaggio di errore. In tal caso, RuntimeException di solito è preferito, poiché non è un'eccezione controllata. Esistono altre classi di eccezioni per classi comuni di errori:

- UnsupportedOperationException - una determinata operazione non è supportata
- IllegalArgumentException : un valore di parametro non valido è stato passato a un metodo
- IllegalStateException : la tua API ha raggiunto internamente una condizione che non dovrebbe mai accadere o che si verifica a seguito dell'utilizzo dell'API in modo non valido

I casi in cui **si** vuole utilizzare una classe eccezione personalizzata sono i seguenti:

- Stai scrivendo un'API o una libreria per l'utilizzo da parte di altri e desideri consentire agli utenti della tua API di essere in grado di catturare e gestire in modo specifico le eccezioni dalla tua API e *di distinguere tali eccezioni da altre eccezioni più generiche* .
- Stai lanciando eccezioni per un **tipo specifico di errore** in una parte del tuo programma, che vuoi catturare e gestire in un'altra parte del tuo programma, e vuoi essere in grado di distinguere questi errori da altri errori più generici.

È possibile creare eccezioni personalizzate estendendo `RuntimeException` per un'eccezione non controllata o verificata l'eccezione estendendo qualsiasi `Exception` *che non sia anche sottoclasse di RuntimeException* , in quanto:

> Le sottoclassi di Eccezione che non sono anche sottoclassi di RuntimeException sono eccezioni controllate

```
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;
    }
}
```

Quelli possono essere usati solo come eccezioni predefinite:

```
void validateString(String value){
    if (value.length() > 30){
        throw new StringTooLongException(value, 30);
    }
}
```

E i campi possono essere utilizzati dove viene rilevata e gestita l'eccezione:

```
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 );
    }
}
```

Tieni presente che, secondo la documentazione Java di Oracle :

> [...] Se ci si può ragionevolmente aspettare che un client recuperi da un'eccezione, rendi un'eccezione controllata. Se un client non può eseguire operazioni di ripristino dall'eccezione, renderlo un'eccezione non controllata.

Di Più:

- Perché RuntimeException non richiede una gestione delle eccezioni esplicita?

La dichiarazione try-with-resources

Java SE 7

Come illustra l'esempio di dichiarazione try-catch-final , il cleanup delle risorse che utilizza una clausola `finally` richiede una quantità significativa di codice "boiler-plate" per implementare correttamente le edge case. Java 7 fornisce un modo molto più semplice per affrontare questo problema nella forma *dell'istruzione try-with-resources* .

Cos'è una risorsa?

Java 7 ha introdotto l'interfaccia `java.lang.AutoCloseable` per consentire la gestione delle classi utilizzando l'istruzione *try-with-resources* . Le istanze di classi che implementano `AutoCloseable` sono indicate come *risorse* . Questi in genere devono essere smaltiti in modo tempestivo piuttosto che affidarsi al garbage collector per smaltirli.

L'interfaccia `AutoCloseable` definisce un singolo metodo:

```
public void close() throws Exception
```

Un metodo `close()` dovrebbe disporre della risorsa in modo appropriato. La specifica afferma che dovrebbe essere sicuro chiamare il metodo su una risorsa che è già stata eliminata. Inoltre, le classi che implementano l' `Autocloseable` sono *fortemente incoraggiate* a dichiarare il metodo `close()` per generare un'eccezione più specifica di `Exception` , o nessuna eccezione.

Una vasta gamma di classi e interfacce Java standard implementano `AutoCloseable` . Questi includono:

- `InputStream` , `OutputStream` e le loro sottoclassi
- `Reader` , `Writer` e le loro sottoclassi
- `Socket` e `ServerSocket` e relative sottoclassi
- `Channel` e le sue sottoclassi, e
- il JDBC interfaccia `Connection` , `Statement` e `ResultSet` e le loro sottoclassi.

Anche le classi di applicazioni e di terze parti possono farlo.

La dichiarazione base di prova con la risorsa

La sintassi di un *try-with-resources* si basa su forme classiche *try-catch* , *try-finally* e *try-catch-finally* . Ecco un esempio di una forma "base"; cioè la forma senza un `catch` o, `finally` .

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}
```

Le risorse da gestire sono dichiarate come variabili nella `(...)` sezione dopo la clausola `try` . Nell'esempio sopra, dichiariamo un `stream` variabili di risorsa e lo inizializziamo su `PrintStream` appena creato.

Una volta che le variabili della risorsa sono state inizializzate, viene eseguito il blocco `try` . Al termine, `stream.close()` verrà chiamato automaticamente per garantire che la risorsa non perda. Si noti che la chiamata `close()` avviene indipendentemente dal completamento del blocco.

Le dichiarazioni avanzate di try-with-resource

L'istruzione *try-with-resources* può essere migliorata con `catch` blocchi `catch` e `finally` , come con la sintassi pre-Java 7 *try-catch-finally* . Il seguente frammento di codice aggiunge un blocco `catch` al precedente per gestire l' `PrintStream` `FileNotFoundException` che può essere `PrintStream` costruttore `PrintStream` :

```
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");
}
```

Se l'inizializzazione della risorsa o il blocco try genera l'eccezione, verrà eseguito il blocco `catch` . Il blocco `finally` verrà sempre eseguito, come in una dichiarazione *try-catch-finally* convenzionale.

Ci sono un paio di cose da notare però:

- La variabile di risorsa è *fuori ambito* nel `catch` e `finally` blocchi.
- La pulizia delle risorse avverrà prima che l'istruzione tenti di far corrispondere il blocco `catch` .
- Se la pulizia automatica delle risorse ha generato un'eccezione, *potrebbe* essere catturata in uno dei blocchi `catch` .

Gestire più risorse

I frammenti di codice sopra mostrano una singola risorsa che viene gestita. In effetti, *try-with-resources* può gestire più risorse in un'unica istruzione. Per esempio:

```
try (InputStream is = new FileInputStream(file1);
     OutputStream os = new FileOutputStream(file2)) {
    // Copy 'is' to 'os'
}
```

Questo si comporta come ti aspetteresti. Sia `is` che `os` vengono chiusi automaticamente alla fine del blocco `try` . Ci sono un paio di punti da notare:

- Le inizializzazioni si verificano nell'ordine di codice e gli inizializzatori di variabili di risorse successive possono utilizzare i valori di quelli precedenti.

- Tutte le variabili di risorsa inizializzate correttamente verranno eliminate.
- Le variabili delle risorse vengono pulite in *ordine inverso* rispetto alle loro dichiarazioni.

Pertanto, nell'esempio di cui sopra, `is` è inizializzato prima `os` e ripulito dopo di essa, e `is` verrà pulito se c'è un'eccezione durante l'inizializzazione `os` .

Equivalenza di try-with-resource e try-catch-finally classico

La specifica del linguaggio Java specifica il comportamento delle forme *try-with-resource* in termini della classica dichiarazione *try-catch-finally* . (Si prega di fare riferimento al JLS per tutti i dettagli.)

Ad esempio, questa base di *prova con risorsa* :

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}
```

è definito come equivalente a questo *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);
        }
    }
}
```

(Il JLS specifica che le variabili `t` e `primaryException` effettive saranno invisibili al normale codice Java.)

La forma migliorata di *try-with-resources* è specificata come un'equivalenza con il modulo base. Per esempio:

```
try (PrintStream stream = new PrintStream(fileName)) {
    stream.println("Hello world!");
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}
```

è 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");
}
```

Creazione e lettura di stacktraces

Quando viene creato un oggetto eccezione (cioè quando lo si è `new`), il costruttore `Throwable` acquisisce informazioni sul contesto in cui è stata creata l'eccezione. In seguito, queste informazioni possono essere visualizzate sotto forma di stacktrace, che può essere utilizzato per aiutare a diagnosticare il problema che ha causato l'eccezione in primo luogo.

Stampa di uno stacktrace

Stampare uno stacktrace è semplicemente una questione di chiamare il metodo `printStackTrace()` . Per esempio:

```
try {
    int a = 0;
    int b = 0;
    int c = a / b;
} catch (ArithmeticException ex) {
    // This prints the stacktrace to standard output
    ex.printStackTrace();
}
```

Il metodo `printStackTrace()` senza argomenti verrà stampato sull'output standard dell'applicazione; cioè l'attuale `System.out` . Esistono anche `printStackTrace(PrintStream)` e `printStackTrace(PrintWriter)` che vengono stampati su uno `Stream` o su un `Writer` specificato.

Gli appunti:

1. Lo stacktrace non include i dettagli dell'eccezione stessa. Puoi usare il metodo `toString()`

per ottenere quei dettagli; per esempio

```
// Print exception and stacktrace
System.out.println(ex);
ex.printStackTrace();
```

2. La stampa Stacktrace dovrebbe essere usata con parsimonia; see Pitfall - Stacktraces eccessivi o inappropriati . È spesso preferibile utilizzare un framework di registrazione e passare l'oggetto di eccezione da registrare.

Capire uno stacktrace

Considera il seguente semplice programma composto da due classi in due file. (Abbiamo mostrato i nomi dei file e i numeri di riga aggiunti a scopo illustrativo.)

```
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      }
```

Quando questi file sono compilati ed eseguiti, otterremo il seguente output.

```
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)
```

Leggiamo questa riga alla volta per capire cosa ci sta dicendo.

La riga n. 1 ci dice che il thread chiamato "main" è terminato a causa di un'eccezione non rilevata. Il nome completo dell'eccezione è `java.lang.ArithmeticException` e il messaggio di eccezione è "/ per zero".

Se cerchiamo i javadoc per questa eccezione, si dice:

> Gettato quando si è verificata una condizione aritmetica eccezionale. Ad esempio, un intero "divide per zero" genera un'istanza di questa classe.

In effetti, il messaggio "/ per zero" indica chiaramente che la causa dell'eccezione è che alcuni codici hanno tentato di dividere qualcosa per zero. Ma cosa?

Le restanti 3 linee sono la traccia dello stack. Ogni linea rappresenta una chiamata al metodo (o al costruttore) nello stack delle chiamate e ognuna ci dice tre cose:

- il nome della classe e del metodo che è stato eseguito,
- il nome file del codice sorgente,
- il numero di riga del codice sorgente dell'istruzione che si stava eseguendo

Queste linee di uno stacktrace sono elencate con il frame per la chiamata corrente in alto. Il frame superiore nell'esempio sopra riportato è nel metodo `Test.bar` e alla riga 9 del file Test.java. Questa è la seguente riga:

```
    return a / b;
```

Se guardiamo prima un paio di righe nel file dove `b` è inizializzato, è chiaro che `b` avrà il valore zero. Possiamo dire senza alcun dubbio che questa è la causa dell'eccezione.

Se dovessimo andare oltre, possiamo vedere dallo stacktrace che `bar()` stato chiamato da `foo()` alla riga 3 di Test.java, e che `foo()` stato a sua volta chiamato da `Main.main()` .

Nota: i nomi di classi e metodi nei frame di stack sono i nomi interni per le classi e i metodi. Dovrai riconoscere i seguenti casi insoliti:

- Una classe nidificata o interiore avrà l'aspetto di "OuterClass $ InnerClass".
- Una classe interna anonima avrà l'aspetto di "OuterClass $ 1", "OuterClass $ 2", eccetera.
- Quando viene eseguito il codice in un costruttore, inizializzatore campo istanza o un blocco inizializzatore istanza, il nome del metodo sarà "".
- Quando viene eseguito il codice di un inizializzatore di campo statico o di un blocco di inizializzazione statico, il nome del metodo sarà "".

(In alcune versioni di Java, il codice di formattazione dello stacktrace rileverà ed eliderà sequenze ripetute dello stackframe, come può accadere quando un'applicazione fallisce a causa della ricorsione eccessiva.)

Eccezione di concatenamento e stacker nidificati

Java SE 1.4

Il concatenamento di eccezioni si verifica quando un pezzo di codice cattura un'eccezione e quindi crea e ne genera uno nuovo, passando la prima eccezione come causa. Ecco un esempio:

```
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  }
```

Quando la classe sopra è compilata ed eseguita, otteniamo il seguente stacktrace:

```
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
```

Lo stacktrace inizia con il nome della classe, il metodo e lo stack di chiamate per l'eccezione che (in questo caso) ha causato l'arresto anomalo dell'applicazione. Questo è seguito da una riga "Causato da:" che segnala l'eccezione di `cause` . Vengono riportati il nome della classe e il messaggio, seguiti dai frame dello stack dell'eccezione causa. La traccia termina con un "... N more" che indica che gli ultimi N frame sono gli stessi dell'eccezione precedente.

"Caused by:" è incluso solo nell'output quando la `cause` dell'eccezione primaria non è `null`). Le eccezioni possono essere concatenate indefinitamente, e in tal caso lo stacktrace può avere più tracce "Causate da:".

Nota: il meccanismo di `cause` stato esposto solo nell'API `Throwable` in Java 1.4.0. Prima di ciò, il concatenamento delle eccezioni doveva essere implementato dall'applicazione utilizzando un campo di eccezioni personalizzato per rappresentare la causa e un metodo `printStackTrace` personalizzato.

Catturare uno stacktrace come una stringa

A volte, un'applicazione deve essere in grado di acquisire uno stacktrace come `String` Java, in modo che possa essere utilizzato per altri scopi. L'approccio generale per fare ciò è creare un `OutputStream` o un `Writer` temporaneo che scriva su un buffer in memoria e lo passi a `printStackTrace(...)` .

Le librerie Apache Commons e Guava forniscono metodi di utilità per acquisire uno stacktrace come stringa:

```
org.apache.commons.lang.exception.ExceptionUtils.getStackTrace(Throwable)

com.google.common.base.Throwables.getStackTraceAsString(Throwable)
```

Se non è possibile utilizzare librerie di terze parti nella propria base di codice, utilizzare il seguente metodo per eseguire l'attività:

```
   /**
     * 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();
    }
```

Si noti che se si intende analizzare lo stacktrace, è più semplice utilizzare `getStackTrace()` e `getCause()` piuttosto che tentare di analizzare uno stacktrace.

Gestione di InterruptedException

`InterruptedException` è una bestia che confonde - si presenta in metodi apparentemente innocui come `Thread.sleep()` , ma `Thread.sleep()` modo errato porta a un codice difficile da gestire che si comporta male negli ambienti concorrenti.

Nella sua forma più semplice, se viene rilevata un'interruzione di `InterruptedException` , significa che qualcuno, da qualche parte, ha chiamato `Thread.interrupt()` sul thread in cui è attualmente in esecuzione il codice. Potresti essere incline a dire "È il mio codice! Non lo interromperò mai! " e quindi fare qualcosa del genere:

```
// Bad. Don't do this.
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  // disregard
}
```

Ma questo è esattamente il modo sbagliato di gestire un evento "impossibile" che si verifica. Se sai che la tua applicazione non incontrerà mai un'interruzione di `InterruptedException` , dovresti trattare tale evento come una grave violazione delle ipotesi del tuo programma e uscire il prima possibile.

Il modo corretto di gestire un interrupt "impossibile" è come questo:

```
// When nothing will interrupt your code
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  Thread.currentThread().interrupt();
  throw new AssertionError(e);
}
```

Questo fa due cose; ripristina innanzitutto lo stato di interruzione del thread (come se l'

`InterruptedException` non fosse stato gettato in primo luogo), quindi lancia un `AssertionError` indica che gli invarianti di base della tua applicazione sono stati violati. Se si è certi che non si interromperà mai il thread, questo codice viene eseguito in questo modo, poiché il blocco `catch` non dovrebbe mai essere raggiunto.

L'uso della classe `Uninterruptibles` di Guava aiuta a semplificare questo schema; chiamando `Uninterruptibles.sleepUninterruptibly()` ignora lo stato interrotto di un thread fino a quando la durata del sonno è scaduta (a quel punto viene ripristinata per le chiamate successive da ispezionare e lanciare la propria `InterruptedException`). Se sai che non interromperesti mai questo codice, eviterai in modo sicuro di dover avvolgere le tue chiamate a riposo in un blocco try-catch.

Più spesso, tuttavia, non è possibile garantire che il thread non verrà mai interrotto. In particolare se stai scrivendo un codice che verrà eseguito da un `Executor` o da qualche altra gestione dei thread, è fondamentale che il tuo codice risponda prontamente agli interrupt, altrimenti l'applicazione si fermerà o si bloccherà.

In questi casi, la cosa migliore da fare è in genere consentire a `InterruptedException` di propagare lo stack delle chiamate, aggiungendo una `throws InterruptedException` di `throws InterruptedException` a ciascun metodo a turno. Questo può sembrare imbarazzante, ma in realtà è una proprietà desiderabile - le firme del tuo metodo ora indicano ai chiamanti che risponderanno prontamente alle interruzioni.

```
// Let the caller determine how to handle the interrupt if you're unsure
public void myLongRunningMethod() throws InterruptedException {
  ...
}
```

In casi limitati (ad es. Durante l'override di un metodo che non `throw` eccezioni controllate) è possibile ripristinare lo stato interrotto senza generare un'eccezione, aspettandosi che qualsiasi codice venga eseguito accanto a gestire l'interrupt. Questo ritarda la gestione dell'interruzione ma non la sopprime completamente.

```
// 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 gerarchia delle eccezioni Java - Eccezioni non selezionate e controllate

Tutte le eccezioni Java sono istanze di classi nella gerarchia di classi Exception. Questo può essere rappresentato come segue:

- `java.lang.Throwable` - Questa è la classe base per tutte le classi di eccezioni. I suoi metodi e costruttori implementano una gamma di funzionalità comuni a tutte le eccezioni.
    - `java.lang.Exception` - Questa è la superclasse di tutte le normali eccezioni.
        - varie classi di eccezioni standard e personalizzate.

        - `java.lang.RuntimeException` - Questa è la superclasse di tutte le eccezioni normali che sono *eccezioni non controllate* .
            - varie classi di eccezioni di runtime standard e personalizzate.
    - `java.lang.Error` - Questa è la superclasse di tutte le eccezioni di "errore fatale".

Gli appunti:

1. La distinzione tra le eccezioni *controllate* e *non controllate* è descritta di seguito.
2. La `Throwable` , `Exception` e `RuntimeException` deve essere considerata `abstract` ; vedi Pitfall - Throwable Throwable, Exception, Error o RuntimeException .
3. Le eccezioni di `Error` vengono generate dalla JVM in situazioni in cui non sarebbe sicuro o imprudente per un'applicazione tentare di ripristinare.
4. Non sarebbe saggio dichiarare sottotipi personalizzati di `Throwable` . Gli strumenti e le librerie Java possono assumere che `Error` ed `Exception` sono gli unici sottotipi diretti di `Throwable` e si comportano `Throwable` se tale presupposto non è corretto.

Controllato contro Eccezioni non selezionate

Una delle critiche al supporto delle eccezioni in alcuni linguaggi di programmazione è che è difficile sapere quali eccezioni potrebbero generare un determinato metodo o procedura. Dato che un'eccezione non gestita può causare l'arresto anomalo di un programma, ciò può rendere le eccezioni una fonte di fragilità.

Il linguaggio Java affronta questo problema con il meccanismo di eccezione verificato. Innanzitutto, Java classifica le eccezioni in due categorie:

- Le eccezioni controllate rappresentano in genere gli eventi previsti che un'applicazione dovrebbe essere in grado di gestire. Ad esempio, `IOException` e i suoi sottotipi rappresentano condizioni di errore che possono verificarsi nelle operazioni di I / O. Gli esempi includono, l'apertura di file non avviene perché un file o una directory non esiste, le letture e le scritture di rete non funzionano perché una connessione di rete è stata interrotta e così via.

- Le eccezioni non controllate tipicamente rappresentano eventi imprevisti che un'applicazione non è in grado di gestire. Questi sono in genere il risultato di un bug nell'applicazione.

(Di seguito, "gettato" si riferisce a qualsiasi eccezione lanciata esplicitamente (da un'istruzione `throw`), o implicitamente (in un dereferenziamento fallito, digitare cast e così via). Allo stesso modo, "propagated" si riferisce a un'eccezione che è stata lanciata in un chiamata nidificata e non catturata all'interno di quella chiamata. Il seguente codice di esempio illustrerà questo.)

La seconda parte del meccanismo di eccezione verificata è che esistono restrizioni sui metodi in cui può verificarsi un'eccezione verificata:

- Quando un'eccezione controllata viene lanciata o propagata in un metodo, *deve* essere rilevata dal metodo o elencata nella clausola di `throws` del metodo. (Il significato della clausola `throws` è descritto in questo esempio).
- Quando un'eccezione controllata viene lanciata o propagata in un blocco di inizializzazione, deve essere catturata dal blocco.
- Un'eccezione verificata non può essere propagata da una chiamata di metodo in

un'espressione di inizializzazione del campo. (Non c'è modo di cogliere tale eccezione).

In breve, un'eccezione controllata deve essere gestita o dichiarata.

Queste restrizioni non si applicano alle eccezioni non controllate. Ciò include tutti i casi in cui un'eccezione viene lanciata implicitamente, poiché tutti questi casi generano eccezioni non controllate.

Esempi di eccezioni controllati

Questi snippet di codice hanno lo scopo di illustrare le restrizioni delle eccezioni controllate. In ogni caso, mostriamo una versione del codice con un errore di compilazione e una seconda versione con l'errore corretto.

```
// 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.
}
```

Il primo esempio mostra come le eccezioni verificate esplicitamente generate possono essere dichiarate come "lanciate" se non devono essere gestite nel metodo.

```
// 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
    }
}
```

Il secondo esempio mostra come può essere gestita un'eccezione verificata propagata.

```
// 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());
    }
}
```

L'esempio finale mostra come gestire un'eccezione controllata in un inizializzatore di campo statico.

```
// 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;
    }
}
```

Si noti che in questo ultimo caso, abbiamo anche a che fare con i problemi che `is` non possono essere assegnati a più di una volta, ma anche deve essere assegnato a, anche nel caso di un'eccezione.

introduzione

Le eccezioni sono errori che si verificano quando un programma è in esecuzione. Considera il programma Java sotto il quale si dividono due numeri interi.

```
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);
    }
}
```

Ora compiliamo ed eseguiamo il codice precedente e vediamo l'output per una divisione tentata per zero:

```
Input two integers
7 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Division.main(Disivion.java:14)
```

La divisione per zero è un'operazione non valida che produce un valore che non può essere rappresentato come un numero intero. Java si occupa di questo *lanciando* ***un'eccezione*** . In questo caso, l'eccezione è un'istanza della classe *ArithmeticException* .

Nota: l'esempio sulla creazione e la lettura delle tracce dello stack spiega cosa significa l'output dopo i due numeri.

L'utilità di *un'eccezione* è il controllo del flusso che consente. Senza usare eccezioni, una soluzione tipica a questo problema potrebbe essere quella di verificare prima se `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);
    }
}
```

Questo stampa il messaggio `You cannot divide by zero.` alla console e chiude il programma in modo aggraziato quando l'utente tenta di dividere per zero. Un modo equivalente di affrontare questo problema tramite la *gestione delle eccezioni* sarebbe quello di sostituire il controllo del

flusso `if` con un blocco `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 blocco catch try viene eseguito come segue:

1. Inizia l'esecuzione del codice nel blocco `try` .
2. Se si verifica *un'eccezione* nel blocco try, interrompere immediatamente e verificare se questa eccezione è *catturata* dal blocco `catch` (in questo caso, quando Exception è un'istanza di `ArithmeticException`).
3. Se l'eccezione viene *catturata* , viene assegnata alla variabile `e` e il blocco `catch` viene eseguito.
4. Se il blocco `try` o `catch` è completato (ovvero non si verificano eccezioni non rilevate durante l'esecuzione del codice), continuare ad eseguire il codice sotto il blocco `try-catch` .

In genere, è consigliabile utilizzare la *gestione delle eccezioni* come parte del normale controllo del flusso di un'applicazione in cui il comportamento sarebbe altrimenti indefinito o inatteso. Ad esempio, invece di restituire `null` quando un metodo fallisce, solitamente è meglio *lanciare un'eccezione in* modo che l'applicazione che fa uso del metodo possa definire il proprio controllo di flusso per la situazione tramite la *gestione* delle *eccezioni* del tipo illustrato sopra. In un certo senso, questo aggira il problema di dover restituire un particolare *tipo* , poiché uno qualsiasi dei molteplici tipi di *eccezioni* può essere *lanciato* per indicare il problema specifico che si è verificato.

Per ulteriori consigli su come e in che modo non usare le eccezioni, fare riferimento a Insidie di Java - Utilizzo delle eccezioni

Restituisci le dichiarazioni in try catch block

Sebbene sia una cattiva pratica, è possibile aggiungere più istruzioni di ritorno in un blocco di gestione delle eccezioni:

```
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;
    }
}
```

Questo metodo restituirà sempre 7 poiché il blocco finally associato al blocco try / catch viene eseguito prima che venga restituito qualcosa. Ora, come finalmente ha `return 7;` , questo valore sostituisce i valori di risposta try / catch.

Se il blocco catch restituisce un valore primitivo e tale valore primitivo viene successivamente modificato nel blocco finally, verrà restituito il valore restituito nel blocco catch e le modifiche dal blocco finally verranno ignorate.

L'esempio seguente stamperà "0", non "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;
        }
    }
}
```

Funzionalità avanzate di Eccezioni

Questo esempio illustra alcune funzioni avanzate e casi d'uso per Eccezioni.

Esaminando programmaticamente il callstack

Java SE 1.4

L'uso principale degli stacktraces delle eccezioni consiste nel fornire informazioni su un errore dell'applicazione e il relativo contesto in modo che il programmatore possa diagnosticare e risolvere il problema. A volte può essere usato per altre cose. Ad esempio, una classe `SecurityManager` potrebbe dover esaminare lo stack di chiamate per decidere se il codice che sta

effettuando una chiamata deve essere considerato attendibile.

È possibile utilizzare le eccezioni per esaminare lo stack di chiamate in modo programmatico come segue:

```
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());
```

Ci sono alcuni avvertimenti importanti su questo:

1. Le informazioni disponibili in `StackTraceElement` sono limitate. Non ci sono più informazioni disponibili di quelle visualizzate da `printStackTrace` . (I valori delle variabili locali nel frame non sono disponibili.)

2. I javadoc per `getStackTrace()` indicano che una JVM può lasciare i frame:

   > Alcune macchine virtuali possono, in alcune circostanze, omettere uno o più frame dello stack dall'analisi dello stack. Nel caso estremo, una macchina virtuale che non ha informazioni sulla traccia di stack relative a questo gettabile è autorizzata a restituire una matrice a lunghezza zero da questo metodo.

Ottimizzazione della costruzione di eccezioni

Come accennato altrove, la costruzione di un'eccezione è piuttosto costosa in quanto comporta l'acquisizione e la registrazione di informazioni su tutti i frame di stack sul thread corrente. A volte, sappiamo che quell'informazione non verrà mai usata per una determinata eccezione; ad esempio, lo stacktrace non verrà mai stampato. In tal caso, esiste un trucco di implementazione che è possibile utilizzare in un'eccezione personalizzata per impedire che le informazioni vengano acquisite.

Le informazioni sul frame dello stack necessarie per gli stacktraces, vengono acquisite quando i costruttori `Throwable` chiamano il metodo `Throwable.fillInStackTrace()` . Questo metodo è `public` , il che significa che una sottoclasse può sovrascriverla. Il trucco è scavalcare il metodo ereditato da `Throwable` con uno che non fa nulla; per esempio

```
public class MyException extends Exception {
    // constructors

    @Override
    public void fillInStackTrace() {
        // do nothing
    }
}
```

Il problema con questo approccio è che un'eccezione che sovrascrive `fillInStackTrace()` non può mai acquisire lo stacktrace ed è inutile negli scenari in cui ne hai bisogno.

Cancellazione o sostituzione dello stacktrace

Java SE 1.4

In alcune situazioni, lo stacktrace per un'eccezione creata nel modo normale contiene informazioni errate o informazioni che lo sviluppatore non desidera rivelare all'utente. Per questi scenari, è possibile utilizzare `Throwable.setStackTrace` per sostituire la matrice di oggetti `StackTraceElement` che contiene le informazioni.

Ad esempio, è possibile utilizzare quanto segue per eliminare le informazioni sullo stack di un'eccezione:

```
 exception.setStackTrace(new StackTraceElement[0]);
```

Eccezioni soppresse

Java SE 7

Java 7 ha introdotto il costrutto *try-with-resources* e il concetto associato di soppressione delle eccezioni. Considera il seguente frammento:

```
try (Writer w = new BufferedWriter(new FileWriter(someFilename))) {
    // do stuff
    int temp = 0 / 0;    // throws an ArithmeticException
}
```

Quando viene lanciata l'eccezione, la `try` chiamerà `close()` su `w` che svuoterà qualsiasi output bufferizzato e quindi chiuderà `FileWriter` . Ma cosa succede se viene generata una `IOException` mentre si scarica l'output?

Quello che succede è che ogni eccezione che viene lanciata mentre si ripulisce una risorsa viene *soppressa* . L'eccezione viene rilevata e aggiunta all'elenco delle eccezioni soppresse dell'eccezione primaria. Quindi il *try-with-resources* continuerà con la pulizia delle altre risorse. Infine, l'eccezione primaria verrà riconsiderata.

Un modello simile si verifica se un'eccezione viene generata durante l'inizializzazione della risorsa o se il blocco `try` completato normalmente. La prima eccezione generata diventa l'eccezione principale e quelli successivi derivanti dalla pulizia vengono eliminati.

Le eccezioni soppresse possono essere recuperate dall'oggetto eccezione principale chiamando `getSuppressedExceptions` .

Le dichiarazioni try-finally e try-catch-finally

L'istruzione `try...catch...finally` combina la gestione delle eccezioni con il codice clean-up. Il blocco `finally` contiene il codice che verrà eseguito in tutte le circostanze. Questo li rende adatti per la gestione delle risorse e altri tipi di pulizia.

Prova-finalmente

Ecco un esempio della forma più semplice (`try...finally`):

```
try {
    doSomething();
} finally {
    cleanUp();
}
```

Il comportamento del `try...finally` è il seguente:

- Il codice nel blocco `try` viene eseguito.
- Se non è stata lanciata alcuna eccezione nel blocco `try` :
    - Il codice nel blocco `finally` viene eseguito.
    - Se il blocco `finally` genera un'eccezione, quell'eccezione viene propagata.
    - Altrimenti, il controllo passa alla successiva dichiarazione dopo il `try...finally` .
- Se è stata generata un'eccezione nel blocco try:
    - Il codice nel blocco `finally` viene eseguito.
    - Se il blocco `finally` genera un'eccezione, quell'eccezione viene propagata.
    - Altrimenti, l'eccezione originale continua a propagarsi.

Il codice all'interno del blocco `finally` verrà sempre eseguito. (Le uniche eccezioni sono se viene chiamato `System.exit(int)` o se i panni della JVM.) Quindi un blocco `finally` è il codice posto corretto che deve sempre essere eseguito; ad esempio chiudere file e altre risorse o rilasciare blocchi.

try-catch-finally

Il nostro secondo esempio mostra come `catch` e, `finally` può essere utilizzato insieme. Illustra anche che pulire le risorse non è semplice.

```
// 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
        }
    }
}
```

L'insieme completo di (ipotetici) comportamenti di `try...catch...finally` in questo esempio sono

troppo complicati per descrivere qui. La versione semplice è che il codice nel blocco `finally` verrà sempre eseguito.

Guardando questo dal punto di vista della gestione delle risorse:

- Dichiariamo la "risorsa" (cioè la variabile del `reader`) prima del blocco `try` modo che sia in scope per il blocco `finally` .
- Inserendo il `new FileReader(...)` , il `catch` è in grado di gestire qualsiasi eccezione `IOError` generata quando si apre il file.
- Abbiamo bisogno di un `reader.close()` nel blocco `finally` perché ci sono alcuni percorsi di eccezione che non possiamo intercettare né nel blocco `try` né nel blocco `catch` .
- Tuttavia, poiché un'eccezione *potrebbe* essere stata lanciata prima che il `reader` fosse inizializzato, abbiamo anche bisogno di un test `null` esplicito.
- Infine, la chiamata a `reader.close()` potrebbe (ipoteticamente) generare un'eccezione. Non ci interessa, ma se non rileviamo l'eccezione alla fonte, avremmo bisogno di occuparci ulteriormente dello stack delle chiamate.

Java SE 7

Java 7 e versioni successive forniscono una sintassi alternativa try-with-resources che semplifica notevolmente la pulizia delle risorse.

La clausola 'getta' in una dichiarazione di metodo

Il meccanismo delle *eccezioni controllate* di Java richiede che il programmatore dichiari che determinati metodi *potrebbero* generare eccezioni controllate specificate. Questo viene fatto usando la clausola `throws` . Per esempio:

```
public class OddNumberException extends Exception { // a checked exception
}

public void checkEven(int number) throws OddNumberException {
    if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

Il `throws OddNumberException` dichiara che una chiamata a `checkEven` *potrebbe* generare un'eccezione di tipo `OddNumberException` .

Una clausola `throws` può dichiarare un elenco di tipi e può includere eccezioni non verificate e eccezioni controllate.

```
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();
    }
}
```

Qual è il punto di dichiarare le eccezioni non controllate come generate?

La clausola `throws` in una dichiarazione di metodo ha due scopi:

1. Indica al compilatore quali eccezioni vengono lanciate in modo che il compilatore possa segnalare eccezioni non verificate (controllate) come errori.

2. Indica a un programmatore che sta scrivendo il codice che chiama il metodo quali eccezioni aspettarsi. A tale scopo, fa spesso intendere di includere eccezioni non selezionate in una lista di `throws` .

Nota: che la lista dei `throws` viene anche utilizzata dallo strumento javadoc durante la generazione della documentazione API e da un tipico suggerimento del metodo "testo hover" dell'IDE.

Tiri e metodo di esclusione

La clausola `throws` fa parte della firma di un metodo allo scopo di sovrascrivere il metodo. Un metodo di override può essere dichiarato con lo stesso insieme di eccezioni verificate come generate dal metodo sottoposto a override o con un sottoinsieme. Tuttavia, il metodo di sovrascrittura non può aggiungere eccezioni controllate supplementari. Per esempio:

```
@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
```

Il motivo di questa regola è che se un metodo sovrascritto può generare un'eccezione verificata che il metodo sottoposto a override non è in grado di generare, ciò interromperà la sostituibilità del tipo.

Leggi Eccezioni e gestione delle eccezioni online: https://riptutorial.com/it/java/topic/89/eccezioni-e-gestione-delle-eccezioni

Capitolo 52: Edizioni, versioni, rilasci e distribuzioni Java

Examples

Differenze tra le distribuzioni Java SE JRE o Java SE JDK

Le versioni Sun / Oracle di Java SE sono disponibili in due forme: JRE e JDK. In termini semplici, i JRE supportano l'esecuzione di applicazioni Java e i JDK supportano anche lo sviluppo Java.

Java Runtime Environment

Le distribuzioni Java Runtime Environment o JRE sono costituite dall'insieme di librerie e strumenti necessari per eseguire e gestire le applicazioni Java. Gli strumenti in un tipico JRE moderno includono:

- Il comando `java` per l'esecuzione di un programma Java in una JVM (Java Virtual Machine)
- Il comando `jjs` per l'esecuzione del motore JavaScript Nashorn.
- Il comando `keytool` per manipolare i keystore Java.
- Il comando `policytool` per la modifica delle politiche di sicurezza sandbox di sicurezza.
- Gli strumenti `pack200` e `unpack200` per l'imballaggio e il disimballaggio del file "pack200" per l'implementazione web.
- I `orbd` , `rmid` , `rmiregistry` e `tnameserv` che supportano le applicazioni Java CORBA e RMI.

Gli installer di "Desktop JRE" includono un plugin Java adatto per alcuni browser web. Questo è volutamente lasciato fuori da "Server JRE" installers.linux syscall read benchmarku

Dall'aggiornamento 6 di Java 7 in poi, i programmi di installazione JRE hanno incluso JavaFX (versione 2.2 o successiva).

Kit di sviluppo Java

Una distribuzione Java Development Kit o JDK include gli strumenti JRE e strumenti aggiuntivi per lo sviluppo di software Java. Gli strumenti aggiuntivi includono in genere:

- Il comando `javac` , che compila il codice sorgente Java (".java") in file bytecode (".class").
- Gli strumenti per creare file JAR come `jar` e `jarsigner`
- Strumenti di sviluppo come:
  - `appletviewer` per l'esecuzione di applet
  - `idlj` il compilatore `idlj` CORBA a Java
  - `javah` il generatore di stub JNI
  - `native2ascii` per la conversione del set di caratteri del codice sorgente Java
  - `schemagen` il generatore di schemi da Java a XML (parte di JAXB)
  - `serialver` genera una stringa di versione di serializzazione di oggetti Java.

    - gli strumenti di supporto `wsgen` e `wsimport` per JAX-WS
- Strumenti diagnostici come:
    - `jdb` il debugger Java di base
    - `jmap` e `jhat` per il dumping e l'analisi di un heap Java.
    - `jstack` per ottenere un dump dello stack di thread.
    - `javap` per l'esame dei file ".class".
- Strumenti di gestione e monitoraggio delle applicazioni come:
    - `jconsole` una console di gestione,
    - `jstat` , `jstatd` , `jinfo` e `jps` per il monitoraggio dell'applicazione

Una tipica installazione Sun / Oracle JDK include anche un file ZIP con il codice sorgente delle librerie Java. Prima di Java 6, questo era l'unico codice sorgente Java disponibile pubblicamente.

Da Java 6 in poi, il codice sorgente completo per OpenJDK è disponibile per il download dal sito OpenJDK. Solitamente non è incluso nei pacchetti JDK (Linux), ma è disponibile come pacchetto separato.

Qual è la differenza tra Oracle Hotspot e OpenJDK

Ortogonale alla dicotomia JRE versus JDK, esistono due tipi di rilascio Java ampiamente disponibili:

- Le versioni di Hotspot Oracle sono quelle scaricate dai siti di download di Oracle.
- Le versioni di OpenJDK sono quelle che vengono create (in genere da provider di terze parti) dai repository di origine OpenJDK.

In termini funzionali, c'è poca differenza tra una versione di Hotspot e una versione di OpenJDK. Ci sono alcune funzionalità extra "enterprise" in Hotspot che i clienti Java (a pagamento) Oracle possono abilitare, ma a parte questo la tecnologia è presente sia in Hotspot che in OpenJDK.

Un altro vantaggio di Hotspot rispetto a OpenJDK è che le versioni di patch per Hotspot tendono ad essere disponibili un po 'prima. Ciò dipende anche da quanto agile sia il tuo provider OpenJDK; Ad esempio, quanto tempo impiega un team di build di una distribuzione Linux per preparare e QA una nuova build OpenJDK, e portarla nei loro repository pubblici.

Il rovescio della medaglia è che le versioni di Hotspot non sono disponibili dai repository di pacchetti per la maggior parte delle distribuzioni Linux. Ciò significa che mantenere il tuo software Java aggiornato su una macchina Linux di solito è più utile se usi Hotspot.

Differenze tra Java EE, Java SE, Java ME e JavaFX

La tecnologia Java è sia un linguaggio di programmazione che una piattaforma. Il linguaggio di programmazione Java è un linguaggio orientato agli oggetti di alto livello con una sintassi e uno stile particolari. Una piattaforma Java è un particolare ambiente in cui vengono eseguite le applicazioni del linguaggio di programmazione Java.

Ci sono diverse piattaforme Java. Molti sviluppatori, anche gli sviluppatori di linguaggi di programmazione Java di lunga data, non capiscono in che modo le diverse piattaforme si

relazionano tra loro.

Le piattaforme Java Programming Language

Esistono quattro piattaforme del linguaggio di programmazione Java:

- Piattaforma Java, Standard Edition (Java SE)
- Piattaforma Java, Enterprise Edition (Java EE)
- Piattaforma Java, Micro Edition (Java ME)
- FX Java

Tutte le piattaforme Java sono costituite da una Java Virtual Machine (VM) e un'interfaccia di programmazione delle applicazioni (API). Java Virtual Machine è un programma, per una particolare piattaforma hardware e software, che esegue applicazioni di tecnologia Java. Un'API è una raccolta di componenti software che è possibile utilizzare per creare altri componenti o applicazioni software. Ogni piattaforma Java fornisce una macchina virtuale e un'API e questo consente alle applicazioni scritte per quella piattaforma di funzionare su qualsiasi sistema compatibile con tutti i vantaggi del linguaggio di programmazione Java: indipendenza dalla piattaforma, potenza, stabilità, facilità di sviluppo e sicurezza.

Java SE

Quando molte persone pensano al linguaggio di programmazione Java, pensano all'API Java SE. L'API di Java SE fornisce le funzionalità principali del linguaggio di programmazione Java. Definisce tutto, dai tipi e oggetti di base del linguaggio di programmazione Java alle classi di alto livello utilizzate per il networking, la sicurezza, l'accesso al database, lo sviluppo dell'interfaccia grafica utente (GUI) e l'analisi XML.

Oltre all'API di base, la piattaforma Java SE è composta da una macchina virtuale, strumenti di sviluppo, tecnologie di implementazione e altre librerie di classi e toolkit comunemente utilizzati nelle applicazioni di tecnologia Java.

Java EE

La piattaforma Java EE è costruita sulla piattaforma Java SE. La piattaforma Java EE fornisce un ambiente API e di runtime per lo sviluppo e l'esecuzione di applicazioni di rete su larga scala, multilivello, scalabili, affidabili e sicure.

Java ME

La piattaforma Java ME fornisce un'API e una macchina virtuale di dimensioni ridotte per l'esecuzione di applicazioni di linguaggio di programmazione Java su dispositivi di piccole dimensioni, come i telefoni cellulari. L'API è un sottoinsieme dell'API Java SE, insieme a librerie di classi speciali utili per lo sviluppo di applicazioni su dispositivi di piccole dimensioni. Le applicazioni Java ME sono spesso client dei servizi della piattaforma Java EE.

FX Java

La tecnologia Java FX è una piattaforma per la creazione di applicazioni Internet ricche scritte in Java FX ScriptTM. Java FX Script è un linguaggio dichiarativo con tipizzazione statica che viene compilato in bytecode con tecnologia Java, che può quindi essere eseguito su una VM Java. Le applicazioni scritte per la piattaforma Java FX possono includere e collegare classi di linguaggio di programmazione Java e possono essere clienti di servizi di piattaforma Java EE.

- Tratto dalla documentazione di Oracle

Versioni di Java SE

Storia della versione di SE Java

La seguente tabella fornisce la tempistica per le principali versioni significative della piattaforma Java SE.

Java SE Versione [1]	Nome in codice	Fine vita (gratis [2])	Data di rilascio
Java SE 9 (accesso anticipato)	*Nessuna*	futuro	2017-07-27 (stimato)
Java SE 8	*Nessuna*	futuro	2014/03/18
Java SE 7	Delfino	2015/04/14	2011-07-28
Java SE 6	Mustang	2013/04/16	2006-12-23
Java SE 5	Tigre	2009-11-04	2004-10-04
Java SE 1.4.2	Mantide	prima del 2009-11-04	2003-06-26
Java SE 1.4.1	Tramoggia / cavalletta	prima del 2009-11-04	2002/09/16
Java SE 1.4	smeriglio	prima del 2009-11-04	2002/02/06

Java SE Versione [1]	Nome in codice	Fine vita (gratis [2])	Data di rilascio
Java SE 1.3.1	Coccinella	prima del 2009-11-04	2001-05-17
Java SE 1.3	Gheppio	prima del 2009-11-04	2000/05/08
Java SE 1.2	Terreno di gioco	prima del 2009-11-04	1998/12/08
Java SE 1.1	sparkler	prima del 2009-11-04	1997/02/19
Java SE 1.0	Quercia	prima del 2009-11-04	1996/01/21

Note:

1. I collegamenti sono alle copie online della documentazione relativa alle versioni sul sito Web di Oracle. La documentazione di molte versioni precedenti non è più online, anche se in genere può essere scaricata dagli Oracle Java Archives.
2. La maggior parte delle versioni storiche di Java SE ha superato le date ufficiali di "fine vita". Quando una versione di Java supera questo traguardo, Oracle smette di fornire aggiornamenti gratuiti per questo. Gli aggiornamenti sono ancora disponibili per i clienti con contratti di supporto.

Fonte:

- Date di rilascio di JDK di Roedy Green di Canadian Mind Products

Caratteristiche della versione Java SE

Versione Java SE	Mette in risalto
Java SE 8	Espressioni lambda e flussi ispirati a MapReduce. Il motore Javascript di Nashorn. Annotazioni su tipi e annotazioni ripetute. Estensioni aritmetiche senza segno. Nuove API di data e ora. Librerie JNI collegate in modo statico. Launcher JavaFX. Rimozione di PermGen.
Java SE 7	String switch, *try-with-resource* , diamond (<>), miglioramenti letterali numerici e miglioramenti nella gestione delle eccezioni / rethrowing. Miglioramenti alla libreria di concorrenza. Supporto avanzato per i file system nativi. Timsort. Algoritmi crittografici ECC. Migliorato supporto per la grafica 2D (GPU). Annotazioni inseribili

Versione Java SE	Mette in risalto
Java SE 6	Miglioramenti significativi delle prestazioni alla piattaforma JVM e Swing. API di scripting language e motore di Mozilla Rhino Javascript. JDBC 4.0. API del compilatore. JAXB 2.0. Supporto dei servizi Web (JAX-WS)
Java SE 5	Generics, annotazioni, auto-boxing, classi `enum` , varargs, enhanced `for` loops e importazioni statiche. Specifica del modello di memoria Java. Miglioramenti dell'oscillazione e dell'RMI. Aggiunta del pacchetto `java.util.concurrent.*` E dello `Scanner` .
Java SE 1.4	La parola chiave `assert` . Classi di espressioni regolari. Eccezione concatenata. API NIO: I / O non bloccante, `Buffer` e `Channel` . `java.util.logging.*` API. API I / O immagine. XML integrato e XSLT (JAXP). Sicurezza integrata e crittografia (JCE, JSSE, JAAS). Java Web Start integrato. API delle preferenze.
Java SE 1.3	HotSpot JVM incluso. Integrazione CORBA / RMI. JNDI (Java Naming and Directory Interface). Struttura debugger (JPDA). API JavaSound. API proxy.
Java SE 1.2	La parola chiave `strictfp` . API Swing. Il plugin Java (per i browser Web). Interoperabilità CORBA. Quadro delle collezioni.
Java SE 1.1	Classi interne Riflessione. JDBC. RMI. Stream Unicode / di caratteri. Supporto per l'internazionalizzazione Revisione del modello di eventi AWT. JavaBeans.

Fonte:

- Wikipedia: cronologia delle versioni di Java

Leggi Edizioni, versioni, rilasci e distribuzioni Java online:
https://riptutorial.com/it/java/topic/8973/edizioni--versioni--rilasci-e-distribuzioni-java

Capitolo 53: Elaborazione degli argomenti della riga di comando

Sintassi

- public static void main (String [] args)

Parametri

Parametro	Dettagli
args	Gli argomenti della riga di comando. Supponendo che il metodo `main` sia invocato dal launcher Java, `args` sarà non nullo e non avrà elementi `null` .

Osservazioni

Quando una normale applicazione Java viene avviata usando il comando `java` (o equivalente), verrà chiamato un metodo `main` , passando gli argomenti dalla riga di comando dell'array `args` .

Sfortunatamente, le librerie di classi Java SE non forniscono alcun supporto diretto per l'elaborazione degli argomenti dei comandi. Questo ti lascia due alternative:

- Implementa l'argomento elaborando a mano in Java.
- Utilizza una libreria di terze parti.

Questo argomento tenterà di coprire alcune delle più popolari librerie di terze parti. Per un elenco completo delle alternative, vedere questa risposta alla domanda StackOverflow "Come analizzare gli argomenti della riga di comando in Java?" .

Examples

Elaborazione degli argomenti mediante GWT ToolBase

Se si desidera analizzare argomenti di riga di comando più complessi, ad esempio con parametri facoltativi, il migliore è utilizzare l'approccio GWT di Google. Tutte le classi sono disponibili al pubblico presso:

https://gwt.googlesource.com/gwt/+/2.8.0-beta1/dev/core/src/com/google/gwt/util/tools/ToolBase.java

Un esempio per la gestione della riga di comando `myprogram -dir "~/Documents" -port 8888` è:

```
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` ha anche un metodo `isRequired()` che può essere sovrascritto per dire che l'argomento della riga di comando è richiesto (il valore predefinito restituito è `false` modo che l'argomento sia facoltativo.

Elaborazione di argomenti a mano

Quando la sintassi della riga di comando per un'applicazione è semplice, è ragionevole eseguire l'argomento del comando elaborando interamente nel codice personalizzato.

In questo esempio, presenteremo una serie di semplici case study. In ogni caso, il codice genererà messaggi di errore se gli argomenti non sono accettabili e quindi chiamerà `System.exit(1)` per indicare alla shell che il comando non è riuscito. (Assumeremo in ogni caso che il codice Java è invocato usando un wrapper il cui nome è "myapp".)

Un comando senza argomenti

In questo caso di studio, il comando non richiede argomenti. Il codice illustra che `args.length` ci fornisce il numero di argomenti della riga di 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 due argomenti

In questo caso di studio, il comando richiede esattamente due argomenti.

```
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]);
    }
}
```

Notare che se si trascurasse di controllare `args.length` , il comando si `args.length` anomalo se l'utente lo eseguiva con troppi pochi argomenti da riga di comando.

Un comando con opzioni "flag" e almeno un argomento

In questo caso di studio, il comando ha un paio di opzioni (opzionali) di flag e richiede almeno un argomento dopo le opzioni.

```
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);
    }
}
```

Come potete vedere, l'elaborazione degli argomenti e delle opzioni diventa piuttosto ingombrante se la sintassi del comando è complicata. Si consiglia di utilizzare una libreria "parsing della riga di comando"; vedi gli altri esempi.

Leggi Elaborazione degli argomenti della riga di comando online:
https://riptutorial.com/it/java/topic/4775/elaborazione-degli-argomenti-della-riga-di-comando

Capitolo 54: elenchi

introduzione

Una *lista* è una raccolta di valori *ordinati* . In Java, le liste fanno parte di Java Collections Framework . Le liste implementano l'interfaccia `java.util.List` , che estende `java.util.Collection` .

Sintassi

- ls.add (elemento E); // Aggiunge un elemento
- ls.remove (elemento E); // Rimuove un elemento
- for (E element: ls) {} // Iterates su ogni elemento
- ls.toArray (new String [ls.length]); // Converte un elenco di stringhe in un array di stringhe
- ls.get (int index); // Restituisce l'elemento nell'indice specificato.
- ls.set (indice int, elemento E); // Sostituisce l'elemento in una posizione specificata.
- ls.isEmpty (); // Restituisce true se la matrice non contiene elementi, altrimenti restituisce false.
- ls.indexOf (Object o); // Restituisce l'indice della prima posizione dell'elemento specificato o, o, se non è presente, restituisce -1.
- ls.lastIndexOf (Object o); // Restituisce l'indice dell'ultima posizione dell'elemento specificato o, oppure, se non è presente, restituisce -1.
- ls.size (); // Restituisce il numero di elementi nella lista.

Osservazioni

Un elenco è un oggetto che memorizza una raccolta di valori ordinata. "Ordinato" significa che i valori sono memorizzati in un ordine particolare: un elemento viene prima, uno viene al secondo e così via. I singoli valori sono comunemente chiamati "elementi". Gli elenchi Java in genere forniscono queste funzionalità:

- Le liste possono contenere zero o più elementi.
- Le liste possono contenere valori duplicati. In altre parole, un elemento può essere inserito in una lista più di una volta.
- Gli elenchi memorizzano i loro elementi in un ordine particolare, ovvero un elemento viene prima, uno viene dopo e così via.
- Ogni elemento ha un *indice che* indica la sua posizione all'interno della lista. Il primo elemento ha indice 0, il successivo ha indice 1 e così via.
- Le liste consentono di inserire elementi all'inizio, alla fine o in qualsiasi indice all'interno dell'elenco.
- Verificare se un elenco contiene un valore particolare generalmente significa esaminare ciascun elemento nell'elenco. Ciò significa che il tempo per eseguire questo controllo è O (n) , proporzionale alla dimensione della lista.

L'aggiunta di un valore a un elenco in un punto diverso dalla fine sposterà tutti i seguenti elementi

"in basso" o "a destra". In altre parole, l'aggiunta di un elemento nell'indice *n* sposta l'elemento che era al momento dell'indice *n* all'indice *n + 1* e così via. Per esempio:

```
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

Ordinamento di un elenco generico

La classe `Collections` offre due metodi statici standard per ordinare un elenco:

- `sort(List<T> list)` applicabile agli elenchi in cui `T extends Comparable<? super T>` , e
- `sort(List<T> list, Comparator<? super T> c)` applicabile a liste di qualsiasi tipo.

L'applicazione del primo richiede la modifica della classe di elementi di elenco ordinati, il che non è sempre possibile. Potrebbe anche non essere desiderabile poiché, sebbene fornisca l'ordinamento predefinito, altri ordini di ordinamento potrebbero essere richiesti in circostanze diverse, o l'ordinamento è solo un'attività a parte.

Considera che abbiamo un'attività di ordinare oggetti che sono istanze della seguente classe:

```
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);
    }
}
```

Per utilizzare `Collections.sort(List<User> list)` è necessario modificare la classe `User` per implementare l'interfaccia `Comparable` . Per esempio

```
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);
    }
}
```

(A parte: molte classi Java standard come `String` , `Long` , `Integer` implementano l'interfaccia `Comparable` rende gli elenchi di quegli elementi ordinabili per impostazione predefinita e semplifica l'implementazione di `compare` o `compareTo` in altre classi.)

Con la modifica sopra, possiamo facilmente ordinare un elenco di oggetti `User` base *all'ordine naturale delle* classi. (In questo caso, abbiamo definito che essere ordinati in base ai valori `id`). Per esempio:

```
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]
```

Tuttavia, supponiamo di voler ordinare gli oggetti `User` per `name` anziché per `id` . In alternativa, supponiamo di non essere stato in grado di modificare la classe per renderla `Comparable` .

È qui che il metodo di `sort` con l'argomento `Comparator` è utile:

```
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

In Java 8 puoi usare una *lambda* invece di una classe anonima. Quest'ultimo si riduce ad un unico rivestimento:

```
Collections.sort(users, (l, r) -> l.username.compareTo(r.username));
```

Inoltre, Java 8 aggiunge un metodo di `sort` predefinito all'interfaccia `List` , che semplifica ulteriormente l'ordinamento.

```
users.sort((l, r) -> l.username.compareTo(r.username))
```

Creare una lista

Dando alla tua lista un tipo

Per creare un elenco è necessario un tipo (qualsiasi classe, ad esempio `String`). Questo è il tipo della tua `List` . L' `List` memorizzerà solo oggetti del tipo specificato. Per esempio:

```
List<String> strings;
```

Può memorizzare `"string1"` , `"hello world!"` , `"goodbye"` , ecc., ma non può memorizzare `9.2` , tuttavia:

```
List<Double> doubles;
```

Può memorizzare `9.2` , ma non `"hello world!"` .

Inizializzazione della lista

Se provate ad aggiungere qualcosa agli elenchi sopra, otterrete una NullPointerException, perché le `strings` e i `doubles` uguali sono **nulli** !

Esistono due modi per inizializzare un elenco:

Opzione 1: utilizza una classe che implementa List

`List` è un'interfaccia, il che significa che non ha un costruttore, piuttosto metodi che una classe deve sovrascrivere. `ArrayList` è l' `List` più comunemente usato, anche se `LinkedList` è anche comune. Quindi inizializziamo la nostra lista in questo modo:

```
List<String> strings = new ArrayList<String>();
```

o

```
List<String> strings = new LinkedList<String>();
```

Java SE 7

A partire da Java SE 7, è possibile utilizzare un *operatore di diamante* :

```
List<String> strings = new ArrayList<>();
```

o

```
List<String> strings = new LinkedList<>();
```

Opzione 2: usa la classe Collections

La classe `Collections` fornisce due metodi utili per la creazione di elenchi senza una variabile di `List` :

- `emptyList()` : restituisce una lista vuota.
- `singletonList(T)` : crea una lista di tipo T e aggiunge l'elemento specificato.

E un metodo che utilizza un `List` esistente per riempire i dati in:

- `addAll(L, T...)` : aggiunge tutti gli elementi specificati all'elenco passato come primo parametro.

Esempi:

```
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);
```

Operazioni di accesso posizionale

L'API List ha otto metodi per le operazioni di accesso posizionale:

- `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)`

Quindi, se abbiamo una lista:

```
List<String> strings = new ArrayList<String>();
```

E volevamo aggiungere le stringhe "Ciao mondo!" e "Arrivederci mondo!" ad esso, lo faremmo come tale:

```
strings.add("Hello world!");
strings.add("Goodbye world!");
```

E la nostra lista conterrebbe i due elementi. Ora diciamo che volevamo aggiungere "Programma in corso!" in **cima** alla lista. Lo faremmo in questo modo:

```
strings.add(0, "Program starting!");
```

NOTA: il primo elemento è 0.

Ora, se volessimo rimuovere il "mondo degli addii!" linea, potremmo farlo in questo modo:

```
strings.remove("Goodbye world!");
```

E se volessimo rimuovere la prima riga (che in questo caso sarebbe "Programma in avvio!", Potremmo farlo in questo modo:

```
strings.remove(0);
```

Nota:

1. L'aggiunta e la rimozione di elementi di elenco modifica l'elenco e questo può portare a una `ConcurrentModificationException` se l'elenco viene iterato contemporaneamente.
2. L'aggiunta e la rimozione di elementi può essere `O(1)` o `O(N)` seconda della classe di elenco, del metodo utilizzato e dell'aggiunta / rimozione di un elemento all'inizio, alla fine o al centro dell'elenco.

Per recuperare un elemento della lista in una posizione specifica puoi usare `E get(int index);` metodo dell'API List. Per esempio:

```
strings.get(0);
```

restituirà il primo elemento della lista.

Puoi sostituire qualsiasi elemento in una posizione specificata usando il `set(int index, E element);` . Per esempio:

```
strings.set(0,"This is a replacement");
```

Questo imposterà la stringa "Questa è una sostituzione" come il primo elemento della lista.

Nota: il metodo set sovrascrive l'elemento nella posizione 0. Non aggiungerà la nuova stringa nella posizione 0 e spingerà quella precedente nella posizione 1.

L' `int indexOf(Object o);` restituisce la posizione della prima occorrenza dell'oggetto passato come argomento. Se non ci sono occorrenze dell'oggetto nella lista, viene restituito il valore -1. In continuazione dell'esempio precedente se si chiama:

```
strings.indexOf("This is a replacement")
```

lo 0 dovrebbe essere restituito quando impostiamo la stringa "This is a replacement" nella posizione 0 della nostra lista. Nel caso in cui ci siano più di un'occorrenza nella lista quando `int indexOf(Object o);` viene chiamato quindi come detto verrà restituito l'indice della prima occorrenza. Chiamando `int lastIndexOf(Object o)` è possibile recuperare l'indice dell'ultima occorrenza nell'elenco. Quindi se aggiungiamo un altro "Questa è una sostituzione":

```
strings.add("This is a replacement");
strings.lastIndexOf("This is a replacement");
```

Questa volta verrà restituito 1 e non lo 0;

Iterare su elementi in una lista

Per esempio, diciamo che abbiamo una lista di tipo String che contiene quattro elementi: "ciao", "come", "sono", "tu?"

Il modo migliore per scorrere su ogni elemento è utilizzare un ciclo for-each:

```
public void printEachElement(List<String> list){
    for(String s : list){
        System.out.println(s);
    }
}
```

Quale stamperebbe:

```
hello,
how
are
you?
```

Per stamparli tutti nella stessa riga, puoi usare StringBuilder:

```
public void printAsLine(List<String> list){
    StringBuilder builder = new StringBuilder();
    for(String s : list){
        builder.append(s);
    }
    System.out.println(builder.toString());
}
```

Stamperà:

```
hello, how are you?
```

In alternativa, è possibile utilizzare l'indicizzazione degli elementi (come descritto in Accedere all'elemento con l'indice ith da ArrayList) per iterare un elenco. Attenzione: questo approccio è inefficiente per gli elenchi collegati.

Rimozione di elementi dall'elenco B presenti nell'elenco A

Supponiamo di avere 2 elenchi A e B, e tu vuoi rimuovere da **B** tutti gli elementi che hai in **A** il metodo in questo caso è

```
 List.removeAll(Collection c);
```

#Esempio:

```
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);
    }
```

questo stamperà

A: [1, 3, 4, 7, 5, 2]

B: [13, 32, 533, 3, 4, 2]

B cancellato: [13, 32, 533]

Trovare elementi comuni tra 2 elenchi

Supponiamo di avere due liste: A e B, e devi trovare gli elementi che esistono in entrambi gli elenchi.

Puoi farlo invocando semplicemente il metodo `List.retainAll()` .

Esempio:

```
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);

}
```

Convertire un elenco di numeri interi in un elenco di stringhe

```
List<Integer> nums = Arrays.asList(1, 2, 3);
List<String> strings = nums.stream()
    .map(Object::toString)
    .collect(Collectors.toList());
```

Questo è:

1. Crea un flusso dall'elenco
2. Mappare ogni elemento usando `Object::toString`
3. Raccogli i valori di `String` in un `List` utilizzando `Collectors.toList()`

Creazione, aggiunta e rimozione di elementi da un ArrayList

`ArrayList` è una delle strutture dati integrate in Java. È una matrice dinamica (in cui non è necessario dichiarare la dimensione della struttura dati prima) per la memorizzazione di elementi (oggetti).

Estende la classe `AbstractList` e implementa l'interfaccia `List` . Un `ArrayList` può contenere elementi duplicati dove mantiene l'ordine di inserimento. Va notato che la classe `ArrayList` non è sincronizzata, quindi è necessario prestare attenzione quando si maneggia la concorrenza con `ArrayList` . `ArrayList` consente l'accesso casuale perché l'array funziona alla base dell'indice. La manipolazione è lenta in `ArrayList` causa dello spostamento che si verifica spesso quando un elemento viene rimosso dall'elenco di array.

Un `ArrayList` può essere creato come segue:

```
List<T> myArrayList = new ArrayList<>();
```

Dove `T` (Generics) è il tipo che verrà archiviato all'interno di `ArrayList` .

Il tipo di `ArrayList` può essere qualsiasi oggetto. Il tipo non può essere un tipo primitivo (usa invece le loro *classi wrapper*).

Per aggiungere un elemento a `ArrayList` , utilizzare il metodo `add()` :

```
myArrayList.add(element);
```

O per aggiungere elementi a un determinato indice:

```
myArrayList.add(index, element); //index of the element should be an int (starting from 0)
```

Per rimuovere un oggetto da `ArrayList` , utilizzare il metodo `remove()` :

```
myArrayList.remove(element);
```

O per rimuovere un elemento da un certo indice:

```
myArrayList.remove(index); //index of the element should be an int (starting from 0)
```

Sostituzione sul posto di un elemento di elenco

Questo esempio riguarda la sostituzione di un elemento `List` assicurando che l'elemento di sostituzione si trovi nella stessa posizione dell'elemento che viene sostituito.

Questo può essere fatto usando questi metodi:

- set (indice int, tipo T)
- int indexOf (tipo T)

Considera una `ArrayList` contenente gli elementi "Programma in corso!", "Ciao mondo!" e "Arrivederci mondo!"

```
List<String> strings = new ArrayList<String>();
strings.add("Program starting!");
strings.add("Hello world!");
strings.add("Goodbye world!");
```

Se conosciamo l'indice dell'elemento che vogliamo sostituire, possiamo semplicemente usare `set` come segue:

```
strings.set(1, "Hi world");
```

Se non conosciamo l'indice, possiamo prima cercarlo. Per esempio:

```
int pos = strings.indexOf("Goodbye world!");
if (pos >= 0) {
    strings.set(pos, "Goodbye cruel world!");
}
```

Gli appunti:

1. L'operazione `set` non causerà una `ConcurrentModificationException` .
2. L'operazione `set` è veloce (`O(1)`) per `ArrayList` ma è lenta (`O(N)`) per una `LinkedList` .
3. Una ricerca `indexOf` su `ArrayList` o `LinkedList` è lenta (`O(N)`).

Rendere una lista non modificabile

La classe Collections fornisce un modo per rendere un elenco non modificabile:

```
List<String> ls = new ArrayList<String>();
List<String> unmodifiableList = Collections.unmodifiableList(ls);
```

Se vuoi un elenco non modificabile con un oggetto puoi usare:

```
List<String> unmodifiableList = Collections.singletonList("Only string in the list");
```

Spostare gli oggetti nella lista

La classe Collections ti consente di spostare gli oggetti nella lista usando vari metodi (ls è la lista):

Inversione di una lista:

```
Collections.reverse(ls);
```

Ruotare le posizioni degli elementi in una lista

Il metodo di rotazione richiede un argomento intero. Questo è il numero di punti per spostarlo lungo la linea. Un esempio di questo è qui sotto:

```
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();
```

Questo stamperà "ciao, come stai?"

Mescolare elementi in una lista

Usando la stessa lista sopra, possiamo mescolare gli elementi in una lista:

```
Collections.shuffle(ls);
```

Possiamo anche dargli un oggetto java.util.Random che usa per posizionare in modo casuale gli oggetti in punti:

```
Random random = new Random(12);
Collections.shuffle(ls, random);
```

Classi che implementano List - Pro e Contro

L'interfaccia `List` è implementata da classi diverse. Ognuno di loro ha il suo modo di implementarlo con strategie diverse e fornire pro e contro diversi.

Classi che implementano List

Queste sono tutte le classi `public` in Java SE 8 che implementano l'interfaccia `java.util.List` :

1. **Classi astratte:**
    - AbstractList
    - AbstractSequentialList
2. **Classi concrete:**
    - Lista di array
    - AttributeList
    - CopyOnWriteArrayList
    - Lista collegata
    - RoleList
    - RoleUnresolvedList

- Pila
- Vettore

Pro e contro di ogni implementazione in termini di complessità temporale

Lista di array

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

ArrayList è un'implementazione di array ridimensionabile dell'interfaccia List. Memorizzando la lista in un array, *ArrayList* fornisce metodi (oltre ai metodi che implementano l'interfaccia *List*) per manipolare la dimensione dell'array.

Inizializza ArrayList of Integer con dimensione 100

```
List<Integer> myList = new ArrayList<Integer>(100); // Constructs an empty list with the
specified initial capacity.
```

- PROFESSIONISTI:

Le operazioni di dimensione, isEmpty, ***get*** , ***set*** , iterator e listIterator vengono eseguite in tempo costante. Quindi ottenere e impostare ogni elemento dell'elenco ha lo stesso *costo in termini di tempo* :

```
int e1 = myList.get(0);  //   \
int e2 = myList.get(10); //    | => All the same constant cost => O(1)
myList.set(2,10);        //   /
```

- CONS:

L'implementazione di un array (struttura statica) che aggiunge elementi alle dimensioni dell'array ha un costo elevato a causa del fatto che è necessario eseguire una nuova allocazione per tutto l'array. Tuttavia, dalla documentazione :

> L'operazione di aggiunta viene eseguita in tempo costante ammortizzato, ovvero, l'aggiunta di n elementi richiede tempo O (n)

La rimozione di un elemento richiede O (n) tempo.

AttributeList

Alla venuta

CopyOnWriteArrayList

Alla venuta

Lista collegata

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

LinkedList è implementato da un elenco collegato doppiamente una struttura di dati collegata che consiste in un insieme di record collegati in sequenza chiamati nodi.

Iitialize LinkedList of Integer

```
List<Integer> myList = new LinkedList<Integer>(); // Constructs an empty list.
```

- PROFESSIONISTI:

L'aggiunta o la rimozione di un elemento nella parte anteriore dell'elenco o alla fine è costante.

```
myList.add(10);  // \
myList.add(0,2); //  | => constant time => O(1)
myList.remove(); // /
```

- CONS: dalla documentazione :

> Le operazioni che indicizzano nella lista attraverseranno la lista dall'inizio o dalla fine, a seconda di quale sia più vicino all'indice specificato.

Operazioni come:

```
myList.get(10);    // \
myList.add(11,25); //  | => worst case done in O(n/2)
myList.set(15,35); // /
```

RoleList

Alla venuta

RoleUnresolvedList

Alla venuta

Pila

Alla venuta

Vettore

Alla venuta

Leggi elenchi online: https://riptutorial.com/it/java/topic/2989/elenchi

Capitolo 55: Enum che inizia con il numero

introduzione

Java non consente il nome di enum per iniziare con un numero come 100A, 25K. In tal caso, possiamo aggiungere il codice con _ (carattere di sottolineatura) o qualsiasi modello consentito e verificarlo.

Examples

Enum con nome all'inizio

```
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);
    }
}
```

Leggi Enum che inizia con il numero online: https://riptutorial.com/it/java/topic/10719/enum-che-inizia-con-il-numero

Capitolo 56: Enums

introduzione

Le enumerazioni Java (dichiarate usando la parola chiave `enum`) sono sintassi abbreviata per quantità considerevoli di costanti di una singola classe.

Sintassi

- [public / protected / private] enum Enum_name {// Dichiara un nuovo enum.
- ENUM_CONSTANT_1 [, ENUM_CONSTANT_2 ...]; // Dichiara le costanti enum. Questa deve essere la prima riga all'interno dell'enumerazione e deve essere separata da virgole, con un punto e virgola alla fine.
- ENUM_CONSTANT_1 (param) [, ENUM_CONSTANT_2 (param) ...]; // Dichiara le costanti enum con i parametri. I tipi di parametri devono corrispondere al costruttore.
- ENUM_CONSTANT_1 {...} [, ENUM_CONSTANT_2 {...} ...]; // Dichiara le costanti enum con metodi sottoposti a override. Questo deve essere fatto se l'enum contiene metodi astratti; tutti questi metodi devono essere implementati.
- ENUM_CONSTANT.name () // Restituisce una stringa con il nome della costante enum.
- ENUM_CONSTANT.ordinal () // Restituisce l'ordinale di questa costante di enumerazione, la sua posizione nella sua dichiarazione enum, dove alla costante iniziale è assegnato un ordinale pari a zero.
- Enum_name.values () // Restituisce una nuova matrice (di tipo Enum_name []) contenente ogni costante di quell'enumerazione ogni volta che viene chiamata.
- Enum_name.valueOf ("ENUM_CONSTANT") // L'inverso di ENUM_CONSTANT.name () - restituisce la costante enum con il nome specificato.
- Enum.valueOf (Enum_name.class, "ENUM_CONSTANT") // Un sinonimo del precedente: L'inverso di ENUM_CONSTANT.name () - restituisce la costante enum con il nome specificato.

Osservazioni

restrizioni

Enum estende sempre `java.lang.Enum` , quindi è impossibile per un enum estendere una classe. Tuttavia, possono implementare molte interfacce.

Consigli e trucchi

A causa della loro rappresentazione specializzata, ci sono mappe e insiemi più efficienti che possono essere usati con le enumerhe come chiavi. Questi saranno spesso più veloci delle loro controparti non specializzate.

Examples

Dichiarare e usare un enum di base

Enum può essere considerato come lo zucchero di sintassi per una classe sigillata che viene istanziata solo un numero di volte noto in fase di compilazione per definire un insieme di costanti.

Un enum semplice per elencare le diverse stagioni sarebbe dichiarato come segue:

```
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

Anche se le costanti enum non devono necessariamente essere in maiuscolo, è convenzione Java che i nomi delle costanti sono interamente in maiuscolo, con le parole separate da caratteri di sottolineatura.

Puoi dichiarare un Enum nel suo file:

```
/**
 * This enum is declared in the Season.java file.
*/
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

Ma puoi anche dichiararlo in un'altra classe:

```
 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
    }

}
```

Infine, non puoi dichiarare un Enum all'interno di un corpo o costruttore di un metodo:

```
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;
    }

}
```

Le costanti enum duplicate non sono consentite:

```
public enum Season {
    WINTER,
    WINTER, //Compile Time Error : Duplicate Constants
    SPRING,
    SUMMER,
    FALL
}
```

Ogni costante di enum è **`public`**, **`static`** e **`final`** di default. Poiché ogni costante è `static`, è possibile accedervi direttamente utilizzando il nome dell'enumerazione.

Le costanti Enum possono essere passate come parametri del metodo:

```
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"
```

È possibile ottenere un array di costanti enum utilizzando il metodo `values()` . I valori sono garantiti per essere nell'ordine di dichiarazione nell'array restituito:

```
Season[] seasons = Season.values();
```

Nota: questo metodo alloca un nuovo array di valori ogni volta che viene chiamato.

Per scorrere le costanti enum:

```
public static void enumIterate() {
    for (Season s : Season.values()) {
        System.out.println(s.name());
    }
}
```

È possibile utilizzare le enumerazioni in un'istruzione `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;
    }
}
```

Puoi anche confrontare le costanti enum usando `==` :

```
Season.FALL == Season.WINTER    // false
Season.SPRING == Season.SPRING  // true
```

Un altro modo per confrontare le costanti enum è usando `equals()` come sotto, che è considerato una cattiva pratica in quanto si può facilmente cadere in insidie come segue:

```
Season.FALL.equals(Season.FALL); // true
Season.FALL.equals(Season.WINTER); // false
Season.FALL.equals("FALL"); // false and no compiler error
```

Inoltre, sebbene l'insieme di istanze `enum` non possa essere modificato in fase di esecuzione, le istanze stesse non sono intrinsecamente immutabili poiché come qualsiasi altra classe, un `enum` può contenere campi mutabili come illustrato di seguito.

```
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
```

Tuttavia, una buona pratica è rendere le istanze `enum` immutabili, cioè quando non hanno campi aggiuntivi o tutti questi campi sono contrassegnati come `final` e sono immutabili. Ciò garantirà che per tutta la durata dell'applicazione l' `enum` non perderà memoria e che è sicuro utilizzare le sue istanze su tutti i thread.

`Enum` implementa implicitamente `Serializable` e `Comparable` perché la classe `Enum` esegue:

```
public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable
```

Enum con costruttori

Un `enum` non può avere un costruttore pubblico; tuttavia, i costruttori privati sono accettabili (i costruttori per le enumerazioni sono pacchetti-privati per impostazione predefinita):

```
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
```

Si consiglia di mantenere tutti i campi privati e di fornire metodi getter, in quanto vi è un numero finito di istanze per un enum.

Se invece dovessi implementare un `Enum` come `class` , assomiglierebbe a questo:

```
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
```

Le costanti Enum sono tecnicamente mutabili, quindi è possibile aggiungere un setter per modificare la struttura interna di una costante enum. Tuttavia, questa è considerata una pratica molto negativa e dovrebbe essere evitata.

La migliore pratica è rendere immutabili i campi Enum, con `final` :

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value){
        this.value = value;
    }

    ...

}
```

È possibile definire più costruttori nella stessa enumerazione. Quando lo fai, gli argomenti che passi nella tua dichiarazione enum decidono quale costruttore è chiamato:

```
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: tutti i campi enumerati non primitivi dovrebbero implementare `Serializable` perché la classe `Enum` fa.

Utilizzo di metodi e blocchi statici

Un enum può contenere un metodo, proprio come qualsiasi classe. Per vedere come funziona, dichiareremo un enume come questo:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;
}
```

Prendiamo un metodo che restituisce l'enum nella direzione opposta:

```
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;
        }
    }
}
```

Questo può essere ulteriormente migliorato attraverso l'uso di campi e blocchi di inizializzazione statici:

```
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;
    }
}
```

In questo esempio, la direzione opposta viene memorizzata in un campo di istanza privato `opposite` , che viene inizializzato staticamente la prima volta che viene utilizzata una `Direction` . In questo caso particolare (perché `NORTH` riferimento al `SOUTH` e viceversa), non possiamo usare Enum con costruttori qui (Costruttori `NORTH(SOUTH), SOUTH(NORTH), EAST(WEST), WEST(EAST)` sarebbe più elegante e consentirebbe il `opposite` essere dichiarato `final` , ma sarebbe autoreferenziale e quindi non consentito).

Interfaccia di implementazioni

Questa è una `enum` che è anche una funzione callable che verifica gli input di `String` contro i modelli di espressioni regolari precompilati.

```
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"));
    }
}
```

Ogni membro dell'enum può anche implementare il metodo:

```
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
    }
}
```

Enum Polymorphism Pattern

Quando un metodo deve accettare un insieme "enensibile" di valori `enum` , il programmatore può applicare il polimorfismo come in una `class` normale creando un'interfaccia che sarà utilizzata in qualsiasi altro punto in cui l' `enum` deve essere usato:

```
public interface ExtensibleEnum {
    String name();
}
```

In questo modo, qualsiasi `enum` taggato da (implementando) l'interfaccia può essere usato come parametro, consentendo al programmatore di creare una quantità variabile di `enum` che sarà accettata dal metodo. Questo può essere utile, ad esempio, nelle API in cui esiste un `enum` predefinito (non modificabile) e l'utente di queste API desidera "estendere" l' `enum` con più valori.

Una serie di valori di enum predefiniti può essere definita come segue:

```
public enum DefaultValues implements ExtensibleEnum {
    VALUE_ONE, VALUE_TWO;
}
```

Valori aggiuntivi possono quindi essere definiti in questo modo:

```
public enum ExtendedValues implements ExtensibleEnum {
    VALUE_THREE, VALUE_FOUR;
}
```

Esempio che mostra come usare le enumerazioni - si noti come `printEnum()` accetta valori da

entrambi i `enum` tipi:

```
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: questo modello non impedisce di ridefinire i valori enum, che sono già definiti in un enum, in un altro enum. Questi valori di enumerazione sarebbero quindi diversi istanze. Inoltre, non è possibile utilizzare switch-on-enum poiché tutto ciò che abbiamo è l'interfaccia, non il vero `enum` .

Enums con metodi astratti

Enums può definire metodi astratti, che ogni membro `enum` è tenuto a implementare.

```
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);
}
```

Ciò consente a ciascun membro di enum di definire il proprio comportamento per una determinata operazione, senza dover attivare i tipi in un metodo nella definizione di livello superiore.

Si noti che questo modello è una forma breve di ciò che viene tipicamente ottenuto usando il polimorfismo e / o l'implementazione di interfacce.

Documentare le enumerazioni

Non sempre il nome `enum` è abbastanza chiaro per essere compreso. Per documentare un `enum` , usa javadoc standard:

```
/**
 * 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;
    }
}
```

Ottenere i valori di un enum

Ogni classe enum contiene un metodo statico implicito denominato `values()` . Questo metodo restituisce una matrice contenente tutti i valori di quell'enumerazione. È possibile utilizzare questo metodo per scorrere i valori. È importante notare tuttavia che questo metodo restituisce un **nuovo** array ogni volta che viene chiamato.

```
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());
        }
    }
}
```

Se hai bisogno di un `Set` puoi anche usare `EnumSet.allOf(Day.class)` .

Enum come parametro di tipo limitato

Quando si scrive una classe con generics in java, è possibile assicurarsi che il parametro type sia un enum. Poiché tutte le enumerazioni estendono la classe `Enum` , è possibile utilizzare la seguente sintassi.

```
public class Holder<T extends Enum<T>> {
    public final T value;

    public Holder(T init) {
        this.value = init;
    }
}
```

In questo esempio, il tipo `T` *deve* essere un enum.

Ottieni costante enum per nome

Diciamo che abbiamo un enum `DayOfWeek` :

```
enum DayOfWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}
```

Un enum è compilato con un metodo built-in statico `valueOf()` che può essere usato per cercare una costante con il suo nome:

```
String dayName = DayOfWeek.SUNDAY.name();
assert dayName.equals("SUNDAY");

DayOfWeek day = DayOfWeek.valueOf(dayName);
assert day == DayOfWeek.SUNDAY;
```

Questo è anche possibile usando un tipo enum dinamico:

```
Class<DayOfWeek> enumType = DayOfWeek.class;
DayOfWeek day = Enum.valueOf(enumType, "SUNDAY");
assert day == DayOfWeek.SUNDAY;
```

Entrambi questi metodi `valueOf()` generano un `IllegalArgumentException` se l'enum specificato non ha una costante con un nome corrispondente.

La libreria Guava fornisce un metodo di supporto `Enums.getIfPresent()` che restituisce un `Optional` Guava per eliminare la gestione delle eccezioni esplicite:

```
DayOfWeek defaultDay = DayOfWeek.SUNDAY;
DayOfWeek day = Enums.valueOf(DayOfWeek.class, "INVALID").or(defaultDay);
assert day == DayOfWeek.SUNDAY;
```

Implementa il pattern Singleton con un enum a elemento singolo

Le costanti Enum vengono istanziate quando un enum viene referenziato per la prima volta. Pertanto, ciò consente di implementare il modello di progettazione software Singleton con un enum a elemento singolo.

```
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
    }
}
```

Secondo il libro "Effective Java" di Joshua Bloch, un enum a elemento singolo è il modo migliore per implementare un singleton. Questo approccio ha i seguenti vantaggi:

- sicurezza del filo
- garanzia di singola istanziazione
- serializzazione pronta all'uso

E come mostrato nella sezione implements interface questo singleton potrebbe anche implementare una o più interfacce.

Enum con proprietà (campi)

Nel caso in cui vogliamo usare `enum` con più informazioni e non solo come valori costanti, e vogliamo essere in grado di confrontare due enumerazioni.

Considera il seguente esempio:

```
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;
    }

}
```

Qui abbiamo definito un `Enum` chiamato `Coin` che rappresenta il suo valore. Con il metodo `isGreaterThan` possiamo confrontare due `enum` :

```
Coin penny = Coin.PENNY;
Coin dime = Coin.DIME;

System.out.println(penny.isGreaterThan(dime)); // prints: false
System.out.println(dime.isGreaterThan(penny)); // prints: true
```

Converti enum in stringa

A volte vuoi convertire il tuo enum in una stringa, ci sono due modi per farlo.

Supponiamo di avere:

```
public enum Fruit {
    APPLE, ORANGE, STRAWBERRY, BANANA, LEMON, GRAPE_FRUIT;
}
```

Quindi, come possiamo convertire qualcosa come `Fruit.APPLE` in `"APPLE"` ?

Converti usando il `name()`

`name()` è un metodo interno in `enum` che restituisce la rappresentazione `String` dell'enumerazione, la `String` ritorno rappresenta ***esattamente il*** modo in cui è stato definito il valore enum.

Per esempio:

```
System.out.println(Fruit.BANANA.name());      // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.name()); // "GRAPE_FRUIT"
```

Converti usando `toString()`

`toString()` è, *per impostazione predefinita* , sottoposto a override per avere lo stesso comportamento di `name()`

Tuttavia, `toString()` è probabilmente sovrascritto dagli *sviluppatori* per farlo stampare una `String` più user-friendly

Non usare `toString()` se vuoi fare il check-in del tuo codice, `name()` è molto più stabile per quello. Utilizzare `toString()` quando si intende emettere il valore su logs o stdout o qualcosa del genere

Di default:

```
System.out.println(Fruit.BANANA.toString());      // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "GRAPE_FRUIT"
```

Esempio di sovrascrittura

```
System.out.println(Fruit.BANANA.toString());      // "Banana"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "Grape Fruit"
```

Enum corpo specifico costante

In un `enum` è possibile definire un comportamento specifico per una particolare costante `enum` che sovrascrive il comportamento predefinito `enum` , questa tecnica è nota come *corpo specifico costante* .

Supponiamo che tre studenti di pianoforte - John, Ben e Luke - siano definiti in un `enum` chiamato `PianoClass` , come segue:

```
    enum PianoClass {
    JOHN, BEN, LUKE;
        public String getSex() {
            return "Male";
        }
        public String getLevel() {
            return "Beginner";
        }
    }
```

E un giorno arrivano altri due studenti - Rita e Tom - con un sesso (femmina) e un livello (intermedio) che non corrispondono ai precedenti:

```
    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
        }
    }
```

cosicché aggiungere semplicemente i nuovi studenti alla dichiarazione costante, come segue, non è corretto:

```
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
```

È possibile definire un comportamento specifico per ciascuna costante, Rita e Tom, che sovrascrivono il comportamento predefinito `PianoClass2` come segue:

```
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";
    }
}
```

e ora il livello di Tom e il sesso di Rita sono come dovrebbero essere:

```
PianoClass3 tom = PianoClass3.TOM;
PianoClass3 rita = PianoClass3.RITA;
System.out.println(tom.getLevel()); // prints Intermediate
System.out.println(rita.getSex()); // prints Female
```

Un altro modo per definire il corpo specifico del contenuto è utilizzando il costruttore, ad esempio:

```
enum Friend {
    MAT("Male"),
    JOHN("Male"),
    JANE("Female");

    private String gender;

    Friend(String gender) {
        this.gender = gender;
    }

    public String getGender() {
        return this.gender;
    }
}
```

e utilizzo:

```
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
```

Zero istanza enum

```
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...
}
```

Proprio come `enum` può essere usato per singleton (1 classi di istanze), può essere usato per classi di utilità (0 classi di istanze). Assicurati di terminare la lista (vuota) di costanti enum con a `;` .

Vedi la domanda Zero istanza enum vs costruttori privati per prevenire l'istanziazione per una discussione su pro e contro rispetto ai costruttori privati.

Enum con campi statici

Se è richiesta la classe enum per avere campi statici, tenere presente che vengono creati **dopo** i valori enum stessi. Ciò significa che il codice seguente genererà una `NullPointerException` :

```
enum Example {
    ONE(1), TWO(2);

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

    private Example(int value) {
        integers.put(this.name(), value);
    }
}
```

Un modo possibile per risolvere questo problema:

```
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);
    }
}
```

Non inizializzare il campo statico:

```
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()));
    }
}
```

initialisisation:

- crea i valori enum
    - come effetto collaterale putValue () chiamato che inizializza interi
- i valori statici sono impostati
    - interi = null; // viene eseguito dopo l'enumerazione, quindi il contenuto degli interi viene perso

Confronta e contiene per i valori Enum

Enum contiene solo costanti e può essere confrontato direttamente con `==` . Quindi, è necessario solo un controllo di riferimento, non è necessario utilizzare il metodo `.equals` . Inoltre, se `.equals` utilizzato in modo errato, può aumentare `NullPointerException` mentre non è il caso con il controllo `==` .

```
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!");
        }

    }
}
```

Per raggruppare, completare, estendere i valori enum abbiamo classe `EnumSet` che contiene metodi diversi.

- `EnumSet#range` : per ottenere sottoinsieme di enum per intervallo definito da due endpoint
- `EnumSet#of` : insieme di enumerazioni specifiche senza intervallo. Esistono più overload `of` metodi.
- `EnumSet#complementOf` : set di enum che è complementare ai valori enum forniti nel parametro method

```
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");
      }
  }
}
```

Leggi Enums online: https://riptutorial.com/it/java/topic/155/enums

Capitolo 57: Eredità

introduzione

L'ereditarietà è una caratteristica di base orientata agli oggetti in cui una classe acquisisce e si estende alle proprietà di un'altra classe, utilizzando la parola chiave `extends` . Per Interfacce e le `implements` parole chiave, vedere le interfacce .

Sintassi

- class ClassB estende ClassA {...}
- classe ClassB implementa InterfaceA {...}
- interfaccia InterfaceB estende InterfaceA {...}
- class ClassB estende le implementazioni di ClassA InterfaceC, InterfaceD {...}
- abstract class AbstractClassB estende ClassA {...}
- abstract class AbstractClassB estende AbstractClassA {...}
- abstract class AbstractClassB estende le implementazioni di ClassA InterfaceC, InterfaceD {...}

Osservazioni

L'ereditarietà è spesso combinata con i generici in modo che la classe base abbia uno o più parametri di tipo. Vedi Creazione di una classe generica .

Examples

Classi astratte

Una classe astratta è una classe contrassegnata con la parola chiave `abstract` . Esso, contrariamente alla classe non astratta, può contenere metodi astratti - implementazione-meno. È tuttavia valido creare una classe astratta senza metodi astratti.

Una classe astratta non può essere istanziata. Può essere sottoclassato (esteso) finché la sottoclasse è anch'essa astratta, o implementa tutti i metodi contrassegnati come astratti da super classi.

Un esempio di una classe astratta:

```
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 classe deve essere contrassegnata come astratta, quando ha almeno un metodo astratto. Un metodo astratto è un metodo che non ha implementazione. Altri metodi possono essere dichiarati all'interno di una classe astratta che ha implementazione al fine di fornire codice comune per qualsiasi sottoclasse.

Il tentativo di creare un'istanza di questa classe fornirà un errore di compilazione:

```
//error: Component is abstract; cannot be instantiated
Component myComponent = new Component();
```

Tuttavia una classe che estende `Component` e fornisce un'implementazione per tutti i suoi metodi astratti e può essere istanziata.

```
public class Button extends Component {

    @Override
    public void render() {
        //render a button
    }
}

public class TextBox extends Component {

    @Override
    public void render() {
        //render a textbox
    }
}
```

Anche le istanze di classi ereditanti possono essere espresse come classe genitore (normale ereditarietà) e forniscono un effetto polimorfico quando viene chiamato il metodo astratto.

```
Component myButton = new Button();
Component myTextBox = new TextBox();

myButton.render(); //renders a button
myTextBox.render(); //renders a text box
```

Classi astratte vs interfacce

Le classi e le interfacce astratte forniscono entrambi un modo per definire le firme dei metodi, mentre richiedono la classe di estensione / implementazione per fornire l'implementazione.

Ci sono due differenze chiave tra classi astratte e interfacce:

- Una classe può estendere solo una singola classe, ma può implementare molte interfacce.
- Una classe astratta può contenere campi di istanza (non `static`), ma le interfacce possono contenere solo campi `static` .

Java SE 8

I metodi dichiarati nelle interfacce non potevano contenere implementazioni, quindi sono state utilizzate classi astratte quando era utile fornire metodi aggiuntivi quali implementazioni chiamate metodi astratti.

Java SE 8

Java 8 consente alle interfacce di contenere metodi predefiniti , solitamente implementati utilizzando gli altri metodi dell'interfaccia , rendendo le interfacce e le classi astratte ugualmente potenti in questo senso.

Sottoclassi anonime di classi astratte

Per comodità java consente l'istanziazione di istanze anonime di sottoclassi di classi astratte, che forniscono implementazioni per i metodi astratti sulla creazione del nuovo oggetto. Usando l'esempio precedente questo potrebbe assomigliare a questo:

```
Component myAnonymousComponent = new Component() {
    @Override
    public void render() {
        // render a quick 1-time use component
    }
}
```

Eredità statica

Il metodo statico può essere ereditato in modo simile ai metodi normali, tuttavia a differenza dei metodi normali è impossibile creare metodi " astratti " per forzare l'override del metodo statico. Scrivere un metodo con la stessa firma di un metodo statico in una super classe sembra essere una forma di sovrascrittura, ma in realtà ciò crea semplicemente una nuova funzione che nasconde l'altro.

```
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");
    }

}
```

L'esecuzione di una di queste classi produce l'output:

```
Hello
BaseClass's num: 5
Hello
Hey
Static says Hi
StaticOverride's num: test
```

Si noti che, a differenza dell'ereditarietà normale, i metodi di ereditarietà statica non sono nascosti. Puoi sempre chiamare il metodo `sayHello` base usando `BaseClass.sayHello()` . Ma le classi ereditano metodi statici *se* nella sottoclasse non vengono trovati metodi con la stessa firma. Se le firme di due metodi variano, entrambi i metodi possono essere eseguiti dalla sottoclasse, anche se il nome è lo stesso.

I campi statici si nascondono l'un l'altro in modo simile.

Usare 'final' per limitare l'ereditarietà e la sovrascrittura

Classi finali

Se utilizzato in una dichiarazione di `class` , il modificatore `final` impedisce che vengano dichiarate altre classi che `extend` la classe. Una classe `final` è una classe "leaf" nella gerarchia delle classi di ereditarietà.

```
// This declares a final class
final class MyFinalClass {
    /* some code */
}

// Compilation error: cannot inherit from final MyFinalClass
class MySubClass extends MyFinalClass {
    /* more code */
```

```
}
```

Casi d'uso per le classi finali

Le classi finali possono essere combinate con un costruttore `private` per controllare o impedire l'istanziazione di una classe. Questo può essere usato per creare una cosiddetta "utility class" che definisce solo membri statici; cioè costanti e metodi statici.

```
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;
    }

}
```

Anche le classi immutabili dovrebbero essere dichiarate `final` . (Una classe immutabile è quella le cui istanze non possono essere modificate dopo che sono state create, vedere l'argomento I mmutable Objects .) Facendo ciò, è impossibile creare una sottoclasse mutabile di una classe immutabile. Ciò violerebbe il Principio di sostituzione di Liskov che richiede che un sottotipo obbedisca al "contratto comportamentale" dei suoi supertipi.

Dal punto di vista pratico, dichiarare che una classe immutabile è `final` rende più facile ragionare sul comportamento del programma. Affronta anche i problemi di sicurezza nello scenario in cui un codice non affidabile viene eseguito in una sandbox di sicurezza. (Ad esempio, dal momento che `String` è dichiarata `final` , una classe fidata non ha bisogno di preoccuparsi che possa essere ingannato ad accettare sottoclassi mutabili, che il chiamante non sicuro potrebbe quindi cambiare surrettiziamente).

Uno svantaggio delle classi `final` è che non funzionano con alcuni framework di derisione come Mockito. Aggiornamento: la versione 2 di Mockito ora supporta il mocking delle classi finali.

Metodi finali

Il modificatore `final` può anche essere applicato ai metodi per impedire che vengano sovrascritti nelle sottoclassi:

```
public class MyClassWithFinalMethod {

    public final void someMethod() {
    }
}

public class MySubClass extends MyClassWithFinalMethod {

    @Override
```

```
    public void someMethod() { // Compiler error (overridden method is final)
    }
}
```

I metodi finali vengono in genere utilizzati quando si desidera limitare ciò che una sottoclasse può modificare in una classe senza proibire completamente le sottoclassi.

Il modificatore `final` può essere applicato anche alle variabili, ma il significato di `final` per le variabili non è correlato all'ereditarietà.

Il principio di sostituzione di Liskov

La sostituibilità è un principio nella programmazione orientata agli oggetti introdotta da Barbara Liskov in una conferenza del 1987 che afferma che, se la classe `B` è una sottoclasse di classe `A` , allora dovunque `A` è previsto, `B` può essere usato invece:

```
class A {...}
class B extends A {...}

public void method(A obj) {...}

A a = new B(); // Assignment OK
method(new B()); // Passing as parameter OK
```

Ciò si applica anche quando il tipo è un'interfaccia, in cui non è necessario alcun rapporto gerarchico tra gli oggetti:

```
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
```

Ora l'elenco contiene oggetti che non appartengono alla stessa gerarchia di classi.

Eredità

Con l'uso della parola chiave `extends` tra classi, tutte le proprietà della superclasse (anche note come *Parent Class* o *Base Class*) sono presenti nella sottoclasse (anche conosciuta come *Child Class* o *Derived Class*)

```
public class BaseClass {

    public void baseMethod(){
        System.out.println("Doing base class stuff");
    }
}

public class SubClass extends BaseClass {

}
```

Le istanze di `SubClass` hanno *ereditato* il metodo `baseMethod()` :

```
SubClass s = new SubClass();
s.baseMethod();  //Valid, prints "Doing base class stuff"
```

Contenuti aggiuntivi possono essere aggiunti a una sottoclasse. Ciò consente funzionalità aggiuntive nella sottoclasse senza alcuna modifica alla classe base o alle altre sottoclassi della stessa classe 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"
```

I campi sono anche ereditati:

```
public class BaseClassWithField {

    public int x;

}

public class SubClassWithField extends BaseClassWithField {

    public SubClassWithField(int x) {
        this.x = x; //Can access fields
    }
}
```

campi e metodi `private` esistono ancora all'interno della sottoclasse, ma non sono accessibili:

```
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.
```

In Java, ogni classe può estendere al massimo una classe.

```
public class A{}
public class B{}
public class ExtendsTwoClasses extends A, B {} //Illegal
```

Questo è noto come ereditarietà multipla, e mentre è legale in alcune lingue, Java non lo consente con le classi.

Come risultato di ciò, ogni classe ha una catena ancestrale di classi che conduce ad `Object` , da cui tutte le classi discendono.

Ereditarietà e metodi statici

In Java, entrambe le classi genitore e figlio possono avere metodi statici con lo stesso nome. Ma in questi casi l'implementazione del metodo statico in child sta nascondendo l'implementazione della classe genitore, non è un metodo prioritario. Per esempio:

```
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");
    }
  }
}
```

I metodi statici sono associati a una classe non a un'istanza e questo binding di metodo avviene in fase di compilazione. Dal momento che nella prima chiamata a `staticMethod()` , genitore di

riferimento di classe `p` è stato utilizzato, `Parent` versione s' di `staticMethod()` viene richiamato. Nel secondo caso, si gettò `p` in `Child` di classe, `Child` 's `staticMethod()` eseguito.

Ombreggiamento variabile

Le variabili sono SHADOWED e i metodi sono OVERRIDDEN. Quale variabile verrà utilizzata dipende dalla classe di dichiarata della variabile. Quale metodo verrà utilizzato dipende dalla classe effettiva dell'oggetto a cui fa riferimento la variabile.

```
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
    }
}
```

Restringimento e ampliamento dei riferimenti a oggetti

Trasmettere un'istanza di una classe base ad una sottoclasse come in: `b = (B) a;` è chiamato *restringimento* (come si sta tentando di restringere l'oggetto della classe base a un oggetto di classe più specifico) e ha bisogno di un cast di tipo esplicito.

Trasmettere un'istanza di una sottoclasse a una classe base come in: `A a = b;` si chiama *allargamento* e non ha bisogno di un cast di tipo.

Per illustrare, considera le seguenti dichiarazioni di classe e il codice di prova:

```
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 dichiarazione `Vehicle vehicle = new Car();` è una dichiarazione Java valida. Ogni istanza di `Car` è anche un `Vehicle` . Pertanto, l'assegnazione è legale senza la necessità di un cast esplicito del tipo.

D'altra parte, `Car c = vehicle;` non è valido. Il tipo statico della variabile del `vehicle` è `Vehicle` che significa che potrebbe fare riferimento a un'istanza di `Car` , Truck , MotorCycle , `or any other current or future subclass of` veicolo . `(Or indeed, an instance of` Veicolo `itself, since we did not declare it as an class.) The assignment cannot be allowed, since that might lead to` astratta `class.) The assignment cannot be allowed, since that might lead to` all'automobile che si `referring to a` un'istanza di Truck`.

Per evitare questa situazione, dobbiamo aggiungere un cast di tipo esplicito:

```
Car c = (Car) vehicle;
```

Il cast del tipo dice al compilatore che ci *aspettiamo che* il valore del `vehicle` sia `Car` o una sottoclasse di `Car` . Se necessario, il compilatore inserirà il codice per eseguire un controllo del tipo di esecuzione. Se il controllo fallisce, verrà generata una `ClassCastException` momento dell'esecuzione del codice.

Nota che non tutti i cast di tipi sono validi. Per esempio:

```
String s = (String) vehicle;  // not valid
```

Il compilatore Java sa che un'istanza di tipo compatibile con `Vehicle` *non può mai essere* compatibile con `String` . Il cast del tipo non potrebbe mai avere successo, e il JLS impone che questo dia in un errore di compilazione.

Programmazione su un'interfaccia

L'idea alla base della programmazione di un'interfaccia è di basare il codice principalmente sulle interfacce e utilizzare solo classi concrete al momento dell'istanziazione. In questo contesto, un

buon codice che riguarda, ad esempio, le raccolte Java avrà un aspetto simile a questo (non che il metodo stesso sia di qualche utilità, solo illustrazione):

```
public <T> Set<T> toSet(Collection<T> collection) {
  return Sets.newHashSet(collection);
}
```

mentre il codice errato potrebbe apparire come questo:

```
public <T> HashSet<T> toSet(ArrayList<T> collection) {
  return Sets.newHashSet(collection);
}
```

Non solo il primo può essere applicato a una più ampia scelta di argomenti, i suoi risultati saranno più compatibili con il codice fornito da altri sviluppatori che generalmente aderiscono al concetto di programmazione di un'interfaccia. Tuttavia, i motivi più importanti per utilizzare il primo sono:

- il più delle volte il contesto in cui viene utilizzato il risultato non richiede e non dovrebbe richiedere molti dettagli come prevede l'implementazione concreta;
- aderire a un'interfaccia costringe un codice più pulito e meno hacks come un altro metodo pubblico viene aggiunto a una classe che serve uno scenario specifico;
- il codice è più testabile in quanto le interfacce sono facilmente raggirabili;
- infine, il concetto aiuta anche se è prevista solo un'implementazione (almeno per testabilità).

Quindi, come si può facilmente applicare il concetto di programmazione a un'interfaccia quando si scrive un nuovo codice avendo in mente una particolare implementazione? Un'opzione che usiamo comunemente è una combinazione dei seguenti modelli:

- programmazione su un'interfaccia
- fabbrica
- costruttore

L'esempio seguente basato su questi principi è una versione semplificata e troncata di un'implementazione RPC scritta per un numero di protocolli diversi:

```
public interface RemoteInvoker {
  <RQ, RS> CompletableFuture<RS> invoke(RQ request, Class<RS> responseClass);
}
```

L'interfaccia di cui sopra non dovrebbe essere istanziata direttamente tramite una factory, ma deriviamo ulteriori interfacce concrete, una per l'invocazione HTTP e una per AMQP, ognuna delle quali ha una factory e un builder per costruire istanze, che a loro volta sono anche esempi di l'interfaccia di cui sopra:

```
public interface AmqpInvoker extends RemoteInvoker {
  static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
    return new AmqpInvokerBuilder(instanceId, factory);
  }
}
```

Le istanze di `RemoteInvoker` per l'uso con AMQP possono ora essere costruite con la stessa facilità con cui (o sono più coinvolte a seconda del costruttore):

```
RemoteInvoker invoker = AmqpInvoker.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

E l'invocazione di una richiesta è facile come:

```
Response res = invoker.invoke(new Request(data), Response.class).get();
```

A causa di Java 8 che consente il posizionamento di metodi statici direttamente nelle interfacce, lo stabilimento intermedio è diventato implicito nel codice sopra sostituito con `AmqpInvoker.with()` . In Java precedente alla versione 8, lo stesso effetto può essere ottenuto con una classe `Factory` interna:

```
public interface AmqpInvoker extends RemoteInvoker {
  class Factory {
    public static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
      return new AmqpInvokerBuilder(instanceId, factory);
    }
  }
}
```

L'istanziazione corrispondente si trasformerebbe quindi in:

```
RemoteInvoker invoker = AmqpInvoker.Factory.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

Il builder usato sopra potrebbe apparire come questo (sebbene si tratti di una semplificazione in quanto quello attuale consente di definire fino a 15 parametri che si discostano dai valori predefiniti). Nota che il costrutto non è pubblico, quindi è specificamente utilizzabile solo dalla precedente interfaccia di `AmqpInvoker` :

```
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);
  }
}
```

Generalmente, un builder può anche essere generato usando uno strumento come FreeBuilder.

Infine, l'implementazione standard (e l'unica prevista) di questa interfaccia è definita come una classe package-local per imporre l'uso dell'interfaccia, della factory e del builder:

```
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) {
    ...
  }
}
```

Nel frattempo, questo schema si è dimostrato molto efficace nello sviluppo di tutto il nostro nuovo codice, non importa quanto semplice o complessa sia la funzionalità.

Uso astratto della classe e dell'interfaccia: capacità "Is-a" rispetto a "Has-a"

Quando utilizzare le classi astratte: per implementare lo stesso comportamento o un comportamento diverso tra più oggetti correlati

Quando utilizzare le interfacce: per implementare un contratto con più oggetti non correlati

Le classi astratte creano relazioni "è un" mentre le interfacce forniscono "ha" una capacità.

Questo può essere visto nel codice qui sotto:

```
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;
    }
}
```

produzione:

```
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
```

Note chiave:

1. `Animal` è una classe astratta con attributi condivisi: `name` e `lifeExpectancy` Metodi `lifeExpectancy` e astratti: `remember()` e `protectOwner()` . `Dog` e `Cat` sono `Animals` che hanno implementato i metodi `remember()` e `protectOwner()` .

2. `Cat` può `climb()` ma `Dog` non può. `Dog` può `think()` ma `Cat` non può. Queste funzionalità specifiche vengono aggiunte a `Cat` and `Dog` mediante l'implementazione.

3. `Man` non è un `Animal` ma può `Think` , `Learn` , `Apply` e `Climb` .

4. `Cat` non è un `Man` ma può `Climb` .

5. `Dog` non è un `Man` ma può `Learn`

6. `Man` non è né un `Cat` né un `Dog` ma può avere alcune delle capacità degli ultimi due senza estendere `Animal` , `Cat` o `Dog` . Questo è fatto con le interfacce.

7. Anche se `Animal` è una classe astratta, ha un costruttore, a differenza di un'interfaccia.

TL; DR:

Le classi non correlate possono avere capacità attraverso le interfacce, ma le classi correlate cambiano il comportamento attraverso l'estensione delle classi di base.

Fare riferimento alla pagina della documentazione Java per capire quale utilizzare in un caso d'uso specifico.

Prendi in considerazione l'utilizzo di classi astratte se ...

1. Vuoi condividere il codice tra diverse classi strettamente correlate.
2. Ti aspetti che le classi che estendono la tua classe astratta abbiano molti metodi o campi comuni o richiedano modificatori di accesso diversi da quelli pubblici (come protetti e privati).
3. Si desidera dichiarare campi non statici o non finali.

Prendi in considerazione l'utilizzo di interfacce se ...

1. Ti aspetti che le classi non correlate implementino la tua interfaccia. Ad esempio, molti oggetti non collegati possono implementare l'interfaccia `Serializable` .
2. Si desidera specificare il comportamento di un particolare tipo di dati ma non si preoccupano di chi implementa il suo comportamento.
3. Vuoi sfruttare l'ereditarietà multipla del tipo.

Overriding in Inheritance

Overriding in Inheritance viene utilizzato quando si utilizza un metodo già definito da una super classe in una sottoclasse, ma in un modo diverso rispetto al modo in cui il metodo è stato originariamente progettato nella super classe. L'override consente all'utente di riutilizzare il codice utilizzando il materiale esistente e modificandolo in base alle esigenze dell'utente.

L'esempio seguente mostra come `ClassB` sovrascrive la funzionalità di `ClassA` modificando ciò che viene inviato tramite il metodo di stampa:

Esempio:

```
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");
    }
}
```

Produzione:

UN

B

Leggi Eredità online: https://riptutorial.com/it/java/topic/87/eredita

Capitolo 58: espressioni

introduzione

Le espressioni in Java sono il costrutto principale per fare calcoli.

Osservazioni

Per un riferimento sugli operatori che possono essere utilizzati nelle espressioni, vedere Operatori .

Examples

Precedenza dell'operatore

Quando un'espressione contiene più operatori, può potenzialmente essere letta in diversi modi. Ad esempio, l'espressione matematica `1 + 2 x 3` potrebbe essere letta in due modi:

1. Aggiungi `1` e `2` e moltiplica il risultato per `3` . Questo dà la risposta `9` . Se aggiungessimo le parentesi, sembrerebbe `( 1 + 2 ) x 3` .
2. Aggiungi `1` al risultato di moltiplicare `2` e `3` . Questo dà la risposta `7` . Se aggiungessimo le parentesi, sembrerebbe `1 + ( 2 x 3 )` .

In matematica, la convenzione è leggere l'espressione nel secondo modo. La regola generale è che la moltiplicazione e la divisione vengono eseguite prima dell'addizione e della sottrazione. Quando viene usata una notazione matematica più avanzata, o il significato è "autoevidente" (per un matematico addestrato!), O si aggiungono parentesi per disambiguare. In entrambi i casi, l'efficacia della notazione per trasmettere il significato dipende dall'intelligenza e dalla conoscenza condivisa dei matematici.

Java ha le stesse regole chiare su come leggere un'espressione, in base alla *precedenza* degli operatori che vengono utilizzati.

In generale, a ciascun operatore viene attribuito un valore di *precedenza* ; vedi la tabella qui sotto.

Per esempio:

```
1 + 2 * 3
```

La precedenza di `+` è inferiore alla precedenza di `*` , quindi il risultato dell'espressione è 7, non 9.

Descrizione	Operatori / costrutti (primario)	Precedenza	Associatività

Qualifier parentesi	nome `.` nome
	`(` expr `)`
	`new`
		15	Da sinistra a destra

Descrizione	Operatori / costrutti (primario)	Precedenza	Associatività
Creazione di istanze Accesso al campo Accesso alla matrice Invocazione del metodo Metodo di riferimento	 primario `.` nome primario `[` expr `]` primario `(` expr, ... `)` primario `::` nome		
Incremento della posta	expr `++` , expr `--`	14	-
Pre incremento unario Cast [1]	`++` expr, `--` expr, `+` Expr, `-` espr, `~` espr, `!` expr, `(` tipo `)` expr	13	- Da destra a sinistra Da destra a sinistra
moltiplicativo	* /%	12	Da sinistra a destra
Additivo	+ -	11	Da sinistra a destra
Cambio	<< >> >>>	10	Da sinistra a destra
relazionale	<> <=> = `instanceof`	9	Da sinistra a destra
Uguaglianza	==! =	8	Da sinistra a destra
Bitwise AND	&	7	Da sinistra a destra
OR esclusivo bit per bit	^	6	Da sinistra a destra
Bitwise incluso OR	|	5	Da sinistra a destra
AND logico	&&	4	Da sinistra a destra
OR logico	||	3	Da sinistra a destra

Descrizione	Operatori / costrutti (primario)	Precedenza	Associatività
Condizionale [1]	? :	2	Da destra a sinistra
assegnazione Lambda [1]	= * = / =% = + = - = << = >> = >>> = & = ^ = | = ->	1	Da destra a sinistra

[1] La precedenza dell'espressione lambda è complessa, in quanto può verificarsi anche dopo un cast o come terza parte dell'operatore ternario condizionale.

Espressioni costanti

Un'espressione costante è un'espressione che produce un tipo primitivo o una stringa e il cui valore può essere valutato al momento della compilazione su un valore letterale. L'espressione deve valutare senza generare un'eccezione e deve essere composta solo dal seguente:

- Letterali primitivi e stringhe.
- Digita i cast ai tipi primitivi o `String` .
- I seguenti operatori unari: `+` , `-` , `~` e `!` .
- I seguenti operatori binari: `*` , `/` , `%` , `+` , `-` , `<<` , `>>` , `>>>` , `<` , `<=` , `>` , `>=` , `==` `!=` , `&` , `^` , `|` , `&&` e `||` .
- L'operatore condizionale ternario `?` `:`
- Espressioni costanti parentesi
- Nomi semplici che si riferiscono a variabili costanti. (Una variabile costante è una variabile dichiarata come `final` cui l'espressione di inizializzazione è di per sé un'espressione costante).
- Nomi qualificati del modulo `<TypeName>` . `<Identifier>` che si riferisce a variabili costanti.

Si noti che l'elenco precedente *esclude* `++` e `--` , gli operatori di assegnazione, `class` e `instanceof` , chiamate di metodi e riferimenti a variabili o campi generali.

Espressioni costanti di tipo `String` risultato in un "internati" `String` , e operazioni a virgola mobile a espressioni costanti vengono valutate con FP-rigide semantica.

Utilizza per le espressioni costanti

Le espressioni costanti possono essere utilizzate (quasi) ovunque sia possibile utilizzare un'espressione normale. Tuttavia, hanno un significato speciale nei seguenti contesti.

Le espressioni costanti sono obbligatorie per le etichette case nelle istruzioni switch. Per esempio:

```
switch (someValue) {
case 1 + 1:             // OK
case Math.min(2, 3):    // Error - not a constant expression
    doSomething();
}
```

Quando l'espressione sul lato destro di un compito è un'espressione costante, il compito può eseguire una conversione restringimento primitiva. Ciò è consentito a condizione che il valore dell'espressione costante rientri nell'intervallo del tipo sul lato sinistro. (Vedi JLS 5.1.3 e 5.2) Ad esempio:

```
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
```

Quando un'espressione costante viene utilizzata come condizione in una `do` , `while` o `for` , influisce sull'analisi di leggibilità. Per esempio:

```
while (false) {
    doSomething();            // Error - statenent not reachable
}
boolean flag = false;
while (flag) {
    doSomething();            // OK
}
```

(Si noti che questo non si applica `if` istruzioni.Il compilatore Java consente al blocco `then` o `else` di un'istruzione `if` di essere irraggiungibile.Questo è l'analogo Java della compilazione condizionale in C e C ++.)

Infine, `static final` campi `static final` di una classe o di un'interfaccia con inizializzatori di espressioni costanti vengono inizializzati con entusiasmo. Pertanto, è garantito che queste costanti saranno osservate nello stato inizializzato, anche quando c'è un ciclo nel grafico delle dipendenze di inizializzazione della classe.

Per ulteriori informazioni, fare riferimento a JLS 15.28. Espressioni costanti .

Ordine di valutazione dell'espressione

Le espressioni Java sono valutate seguendo le seguenti regole:

- Gli operandi sono valutati da sinistra a destra.
- Gli operandi di un operatore vengono valutati prima dell'operatore.
- Gli operatori vengono valutati in base alla precedenza degli operatori
- Gli elenchi degli argomenti vengono valutati da sinistra a destra.

Semplice esempio

Nell'esempio seguente:

```
int i = method1() + method2();
```

l'ordine di valutazione è:

1. L'operando di sinistra di `=` operator viene valutato all'indirizzo di `i` .
2. Viene valutato l'operando di sinistra dell'operatore `+` (`method1()`).
3. Viene valutato l'operando destro dell'operatore `+` (`method2()`).
4. L'operazione `+` viene valutata.
5. L'operazione `=` viene valutata, assegnando il risultato dell'aggiunta a `i` .

Notare che se gli effetti delle chiamate sono osservabili, si sarà in grado di osservare che la chiamata al `method1` verifica prima della chiamata a `method2` .

Esempio con un operatore che ha un effetto collaterale

Nell'esempio seguente:

```
int i = 1;
intArray[i] = ++i + 1;
```

l'ordine di valutazione è:

1. Viene valutato l'operando di sinistra di `=` operatore. Questo dà l'indirizzo di `intArray[1]` .
2. Il pre-incremento viene valutato. Questo aggiunge `1` a `i` e valuta a `2` .
3. Viene valutato l'operando di destra del `+` .
4. L'operazione `+` viene valutata su: `2 + 1` -> `3` .
5. L'operazione `=` viene valutata, assegnando `3` a `intArray[1]` .

Si noti che poiché l'operando di sinistra di `=` viene valutato per primo, non è influenzato dall'effetto collaterale della sottoespressione di `++i` .

Riferimento:

- JLS 15.7 - Ordine di valutazione

Nozioni di base sull'espressione

Le espressioni in Java sono il costrutto principale per fare calcoli. Ecco alcuni esempi:

```
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
```

In generale, un'espressione è costituita dalle seguenti forme:

- Nomi delle espressioni che consistono in:
  - Identificatori semplici; ad esempio `someIdentifier`
  - Identificatori qualificati; es. `MyClass.someField`
- Primari che consistono in:
  - letterali; ad es. `1` , `1.0` , `'X'` , `"hello"` , `false` e `null`
  - Espressioni letterali di classe; es. `MyClass.class`
  - `this` e `<TypeName> . this`
  - Espressioni parentesi; es. `( a + b )`
  - Espressioni di creazione di istanze di classe; es. `new MyClass(1, 2, 3)`
  - Espressioni di creazione di istanze di array; ad esempio `new int[3]`
  - Espressioni di accesso al campo; ad esempio `obj.someField` o `this.someField`
  - Espressioni di accesso alla matrice; es. `vector[21]`
  - Invocazioni di metodi; es. `obj.doIt(1, 2, 3)`
  - Riferimenti al metodo (Java 8 e versioni successive); ad es. `MyClass::doIt`
- Espressioni di operatore unario; eg `!a` o `i++`
- Espressioni di operatori binari; es. `a + b obj == null`
- Espressioni dell'operatore ternario; ad esempio `(obj == null) ? 1 : obj.getCount()`
- Espressioni Lambda (Java 8 e successive); es. `obj -> obj.getCount()`

I dettagli delle diverse forme di espressione possono essere trovati in altri argomenti.

- L'argomento Operatori copre espressioni di operatori unari, binari e ternari.
- L'argomento espressioni Lambda copre espressioni lambda e espressioni di riferimento del metodo.
- L'argomento Classi e oggetti copre le espressioni di creazione di istanze di classe.
- L'argomento Array copre espressioni di accesso alla matrice e espressioni di creazione di istanze di array.
- L'argomento Literals copre i diversi tipi di espressioni letterali.

Il tipo di un'espressione

Nella maggior parte dei casi, un'espressione ha un tipo statico che può essere determinato in fase di compilazione esaminando e le sue sottoespressioni. Questi sono indicati come espressioni *stand-alone* .

Tuttavia, (in Java 8 e versioni successive) i seguenti tipi di espressioni possono essere *espressioni poligonali* :

- Espressioni parentesi
- Espressioni di creazione di istanze di classe
- Espressioni di richiamo del metodo
- Espressioni di riferimento del metodo
- Espressioni condizionali
- Espressioni Lambda

Quando un'espressione è un'espressione poli, il suo tipo può essere influenzato dal *tipo di destinazione* dell'espressione; cioè per cosa viene usato.

Il valore di un'espressione

Il valore di un'espressione è l'assegnazione compatibile con il suo tipo. L'eccezione a questo è quando si è verificato l' *inquinamento da cumulo* ; ad es. perché gli avvisi di "non sicura conversione" sono stati (inappropriatamente) soppressi o ignorati.

Dichiarazioni di espressione

A differenza di molti altri linguaggi, in genere Java non consente di utilizzare espressioni come istruzioni. Per esempio:

```
public void compute(int i, int j) {
    i + j;   // ERROR
}
```

Poiché il risultato della valutazione di un'espressione come non può essere utilizzato e poiché non può influenzare l'esecuzione del programma in alcun altro modo, i progettisti Java hanno preso la posizione che tale utilizzo è un errore o un errore.

Tuttavia, questo non si applica a tutte le espressioni. Un sottoinsieme di espressioni sono (di fatto) legali come affermazioni. Il set comprende:

- Espressione dell'assegnazione, compresi gli incarichi *operativi e gli* incarichi.
- Pre e post espressioni di incremento e decremento.
- Chiamate di metodo (`void` o non `void`).
- Espressioni di creazione di istanze di classe.

Leggi espressioni online: https://riptutorial.com/it/java/topic/8167/espressioni

Capitolo 59: Espressioni regolari

introduzione

Un'espressione regolare è una sequenza speciale di caratteri che aiuta a trovare corrispondenze o trovare altre stringhe o gruppi di stringhe, utilizzando una sintassi specializzata contenuta in un modello. Java ha il supporto per l'uso regolare delle espressioni tramite il pacchetto `java.util.regex` . Questo argomento è quello di introdurre e aiutare gli sviluppatori a capire di più con esempi su come le espressioni regolari devono essere utilizzate in Java.

Sintassi

- Pattern patternName = Pattern.compile (regex);
- Matcher matcherName = patternName.matcher (textToSearch);
- matcherName.matches () // Restituisce vero se textToSearch corrisponde esattamente all'espressione regolare
- matcherName.find () // Cerca tra textToSearch per la prima istanza di una sottostringa che corrisponde alla regex. Le chiamate successive cercheranno il resto della stringa.
- matcherName.group (groupNum) // Restituisce la sottostringa all'interno di un gruppo di cattura
- matcherName.group (groupName) // Restituisce la sottostringa all'interno di un gruppo di acquisizione denominato (Java 7+)

Osservazioni

importazioni

Prima di poter utilizzare Regex è necessario aggiungere le seguenti importazioni:

```
import java.util.regex.Matcher
import java.util.regex.Pattern
```

insidie

In java, una barra rovesciata viene sfuggita con una doppia barra rovesciata, quindi una barra rovesciata nella stringa regex deve essere immessa come una doppia barra rovesciata. Se è necessario sfuggire a una doppia barra rovesciata (per abbinare una singola barra inversa con la regex, è necessario inserirla come una barra rovesciata quadrupla.

Simboli importanti spiegati

Personaggio	Descrizione
`*`	Abbina il carattere precedente o la sottoespressione 0 o più volte
`+`	Abbina il carattere o la sottoespressione precedente 1 o più volte
`?`	Abbina il carattere precedente o la sottoespressione 0 o 1 volte

Ulteriori letture

L' argomento espressioni regolari contiene ulteriori informazioni sulle espressioni regolari.

Examples

Utilizzo dei gruppi di cattura

Se è necessario estrarre una parte di stringa dalla stringa di input, possiamo usare i **gruppi** di **cattura** di regex.

Per questo esempio, inizieremo con un semplice numero di telefono regex:

```
\d{3}-\d{3}-\d{4}
```

Se le parentesi sono aggiunte alla regex, ciascuna serie di parentesi viene considerata un *gruppo di cattura* . In questo caso, stiamo usando quelli che sono chiamati gruppi di cattura numerati:

```
(\d{3})-(\d{3})-(\d{4})
^-----^ ^-----^ ^-----^
Group 1 Group 2 Group 3
```

Prima di poterlo utilizzare in Java, non dobbiamo dimenticare di seguire le regole di Stringhe, sfuggire alle barre inverse, con il seguente schema:

```
"(\\d{3})-(\\d{3})-(\\d{4})"
```

Per prima cosa dobbiamo compilare il modello regex per creare un `Pattern` e quindi abbiamo bisogno di un `Matcher` per abbinare la nostra stringa di input con il pattern:

```
Pattern phonePattern = Pattern.compile("(\\d{3})-(\\d{3})-(\\d{4})");
Matcher phoneMatcher = phonePattern.matcher("abcd800-555-1234wxyz");
```

Successivamente, il Matcher deve trovare la prima sottosequenza che corrisponde alla regex:

```
phoneMatcher.find();
```

Ora, usando il metodo di gruppo, possiamo estrarre i dati dalla stringa:

```
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()` può essere utilizzato al posto di `Matcher.group(0)` .

Java SE 7

Java 7 ha introdotto i gruppi di acquisizione denominati. I gruppi di cattura con nome funzionano allo stesso modo dei gruppi di cattura numerati (ma con un nome anziché un numero), sebbene vi siano lievi modifiche alla sintassi. L'utilizzo dei gruppi di acquisizione con nome migliora la leggibilità.

Possiamo modificare il codice precedente per utilizzare i gruppi denominati:

```
(?<AreaCode>\d{3})-(\d{3})-(\d{4})
^----------------^ ^-----^ ^-----^
AreaCode           Group 2 Group 3
```

Per ottenere il contenuto di "AreaCode", possiamo invece utilizzare:

```
String aCode = phoneMatcher.group("AreaCode"); //"800"
```

Usare espressioni regolari con comportamento personalizzato compilando Pattern con flag

Un `Pattern` può essere compilato con flag, se la regex è usata come `String` letterale, usa i modificatori in linea:

```
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".
```

Personaggi di fuga

Generalmente

Per usare caratteri specifici dell'espressione regolare (`?+|` Ecc.) Nel loro significato letterale, devono essere sfuggiti. Nell'espressione regolare comune ciò è fatto da una barra rovesciata `\` . Tuttavia, poiché ha un significato speciale in Java Strings, devi usare una doppia barra rovesciata `\\` .

Questi due esempi non funzioneranno:

```
"???".replaceAll ("?", "!"); //java.util.regex.PatternSyntaxException
"???".replaceAll ("\?", "!"); //Invalid escape sequence
```

Questo esempio funziona

```
"???".replaceAll ("\\?", "!"); //"!!!"
```

Divisione di una stringa delimitata da un tubo

Questo non restituisce il risultato atteso:

```
"a|b".split ("|"); // [a, |, b]
```

Ciò restituisce il risultato previsto:

```
"a|b".split ("\\|"); // [a, b]
```

Escaping backslash `\`

Questo darà un errore:

```
"\\".matches("\\"); // PatternSyntaxException
"\\".matches("\\\"); // Syntax Error
```

Questo funziona:

```
"\\".matches("\\\\"); // true
```

Corrispondenza con una regex letterale.

Se è necessario abbinare i caratteri che fanno parte della sintassi delle espressioni regolari, è possibile contrassegnare tutto o parte del modello come un regex letterale.

`\Q` segna l'inizio della regex letterale. `\E` segna la fine della regex letterale.

```
// 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
```

Un modo più semplice per farlo senza dover ricordare le sequenze di escape `\Q` e `\E` è usare

`Pattern.quote()`

```
"[123".matches(Pattern.quote("[") + "123"); // returns true
```

Non corrisponde a una determinata stringa

Per abbinare qualcosa che *non* contiene una determinata stringa, si può usare lookahead negativo:

Sintassi Regex: `(?!string-to-not-match)`

Esempio:

```
//not matching "popcorn"
String regexString = "^(?!popcorn).*$";
System.out.println("[popcorn] " + ("popcorn".matches(regexString) ? "matched!" : "nope!"));
System.out.println("[unicorn] " + ("unicorn".matches(regexString) ? "matched!" : "nope!"));
```

Produzione:

```
[popcorn] nope!
[unicorn] matched!
```

Abbinare una barra rovesciata

Se vuoi abbinare una barra rovesciata nella tua espressione regolare, dovrai fuggire.

Il backslash è un carattere di escape nelle espressioni regolari. Puoi usare '\\' per fare riferimento a una singola barra rovesciata in un'espressione regolare.

Tuttavia, il backslash è *anche* un carattere di escape nelle stringhe letterali Java. Per fare un'espressione regolare da una stringa letterale, devi sfuggire a ciascuna delle *sue* barre retroverse. In una stringa letterale '\\\\' può essere usato per creare un'espressione regolare con '\\', che a sua volta può corrispondere a '\'.

Ad esempio, considera l'abbinamento di stringhe come "C: \ dir \ myfile.txt". Un'espressione regolare `([A-Za-z]):\\(.*)` Corrisponderà e fornirà la lettera di unità come gruppo di cattura. Notare il backslash raddoppiato.

Per esprimere quel modello in una stringa letterale Java, è necessario che tutti i backslash nell'espressione regolare debbano essere sottoposti a escape.

```
    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:.
    }
```

Se vuoi abbinare *due* backslash, ti troverai usando otto in una stringa letterale, per rappresentare quattro nell'espressione regolare, per abbinarne due.

```
    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'.
    }
```

Leggi Espressioni regolari online: https://riptutorial.com/it/java/topic/135/espressioni-regolari

Capitolo 60: File I / O

introduzione

I / O Java (Input e Output) viene utilizzato per elaborare l'input e produrre l'output. Java usa il concetto di stream per rendere veloce l'operazione I / O. Il pacchetto java.io contiene tutte le classi necessarie per le operazioni di input e output. La gestione dei file viene eseguita anche in Java dall'API I / O Java.

Examples

Lettura di tutti i byte su un byte []

Java 7 ha introdotto la classe Files molto utile

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();
}
```

Leggere un'immagine da un file

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

...

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

Scrivere un byte [] in un file

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 maggior parte delle API di file java.io accetta sia `String` s che `File` s come argomenti, quindi si potrebbe anche usare

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

Stream vs Writer / Reader API

Gli stream forniscono l'accesso più diretto al contenuto binario, quindi qualsiasi implementazione `InputStream` / `OutputStream` funziona sempre su `int` s e `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);
}
```

Ci sono alcune eccezioni, probabilmente in particolare `PrintStream` che aggiunge la "capacità di stampare le rappresentazioni di vari valori di dati convenientemente". Ciò consente di utilizzare `System.out` sia come `InputStream` binario sia come output testuale utilizzando metodi come `System.out.println()` .

Inoltre, alcune implementazioni di stream funzionano come interfaccia per contenuti di livello superiore come oggetti Java (vedi Serializzazione) o tipi nativi, ad esempio `DataOutputStream` / `DataInputStream` .

Con le classi `Writer` e `Reader` , Java fornisce anche un'API per i flussi di caratteri espliciti. Sebbene la maggior parte delle applicazioni baserà queste implementazioni sui flussi, l'API del flusso di caratteri non espone alcun metodo per il contenuto 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();
```

Ogni volta che è necessario codificare i caratteri in dati binari (ad esempio quando si utilizzano le classi `InputStreamWriter` / `OutputStreamWriter`), è necessario specificare un set di caratteri se non si desidera dipendere dal set di caratteri predefinito della piattaforma. In caso di dubbi, utilizzare una codifica compatibile con Unicode, ad esempio UTF-8, supportata su tutte le piattaforme Java. Pertanto, si dovrebbe probabilmente stare lontano da classi come `FileWriter` e `FileReader` poiché utilizzano sempre il set di caratteri della piattaforma predefinito. Un modo migliore per accedere ai file utilizzando i flussi di caratteri è questo:

```
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 dei `Reader` più comunemente usati è `BufferedReader` che fornisce un metodo per leggere intere righe di testo da un altro lettore ed è presumibilmente il modo più semplice per leggere un flusso di caratteri riga per riga:

```
// 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);
}
```

Lettura di un intero file in una sola volta

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

\ Z è il simbolo EOF (Fine del file). Se impostato come delimitatore, lo scanner leggerà il riempimento fino al raggiungimento del flag EOF.

Lettura di un file con uno scanner

Leggere un file riga per riga

```
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();
        }
    }
}
```

parola per parola

```
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();
        }
    }
}
```

e puoi anche cambiare il delimitatore usando il metodo scanner.useDelimeter ()

Iterazione su una directory e filtro per estensione di file

```
    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());
```

```
            }
        }
    }
```

Migrazione da java.io.File a Java 7 NIO (java.nio.file.Path)

Questi esempi presumono che tu sappia già che cos'è NIO di Java 7 in generale e che sei abituato a scrivere codice usando `java.io.File` . Utilizzare questi esempi come mezzo per trovare rapidamente più documentazione NIO-centrica per la migrazione.

C'è molto di più nell'NIO di Java 7 come i file mappati in memoria o l' apertura di un file ZIP o JAR usando FileSystem . Questi esempi copriranno solo un numero limitato di casi d'uso di base.

Come regola di base, se si è abituati a eseguire un'operazione di lettura / scrittura di file system utilizzando un metodo di istanza `java.io.File` , lo si troverà come metodo statico all'interno di `java.nio.file.Files` .

Indica un percorso

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

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

Percorsi relativi ad un altro percorso

```
// 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");
```

Conversione di file da / a Path per l'uso con le librerie

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

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

Controlla se il file esiste ed eliminalo se lo fa

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

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

Scrivi su un file tramite un OutputStream

Esistono diversi modi per scrivere e leggere da un file utilizzando NIO per diverse prestazioni e vincoli di memoria, leggibilità e casi d'uso, come `FileChannel` , `Files.write(Path path, byte\[\] bytes, OpenOption... options)` . .. In questo esempio, viene coperto solo `OutputStream` , ma sei fortemente incoraggiato a conoscere i file mappati in memoria e i vari metodi statici disponibili in `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));
    }
}
```

Iterazione su ogni file all'interno di una cartella

```
// -> 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;
    }
});
```

Iterazione della cartella ricorsiva

```
// -> 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);
        }
    }
```

```
}
```

File lettura / scrittura utilizzando FileInputStream / FileOutputStream

Scrivi in un file 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();
}
```

Leggi dal file 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();
}
```

Si noti che da Java 1.7 è stata introdotta la dichiarazione try-with-resources che ha reso molto più semplice l'implementazione dell'operazione reading \ writing:

Scrivi in un file 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();
}
```

Leggi dal file 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();
}
```

Lettura da un file binario

Puoi leggere un file binario usando questo pezzo di codice in tutte le versioni recenti di 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();
```

Se si utilizza Java 7 o versioni successive, esiste un modo più semplice di utilizzare l' `nio API` :

Java SE 7

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

Blocco

Un file può essere bloccato utilizzando l'API `FileChannel` che può essere acquisita dai `streams` e dai `readers` Input Output

Esempio con `streams`

// Apre un flusso di file FileInputStream ios = new FileInputStream (filename);

```
    // 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
```

Copia di un file usando InputStream e OutputStream

Possiamo copiare direttamente i dati da un'origine a un data sink usando un loop. In questo esempio, stiamo leggendo i dati da un InputStream e, allo stesso tempo, scrivendo su un OutputStream. Una volta che abbiamo finito di leggere e scrivere, dobbiamo chiudere la risorsa.

```
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();
        }
    }
}
```

Leggere un file usando Channel e Buffer

`Channel` utilizza un `Buffer` per leggere / scrivere dati. Un buffer è un contenitore di dimensioni fisse in cui è possibile scrivere un blocco di dati contemporaneamente. `Channel` è molto più veloce di I / O basato sul flusso.

Per leggere i dati da un file utilizzando il `Channel` è necessario avere i seguenti passaggi:

1. Abbiamo bisogno di un'istanza di `FileInputStream` . `FileInputStream` ha un metodo chiamato `getChannel()` che restituisce un canale.
2. Chiama il metodo `getChannel()` di FileInputStream e acquisisci Canale.
3. Crea un ByteBuffer. ByteBuffer è un contenitore di byte di dimensioni fisse.

4. Il canale ha un metodo di lettura e dobbiamo fornire un ByteBuffer come argomento per questo metodo di lettura. ByteBuffer ha due modalità: umore di sola lettura e umore di sola scrittura. Possiamo cambiare la modalità usando la chiamata al metodo `flip()` . Il buffer ha una posizione, un limite e una capacità. Una volta creato un buffer con una dimensione fissa, il suo limite e la sua capacità sono uguali a quelli della dimensione e la posizione inizia da zero. Mentre un buffer è scritto con dati, la sua posizione aumenta gradualmente. Cambiare modalità significa, cambiare la posizione. Per leggere i dati dall'inizio di un buffer, dobbiamo impostare la posizione a zero. flip () metodo cambia la posizione
5. Quando chiamiamo il metodo di lettura del `Channel` , riempie il buffer usando i dati.
6. Se abbiamo bisogno di leggere i dati dal `ByteBuffer` , dobbiamo capovolgere il buffer per cambiarne la modalità in sola scrittura in modalità di sola lettura e quindi continuare a leggere i dati dal buffer.
7. Quando non ci sono più dati da leggere, il metodo `read()` del canale restituisce 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();
  }
 }
}
```

Copia di un file utilizzando il canale

Possiamo usare `Channel` per copiare il contenuto del file più velocemente. Per fare ciò, possiamo

usare il metodo `transferTo()` di `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();
        }
    }
}
```

Lettura di un file utilizzando BufferedInputStream

Leggendo il file usando `BufferedInputStream` generalmente più velocemente di `FileInputStream` perché mantiene un buffer interno per memorizzare i byte letti dal flusso di input sottostante.

```
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();
        }

    }
```

```
}
```

Scrivere un file usando Channel e Buffer

Per scrivere i dati su un file utilizzando `Channel` è necessario avere i seguenti passaggi:

1. Per prima cosa, dobbiamo ottenere un oggetto di `FileOutputStream`
2. Acquisisci `FileChannel` chiamando il metodo `getChannel()` da `FileOutputStream`
3. Creare un `ByteBuffer` e quindi riempirlo con i dati
4. Quindi dobbiamo chiamare il metodo `flip()` del `ByteBuffer` e passarlo come argomento del metodo `write()` del `FileChannel`
5. Una volta che abbiamo finito di scrivere, dobbiamo chiudere la risorsa

```
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();
  }
 }
}
```

Scrivere un file usando PrintStream

Possiamo usare la classe `PrintStream` per scrivere un file. Ha diversi metodi che ti permettono di stampare qualsiasi valore di tipo di dati. `println()` metodo `println()` aggiunge una nuova riga. Una volta terminata la stampa, dobbiamo svuotare `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();
        }

    }
}
```

Passare sopra una directory che stampa sottodirectory in esso

```
  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());
      }
    }
  }
```

Aggiunta di directory

Per creare una nuova directory da un'istanza di `File` è necessario utilizzare uno dei seguenti due metodi: `mkdirs()` o `mkdir()` .

- `mkdir()` - Crea la directory chiamata da questo percorso astratto. (fonte)
- `mkdirs()` - Crea la directory nominata da questo percorso astratto, includendo tutte le directory padre necessarie ma inesistenti. Notare che se questa operazione fallisce, potrebbe essere riuscita a creare alcune delle directory madri necessarie. (fonte)

Nota: `createNewFile()` non creerà una nuova directory solo un file.

```
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
```

Blocco o reindirizzamento dell'output / errore standard

A volte una libreria di terze parti progettata in modo inadeguato scriverà una diagnostica indesiderata sui flussi `System.out` o `System.err` . Le soluzioni consigliate per questo sarebbe trovare una libreria migliore o (nel caso di open source) risolvere il problema e contribuire con una patch agli sviluppatori.

Se le soluzioni di cui sopra non sono fattibili, dovresti considerare di reindirizzare i flussi.

Reindirizzamento sulla riga di comando

Su un sistema UNIX, Linux o MacOSX può essere fatto dalla shell usando `>` redirection. Per esempio:

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

Il primo reindirizza lo standard output e l'errore standard a "/ dev / null", che getta via qualsiasi cosa scritta su quei flussi. Il secondo reindirizza l'output standard a "out.log" e l'errore standard a "error.log".

(Per ulteriori informazioni sul reindirizzamento, consultare la documentazione della shell dei comandi che si sta utilizzando. Un consiglio simile si applica a Windows.)

In alternativa, è possibile implementare il reindirizzamento in uno script wrapper o in un file batch che avvia l'applicazione Java.

Reindirizzamento all'interno di un'applicazione Java

È anche possibile ridirigere i flussi *all'interno di* un'applicazione Java utilizzando `System.setOut()` e `System.setErr()` . Ad esempio, il seguente snippet reindirizza l'output standard e l'errore standard in 2 file di registro:

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

Se si desidera eliminare completamente l'output, è possibile creare un flusso di output che "scrive" su un descrittore di file non valido. Questo è funzionalmente equivalente alla scrittura su "/ dev / null" su UNIX.

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

Attenzione: fai attenzione a come usi `setOut` e `setErr` :

1. Il reindirizzamento interesserà l'intera JVM.
2. In questo modo, stai rimuovendo la possibilità dell'utente di reindirizzare gli stream dalla riga di comando.

Accedere ai contenuti di un file ZIP

L'API FileSystem di Java 7 consente di leggere e aggiungere voci da o ad un file Zip utilizzando l'API del file NIO Java allo stesso modo di operare su qualsiasi altro file system.

Il FileSystem è una risorsa che dovrebbe essere chiusa correttamente dopo l'uso, quindi dovrebbe essere usato il blocco try-with-resources.

Lettura da un file esistente

```
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();
}
```

Creare un nuovo file

```
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();
}
```

Leggi File I / O online: https://riptutorial.com/it/java/topic/93/file-i---o

Capitolo 61: File JAR multi-release

introduzione

Una delle funzionalità introdotte in Java 9 è il multi-release Jar (MRJAR) che consente di raggruppare il codice per il targeting di più versioni di Java all'interno dello stesso file Jar. La funzione è specificata in JEP 238 .

Examples

Esempio di contenuto di un file Jar multi-release

Impostando `Multi-Release: true` nel file MANIFEST.MF, il file Jar diventa un Jar multi-release e il runtime Java (purché supporti il formato MRJAR) sceglierà le versioni appropriate delle classi a seconda della versione principale corrente .

La struttura di un simile Jar è la seguente:

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

- Su JDK <9, solo le classi nella voce root sono visibili al runtime Java.
- Su un JDK 9, le classi A e B verranno caricate dalla directory `root/META-INF/versions/9` , mentre C e D verranno caricate dalla voce base.
- Su un JDK 10, la classe A verrebbe caricata dalla directory `root/META-INF/versions/10` .

Creare un vaso multi-release usando lo strumento jar

Il comando `jar` può essere utilizzato per creare un jar multi-release contenente due versioni della stessa classe compilate sia per Java 8 che per Java 9, anche se con un avvertimento che indica che le classi sono identiche:

```
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
```

L'opzione `--release 9` dice a `jar` di includere tutto ciò che segue (il pacchetto `demo` all'interno della

`sampleproject-9`) all'interno di una voce con versione nel MRJAR, cioè sotto `root/META-INF/versions/9` . Il risultato è il seguente contenuto:

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

Cerchiamo ora di creare una classe chiamata Main che stampa l'URL di `SampleClass` e aggiungila per la versione 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);
    }
}
```

Se compiliamo questa classe e ri-eseguiamo il comando jar, otteniamo un errore:

```
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
```

Il motivo è che lo strumento `jar` impedisce l'aggiunta di classi pubbliche alle voci con versione se non vengono aggiunte anche alle voci di base. Questo viene fatto in modo che MRJAR esponga la stessa API pubblica per le diverse versioni di Java. Si noti che in fase di runtime, questa regola non è richiesta. Può essere applicato solo da strumenti come `jar` . In questo caso particolare, lo scopo di `Main` è di eseguire un codice di esempio, quindi possiamo semplicemente aggiungere una copia nella voce base. Se la classe fosse parte di un'implementazione più recente di cui abbiamo bisogno solo per Java 9, potrebbe essere resa non pubblica.

Per aggiungere `Main` alla voce root, dobbiamo prima compilarlo per targetizzare una versione pre-Java 9. Questo può essere fatto usando la nuova opzione `--release` di `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
```

L'esecuzione della classe Main mostra che SampleClass viene caricato dalla directory versioned:

```
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 di una classe caricata all'interno di un Jar multi-release

Dato il seguente Jar multi-release:

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

La seguente classe stampa l'URL della `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);
    }
}
```

Se la classe è compilata e aggiunta alla voce con versione per Java 9 in MRJAR, eseguirla risulterebbe in:

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

Leggi File JAR multi-release online: https://riptutorial.com/it/java/topic/9866/file-jar-multi-release

Capitolo 62: FileUpload in AWS

introduzione

Carica il file nel bucket AWS s3 utilizzando l'API Spring Rest.

Examples

Carica il file nel secchio s3

> Qui creeremo un APi di riposo che prenderà l'oggetto file come parametro multipart dal front-end e lo caricherà nel bucket S3 usando l'API java rest.

Requisito : - chiave di secern e chiave di accesso per il bucket s3 in cui si desidera caricare il file.

codice: - 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;

    }


}
```

Funzione front end

```
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);
});
```

Leggi FileUpload in AWS online: https://riptutorial.com/it/java/topic/10589/fileupload-in-aws

Capitolo 63: Fluent Interface

Osservazioni

obiettivi

L'obiettivo principale di un'interfaccia fluida è la leggibilità aumentata.

Quando viene utilizzato per la costruzione di oggetti, le scelte disponibili per il chiamante possono essere rese chiare e applicate tramite controlli in fase di compilazione. Ad esempio, si consideri il seguente albero di opzioni che rappresentano passi lungo il percorso per costruire un oggetto complesso:

```
A -> B
  -> C -> D -> Done
       -> E -> Done
       -> F -> Done.
       -> G -> H -> I -> Done.
```

Un costruttore che utilizza un'interfaccia fluente consentirebbe al chiamante di vedere facilmente quali opzioni sono disponibili in ogni fase. Ad esempio, **A -> B** è possibile, ma **A -> C** non lo è e risulterebbe in un errore in fase di compilazione.

Examples

Truth - Fluent Testing Framework

Da "Come usare la verità" 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();
```

Ottimo stile di programmazione

Nello stile di programmazione fluente si restituisce `this` metodo da metodi fluenti (setter) che non restituirebbero nulla in uno stile di programmazione non fluente.

Ciò consente di concatenare le diverse chiamate di metodo che rendono il codice più breve e più facile da gestire per gli sviluppatori.

Considera questo codice non fluente:

```
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());
  }
}
```

Siccome i metodi di settaggio non restituiscono nulla, abbiamo bisogno di 4 istruzioni nel metodo `main` per istanziare una `Person` con alcuni dati e stamparla. Con uno stile fluido questo codice può essere cambiato in:

```
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());
  }
}
```

L'idea è di restituire sempre alcuni oggetti per abilitare la costruzione di una catena di chiamate di metodo e di usare nomi di metodi che riflettano il linguaggio naturale. Questo stile fluente rende il codice più leggibile.

Leggi Fluent Interface online: https://riptutorial.com/it/java/topic/5090/fluent-interface

Capitolo 64: FTP (File Transfer Protocol)

Sintassi

- Connessione FTPClient (host InetAddress, porta int)
- Login FTPClient (nome utente stringa, password stringa)
- FTPClient disconnect ()
- FTPReply getReplyStrings ()
- boolean storeFile (String remote, InputStream local)
- OutputStream storeFileStream (String remote)
- booleano setFileType (int fileType)
- booleano completePendingCommand ()

Parametri

parametri	Dettagli
ospite	O il nome host o l'indirizzo IP del server FTP
porta	La porta del server FTP
nome utente	Il nome utente del server FTP
parola d'ordine	La password del server FTP

Examples

Connessione e accesso a un server FTP

Per iniziare a utilizzare FTP con Java, sarà necessario creare un nuovo `FTPClient` e quindi connettersi e accedere al server utilizzando `.connect(String server, int port)` e `.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);
    }
}
```

Ora abbiamo fatto le basi. Ma cosa succede se si verifica un errore durante la connessione al server? Vogliamo sapere quando qualcosa va storto e ricevere il messaggio di errore. Aggiungiamo un po 'di codice per rilevare gli errori durante la connessione.

```
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 {

}
```

Analizziamo ciò che abbiamo appena fatto, passo dopo passo.

```
showServerReply(ftp);
```

Questo si riferisce a una funzione che faremo in un secondo momento.

```
int replyCode = ftp.getReplyCode();
```

Questo preleva il codice di risposta / errore dal server e lo memorizza come un intero.

```
if (!FTPReply.isPositiveCompletion(replyCode)) {
    System.out.printIn("Operation failed. Server reply code: " + replyCode)
    return;
}
```

Controlla il codice di risposta per vedere se c'è stato un errore. Se si è verificato un errore, verrà semplicemente stampato "Operazione non riuscita. Codice di risposta del server:" seguito dal codice di errore. Abbiamo anche aggiunto un blocco try / catch a cui aggiungeremo nel prossimo passaggio. Successivamente, creiamo anche una funzione che controlli `ftp.login()` per gli errori.

```
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");
    }
```

Rompiamo anche questo blocco.

```
boolean success = ftp.login(user, pass);
```

Questo non tenterà solo di accedere al server FTP, ma memorizzerà anche il risultato come booleano.

```
showServerReply(ftp);
```

Questo controllerà se il server ci ha inviato messaggi, ma prima dovremo creare la funzione nel passaggio successivo.

```
if (!success) {
System.out.println("Failed to log into the server");
    return;
} else {
    System.out.println("LOGGED IN SERVER");
}
```

Questa affermazione controllerà se abbiamo effettuato l'accesso con successo; in tal caso, stamperà "LOGGED IN SERVER", altrimenti stamperà "Impossibile accedere al server". Questo è il nostro script finora:

```
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 {

        }
    }
}
```

Ora dopo creiamo il blocco Catch nel caso in cui ci imbattiamo in eventuali errori con l'intero processo.

```
} catch (IOException ex) {
    System.out.println("Oops! Something went wrong.");
    ex.printStackTrace();
}
```

Il blocco catch completato ora stamperà "Oops! Qualcosa è andato storto". e lo stacktrace se c'è un errore. Ora il nostro ultimo passo è creare lo `showServerReply()` stiamo usando da un po 'di tempo.

```
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);
        }
    }
}
```

Questa funzione accetta un `FTPClient` come variabile e lo chiama "ftp". Successivamente memorizza qualsiasi risposta del server dal server in un array di stringhe. Quindi controlla se qualche messaggio è stato memorizzato. Se ce n'è uno, stampa ciascuno di essi come "SERVER: [rispondi]". Ora che abbiamo fatto quella funzione, questo è lo script completato:

```
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();
        }
    }
}
```

Per prima cosa è necessario creare un nuovo `FTPClient` e provare a connettersi al server e ad `.connect(String server, int port)` utilizzando `.connect(String server, int port)` e `.login(String username, String password)` . È importante connettersi e accedere utilizzando un blocco try / catch nel caso in cui il nostro codice non riesca a connettersi con il server. Dovremo anche creare una funzione che controlli e visualizzi tutti i messaggi che possiamo ricevere dal server mentre proviamo a connetterci e ad accedere. Chiameremo questa funzione " `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();
        }
    }
}
```

Dopo questo, ora dovresti avere il tuo server FTP connesso allo script Java.

Leggi FTP (File Transfer Protocol) online: https://riptutorial.com/it/java/topic/5228/ftp--file-transfer-protocol-

Capitolo 65: Funzionalità Java SE 7

introduzione

In questo argomento troverai un riepilogo delle nuove funzionalità aggiunte al linguaggio di programmazione Java in Java SE 7. Ci sono molte altre nuove funzionalità in altri campi come JDBC e Java Virtual Machine (JVM) che non saranno coperti in questo argomento.

Osservazioni

Miglioramenti in Java SE 7

Examples

Nuove funzionalità del linguaggio di programmazione Java SE 7

- Binary Literals : i tipi interi (byte, short, int e long) possono anche essere espressi usando il sistema di numeri binari. Per specificare un valore letterale binario, aggiungi il prefisso 0b o 0B al numero.
- Stringhe nelle dichiarazioni switch : è possibile utilizzare un oggetto String nell'espressione di un'istruzione switch
- L'istruzione try-with-resources : L' istruzione try-with-resources è un'istruzione try che dichiara una o più risorse. Una risorsa è come un oggetto che deve essere chiuso al termine del programma. L'istruzione try-with-resources assicura che ogni risorsa sia chiusa alla fine dell'istruzione. Qualsiasi oggetto che implementa java.lang.AutoCloseable, che include tutti gli oggetti che implementano java.io.Closeable, può essere utilizzato come risorsa.
- Acquisire più tipi di eccezioni e rimandare le eccezioni con il controllo dei tipi migliorato : un singolo blocco catch può gestire più di un tipo di eccezione. Questa funzione può ridurre la duplicazione del codice e ridurre la tentazione di rilevare un'eccezione troppo ampia.
- Underscores in Numeric Literals : qualsiasi numero di caratteri di sottolineatura (_) può apparire ovunque tra cifre in un valore letterale numerico. Questa funzione consente, ad esempio, di separare gruppi di cifre in valori letterali numerici, che possono migliorare la leggibilità del codice.
- Digitare Inference per la creazione di istanze generiche : è possibile sostituire gli argomenti di tipo richiesti per richiamare il costruttore di una classe generica con un set vuoto di parametri di tipo (<>) a condizione che il compilatore possa dedurre gli argomenti di tipo dal contesto. Questa coppia di parentesi angolari è chiamata in modo informale il diamante.
- Errori ed errori del compilatore migliorati quando si utilizzano parametri formali non-reifiable con metodi Varargs

Binary Literals

```
// 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 dichiarazione try-with-resources

L'esempio legge la prima riga da un file. Usa un'istanza di `BufferedReader` per leggere i dati dal file. `BufferedReader` è una risorsa che deve essere chiusa al termine del programma:

```
static String readFirstLineFromFile(String path) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

In questo esempio, la risorsa dichiarata nell'istruzione try-with-resources è una `BufferedReader` . L'istruzione dichiarazione appare tra parentesi immediatamente dopo la parola chiave try. La classe `BufferedReader` , in Java SE 7 e successive, implementa l'interfaccia `java.lang.AutoCloseable` . Poiché l'istanza `BufferedReader` viene dichiarata in un'istruzione try-with-resource, verrà chiusa indipendentemente dal fatto che l'istruzione try venga completata normalmente o in modo brusco (come risultato del metodo `BufferedReader.readLine` che `IOException` una `IOException`).

Sottolinea in Numeric Literals

L'esempio seguente mostra altri modi per utilizzare il carattere di sottolineatura in valori letterali numerici:

```
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;
```

È possibile inserire caratteri di sottolineatura solo tra cifre; non puoi inserire caratteri di sottolineatura nei seguenti luoghi:

- All'inizio o alla fine di un numero
- Adiacente a un punto decimale in un letterale in virgola mobile
- Prima di un suffisso F o L
- Nelle posizioni in cui è prevista una stringa di cifre

Digitare un'inferenza per la creazione di istanze generiche

Puoi usare

```
Map<String, List<String>> myMap = new HashMap<>();
```

invece di

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

Tuttavia, non è possibile utilizzare

```
List<String> list = new ArrayList<>();
list.add("A");

  // The following statement should fail since addAll expects
  // Collection<? extends String>

list.addAll(new ArrayList<>());
```

perché non può essere compilato. Si noti che il diamante spesso funziona nelle chiamate al metodo; tuttavia, si suggerisce di utilizzare il diamante principalmente per dichiarazioni variabili.

Stringhe nelle dichiarazioni switch

```
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;
}
```

Leggi Funzionalità Java SE 7 online: https://riptutorial.com/it/java/topic/8272/funzionalita-java-se-7

Capitolo 66: Funzionalità Java SE 8

introduzione

In questo argomento troverai un riepilogo delle nuove funzionalità aggiunte al linguaggio di programmazione Java in Java SE 8. Ci sono molte altre nuove funzionalità in altri campi come JDBC e Java Virtual Machine (JVM) che non saranno coperti in questo argomento.

Osservazioni

Riferimento: miglioramenti in Java SE 8

Examples

Nuove funzionalità del linguaggio di programmazione Java SE 8

- Lambda Expressions , una nuova funzione linguistica, è stata introdotta in questa versione. Consentono di trattare la funzionalità come argomento del metodo o codice come dati. Le espressioni Lambda consentono di esprimere istanze di interfacce a metodo singolo (denominate interfacce funzionali) in modo più compatto.
    - I riferimenti al metodo forniscono espressioni lambda di facile lettura per metodi che hanno già un nome.
    - I metodi predefiniti consentono di aggiungere nuove funzionalità alle interfacce delle librerie e garantiscono la compatibilità binaria con il codice scritto per le versioni precedenti di tali interfacce.
    - API nuove e migliorate che sfruttano Lambda Expressions e Streams in Java SE 8 descrivono classi nuove e migliorate che sfruttano le espressioni lambda e gli stream.
- Migliore inferenza di tipo: il compilatore Java sfrutta la tipizzazione di destinazione per dedurre i parametri di tipo di una chiamata di metodo generica. Il tipo di destinazione di un'espressione è il tipo di dati che il compilatore Java si aspetta a seconda di dove viene visualizzata l'espressione. Ad esempio, è possibile utilizzare il tipo di destinazione dell'istruzione di assegnazione per l'inferenza di tipo in Java SE 7. Tuttavia, in Java SE 8, è possibile utilizzare il tipo di destinazione per l'inferenza di tipo in più contesti.
    - Digitazione obiettivo in Lambda Expressions
    - Tipo di inferenza
- Le annotazioni ripetute forniscono la possibilità di applicare lo stesso tipo di annotazione più volte per la stessa dichiarazione o tipo di uso.
- Le annotazioni di tipo consentono di applicare un'annotazione ovunque sia utilizzato un tipo, non solo su una dichiarazione. Utilizzato con un sistema di tipo collegabile, questa funzione consente di migliorare la verifica del tipo del codice.
- Riflessione dei parametri del metodo : è possibile ottenere i nomi dei parametri formali di qualsiasi metodo o costruttore con il metodo `java.lang.reflect.Executable.getParameters` . (Le classi Method e Constructor estendono la classe Executable e quindi ereditano il metodo `Executable.getParameters`) Tuttavia, i file `.class` non memorizzano i nomi dei parametri

formali per impostazione predefinita. Per memorizzare i nomi dei parametri formali in un particolare file `.class` e quindi abilitare l'API Reflection per recuperare i nomi dei parametri formali, compilare il file di origine con l'opzione `-parameters` del compilatore javac.

- Date-time-api - Aggiunto nuovo tempo api in `java.time` . Se utilizzato, non è necessario designare il fuso orario.

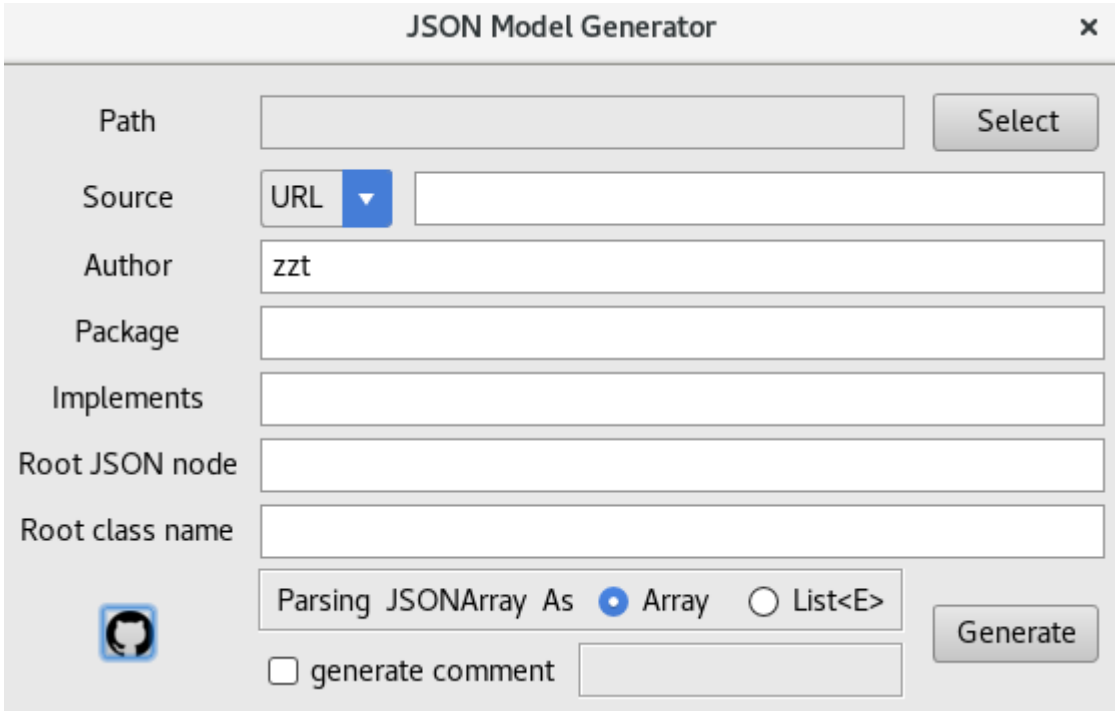
Leggi Funzionalità Java SE 8 online: https://riptutorial.com/it/java/topic/8267/funzionalita-java-se-8

Capitolo 67: Generazione del codice Java

Examples

Genera POJO da JSON

- Installa il plugin JSON Model Genrator di Intellij effettuando una ricerca in Intellij.
- Avvia il plugin da "Strumenti"
- Inserisci il campo dell'interfaccia utente come mostrato di seguito ('Percorso', 'Origine', 'Pacchetto' è richiesto):



- Fare clic sul pulsante "Genera" e il gioco è fatto.

Leggi Generazione del codice Java online: https://riptutorial.com/it/java/topic/9400/generazione-del-codice-java

Capitolo 68: Generazione di numeri casuali

Osservazioni

Niente è veramente casuale e quindi javadoc chiama quei numeri pseudocasuali. Questi numeri sono creati con un generatore di numeri pseudocasuali .

Examples

Pseudo numeri casuali

Java fornisce, come parte del pacchetto `utils` , un generatore di numeri pseudo-casuali di base, opportunamente chiamato `Random` . Questo oggetto può essere utilizzato per generare un valore pseudo-casuale come qualsiasi dei tipi di dati numerici incorporati (`int` , `float` , ecc.). Puoi anche usarlo per generare un valore booleano casuale o una matrice casuale di byte. Un esempio di utilizzo è il seguente:

```
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: questa classe produce solo numeri pseudo-casuali di bassa qualità e non dovrebbe mai essere utilizzata per generare numeri casuali per operazioni crittografiche o altre situazioni in cui la casualità di qualità superiore è fondamentale (per questo, si vorrebbe usare la classe `SecureRandom` , come indicato di seguito). Una spiegazione della distinzione tra casualità "sicura" e "insicura" esula dallo scopo di questo esempio.

Numeri casuali falsi in un intervallo specifico

Il metodo `nextInt(int bound)` di `Random` accetta un limite esclusivo superiore, ovvero un numero che il valore casuale restituito deve essere inferiore a. Tuttavia, solo il metodo `nextInt` accetta un limite; `nextLong` , `nextDouble` ecc. no.

```
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 partire da Java 1.7, puoi anche usare `ThreadLocalRandom` (source). Questa classe fornisce un PRNG sicuro per thread (generatore di numeri pseudo-casuali). Si noti che il metodo `nextInt` di questa classe accetta sia un limite superiore che uno inferiore.

```
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);
```

Si noti che la documentazione ufficiale afferma che `nextInt(int bound)` può fare cose strane quando `bound` è vicino a $2^{30}+1$ (enfasi aggiunta):

> L'algoritmo è leggermente complicato. **Respinge i valori che risulterebbero in una distribuzione non uniforme** (a causa del fatto che 2 ^ 31 non è divisibile per n). La probabilità che un valore venga rifiutato dipende da n. **Il caso peggiore è n = 2 ^ 30 + 1, per cui la probabilità di uno scarto è 1/2, e il numero previsto di iterazioni prima che il ciclo termini è 2.**

In altre parole, la specificazione di un limite diminuirà (leggermente) le prestazioni del metodo `nextInt` e questa riduzione delle prestazioni diventerà più pronunciata quando il `bound` avvicina a metà del valore massimo int.

Generazione di numeri pseudocasuali crittograficamente sicuri

`Random` e `ThreadLocalRandom` sono abbastanza buoni per l'uso quotidiano, ma hanno un grosso problema: si basano su un generatore congruenziale lineare , un algoritmo il cui output può essere predetto piuttosto facilmente. Pertanto, queste due classi **non** sono adatte per usi crittografici (come la generazione di chiavi).

È possibile utilizzare `java.security.SecureRandom` in situazioni in cui è richiesto un PRNG con un output molto difficile da prevedere. Prevedere i numeri casuali creati dalle istanze di questa classe è abbastanza difficile da etichettare la classe come **crittograficamente sicura** .

```
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));
    }
}
```

Oltre a essere crittograficamente sicuro, `SecureRandom` ha un periodo gigantesco di 2^{160} , rispetto al periodo di `Random` di 2^{48} . Ha uno svantaggio di essere considerevolmente più lento di `Random` e altri PRNG lineari come Mersenne Twister e Xorshift , comunque.

Si noti che l'implementazione SecureRandom è dipendente dalla piattaforma e dal fornitore. Il `SecureRandom` predefinito (fornito dal provider `SUN` in `sun.security.provider.SecureRandom`):

- su sistemi simil-Unix, seminato con dati da `/dev/random` e / o `/dev/urandom` .
- su Windows, seminato con chiamate a `CryptGenRandom()` in CryptoAPI .

Seleziona numeri casuali senza duplicati

```
/**
 * 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;
```

```
}
```

Il metodo funziona eseguendo il ciclo su un array che ha le dimensioni della lunghezza richiesta e trova la lunghezza rimanente dei numeri possibili. Imposta un numero casuale di quei possibili numeri `newRandSpot` e trova quel numero all'interno del numero non occupato rimasto. Lo fa scorrendo l'intervallo e controllando per vedere se quel numero è già stato preso.

Ad esempio se l'intervallo è 5 e la lunghezza è 3 e abbiamo già scelto il numero 2. Quindi abbiamo 4 numeri rimanenti in modo da ottenere un numero casuale compreso tra 1 e 4 e passiamo attraverso l'intervallo (5) saltando sopra qualsiasi numero che abbiamo già usato (2).

Ora diciamo che il prossimo numero scelto tra 1 e 4 è 3. Nel primo ciclo otteniamo 1 che non è ancora stato preso così possiamo rimuovere 1 da 3 e renderlo 2. Ora sul secondo ciclo otteniamo 2 che è stato preso quindi non facciamo niente Seguiamo questo modello finché non arriviamo a 4, dove una volta rimosso 1 diventa 0, quindi impostiamo il nuovo numero casuale su 4.

Generazione di numeri casuali con un seme specificato

```
//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);
```

Usare lo stesso seme per generare numeri casuali restituirà sempre gli stessi numeri, quindi impostare un seme diverso per ogni istanza `Random` è una buona idea se non vuoi finire con numeri duplicati.

Un buon metodo per ottenere un `Long` diverso da ogni chiamata è `System.currentTimeMillis()` :

```
Random random = new Random(System.currentTimeMillis());
ThreadLocalRandom.current().setSeed(System.currentTimeMillis());
```

Generazione di numeri casuali usando lang3 apache-common

Possiamo usare `org.apache.commons.lang3.RandomUtils` per generare numeri casuali usando una singola riga.

```
int x = RandomUtils.nextInt(1, 1000);
```

Il metodo `nextInt(int startInclusive, int endExclusive)` accetta un intervallo.

Oltre a int, possiamo generare `long` , `double` , `float` e `bytes` casuali usando questa classe.

`RandomUtils` classe `RandomUtils` contiene i seguenti metodi:

```
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.
```

Leggi Generazione di numeri casuali online: https://riptutorial.com/it/java/topic/890/generazione-di-numeri-casuali

Capitolo 69: Generics

introduzione

I generici sono una funzionalità di programmazione generica che estende il sistema di tipi di Java per consentire a un tipo o un metodo di operare su oggetti di vario tipo fornendo nel contempo la sicurezza del tipo in fase di compilazione. In particolare, il framework delle collezioni Java supporta i generici per specificare il tipo di oggetti memorizzati in un'istanza di raccolta.

Sintassi

- class ArrayList <E> {} // una classe generica con parametro di tipo E
- class HashMap <K, V> {} // una classe generica con due parametri di tipo K e V
- <E> void print (elemento E) {} // un metodo generico con parametro tipo E
- ArrayList <String> nomi; // dichiarazione di una classe generica
- ArrayList <?> Oggetti; // dichiarazione di una classe generica con un parametro di tipo sconosciuto
- new ArrayList <String> () // istanza di una classe generica
- new ArrayList <> () // istanza con tipo di inferenza "diamond" (Java 7 o successivo)

Osservazioni

I generici sono implementati in Java tramite la cancellazione dei tipi, il che significa che durante il runtime le informazioni sul tipo specificate nell'istanza di una classe generica non sono disponibili. Ad esempio, l'istruzione `List<String> names = new ArrayList<>();` produce un oggetto lista da cui il tipo di elemento `String` non può essere recuperato in fase di runtime. Tuttavia, se l'elenco è memorizzato in un campo di tipo `List<String>` o passato a un parametro method / constructor di questo stesso tipo o restituito da un metodo di quel tipo restituito, le informazioni di tipo completo *possono* essere ripristinate in fase di runtime attraverso l'API Java Reflection.

Ciò significa anche che quando si esegue il casting su un tipo generico (es .: `(List<String>) list`), il cast è un *cast non controllato* . Poiché il parametro `<String>` viene cancellato, la JVM non può controllare se un cast da un `List<?>` un `List<String>` è corretto; la JVM vede solo un cast per `List` da `List` in fase di runtime.

Examples

Creazione di una classe generica

Generics abilita classi, interfacce e metodi per prendere altre classi e interfacce come parametri di tipo.

Questo esempio utilizza la classe generica `Param` per prendere un singolo **parametro di tipo** `T` , delimitato da parentesi angolari (`<>`):

```
public class Param<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

Per creare un'istanza di questa classe, fornire un **argomento di tipo** al posto di `T` Ad esempio, `Integer` :

```
Param<Integer> integerParam = new Param<Integer>();
```

L'argomento type può essere qualsiasi tipo di riferimento, inclusi gli array e altri tipi generici:

```
Param<String[]> stringArrayParam;
Param<int[][]> int2dArrayParam;
Param<Param<Object>> objectNestedParam;
```

In Java SE 7 e versioni successive, l'argomento type può essere sostituito con un set vuoto di argomenti type (`<>`) chiamato *diamond* :

Java SE 7

```
Param<Integer> integerParam = new Param<>();
```

A differenza di altri identificatori, i parametri di tipo non hanno vincoli di denominazione. Tuttavia i loro nomi sono comunemente la prima lettera del loro scopo in maiuscolo. (Questo è vero anche in tutti i JavaDocs ufficiali.)
Gli esempi includono `T` per "tipo" , `E` per "elemento" e `K` / `V` per "chiave" / "valore" .

Estendere una classe generica

```
public abstract class AbstractParam<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

`AbstractParam` è una classe astratta dichiarata con un parametro di tipo `T` Quando si estende

questa classe, quel parametro di tipo può essere sostituito da un argomento di tipo scritto all'interno di `<>` , oppure il parametro type può rimanere invariato. Nel primo e nel secondo esempio di seguito, `String` e `Integer` sostituiscono il parametro type. Nel terzo esempio, il parametro type rimane invariato. Il quarto esempio non usa affatto i generici, quindi è simile a se la classe avesse un parametro `Object` . Il compilatore avverte che `AbstractParam` è un tipo non `ObjectParam` , ma compilerà la classe `ObjectParam` . Il quinto esempio ha 2 parametri di tipo (vedi "parametri di tipo multiplo" sotto), scegliendo il secondo parametro come parametro di tipo passato alla superclasse.

```
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> {
    // ...
}
```

Quanto segue è l'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);
```

Si noti che nella classe `Email` , il metodo `T getValue()` agisce come se avesse una firma di `String getValue()` , e il `void setValue(T)` agisce come se fosse stato dichiarato `void setValue(String)` .

È anche possibile creare un'istanza con classe interna anonima con parentesi graffe vuote (`{}`):

```
AbstractParam<Double> height = new AbstractParam<Double>(){};
```

```
height.setValue(198.6);
```

Nota che l' uso del diamante con classi interne anonime non è permesso.

Parametri di tipo multiplo

Java offre la possibilità di utilizzare più di un parametro di tipo in una classe o interfaccia generica. È possibile utilizzare più parametri di tipo in una classe o in un'interfaccia inserendo un **elenco** di tipi **separati da virgole** tra parentesi angolari. Esempio:

```
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;
    }
}
```

L'utilizzo può essere eseguito come di seguito:

```
MultiGenericParam<String, String> aParam = new MultiGenericParam<String, String>("value1",
"value2");
MultiGenericParam<Integer, Double> dayOfWeekDegrees = new MultiGenericParam<Integer,
Double>(1, 2.6);
```

Dichiarazione di un metodo generico

I metodi possono anche avere parametri di tipo generico .

```
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);
    }
}
```

Si noti che non è necessario passare un argomento di tipo effettivo a un metodo generico. Il compilatore deduce l'argomento tipo per noi, in base al tipo di destinazione (ad esempio, la variabile a cui assegniamo il risultato) o ai tipi degli argomenti effettivi. In genere inferirà l'argomento di tipo più specifico che renderà corretta la chiamata.

A volte, anche se raramente, può essere necessario sovrascrivere questo tipo di inferenza con argomenti di tipo esplicito:

```
void usage() {
    consumeObjects(this.<Object>makeList("Jeff", "Atwood").stream());
}

void consumeObjects(Stream<Object> stream) { ... }
```

È necessario in questo esempio perché il compilatore non può "guardare avanti" per vedere che l' `Object` è desiderato per `T` dopo aver chiamato `stream()` e altrimenti inferirebbe `String` basato sugli argomenti `makeList` . Si noti che il linguaggio Java non supporta l'omissione della classe o dell'oggetto su cui viene chiamato il metodo (`this` nell'esempio precedente) quando vengono forniti esplicitamente argomenti tipo.

Il diamante

Java SE 7

Java 7 ha introdotto il *Diamond* [1] per rimuovere alcuni piatti della caldaia attorno all'istanziazione di classe generica. Con Java 7+ puoi scrivere:

```
List<String> list = new LinkedList<>();
```

Dove dovevi scrivere nelle versioni precedenti, questo:

```
List<String> list = new LinkedList<String>();
```

Una limitazione è per le classi anonime , in cui è comunque necessario fornire il parametro type nell'istanza:

```
// 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

Sebbene l'uso del diamante con le classi interne anonime non sia supportato in Java 7 e 8, **verrà incluso come una nuova funzionalità in Java 9** .

Nota:

1 - Alcune persone chiamano `<>` uso di " *operatore* diamante". Questo non è corretto Il diamante non si comporta come un operatore e non è descritto o elencato in nessuna parte del JLS o dei tutorial Java (ufficiali) come operatore. Infatti, `<>` non è nemmeno un token Java distinto. Piuttosto si tratta di un `<` pedina seguito da un `>` Token, ed è legale (anche se cattivo stile) di avere spazi bianchi o commenti tra i due. Il JLS e i Tutorials si riferiscono costantemente a `<>` come "il diamante", e questo è quindi il termine corretto per esso.

Richiesta di più limiti superiori ("estende A & B")

È possibile richiedere un tipo generico per estendere più limiti superiori.

Esempio: vogliamo ordinare una lista di numeri ma `Number` non implementa `Comparable` .

```
public <T extends Number & Comparable<T>> void sortNumbers( List<T> n ) {
  Collections.sort( n );
}
```

In questo esempio `T` deve estendere il `Number` *e* implementare il `Comparable<T>` che dovrebbe adattarsi a tutte le implementazioni del numero incorporato "normale" come `Integer` o `BigDecimal` ma non si adatta a quelle più esotiche come `Striped64` .

Poiché l'ereditarietà multipla non è consentita, è possibile utilizzare al massimo una classe come limite e deve essere la prima elencata. Ad esempio, `<T extends Comparable<T> & Number>` non è consentito perché Comparable è un'interfaccia e non una classe.

Creazione di una classe generica limitata

È possibile limitare i tipi validi utilizzati in una **classe generica** delimitando quel tipo nella definizione della classe. Data la seguente gerarchia di tipi semplici:

```
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";
    }
}
```

Senza **generici limitati** , non possiamo creare una classe contenitore che sia sia generica che sa che ogni elemento è un animale:

```
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 il limite generico nella definizione della classe, questo è ora possibile.

```
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());
```

```
        }
    }
}
```

Ciò limita anche le istanze valide del tipo generico:

```
// 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>();
```

Decidere tra `T`,`? super T`, e `? estende T`

La sintassi per i caratteri jolly limitati generici Java, che rappresentano il tipo sconosciuto da `?` è:

- `? extends T` rappresenta un carattere jolly con limite superiore. Il tipo sconosciuto rappresenta un tipo che deve essere un sottotipo di T o di tipo T stesso.
- `? super T` rappresenta un carattere jolly con limite inferiore. Il tipo sconosciuto rappresenta un tipo che deve essere un supertipo di T o di tipo T stesso.

Come regola generale, dovresti usare

- `? extends T` se hai solo bisogno dell'accesso "lettura" ("input")
- `? super T` se hai bisogno dell'accesso "write" ("output")
- `T` se hai bisogno di entrambi ("modifica")

Usando `extends` o `super` è solitamente *meglio* perché rende il tuo codice più flessibile (come in: consentire l'uso di sottotipi e supertipi), come vedrai di seguito.

```
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) {}
}
```

Il compilatore ora sarà in grado di rilevare alcuni usi errati:

```
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!
}
```

Scegli la `T` giusta `? super T` o `? extends T` è *necessario* per consentire l'uso con sottotipi. Il compilatore può quindi garantire la sicurezza del tipo; non dovrebbe essere necessario eseguire il cast (che non è sicuro per il tipo e potrebbe causare errori di programmazione) se vengono utilizzati correttamente.

Se non è facile da capire, ricorda la regola **PECS** :

> **I**l roditore usa " **E** xtends" e **C** onsumer usa " **S** uper".

(Il produttore ha solo accesso in scrittura e Consumer ha solo accesso in lettura)

Vantaggi della classe generica e dell'interfaccia

Il codice che utilizza i generici ha molti vantaggi rispetto al codice non generico. Di seguito sono riportati i principali vantaggi

Controlli di tipo più potenti al momento della compilazione

Un compilatore Java applica un controllo di tipo forte al codice generico e genera errori se il codice viola la sicurezza del tipo. Correggere gli errori in fase di compilazione è più semplice che correggere errori di runtime, che possono essere difficili da trovare.

Eliminazione di calchi

Il seguente snippet di codice senza generici richiede il casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Quando viene riscritto per *utilizzare i generici* , il codice non richiede il casting:

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0);   // no cast
```

Abilitare i programmatori ad implementare algoritmi generici

Usando i generici, i programmatori possono implementare algoritmi generici che funzionano su raccolte di tipi diversi, possono essere personalizzati e sono sicuri e facili da leggere.

Associazione di parametri generici a più di 1 tipo

I parametri generici possono anche essere associati a più di un tipo usando la sintassi `T extends Type1 & Type2 & ...`

Diciamo che vuoi creare una classe il cui tipo generico dovrebbe implementare sia `Flushable` che `Closeable` , puoi scrivere

```
class ExampleClass<T extends Flushable & Closeable> {
}
```

Ora, `ExampleClass` accetta solo come parametri generici, tipi che implementano sia `Flushable` **che** `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.
```

I metodi di classe possono scegliere di dedurre argomenti di tipo generico come `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:

Non è possibile associare il parametro generico a uno dei tipi utilizzando la clausola *OR* (`|`). È supportata solo la clausola *AND* (`&`). Il tipo generico può estendere solo una classe e molte interfacce. La classe deve essere posizionata all'inizio della lista.

Istanziare un tipo generico

A causa della cancellazione del tipo, quanto segue non funzionerà:

```
public <T> void genericMethod() {
    T t = new T(); // Can not instantiate the type T.
}
```

Il tipo `T` viene cancellato. Dato che, in fase di esecuzione, la JVM non sa cosa fosse originariamente `T` , non sa quale costruttore chiamare.

---

soluzioni alternative

1. Passando alla classe di `T` quando si chiama `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);
```

Che genera eccezioni, poiché non c'è modo di sapere se la classe passata ha un costruttore predefinito accessibile.

Java SE 8

2. Passando un riferimento al costruttore di `T` :

```
public <T> void genericMethod(Supplier<T> cons) {
    T t = cons.get();
}
```

```
genericMethod(String::new);
```

Riferendosi al tipo generico dichiarato all'interno della propria dichiarazione

Come si fa a utilizzare un'istanza di un (eventualmente) ulteriore tipo generico ereditato all'interno di una dichiarazione di metodo nel tipo generico stesso che viene dichiarato? Questo è uno dei problemi che affronterai quando approfondirai un po 'più in profondità i generici, ma ancora piuttosto comune.

Supponiamo di avere un `DataSeries<T>` (interfaccia qui), che definisce una serie di dati generica contenente valori di tipo `T` È complicato lavorare con questo tipo direttamente quando vogliamo eseguire molte operazioni, ad esempio con valori doppi, quindi definiamo `DoubleSeries extends DataSeries<Double>` . Ora supponiamo che il tipo `DataSeries<T>` abbia un metodo `add(values)` che aggiunge un'altra serie della stessa lunghezza e ne restituisce una nuova. Come si impone il tipo di `values` e il tipo di ritorno da `DoubleSeries` anziché `DataSeries<Double>` nella nostra classe derivata?

Il problema può essere risolto aggiungendo un parametro di tipo generico che fa riferimento a ed estende il tipo da dichiarare (applicato a un'interfaccia qui, ma lo stesso è valido per le classi):

```
public interface DataSeries<T, DS extends DataSeries<T, DS>> {
    DS add(DS values);
    List<T> data();
}
```

Qui `T` rappresenta il tipo di dati che la serie detiene, ad es. `Double` e `DS` la serie stessa. Un tipo (o tipo) ereditato può ora essere facilmente implementato sostituendo il parametro sopra citato con un corrispondente tipo derivato, ottenendo così una definizione concreta basata sulla `Double` base della forma:

```
public interface DoubleSeries extends DataSeries<Double, DoubleSeries> {
    static DoubleSeries instance(Collection<Double> data) {
        return new DoubleSeriesImpl(data);
    }
}
```

In questo momento anche un IDE implementerà l'interfaccia di cui sopra con i tipi corretti in

posizione, che, dopo un po 'di riempimento del contenuto potrebbe assomigliare a questo:

```
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);
    }
}
```

Come puoi vedere il metodo `add` viene dichiarato come `DoubleSeries add(DoubleSeries values)` e il compilatore è soddisfatto.

Il modello può essere ulteriormente nidificato, se necessario.

Uso di instanceof con Generics

Usare i generici per definire il tipo in instanceof

Si consideri la seguente classe generica `Example` dichiarato con il parametro formale `<T>` :

```
class Example<T> {
    public boolean isTypeAString(String s) {
        return s instanceof T; // Compilation error, cannot use T as class type here
    }
}
```

Questo darà sempre un errore di compilazione perché non appena il compilatore compila il *codice sorgente Java* nel *bytecode Java* applica un processo noto come *cancellazione del tipo* , che converte tutto il codice generico in codice non generico, rendendo impossibile distinguere tra i tipi T in fase di esecuzione. Il tipo utilizzato con `instanceof` deve essere *reifiable* , il che significa che tutte le informazioni sul tipo devono essere disponibili in fase di esecuzione, e questo di solito non è il caso per i tipi generici.

La seguente classe rappresenta ciò che due diverse classi di `Example` , `Example<String>` ed `Example<Number>` , assomigliano dopo che i generici sono stati cancellati dalla *cancellazione del tipo*

:

```
class Example { // formal parameter is gone
    public boolean isTypeAString(String s) {
        return s instanceof Object; // Both <String> and <Number> are now Object
    }
}
```

Poiché i tipi sono andati, non è possibile per JVM sapere quale tipo è `T`

Eccezione alla regola precedente

È sempre possibile utilizzare un *carattere jolly illimitato* (?) Per specificare un tipo `instanceof` come segue:

```
    public boolean isAList(Object obj) {
        return obj instanceof List<?>;
    }
```

Questo può essere utile per valutare se un'istanza `obj` è 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
```

In effetti, il carattere jolly illimitato è considerato un tipo reifiable.

Utilizzando un'istanza generica con instanceof

L'altro lato della medaglia è che l'uso di un'istanza `t` di `T` con `instanceof` è legale, come mostrato nell'esempio seguente:

```
class Example<T> {
    public boolean isTypeAString(T t) {
        return t instanceof String; // No compilation error this time
    }
}
```

perché dopo la cancellazione del tipo la classe sarà simile alla seguente:

```
class Example { // formal parameter is gone
    public boolean isTypeAString(Object t) {
        return t instanceof String; // No compilation error this time
    }
}
```

Dato che, anche se la cancellazione del tipo avviene comunque, ora la JVM può distinguere tra diversi tipi di memoria, anche se usano lo stesso tipo di riferimento (`Object`), come mostra il seguente frammento:

```
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
```

Diversi modi per implementare un'interfaccia generica (o estendere una classe generica)

Supponiamo che la seguente interfaccia generica sia stata dichiarata:

```
public interface MyGenericInterface<T> {
    public void foo(T t);
}
```

Di seguito sono elencati i possibili modi per implementarlo.

Implementazione di classi non generiche con un tipo specifico

Scegliere un tipo specifico per sostituire il parametro di tipo formale `<T>` di `MyGenericClass` e implementarlo, come nell'esempio seguente:

```
public class NonGenericClass implements MyGenericInterface<String> {
    public void foo(String t) { } // type T has been replaced by String
}
```

Questa classe si occupa solo di `String` , e questo significa che l'utilizzo di `MyGenericInterface` con parametri diversi (ad esempio `Integer` , `Object` ecc.) Non verrà compilato, come mostra il seguente frammento:

```
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
```

Implementazione di classe generica

Dichiarare un'altra interfaccia generica con il parametro di tipo formale `<T>` che implementa `MyGenericInterface` , come segue:

```
public class MyGenericSubclass<T> implements MyGenericInterface<T> {
    public void foo(T t) { } // type T is still the same
    // other methods...
}
```

Si noti che un parametro di tipo formale diverso potrebbe essere stato utilizzato, come segue:

```
public class MyGenericSubclass<U> implements MyGenericInterface<U> { // equivalent to the
previous declaration
    public void foo(U t) { }
```

```
    // other methods...
}
```

Implementazione della classe del tipo non elaborata

Dichiarare una classe non generica che implementa `MyGenericInteface` come *tipo* non `MyGenericInteface` (senza utilizzare generici), come segue:

```
public class MyGenericSubclass implements MyGenericInterface {
    public void foo(Object t) { } // type T has been replaced by Object
    // other possible methods
}
```

In questo modo **non** è consigliabile, poiché non è sicuro al 100% in fase di esecuzione perché mischia il *tipo* non elaborato (della sottoclasse) con *generici* (dell'interfaccia) ed è anche fonte di confusione. I moderni compilatori Java solleveranno un avvertimento con questo tipo di implementazione, tuttavia il codice - per ragioni di compatibilità con JVM precedente (1.4 o precedenti) - verrà compilato.

Tutti i modi elencati sopra sono consentiti anche quando si utilizza una classe generica come supertipo invece di un'interfaccia generica.

Uso di Generics per il cast automatico

Con i generici, è possibile restituire ciò che il chiamante si aspetta:

```
private Map<String, Object> data;
public <T> T get(String key) {
    return (T) data.get(key);
}
```

Il metodo verrà compilato con un avviso. Il codice è in realtà più sicuro di quanto sembri perché il runtime Java eseguirà un cast quando lo si utilizza:

```
Bar bar = foo.get("bar");
```

È meno sicuro quando si utilizzano tipi generici:

```
List<Bar> bars = foo.get("bars");
```

Qui, il cast funzionerà quando il tipo restituito è un qualsiasi tipo di `List` (ad esempio, la restituzione di `List<String>` non innescherebbe un `ClassCastException` , ma alla fine lo si otterrà quando si eliminano elementi dall'elenco).

Per aggirare questo problema, puoi creare un'API che utilizza le chiavi digitate:

```
public final static Key<List<Bar>> BARS = new Key<>("BARS");
```

insieme a questo metodo `put()` :

```
public <T> T put(Key<T> key, T value);
```

Con questo approccio, non è possibile inserire il tipo sbagliato nella mappa, quindi il risultato sarà sempre corretto (a meno che non si creino accidentalmente due chiavi con lo stesso nome ma tipi diversi).

Relazionato:

- Mappa sicura

Ottieni una classe che soddisfi i parametri generici in fase di esecuzione

Molti parametri generici non associati, come quelli utilizzati in un metodo statico, non possono essere ripristinati in fase di esecuzione (vedere *Altri thread* su *Cancellazione*). Tuttavia esiste una strategia comune utilizzata per accedere al tipo che soddisfa un parametro generico su una classe in fase di esecuzione. Ciò consente il codice generico che dipende dall'accesso al tipo *senza* dover inserire informazioni sul tipo tramite ogni chiamata.

sfondo

La parametrizzazione generica su una classe può essere ispezionata creando una classe interna anonima. Questa classe acquisirà le informazioni sul tipo. In generale questo meccanismo è indicato come **token di tipo super** , che sono dettagliati nel post del blog di Neal Gafter .

implementazioni

Tre implementazioni comuni in Java sono:

- TipoToken di Guava
- Spring ParameterizedTypeReference
- Tipo di riferimento di Jackson

Esempio di utilizzo

```
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();
    }
}
```

Leggi Generics online: https://riptutorial.com/it/java/topic/92/generics

Capitolo 70: Gestione della memoria Java

Osservazioni

In Java, gli oggetti vengono allocati nell'heap e la memoria heap viene recuperata dalla garbage collection automatica. Un programma applicativo non può cancellare esplicitamente un oggetto Java.

I principi fondamentali della raccolta dei rifiuti Java sono descritte nella Garbage Collection esempio. Altri esempi descrivono la finalizzazione, come attivare manualmente il garbage collector e il problema delle perdite di archiviazione.

Examples

finalizzazione

Un oggetto Java può dichiarare un metodo `finalize` . Questo metodo viene chiamato poco prima che Java rilasci la memoria per l'oggetto. In genere sarà simile a questo:

```
public class MyClass {

    //Methods for the class

    @Override
    protected void finalize() throws Throwable {
        // Cleanup code
    }
}
```

Tuttavia, ci sono alcune avvertenze importanti sul comportamento della finalizzazione di Java.

- Java non garantisce quando verrà chiamato un metodo `finalize()` .
- Java non garantisce nemmeno che un metodo `finalize()` venga chiamato un po 'di tempo durante la vita dell'applicazione in esecuzione.
- L'unica cosa che è garantita è che il metodo verrà chiamato prima che l'oggetto sia cancellato ... se l'oggetto è cancellato.

Le avvertenze precedenti indicano che è una cattiva idea affidarsi al metodo `finalize` per eseguire azioni di pulizia (o altro) che devono essere eseguite in modo tempestivo. L'eccessivo affidamento sulla finalizzazione può portare a perdite di memoria, perdite di memoria e altri problemi.

In breve, ci sono pochissime situazioni in cui la finalizzazione è in realtà una buona soluzione.

I finalizzatori vengono eseguiti una volta sola

Normalmente, un oggetto viene cancellato dopo che è stato finalizzato. Tuttavia, questo non accade tutto il tempo. Considera il seguente esempio [1] :

```
public class CaptainJack {
    public static CaptainJack notDeadYet = null;

    protected void finalize() {
        // Resurrection!
        notDeadYet = this;
    }
}
```

Quando un'istanza di `CaptainJack` diventa irraggiungibile e il garbage collector tenta di recuperarlo, il metodo `finalize()` assegnerà un riferimento all'istanza alla variabile `notDeadYet` . Ciò renderà l'istanza nuovamente raggiungibile e il garbage collector non lo eliminerà.

Domanda: Il Capitano Jack è immortale?

Risposta: No.

Il problema è che JVM eseguirà un finalizzatore su un oggetto solo una volta nella sua vita. Se assegni `null` a `notDeadYet` facendo sì che un'istanza resurected diventi irraggiungibile ancora una volta, il garbage collector non chiamerà `finalize()` sull'oggetto.

1 - Vedi https://en.wikipedia.org/wiki/Jack_Harkness .

Attivazione manuale di GC

È possibile attivare manualmente il Garbage Collector chiamando

```
System.gc();
```

Tuttavia, Java non garantisce che il Garbage Collector sia eseguito quando la chiamata ritorna. Questo metodo semplicemente "suggerisce" alla JVM (Java Virtual Machine) che si desidera che esegua il garbage collector, ma non lo obbliga a farlo.

Generalmente è considerata una cattiva pratica tentare di attivare manualmente la garbage collection. La JVM può essere eseguita con l'opzione `-XX:+DisableExplicitGC` per disabilitare le chiamate a `System.gc()` . L'attivazione della garbage collection chiamando `System.gc()` può interrompere le normali attività di garbage management / object promotion dell'implementazione del garbage collector specifico in uso da parte della JVM.

Raccolta dei rifiuti

L'approccio C ++: nuovo e cancella

In un linguaggio come C ++, il programma applicativo è responsabile della gestione della memoria utilizzata dalla memoria allocata dinamicamente. Quando un oggetto viene creato nell'heap C ++ utilizzando il `new` operatore, è necessario che ci sia un corrispondente uso dell'operatore `delete` per disporre dell'oggetto:

- Se il programma si dimentica di `delete` un oggetto e semplicemente "dimentica" su di esso,

la memoria associata viene persa per l'applicazione. Il termine per questa situazione è una *perdita di memoria* , e troppa memoria perde una domanda è suscettibile di utilizzare sempre più memoria, e alla fine crash.

- D'altra parte, se un'applicazione tenta di `delete` due volte lo stesso oggetto, o di usare un oggetto dopo che è stato cancellato, l'applicazione rischia di bloccarsi a causa di problemi con il danneggiamento della memoria

In un complicato programma in C ++, l'implementazione della gestione della memoria utilizzando `new` e `delete` può richiedere molto tempo. In effetti, la gestione della memoria è una fonte comune di bug.

L'approccio Java - garbage collection

Java ha un approccio diverso. Invece di un operatore di `delete` esplicita, Java fornisce un meccanismo automatico noto come garbage collection per recuperare la memoria utilizzata da oggetti che non sono più necessari. Il sistema di runtime Java si assume la responsabilità di trovare gli oggetti da smaltire. Questa attività viene eseguita da un componente chiamato *garbage collector* o GC in breve.

In qualsiasi momento durante l'esecuzione di un programma Java, possiamo dividere l'insieme di tutti gli oggetti esistenti in due sottoinsiemi distinti [1] :

- Gli oggetti raggiungibili sono definiti da JLS come segue:

  > Un oggetto raggiungibile è qualsiasi oggetto a cui è possibile accedere in qualsiasi potenziale calcolo continuo da qualsiasi thread attivo.

  In pratica, ciò significa che esiste una catena di riferimenti che parte da una variabile locale in-scope o da una variabile `static` tramite la quale alcuni codici potrebbero essere in grado di raggiungere l'oggetto.

- Gli oggetti non raggiungibili sono oggetti che *non possono* essere raggiunti come sopra.

Qualsiasi oggetto irraggiungibile è *idoneo* per la garbage collection. Questo non significa che *saranno* raccolti dalla spazzatura. Infatti:

- Un oggetto irraggiungibile *non* viene raccolto immediatamente quando diventa irraggiungibile [1] .
- Un oggetto *non* raggiungibile *non può mai* essere sottoposto a garbage collection.

La specifica del linguaggio Java dà molta libertà a un'implementazione JVM per decidere quando raccogliere oggetti non raggiungibili. Inoltre (in pratica) dà il permesso che un'implementazione JVM sia conservativa nel modo in cui rileva oggetti non raggiungibili.

L'unica cosa che garantisce JLS è che nessun oggetto *raggiungibile* sarà mai raccolto.

Cosa succede quando un oggetto diventa irraggiungibile

Prima di tutto, nulla accade in modo specifico quando un oggetto *diventa* irraggiungibile. Le cose accadono solo quando il garbage collector viene eseguito *e* rileva che l'oggetto è irraggiungibile. Inoltre, è comune che una esecuzione GC non rilevi tutti gli oggetti non raggiungibili.

Quando il GC rileva un oggetto irraggiungibile, possono verificarsi i seguenti eventi.

1. Se ci sono `Reference` oggetti che fanno riferimento all'oggetto, tali riferimenti verranno cancellati prima che l'oggetto viene eliminato.

2. Se l'oggetto è *definibile* , sarà finalizzato. Questo succede prima che l'oggetto sia cancellato.

3. L'oggetto può essere cancellato e la memoria che occupa può essere recuperata.

Si noti che esiste una sequenza chiara in cui *possono* verificarsi gli eventi precedenti, ma nulla richiede che il garbage collector esegua la cancellazione finale di qualsiasi oggetto specifico in un intervallo di tempo specifico.

Esempi di oggetti raggiungibili e non raggiungibili

Considera le seguenti classi di esempio:

```
// 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
    }
}
```

Esaminiamo cosa succede quando viene chiamato `test()` . Le dichiarazioni T1, T2 e T3 creano oggetti `Node` e gli oggetti sono tutti raggiungibili tramite le variabili `n1` , `n2` e `n3` rispettivamente. L'istruzione T4 assegna il riferimento all'oggetto 2nd `Node` al campo `next` del primo. Quando ciò è fatto, il 2o `Node` è raggiungibile attraverso due percorsi:

```
 n2 -> Node2
 n1 -> Node1, Node1.next -> Node2
```

Nell'istruzione T5, assegniamo `null` a `n2` . Questo interrompe la prima delle catene di raggiungibilità per `Node2` , ma il secondo rimane ininterrotto, quindi il `Node2` è ancora raggiungibile.

Nell'istruzione T6, assegniamo `null` a `n3` . Questo rompe l'unica catena di raggiungibilità per `Node3` , il che rende `Node3` irraggiungibile. Tuttavia, `Node1` e `Node2` sono entrambi ancora raggiungibili tramite la variabile `n1` .

Infine, quando il metodo `test()` ritorna, le sue variabili locali `n1` , `n2` e `n3` escono dall'ambito e quindi non possono essere accessibili da nulla. Ciò interrompe le restanti catene di raggiungibilità per `Node1` e `Node2` e tutti gli oggetti `Node` non sono né irraggiungibili e sono *idonei* per la garbage collection.

---

1 - Questa è una semplificazione che ignora la finalizzazione e le classi di `Reference` . 2 - Ipoteticamente, un'implementazione di Java potrebbe farlo, ma il costo delle prestazioni di fare ciò lo rende poco pratico.

Impostazione delle dimensioni di heap, PermGen e stack

Quando viene avviata una macchina virtuale Java, è necessario sapere quanto è grande l'heap e le dimensioni predefinite per gli stack di thread. Questi possono essere specificati usando le opzioni della riga di comando sul comando `java` . Per le versioni di Java precedenti a Java 8, è anche possibile specificare la dimensione della regione PermGen dell'heap.

Si noti che PermGen è stato rimosso in Java 8 e, se si tenta di impostare la dimensione PermGen, l'opzione verrà ignorata (con un messaggio di avviso).

Se non si specificano le dimensioni di heap e stack in modo esplicito, la JVM utilizzerà i valori predefiniti calcolati in una versione e in una modalità specifica della piattaforma. Ciò potrebbe comportare che l'applicazione utilizzi troppo poca o troppa memoria. Questo è in genere OK per gli stack di thread, ma può essere problematico per un programma che utilizza molta memoria.

Impostazione delle dimensioni heap, permGen e stack predefinito:

Le seguenti opzioni JVM impostano la dimensione heap:

- `-Xms<size>` - imposta la dimensione heap iniziale
- `-Xmx<size>` : imposta la dimensione massima dell'heap
- `-XX:PermSize<size>` - imposta la dimensione iniziale di PermGen
- `-XX:MaxPermSize<size>` : imposta la dimensione massima di PermGen
- `-Xss<size>` - imposta la dimensione predefinita dello stack di thread

Il parametro `<size>` può essere un numero di byte o può avere un suffisso di `k` , `m` o `g` . Questi ultimi specificano le dimensioni in kilobyte, megabyte e gigabyte rispettivamente.

Esempi:

```
$ java -Xms512m -Xmx1024m JavaApp
$ java -XX:PermSize=64m -XX:MaxPermSize=128m JavaApp
$ java -Xss512k JavaApp
```

Trovare le dimensioni predefinite:

L'opzione `-XX:+printFlagsFinal` può essere utilizzata per stampare i valori di tutti i flag prima di avviare la JVM. Questo può essere usato per stampare i valori di default per le impostazioni di heap e stack come segue:

- Per Linux, Unix, Solaris e Mac OSX

  $ java -XX: + PrintFlagsFinal -version | grep -iE 'HeapSize | PermSize | ThreadStackSize'

- Per Windows:

  java -XX: + PrintFlagsFinal -version | findstr / i "HeapSize PermSize ThreadStackSize"

L'output dei comandi precedenti sarà simile al seguente:

```
uintx InitialHeapSize                          := 20655360        {product}
uintx MaxHeapSize                              := 331350016       {product}
uintx PermSize                                  = 21757952        {pd product}
uintx MaxPermSize                               = 85983232        {pd product}
 intx ThreadStackSize                           = 1024            {pd product}
```

Le dimensioni sono espresse in byte.

Perdite di memoria in Java

Nell'esempio della Garbage collection , abbiamo implicito che Java risolva il problema delle perdite di memoria. Questo non è in realtà vero. Un programma Java può perdere memoria, anche se le cause delle perdite sono piuttosto diverse.

Gli oggetti raggiungibili possono perdere

Si consideri la seguente implementazione dello stack ingenuo.

```
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;
    }
}
```

Quando si `push` un oggetto e poi subito `pop` , ci sarà ancora un riferimento all'oggetto nella `stack` matrice.

La logica dell'implementazione dello stack significa che quel riferimento non può essere restituito a un client dell'API. Se un oggetto è stato scoppiato, possiamo provare che non è *"possibile accedere a qualsiasi potenziale calcolo continuo da qualsiasi thread live"* . Il problema è che una JVM di generazione attuale non può dimostrarlo. Le JVM di generazione corrente non considerano la logica del programma nel determinare se i riferimenti sono raggiungibili. (Per cominciare, non è pratico).

Ma mettendo da parte il problema di cosa significhi veramente la *raggiungibilità* , abbiamo chiaramente una situazione in cui l'implementazione di `NaiveStack` è "aggrappata" a oggetti che dovrebbero essere recuperati. Questa è una perdita di memoria.

In questo caso, la soluzione è semplice:

```
    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;
    }
```

Le cache possono essere perdite di memoria

Una strategia comune per migliorare le prestazioni del servizio è memorizzare i risultati nella cache. L'idea è di tenere un registro delle richieste comuni e dei loro risultati in una struttura di dati in memoria nota come cache. Quindi, ogni volta che viene effettuata una richiesta, si cerca la richiesta nella cache. Se la ricerca ha esito positivo, si restituiscono i risultati salvati corrispondenti.

Questa strategia può essere molto efficace se implementata correttamente. Tuttavia, se implementata in modo errato, una cache può essere una perdita di memoria. Considera il seguente esempio:

```
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;
    }
}
```

Il problema con questo codice è che mentre qualsiasi chiamata a `doRequest` potrebbe aggiungere una nuova voce alla cache, non c'è nulla da rimuovere. Se il servizio riceve continuamente compiti diversi, alla fine la cache consumerà tutta la memoria disponibile. Questa è una forma di perdita di memoria.

Un approccio per risolvere questo è usare una cache con una dimensione massima e buttare fuori vecchie voci quando la cache supera il massimo. (Lanciare la voce meno utilizzata di recente è una buona strategia.) Un altro approccio è creare la cache usando `WeakHashMap` modo che la JVM possa sfrattare le voci della cache se l'heap inizia ad essere troppo pieno.

Leggi Gestione della memoria Java online: https://riptutorial.com/it/java/topic/2804/gestione-della-memoria-java

Capitolo 71: Getter e setter

introduzione

Questo articolo discute di getter e setter; il modo standard per fornire accesso ai dati nelle classi Java.

Examples

Aggiungere getter e setter

L'incapsulamento è un concetto base in OOP. Si tratta di avvolgere dati e codice come una singola unità. In questo caso, è buona norma dichiarare le variabili come `private` e quindi accedervi tramite `Getters` e `Setters` per visualizzarle e / o modificarle.

```
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;
  }
}
```

Non è possibile accedere a queste variabili private direttamente dall'esterno della classe. Quindi sono protetti da accessi non autorizzati. Ma se vuoi vederli o modificarli, puoi usare Getters e Setter.

`getXxx()` metodo `getXxx()` restituirà il valore corrente della variabile `xxx` , mentre è possibile impostare il valore della variabile `xxx` utilizzando `setXxx()` .

La convenzione di denominazione dei metodi è (nella variabile di esempio è chiamata `variableName`):

- Tutte le variabili non `boolean`

```
getVariableName()   //Getter, The variable name should start with uppercase
setVariableName(..) //Setter, The variable name should start with uppercase
```

- variabili `boolean`

```
isVariableName()     //Getter, The variable name should start with uppercase
setVariableName(...) //Setter, The variable name should start with uppercase
```

I getter e i setter pubblici fanno parte della definizione di proprietà di un bean Java.

Usare un setter o un getter per implementare un vincolo

Setter e Getter consentono a un oggetto di contenere variabili private a cui è possibile accedere e modificarle con restrizioni. Per esempio,

```
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;
    }
}
```

In questa classe `Person` , c'è una singola variabile: `name` . È possibile accedere a questa variabile utilizzando il metodo `getName()` e modificata utilizzando il metodo `setName(String)` , tuttavia, l'impostazione di un nome richiede che il nuovo nome abbia una lunghezza maggiore di 2 caratteri e non sia null. L'uso di un metodo setter invece di rendere pubblico il `name` della variabile consente ad altri di impostare il valore del `name` con determinate restrizioni. Lo stesso può essere applicato al metodo getter:

```
public String getName(){
   if(name.length()>16)
      return "Name is too large!";
   else
      return name;
}
```

Nel metodo `getName()` modificato sopra, il `name` viene restituito solo se la sua lunghezza è minore o uguale a 16. In caso contrario, viene restituito `"Name is too large"` . Ciò consente al programmatore di creare variabili che siano raggiungibili e modificabili come desiderano, impedendo alle classi client di modificare indesideratamente le variabili.

Perché usare getter e setter?

Considera una classe base contenente un oggetto con getter e setter in Java:

```
public class CountHolder {
  private int count = 0;
```

```
  public int getCount() { return count; }
  public void setCount(int c) { count = c; }
}
```

Non possiamo accedere alla variabile `count` perché è privata. Ma possiamo accedere ai metodi `getCount()` e `setCount(int)` perché sono pubblici. Per alcuni, questo potrebbe sollevare la domanda; perché introdurre l'intermediario? Perché non limitarsi a renderle pubbliche?

```
public class CountHolder {
  public int count = 0;
}
```

A tutti gli effetti, questi due sono esattamente gli stessi, dal punto di vista della funzionalità. La differenza tra loro è l'estensibilità. Considera cosa dice ogni classe:

- **Primo** : "Ho un metodo che ti darà un valore `int` e un metodo che imposterà quel valore su un altro `int` ".
- **Secondo** : "Ho un `int` che puoi impostare e ottenere come ti pare."

Questi potrebbero sembrare simili, ma il primo è in realtà molto più custodito nella sua natura; esso permette solo di interagire con la sua natura interna in **quanto** impone. Questo lascia la palla nel suo campo; può scegliere come si verificano le interazioni interne. Il secondo ha esposto esternamente la sua implementazione interna e ora non è solo incline agli utenti esterni, ma, nel caso di un'API, si **impegna** a mantenere tale implementazione (o altrimenti a rilasciare un'API compatibile non retrocompatibile).

Consideriamo se vogliamo sincronizzare l'accesso alla modifica e all'accesso al conteggio. Nel primo, questo è semplice:

```
public class CountHolder {
  private int count = 0;

  public synchronized int getCount() { return count; }
  public synchronized void setCount(int c) { count = c; }
}
```

ma nel secondo esempio, questo è ora quasi impossibile senza passare e modificare ogni luogo in cui viene fatto riferimento alla variabile `count` . Peggio ancora, se questo è un oggetto che stai fornendo in una biblioteca per essere consumato da altri, **non** hai un modo di eseguire quella modifica, e sei costretto a fare la difficile scelta di cui sopra.

Quindi elemosina la domanda; le variabili pubbliche sono sempre una buona cosa (o, almeno, non sono cattive)?

Non sono sicuro Da un lato, è possibile visualizzare esempi di variabili pubbliche che hanno superato la prova del tempo (IE: la variabile `out` a cui fa riferimento in `System.out`). Dall'altro, fornire una variabile pubblica non dà alcun beneficio al di fuori del sovraccarico estremamente minimo e della potenziale riduzione del prolisso. La mia linea guida qui sarebbe che, se hai intenzione di rendere pubblica una variabile, dovresti giudicarla contro questi criteri con **estremo**

pregiudizio:

1. La variabile dovrebbe avere alcuna ragione plausibile per cambiare **mai** nella sua attuazione. Questo è qualcosa che è estremamente facile da rovinare (e, anche se lo fai nel modo giusto, i requisiti possono cambiare), motivo per cui i getter / setter sono l'approccio comune. Se si ha una variabile pubblica, è necessario pensarci sopra, specialmente se rilasciata in una libreria / framework / API.
2. La variabile deve essere referenziata abbastanza frequentemente che i guadagni minimi derivanti dalla riduzione della verbosità lo giustificano. Non penso nemmeno che l'overhead per l'utilizzo di un metodo rispetto al riferimento diretto debba essere considerato qui. È troppo trascurabile per quello che valuterei prudentemente per il 99,9% delle applicazioni.

Probabilmente c'è più di quello che non ho considerato in cima alla mia testa. Se hai dubbi, usa sempre getter / setter.

Leggi Getter e setter online: https://riptutorial.com/it/java/topic/3560/getter-e-setter

Capitolo 72: Grafica 2D in Java

introduzione

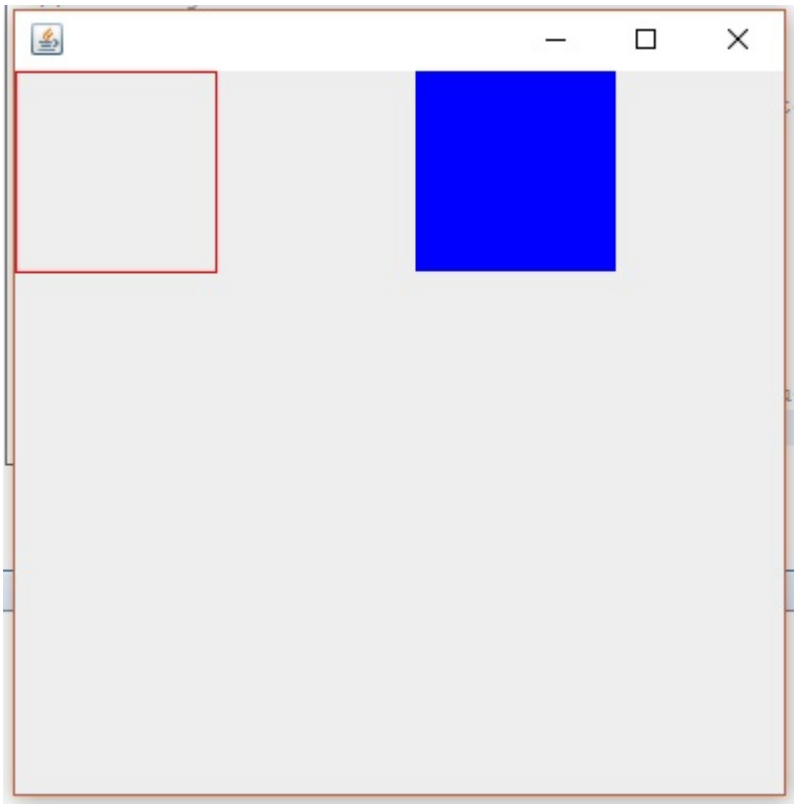
I grafici sono immagini o disegni visivi su una superficie, come un muro, una tela, uno schermo, una carta o una pietra per informare, illustrare o intrattenere. Comprende: rappresentazione grafica dei dati, come nella progettazione e produzione assistita da computer, nella composizione e nelle arti grafiche e nei software educativi e ricreativi. Le immagini generate da un computer si chiamano computer graphics.

L'API Java 2D è potente e complessa. Ci sono molti modi per fare grafica 2D in Java.

Examples

Esempio 1: Disegna e riempie un rettangolo usando Java

Questo è un esempio che stampa il rettangolo e il colore di riempimento nel rettangolo.



https://i.stack.imgur.com/dlC5v.jpg

La maggior parte dei metodi della classe Graphics può essere divisa in due gruppi di base:

1. Disegna e riempie i metodi, consentendo di eseguire il rendering di forme, testo e immagini di base
2. Attributi che impostano i metodi, che influenzano il modo in cui appare il disegno e il riempimento

Esempio di codice: iniziamo questo con un piccolo esempio di disegno di un rettangolo e

riempimento di colore in esso. Qui dichiariamo due classi, una classe è MyPanel e l'altra classe è Test. Nella classe MyPanel utilizziamo i mathod drawRect () e fillRect () per disegnare il rettangolo e riempire il colore in esso. Impostiamo il colore con il metodo setColor (Color.blue). In Second Class testiamo la nostra grafica, che è Test Class, creiamo un Frame e inseriamo MyPanel con p = new MyPanel () oggetto. In esecuzione Test Class vediamo un rettangolo e un rettangolo pieno di colore blu.

Prima classe: 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
    }
}
```

Seconda classe: test

```
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();
    }
}
```

Per ulteriori spiegazioni sul layout del bordo:
https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html

paintComponent ()

- È un metodo principale per dipingere
- Di default, per prima cosa dipinge lo sfondo
- Dopo di ciò, esegue la pittura personalizzata (disegno cerchio, rettangoli ecc.)

Graphic2D fa riferimento a Graphic2D Class

Nota: l'API Java 2D consente di eseguire facilmente le seguenti attività:

- Disegna linee, rettangoli e qualsiasi altra forma geometrica.
- Riempi quelle forme con colori solidi o sfumature e trame.
- Disegna il testo con le opzioni per un controllo preciso sul carattere e sul processo di rendering.
- Disegna le immagini, opzionalmente applicando le operazioni di filtro.
- Applicare operazioni come compositing e trasformazione durante una delle operazioni di rendering di cui sopra.

Esempio 2: disegno e riempimento ovale

```
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 width);
Questo metodo disegnerà un ovale alla posizione xey specificata con altezza e larghezza specificate.

g2.fillOval (int x, int y, int height, int width); Questo metodo riempie un ovale alla posizione xey specificata con altezza e larghezza specificate.

Leggi Grafica 2D in Java online: https://riptutorial.com/it/java/topic/10127/grafica-2d-in-java

Capitolo 73: HttpURLConnection

Osservazioni

- L'utilizzo di HttpUrlConnection su Android richiede l'aggiunta dell'autorizzazione Internet alla tua app (in `AndroidManifest.xml`).
- Esistono anche altri client e librerie Java HTTP, come OkHttp di Square, che sono più facili da usare e possono offrire prestazioni migliori o più funzionalità.

Examples

Ottieni il corpo della risposta da un URL come stringa

```
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();
}
```

Ciò scaricherà i dati di testo dall'URL specificato e li restituirà come una stringa.

Come funziona:

- Per prima cosa, creiamo `HttpUrlConnection` dal nostro URL, con il `new URL(url).openConnection()` . Trasmettiamo `UrlConnection` per tornare a `HttpUrlConnection` , così abbiamo accesso a cose come aggiungere intestazioni (come User Agent) o controllare il codice di risposta. (Questo esempio non lo fa, ma è facile da aggiungere.)

- Quindi, creare `InputStream` base al codice di risposta (per la gestione degli errori)
- Quindi, crea un `BufferedReader` che ci permette di leggere il testo da `InputStream` che otteniamo dalla connessione.
- Ora, aggiungiamo il testo a `StringBuilder` , riga per riga.
- Chiudi l' `InputStream` e restituisci la stringa che ora abbiamo.

Gli appunti:

- Questo metodo genererà una `IoException` di `IoException` in caso di errore (ad esempio un errore di rete o nessuna connessione Internet) e genererà anche *un'eccezione* `MalformedUrlException` *non selezionata* se l'URL specificato non è valido.
- Può essere utilizzato per la lettura da qualsiasi URL che restituisce testo, come pagine Web (HTML), API REST che restituiscono JSON o XML, ecc.
- Vedi anche: Leggi l'URL su String in poche righe di codice Java .

Uso:

È molto semplice:

```
String text = getText(”http://example.com");
//Do something with the text from example.com, in this case the HTML.
```

Dati 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();
    }
}
```

Questo invierà i dati all'URL specificato, quindi leggerà la risposta riga per riga.

Come funziona

- Come al solito otteniamo `HttpURLConnection` da un `URL` .
- Imposta il tipo di contenuto utilizzando `setRequestProperty` , per impostazione predefinita è `application/x-www-form-urlencoded`
- `setDoOutput(true)` indica la connessione che invieremo i dati.
- Quindi otteniamo `OutputStream` chiamando `getOutputStream()` e scrivendo i dati su di esso. Non dimenticare di chiuderlo dopo aver finito.
- Finalmente leggiamo la risposta del server.

Elimina risorsa

```
   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();
        }
    }
```

Questo ELIMINA la risorsa nell'URL specificato, quindi stampa l'intestazione della risposta.

Come funziona

- otteniamo `HttpURLConnection` da un `URL` .
- Imposta il tipo di contenuto utilizzando `setRequestProperty` , per impostazione predefinita è `application/x-www-form-urlencoded`
- `setDoInput(true)` indica alla connessione che intendiamo utilizzare la connessione URL per l'input.
- `setRequestMethod("DELETE")` per eseguire il DELETE HTTP

Finalmente stampiamo l'intestazione della risposta del server.

Controlla se esiste una risorsa

```
/**
 * 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;
}
```

Spiegazione:

Se stai solo controllando se esiste una risorsa, è meglio usare una richiesta HEAD piuttosto che una GET. Ciò evita il sovraccarico del trasferimento della risorsa.

Si noti che il metodo restituisce `true` solo se il codice di risposta è `200` . Se si anticipano le risposte di reindirizzamento (ovvero 3XX), potrebbe essere necessario migliorare il metodo per onorarle.

Esempio:

```
checkIfResourceExists(new URL("http://images.google.com/")); // true
checkIfResourceExists(new URL("http://pictures.google.com/")); // false
```

Leggi HttpURLConnection online: https://riptutorial.com/it/java/topic/156/httpurlconnection

Capitolo 74: I flussi

introduzione

Un `Stream` rappresenta una sequenza di elementi e supporta diversi tipi di operazioni per eseguire calcoli su tali elementi. Con Java 8, l'interfaccia `Collection` ha due metodi per generare un `Stream` : `stream()` e `parallelStream()` . `Stream` operazioni di `Stream` sono intermedie o terminali. Le operazioni intermedie restituiscono un `Stream` modo che più operazioni intermedie possano essere concatenate prima che il `Stream` venga chiuso. Le operazioni terminal sono nulle o restituiscono un risultato non stream.

Sintassi

- collection.stream ()
- Arrays.stream (array)
- Stream.iterate (firstValue, currentValue -> nextValue)
- Stream.generate (() -> valore)
- Stream.of (elementOfT [, elementOfT, ...])
- Stream.empty ()
- StreamSupport.stream (iterable.spliterator (), false)

Examples

Usando i flussi

Un `Stream` è una sequenza di elementi su cui è possibile eseguire operazioni di aggregazione sequenziali e parallele. Qualsiasi `Stream` dato può potenzialmente contenere una quantità illimitata di dati. Di conseguenza, i dati ricevuti da un `Stream` vengono elaborati singolarmente al loro arrivo, anziché eseguire l'elaborazione batch sui dati del tutto. Quando combinati con espressioni lambda forniscono un modo conciso per eseguire operazioni su sequenze di dati utilizzando un approccio funzionale.

Esempio: (vedi funziona su 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);
```

Produzione:

MELA
BANANA

ARANCIA
PERA

Le operazioni eseguite dal codice precedente possono essere riassunte come segue:

1. Crea uno `Stream<String>` contenente un `Stream` ordinato di elementi `String` di `Stream` ordinati utilizzando il metodo statico di fabbrica `Stream.of(values)` .

2. L'operazione `filter()` conserva solo gli elementi che corrispondono a un determinato predicato (gli elementi che vengono testati dal predicato restituiscono true). In questo caso, mantiene gli elementi contenenti una `"a"` . Il predicato è dato come espressione lambda .

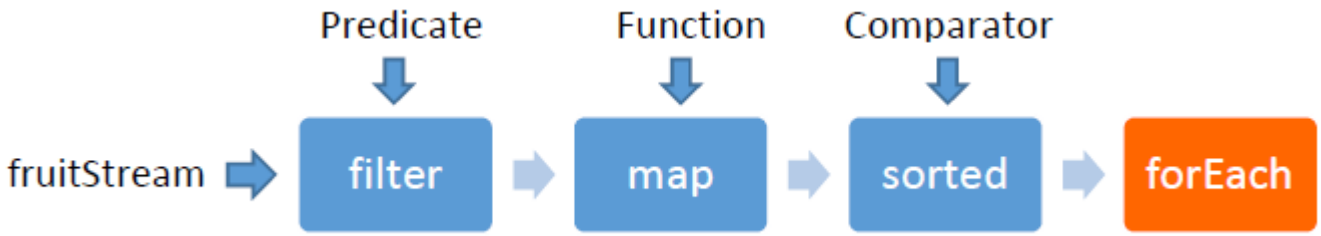
3. L'operazione `map()` trasforma ogni elemento usando una determinata funzione, chiamata mapper. In questo caso, ogni Fruit `String` viene associata alla sua versione `String` maiuscolo utilizzando il metodo method: `String::toUppercase` .

   > **Si noti** che l'operazione `map()` restituirà un flusso con un diverso tipo generico se la funzione di mappatura restituisce un tipo diverso dal proprio parametro di input. Ad esempio su uno `Stream<String>` chiama `.map(String::isEmpty)` restituisce un `Stream<Boolean>`

4. L'operazione sort `sorted()` ordina gli elementi del `Stream` base al loro ordinamento naturale (lessicograficamente, nel caso di `String`).

5. Infine, l'operazione `forEach(action)` esegue un'azione che agisce su ciascun elemento del `Stream` , passandolo a un consumatore . Nell'esempio, ogni elemento viene semplicemente stampato sulla console. Questa operazione è un'operazione terminale, rendendo impossibile l'operazione su di esso.

   > **Si noti** che le operazioni definite sullo `Stream` vengono eseguite a *causa* dell'operazione del terminale. Senza un'operazione terminale, lo stream non viene elaborato. Gli stream non possono essere riutilizzati. Una volta che viene chiamata un'operazione terminale, l'oggetto `Stream` diventa inutilizzabile.



Le operazioni (come visto sopra) sono concatenate per formare ciò che può essere visto come una query sui dati.

Flussi di chiusura

> **Nota che un `Stream` generalmente non deve essere chiuso.** È richiesto solo per

chiudere i flussi che operano su canali IO. La maggior parte dei tipi di `Stream` non funziona su risorse e pertanto non richiede la chiusura.

L'interfaccia `Stream` estende `AutoCloseable` . Gli stream possono essere chiusi chiamando il metodo `close` o usando le istruzioni try-with-resource.

Un esempio di caso d'uso in cui un `Stream` dovrebbe essere chiuso è quando crei un `Stream` di linee da un file:

```
try (Stream<String> lines = Files.lines(Paths.get("somePath"))) {
    lines.forEach(System.out::println);
}
```

L'interfaccia `Stream` dichiara anche il metodo `Stream.onClose()` che consente di registrare i gestori `Runnable` che verranno chiamati quando lo stream viene chiuso. Un caso di utilizzo di esempio è dove il codice che produce un flusso deve sapere quando viene utilizzato per eseguire una pulizia.

```
public Stream<String>streamAndDelete(Path path) throws IOException {
    return Files.lines(path).onClose(() -> someClass.deletePath(path));
}
```

Il gestore di esecuzione verrà eseguito solo se viene chiamato il metodo `close()` , esplicitamente o implicitamente da un'istruzione try-with-resources.

Ordine di elaborazione

L'elaborazione di un oggetto `Stream` può essere sequenziale o parallela .

In una modalità **sequenziale** , gli elementi vengono elaborati nell'ordine della sorgente del `Stream` . Se il `Stream` è ordinato (come un'implementazione `SortedMap` o un `List`), l'elaborazione è garantita per corrispondere all'ordine della fonte. In altri casi, tuttavia, è necessario prestare attenzione a non dipendere dall'ordinamento (si veda: l'ordine di iterazione di Java `HashMap keySet()` coerente?).

Esempio:

```
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
```

Vivi su Ideone

La modalità **parallela** consente l'utilizzo di più thread su più core, ma non vi è alcuna garanzia dell'ordine in cui vengono elaborati gli elementi.

Se più metodi vengono richiamati su un `Stream` sequenziale, non tutti i metodi devono essere richiamati. Ad esempio, se un `Stream` viene filtrato e il numero di elementi è ridotto a uno, non verrà eseguita una chiamata successiva a un metodo come l' `sort` . Ciò può aumentare le prestazioni di un `Stream` sequenziale: un'ottimizzazione che non è possibile con un `Stream` parallelo.

Esempio:

```
// parallel
long howManyOddNumbersParallel = integerList.parallelStream()
                                            .filter(e -> (e % 2) == 1)
                                            .count();

System.out.println(howManyOddNumbersParallel); // Output: 2
```

Vivi su Ideone

Differenze da contenitori (o collezioni)

Mentre alcune azioni possono essere eseguite sia su Containers che su Stream, alla fine hanno scopi diversi e supportano diverse operazioni. I contenitori sono più focalizzati su come gli elementi sono memorizzati e su come è possibile accedere a tali elementi in modo efficiente. Un `Stream` , d'altra parte, non fornisce accesso diretto e manipolazione ai suoi elementi; è più dedicato al gruppo di oggetti come entità collettiva ed esegue operazioni su quell'entità nel suo complesso. `Stream` e `Collection` sono astrazioni di alto livello separate per questi scopi diversi.

Raccogli elementi di un flusso in una raccolta

Raccogli con `toList()` e `toSet()`

Gli elementi di un `Stream` possono essere facilmente raccolti in un contenitore usando l'operazione `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]
```

Altre istanze di raccolta, come un `Set` , possono essere create usando altri metodi built-in di `Collectors` . Ad esempio, `Collectors.toSet()` raccoglie gli elementi di un `Stream` in un `Set` .

Controllo esplicito sull'implementazione di `List` o `Set`

Secondo la documentazione di `Collectors#toList()` e `Collectors#toSet()` , non ci sono garanzie sul

tipo, sulla mutevolezza, sulla serializzabilità o sulla sicurezza del thread `List` o del `Set` restituito.

Per il controllo esplicito dell'implementazione da restituire, è possibile utilizzare `Collectors#toCollection(Supplier)` , in cui il fornitore specificato restituisce una raccolta nuova e vuota.

```
// 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<>()))
);
```

Raccolta di elementi usando toMap

Il raccoglitore accumula elementi in una mappa, dove la chiave è l'id dello studente e il valore è il valore dello studente.

```
  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);
```

Produzione :

```
{1=test1, 2=test2, 3=test3}
```

Collectors.toMap ha un'altra implementazione `Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper, BinaryOperator<U> mergeFunction)` . La funzione mergeFunction viene utilizzata principalmente per selezionare un nuovo valore o mantenere il vecchio valore se la chiave viene ripetuta quando si aggiunge un nuovo membro nella mappa da un elenco.

Spesso la mergeFunction assomiglia a: `(s1, s2) -> s1` per mantenere il valore corrispondente alla chiave ripetuta, o `(s1, s2) -> s2` per inserire un nuovo valore per la chiave ripetuta.

Raccolta di elementi per la mappa delle collezioni

Esempio: da ArrayList a Map <String, List <>>

Spesso è necessario creare una mappa dell'elenco da un elenco principale. Esempio: da uno studente della lista, abbiamo bisogno di fare una mappa dell'elenco delle materie per ogni

studente.

```
    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);
```

Produzione:

```
{ Robert=[LITERATURE],
Sascha=[ENGLISH, MATH, SCIENCE, LITERATURE],
Davis=[MATH, SCIENCE, GEOGRAPHY] }
```

Esempio: da 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);
```

Produzione:

```
{ 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]} }
```

Cheat-sheet

Obbiettivo	Codice
Raccogli a una `List`	`Collectors.toList()`
Raccogli a un `ArrayList` con dimensioni pre-allocate	`Collectors.toCollection(() -> new ArrayList<>(size))`
Raccogli per un `Set`	`Collectors.toSet()`
Raccogli in un `Set` con prestazioni di iterazione migliori	`Collectors.toCollection(() -> new LinkedHashSet<>())`
Raccogli in un `Set<String>` insensibile alle maiuscole / minuscole `Set<String>`	`Collectors.toCollection(() -> new TreeSet<>(String.CASE_INSENSITIVE_ORDER))`
Raccogli a `EnumSet<AnEnum>` (miglior rendimento per enumerazioni)	`Collectors.toCollection(() -> EnumSet.noneOf(AnEnum.class))`
Raccogli a una `Map<K,V>` con chiavi univoche	`Collectors.toMap(keyFunc,valFunc)`
Mappare MyObject.getter () su MyObject univoco	`Collectors.toMap(MyObject::getter, Function.identity())`
Mappare MyObject.getter () su più MyObjects	`Collectors.groupingBy(MyObject::getter)`

Stream infiniti

È possibile generare un `Stream` che non termina. Chiamando un metodo terminale su un `Stream` infinito `Stream` il `Stream` entra in un ciclo infinito. Il metodo `limit` di un `Stream` può essere utilizzato per limitare il numero di termini del `Stream` che Java elabora.

Questo esempio genera un `Stream` di tutti i numeri naturali, a partire dal numero 1. Ogni successivo termine del `Stream` è uno più alto del precedente. Chiamando il metodo limite di questo `Stream` , vengono considerati e stampati solo i primi cinque termini 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);
```

Produzione:

1
2
3
4
5

Un altro modo per generare un flusso infinito è utilizzare il metodo Stream.generate . Questo metodo richiede un lambda di tipo Supplier .

```
// 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);
```

Flussi di consumo

Un `Stream` sarà attraversato solo quando c'è un'operazione *terminale* , come `count()` , `collect()` o `forEach()` . In caso contrario, non verrà eseguita alcuna operazione sul `Stream` .

Nell'esempio seguente, nessuna operazione terminale viene aggiunta al `Stream` , quindi l'operazione `filter()` non verrà invocata e non verrà prodotto alcun output perché `peek()` NON è un'operazione *terminale* .

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).peek(System.out::println);
```

Vivi su Ideone

Questa è una sequenza `Stream` con un'operazione di *terminale* valida, quindi viene prodotta un'uscita.

Puoi anche usare `forEach` invece di `peek` :

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).forEach(System.out::println);
```

Vivi su Ideone

Produzione:

2
4
6
8

Dopo aver eseguito l'operazione del terminale, il `Stream` viene consumato e non può essere riutilizzato.

Sebbene un determinato oggetto flusso non possa essere riutilizzato, è facile creare un `Iterable` riutilizzabile che deleghi ad una pipeline di flusso. Questo può essere utile per restituire una vista modificata di un set di dati live senza dover raccogliere i risultati in una struttura temporanea.

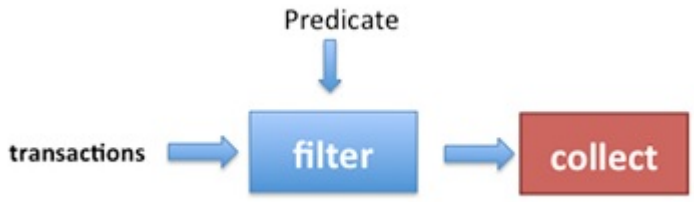
```
List<String> list = Arrays.asList("FOO", "BAR");
Iterable<String> iterable = () -> list.stream().map(String::toLowerCase).iterator();

for (String str : iterable) {
    System.out.println(str);
}
for (String str : iterable) {
    System.out.println(str);
}
```

Produzione:

> foo
> bar
> foo
> bar

Questo funziona perché `Iterable` dichiara un singolo metodo astratto `Iterator<T> iterator()` . Ciò lo rende efficacemente un'interfaccia funzionale, implementata da un lambda che crea un nuovo stream per ogni chiamata.

In generale, un `Stream` funziona come mostrato nell'immagine seguente:



NOTA : i controlli degli argomenti vengono sempre eseguiti, anche senza un'operazione di *terminale* :

```
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()");
}
```

Vivi su Ideone

Produzione:

> Abbiamo ottenuto una NullPointerException in quanto null è stato passato come argomento a filter ()

Creazione di una mappa di frequenza

Il collector `groupingBy(classifier, downstream)` consente la raccolta di elementi `Stream` in una `Map` classificando ciascun elemento in un gruppo ed eseguendo un'operazione downstream sugli elementi classificati nello stesso gruppo.

Un classico esempio di questo principio è l'utilizzo di una `Map` per contare le occorrenze di elementi in un `Stream` . In questo esempio, il classificatore è semplicemente la funzione di identità, che restituisce l'elemento così com'è. L'operazione downstream conta il numero di elementi uguali, usando il `counting()` .

```
Stream.of("apple", "orange", "banana", "apple")
      .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
      .entrySet()
      .forEach(System.out::println);
```

L'operazione downstream è essa stessa un collector (`Collectors.counting()`) che opera su elementi di tipo String e produce un risultato di tipo `Long` . Il risultato della chiamata al metodo `collect` è una `Map<String, Long>` .

Questo produrrebbe il seguente risultato:

banane = 1
arancione = 1
mela = 2

Stream parallelo

Nota: prima di decidere quale `Stream` utilizzare, dare un'occhiata al comportamento di ParallelStream vs Sequential Stream .

Quando si desidera eseguire contemporaneamente operazioni di `Stream` , è possibile utilizzare uno di questi modi.

```
List<String> data = Arrays.asList("One", "Two", "Three", "Four", "Five");
Stream<String> aParallelStream = data.stream().parallel();
```

O:

```
Stream<String> aParallelStream = data.parallelStream();
```

Per eseguire le operazioni definite per lo streaming parallelo, chiamare un operatore di terminale:

```
aParallelStream.forEach(System.out::println);
```

(Un possibile) output dal `Stream` parallelo:

Tre

quattro
Uno
Due
Cinque

L'ordine potrebbe cambiare in quanto tutti gli elementi sono elaborati in parallelo (il che *potrebbe* renderlo più veloce). Usa `parallelStream` quando l'ordine non ha importanza.

Impatto sulle prestazioni

Nel caso in cui sia coinvolta la rete, i `Stream` paralleli possono peggiorare le prestazioni generali di un'applicazione in quanto tutti i `Stream` paralleli utilizzano un comune pool di thread fork-join per la rete.

D'altra parte, parallel `Stream` s può migliorare significativamente le prestazioni in molti altri casi, a seconda del numero di core disponibili nella CPU in esecuzione al momento.

Conversione di un flusso di facoltativo in un flusso di valori

Potrebbe essere necessario convertire un `Stream` emette `Optional` in un `Stream` di valori, emettendo solo valori da `Optional` esistente. (vale a dire: senza valore `null` e senza trattare `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]
```

Creazione di un flusso

Tutti i `Collection<E>` s di java `Collection<E>` hanno i metodi `stream()` e `parallelStream()` da cui è possibile costruire uno `Stream<E>` :

```
Collection<String> stringList = new ArrayList<>();
Stream<String> stringStream = stringList.parallelStream();
```

È possibile creare uno `Stream<E>` da una matrice utilizzando uno dei seguenti due metodi:

```
String[] values = { "aaa", "bbbb", "ddd", "cccc" };
Stream<String> stringStream = Arrays.stream(values);
Stream<String> stringStreamAlternative = Stream.of(values);
```

La differenza tra `Arrays.stream()` e `Stream.of()` è che `Stream.of()` ha un parametro varargs, quindi può essere usato come:

```
Stream<Integer> integerStream = Stream.of(1, 2, 3);
```

Ci sono anche i primitivi `Stream` che puoi usare. Per esempio:

```
IntStream intStream = IntStream.of(1, 2, 3);
DoubleStream doubleStream = DoubleStream.of(1.0, 2.0, 3.0);
```

Questi stream primitivi possono anche essere costruiti usando il metodo `Arrays.stream()` :

```
IntStream intStream = Arrays.stream(new int[]{ 1, 2, 3 });
```

È possibile creare un `Stream` da una matrice con un intervallo specificato.

```
int[] values= new int[]{1, 2, 3, 4, 5};
IntStream intStram = Arrays.stream(values, 1, 3);
```

Si noti che qualsiasi flusso primitivo può essere convertito in stream di tipo in scatola usando il metodo in `boxed` :

```
Stream<Integer> integerStream = intStream.boxed();
```

Questo può essere utile in alcuni casi se si desidera raccogliere i dati poiché il flusso primitivo non ha alcun metodo di `collect` che accetta un `Collector` come argomento.

Riutilizzo delle operazioni intermedie di una catena di flusso

Lo streaming viene chiuso quando viene chiamata un'operazione terminale. Riutilizzare il flusso di operazioni intermedie, quando solo il funzionamento del terminale è solo variabile. potremmo creare un fornitore di stream per costruire un nuovo stream con tutte le operazioni intermedie già configurate.

```
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[]` array `int[]` possono essere convertiti in `List<Integer>` utilizzando gli stream

```
int[] ints = {1,2,3};
List<Integer> list = IntStream.of(ints).boxed().collect(Collectors.toList());
```

Ricerca di statistiche sui flussi numerici

Java 8 fornisce classi chiamate `IntSummaryStatistics` , `DoubleSummaryStatistics` e `LongSummaryStatistics` che forniscono un oggetto di stato per la raccolta di statistiche quali `count` , `min` , `max` , `sum` e `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);
```

Quale risulterà in:

Java SE 8

```
IntSummaryStatistics{count=10, sum=55, min=1, max=10, average=5.500000}
```

Ottieni una fetta di un flusso

Esempio: ottieni un `Stream` di 30 elementi, contenente elementi dal 21 ° al 50 ° (inclusi) di una raccolta.

```
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);
```

Gli appunti:

- `IllegalArgumentException` viene generato se `n` è negativo o `maxSize` è negativo
- sia `skip(long)` che `limit(long)` sono operazioni intermedie
- se un flusso contiene meno di `n` elementi, `skip(n)` restituisce un flusso vuoto
- sia `skip(long)` che `limit(long)` sono operazioni a basso costo su pipeline sequenziali, ma possono essere piuttosto costose su condotte parallele ordinate

Concatenare flussi

Dichiarazione variabile per esempi:

```
Collection<String> abc = Arrays.asList("a", "b", "c");
Collection<String> digits = Arrays.asList("1", "2", "3");
Collection<String> greekAbc = Arrays.asList("alpha", "beta", "gamma");
```

Esempio 1 - Concatena due `Stream` s

```
final Stream<String> concat1 = Stream.concat(abc.stream(), digits.stream());

concat1.forEach(System.out::print);
```

```
// prints: abc123
```

Esempio 2: concatena più di due `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
```

In alternativa, per semplificare il nidificata `concat()` Sintassi il `Stream` s può anche essere concatenato 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
```

Fai attenzione quando costruisci `Stream` s dalla concatenazione ripetuta, perché l'accesso a un elemento di un `Stream` profondamente concatenato può portare a catene di chiamate profonde o persino a `StackOverflowException` .

IntStream a stringa

Java non ha un *flusso di caratteri* , quindi quando si lavora con `String` s e si costruisce un `Stream` di `Character` s, un'opzione è quella di ottenere un `IntStream` di punti di codice usando il metodo `String.codePoints()` . Quindi `IntStream` può essere ottenuto come di seguito:

```
public IntStream stringToIntStream(String in) {
  return in.codePoints();
}
```

È un po 'più complicato fare la conversione in un altro modo, cioè IntStreamToString. Questo può essere fatto come segue:

```
public String intStreamToString(IntStream intStream) {
  return intStream.collect(StringBuilder::new, StringBuilder::appendCodePoint,
StringBuilder::append).toString();
}
```

Ordina utilizzando 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);
```

Produzione:

```
[Sydney, London, New York, Amsterdam, Mumbai, California]
[Amsterdam, California, London, Mumbai, New York, Sydney]
```

È anche possibile utilizzare un meccanismo di confronto diverso in quanto esiste una versione `sorted` sovraccarico che accetta come argomento un comparatore.

Inoltre, puoi usare un'espressione lambda per l'ordinamento:

```
List<String> sortedData2 = data.stream().sorted((s1,s2) ->
s2.compareTo(s1)).collect(Collectors.toList());
```

Ciò produrrebbe `[Sydney, New York, Mumbai, London, California, Amsterdam]`

Puoi usare `Comparator.reverseOrder()` per avere un comparatore che impone il `reverse` dell'ordine naturale.

```
List<String> reverseSortedData =
data.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
```

Flussi di primitivi

Java fornisce `Stream` specializzati per tre tipi di primitive `IntStream` (per `int` s), `LongStream` (per `long` s) e `DoubleStream` (per `double` s). Oltre ad essere implementazioni ottimizzate per le rispettive primitive, forniscono anche diversi metodi terminali specifici, in genere per operazioni matematiche. Per esempio:

```
IntStream is = IntStream.of(10, 20, 30);
double average = is.average().getAsDouble(); // average is 20.0
```

Raccogli i risultati di un flusso in una matrice

Analogico per ottenere una raccolta per uno `Stream` di `collect()` possibile ottenere un array con il metodo `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` è un tipo speciale di riferimento al metodo: un riferimento costruttore.

Trovare il primo elemento che corrisponde a un predicato

È possibile trovare il primo elemento di un `Stream` che corrisponde a una condizione.

Per questo esempio, troveremo il primo numero `Integer` cui quadrato è superiore a `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
```

Questa espressione restituirà un `OptionalInt` con il risultato.

Nota che con un `Stream` infinito, Java continuerà a controllare ogni elemento fino a quando non troverà un risultato. Con un `Stream` , se Java esaurisce gli elementi ma non riesce a trovare un risultato, restituisce un oggetto `OptionalInt` vuoto.

Utilizzo di IntStream per iterare su indici

`Stream` elementi di `Stream` di solito non consentono l'accesso al valore dell'indice dell'oggetto corrente. Per scorrere su un array o `ArrayList` mentre si ha accesso agli indici, utilizzare `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);
```

Il metodo `range(start, endExclusive)` restituisce un altro `IntStream` e `mapToObj(mapper)` restituisce un flusso di `String` .

Produzione:

```
# 1 Jon
# 2 Darin
# 3 Bauke
# 4 Hans
# 5 Marc
```

Questo è molto simile all'uso di un normale `for` ciclo con un contatore, ma con il vantaggio di pipelining e parallelizzazione:

```
for (int i = 0; i < names.length; i++) {
    String newName = String.format("#%d %s", i + 1, names[i]);
```

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

Flattenare i flussi con flatMap ()

Un `Stream` di elementi a sua volta scorrevoli può essere appiattito in un unico `Stream` continuo:

La matrice dell'elenco di elementi può essere convertita in una singola 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]
```

La mappa contenente l'elenco di elementi come valori può essere appiattita in un elenco combinato

```
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` di `Map` può essere appiattito in un singolo `Stream` continuo

```
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]
```

Crea una mappa basata su un flusso

Caso semplice senza chiavi duplicate

```
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}
```

Per rendere le cose più dichiarative, possiamo usare il metodo statico in `Function` interface - `Function.identity()` . Possiamo sostituire questo `element -> element` lambda `element -> element` con `Function.identity()` .

Caso in cui potrebbero esserci chiavi duplicate

Il javadoc per gli stati `Collectors.toMap` :

> Se le chiavi mappate contengono duplicati (in base a `Object.equals(Object)`), viene generata una `IllegalStateException` quando viene eseguita l'operazione di raccolta. Se le chiavi mappate possono avere duplicati, utilizzare invece `toMap(Function, Function, BinaryOperator)` .

```
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}
```

`BinaryOperator` passato a `Collectors.toMap(...)` genera il valore da memorizzare nel caso di una collisione. Può:

- restituisce il vecchio valore, in modo che il primo valore nel flusso abbia la precedenza,
- restituire il nuovo valore, in modo che l'ultimo valore nel flusso abbia la precedenza o
- combinare i valori vecchi e nuovi

Raggruppamento per valore

È possibile utilizzare `Collectors.groupingBy` quando è necessario eseguire l'equivalente di un'operazione "raggruppa per" in cascata del database. Per illustrare, il seguente crea una mappa in cui i nomi delle persone sono mappati ai cognomi:

```
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]}
```

Vivi su Ideone

Generazione di stringhe casuali usando flussi

A volte è utile creare `Strings` casuali, magari come ID sessione per un servizio Web o una password iniziale dopo la registrazione per un'applicazione. Questo può essere facilmente ottenuto usando `Stream` s.

Per prima cosa dobbiamo inizializzare un generatore di numeri casuali. Per migliorare la sicurezza per le `String` generate, è una buona idea usare `SecureRandom` .

Nota : la creazione di un `SecureRandom` è piuttosto costosa, quindi è consigliabile farlo una volta sola e chiamare uno dei suoi metodi `setSeed()` di volta in volta per renderizzarlo.

```
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
```

Quando creiamo `String` casuali, di solito vogliamo che usino solo determinati caratteri (ad es. Solo lettere e cifre). Pertanto possiamo creare un metodo che restituisce un valore `boolean` che può essere successivamente utilizzato per filtrare il `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);
}
```

Quindi possiamo utilizzare l'RNG per generare una stringa casuale di lunghezza specifica contenente il set di caratteri che supera il nostro controllo `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;
```

```
}
```

Utilizzo degli stream per implementare funzioni matematiche

`Stream` `IntStream` , e in particolare quelli `IntStream` , rappresentano un modo elegante per implementare i termini di somma (Σ). Le gamme del `Stream` possono essere utilizzate come limiti della sommatoria.

Ad esempio, l'approssimazione di Pi di Madhava è data dalla formula (Fonte: wikipedia):

$$\pi = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-3)^{-k}}{2k+1} = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-\frac{1}{3})^{k}}{2k+1} = \sqrt{12}\left(\frac{1}{1\cdot 3^0} - \frac{1}{3\cdot 3^1} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots\right)$$

Questo può essere calcolato con una precisione arbitraria. Ad esempio, per 101 termini:

```
double pi = Math.sqrt(12) *
            IntStream.rangeClosed(0, 100)
                     .mapToDouble(k -> Math.pow(-3, -1 * k) / (2 * k + 1))
                     .sum();
```

Nota: con la precisione del `double` , selezionare un limite superiore di 29 è sufficiente per ottenere un risultato indistinguibile da `Math.Pi`

Utilizzo degli stream e dei riferimenti al metodo per scrivere processi di auto-documentazione

I riferimenti ai metodi costituiscono un codice di auto-documentazione eccellente e l'utilizzo dei riferimenti ai metodi con `Stream` s semplifica la lettura e la comprensione dei processi complicati. Considera il seguente codice:

```
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;
}
```

Quest'ultimo metodo riscritto usando i riferimenti di `Stream` s e method è molto più leggibile e ogni fase del processo è facilmente e facilmente comprensibile - non è solo più breve, ma mostra anche a colpo d'occhio quali interfacce e classi sono responsabili del codice in ogni passaggio:

```
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());
}
```

Utilizzo degli stream di Map.Entry per preservare i valori iniziali dopo la mappatura

Quando hai uno `Stream` devi mappare ma vuoi preservare anche i valori iniziali, puoi mappare lo `Stream` su `Map.Entry<K,V>` usando un metodo di utilità come il seguente:

```
public static <K, V> Function<K, Map.Entry<K, V>> entryMapper(Function<K, V> mapper){
    return (k)->new AbstractMap.SimpleEntry<>(k, mapper.apply(k));
}
```

Quindi è possibile utilizzare il convertitore per elaborare i `Stream` che hanno accesso sia ai valori originali sia a quelli mappati:

```
Set<K> mySet;
Function<K, V> transformer = SomeClass::transformerMethod;
Stream<Map.Entry<K, V>> entryStream = mySet.stream()
    .map(entryMapper(transformer));
```

È quindi possibile continuare a elaborare quel `Stream` come di consueto. Ciò evita il sovraccarico di creazione di una raccolta intermedia.

Categorie di operazioni di streaming

Le operazioni di streaming rientrano in due categorie principali, operazioni intermedie e terminali e

due sottocategorie, stateless e stateful.

Operazioni intermedie:

Un'operazione intermedia è sempre *pigra* , ad esempio un semplice `Stream.map` . Non è invocato fino a quando il flusso non viene effettivamente consumato. Questo può essere verificato facilmente:

```
Arrays.asList(1, 2 ,3).stream().map(i -> {
    throw new RuntimeException("not gonna happen");
    return i;
});
```

Le operazioni intermedie sono gli elementi costitutivi comuni di un flusso, concatenati dopo l'origine e sono generalmente seguiti da un'operazione terminale che attiva la catena di flusso.

Operazioni terminalistiche

Le operazioni terminal sono ciò che fa scattare il consumo di un flusso. Alcuni dei più comuni sono `Stream.forEach` o `Stream.collect` . Di solito sono posizionati dopo una catena di operazioni intermedie e sono quasi sempre *desiderosi* .

Operazioni senza stato

Apolidia significa che ogni oggetto viene elaborato senza il contesto di altri elementi. Le operazioni senza stato consentono l'elaborazione efficiente dei flussi di memoria. Le operazioni come `Stream.map` e `Stream.filter` che non richiedono informazioni su altri elementi dello stream sono considerate senza stato.

Operazioni stateful

Statefulness significa che l'operazione su ciascun elemento dipende da (alcuni) altri elementi dello stream. Ciò richiede che uno stato sia preservato. Le operazioni di stato possono interrompersi con flussi lunghi o infiniti. Operazioni come `Stream.sorted` richiedono che l'intero stream venga elaborato prima che venga emesso qualsiasi elemento che si romperà in un flusso di elementi abbastanza lungo. Questo può essere dimostrato da un lungo flusso (**eseguito a proprio rischio**):

```
// works - stateless stream
```

```
long BIG_ENOUGH_NUMBER = 999999999;
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).forEach(System.out::println);
```

Ciò causerà una memoria `Stream.sorted` a causa dello stato di `Stream.sorted` :

```
// Out of memory - stateful stream
IntStream.iterate(0, i -> i +
1).limit(BIG_ENOUGH_NUMBER).sorted().forEach(System.out::println);
```

Conversione di un iteratore in un flusso

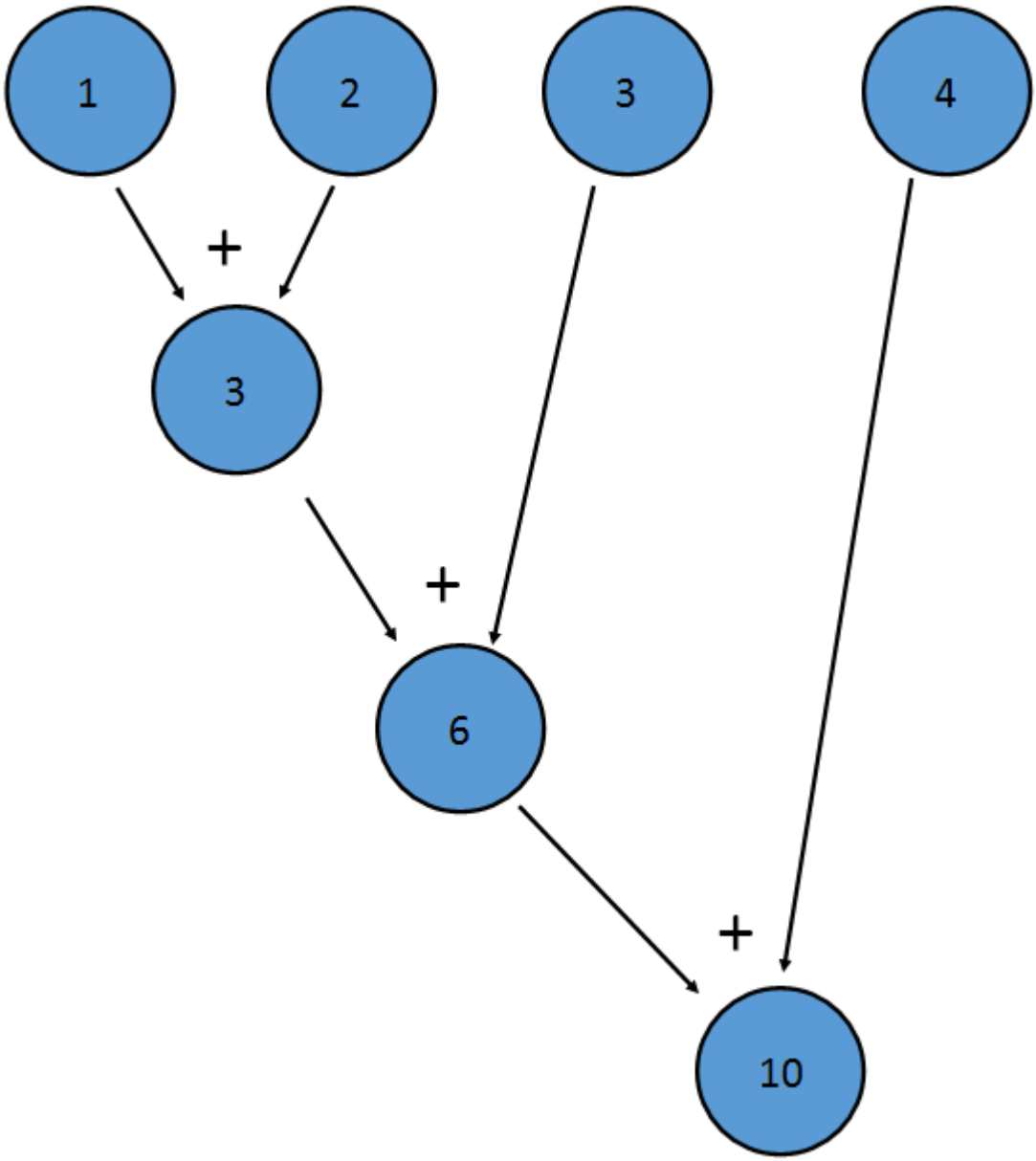
Utilizzare `Spliterators.spliterator()` o `Spliterators.spliteratorUnknownSize()` per convertire un iteratore in uno stream:

```
Iterator<String> iterator = Arrays.asList("A", "B", "C").iterator();
Spliterator<String> spliterator = Spliterators.spliteratorUnknownSize(iterator, 0);
Stream<String> stream = StreamSupport.stream(spliterator, false);
```

Riduzione con stream

La riduzione è il processo di applicazione di un operatore binario a ogni elemento di un flusso per ottenere un valore.

Il metodo `sum()` di un `IntStream` è un esempio di riduzione; applica l'aggiunta a ogni termine del flusso, determinando un valore finale:



Questo è equivalente a `(((1+2)+3)+4)`

Il metodo di `reduce` di un flusso consente di creare una riduzione personalizzata. È possibile utilizzare il metodo `reduce` per implementare il metodo `sum()` :

```
IntStream istr;

//Initialize istr

OptionalInt istr.reduce((a,b)->a+b);
```

La versione `Optional` viene restituita in modo che gli stream vuoti possano essere gestiti in modo appropriato.

Un altro esempio di riduzione è la combinazione di uno `Stream<LinkedList<T>>` in una singola

`LinkedList<T>` :

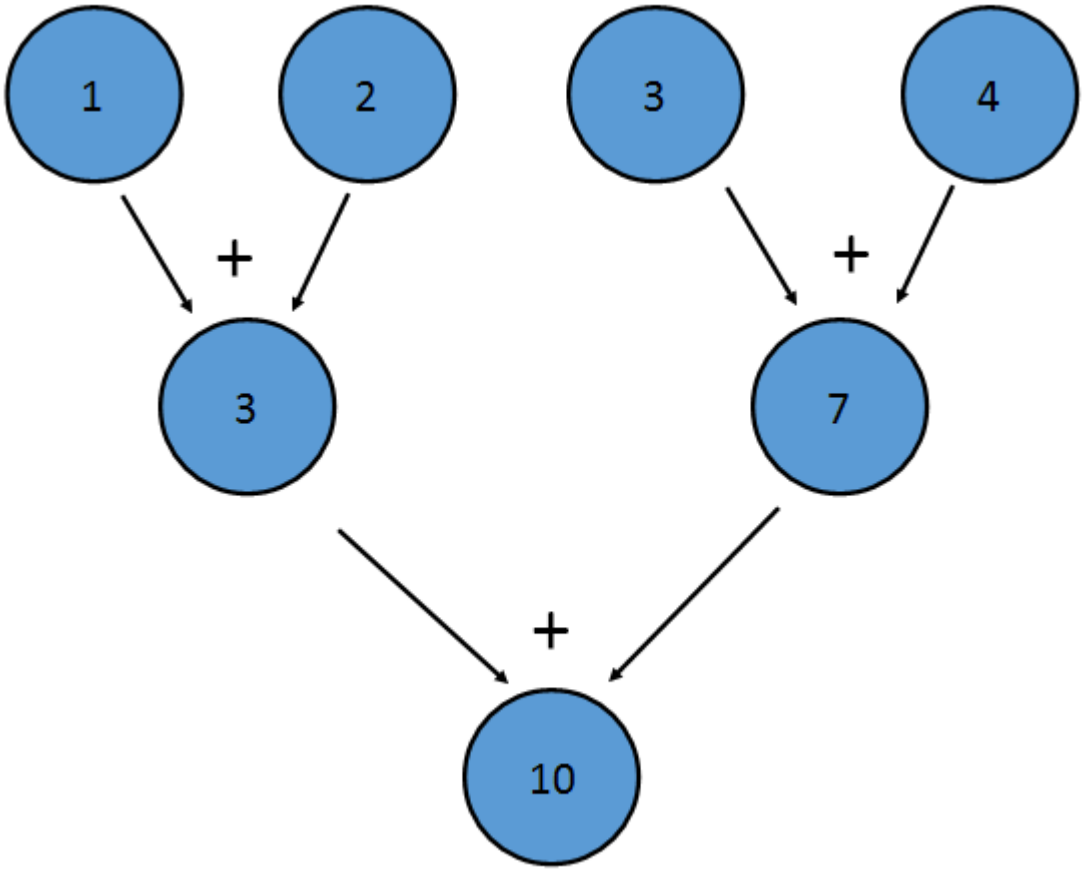
```
Stream<LinkedList<T>> listStream;

//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;
});
```

Puoi anche fornire un *elemento di identità* . Ad esempio, l'elemento identità per addizione è 0, come `x+0==x` . Per la moltiplicazione, l'elemento identità è 1, come `x*1==x` . Nel caso precedente, l'elemento identity è una `LinkedList<T>` vuota `LinkedList<T>` , perché se aggiungi una lista vuota ad un'altra lista, la lista che stai "aggiungendo" non 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;
});
```

Si noti che quando viene fornito un elemento identità, il valore di ritorno non viene incapsulato in un `Optional` -if chiamato su un flusso vuoto, `reduce()` restituirà l'elemento identità.

Anche l'operatore binario deve essere *associativo* , ovvero `(a+b)+c==a+(b+c)` . Questo perché gli elementi possono essere ridotti in qualsiasi ordine. Ad esempio, la riduzione di cui sopra potrebbe essere eseguita in questo modo:



Questa riduzione equivale a scrivere `((1+2)+(3+4))` . La proprietà di associatività consente inoltre a Java di ridurre il `Stream` in parallelo: una porzione del flusso può essere ridotta da ciascun processore, con una riduzione che combina il risultato di ciascun processore alla fine.

Unire uno stream a una singola stringa

Un caso d'uso che si incontra frequentemente, è la creazione di una `String` da uno stream, in cui gli elementi di flusso sono separati da un determinato carattere. Il metodo `Collectors.joining()` può essere utilizzato per questo, come nell'esempio seguente:

```
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);
```

Produzione:

MELA, BANANA, ARANCIA, PERA

Il metodo `Collectors.joining()` può anche gestire pre e postfix:

```
String result = fruitStream.filter(s -> s.contains("e"))
           .map(String::toUpperCase)
           .sorted()
           .collect(Collectors.joining(", ", "Fruits: ", "."));

System.out.println(result);
```

Produzione:

> Frutta: MELA, ARANCIA, PERA.

Vivi su Ideone

Leggi I flussi online: https://riptutorial.com/it/java/topic/88/i-flussi

Capitolo 75: Il comando Java - 'java' e 'javaw'

Sintassi

- `java [ <opt> ... ] <class-name> [ <argument> ... ]`
- `java [ <opt> ... ] -jar <jar-file-pathname> [ <argument> ... ]`

Osservazioni

Il comando `java` viene utilizzato per eseguire un'applicazione Java dalla riga di comando. È disponibile come parte di qualsiasi Java SE JRE o JDK.

Sui sistemi Windows ci sono due varianti del comando `java` :

- La variante `java` avvia l'applicazione in una nuova finestra della console.
- La variante `javaw` avvia l'applicazione senza creare una nuova finestra della console.

Su altri sistemi (ad esempio Linux, Mac OSX, UNIX) viene fornito solo il comando `java` e non viene avviata una nuova finestra della console.

Il simbolo `<opt>` nella sintassi denota un'opzione sulla riga di comando `java` . Gli argomenti "Opzioni Java" e "Opzioni di ridimensionamento heap e stack" coprono le opzioni più comunemente utilizzate. Altri sono trattati nell'argomento Bandi JVM .

Examples

Esecuzione di un file JAR eseguibile

I file JAR eseguibili sono il modo più semplice per assemblare il codice Java in un singolo file che può essere eseguito. * (Nota editoriale: la creazione di file JAR deve essere coperta da un argomento separato). *

Supponendo di avere un file JAR eseguibile con percorso `<jar-path>` , dovresti essere in grado di eseguirlo come segue:

```
java -jar <jar-path>
```

Se il comando richiede argomenti della riga di comando, aggiungili dopo il `<jar-path>` . Per esempio:

```
java -jar <jar-path> arg1 arg2 arg3
```

Se è necessario fornire ulteriori opzioni JVM sulla riga di comando `java` , devono andare *prima* dell'opzione `-jar` . Si noti che l'opzione `-cp` / `-classpath` verrà ignorata se si utilizza `-jar` . Il classpath dell'applicazione è determinato dal manifest del file JAR.

Esecuzione di applicazioni Java tramite una classe "principale"

Quando un'applicazione non è stata pacchettizzata come JAR eseguibile, è necessario fornire il nome di una classe entry-point sulla riga di comando `java` .

Esecuzione della classe HelloWorld

L'esempio "HelloWorld" è descritto in Creazione di un nuovo programma Java . Consiste in una singola classe chiamata `HelloWorld` che soddisfa i requisiti per un punto di ingresso.

Supponendo che il file "HelloWorld.class" (compilato) si trovi nella directory corrente, può essere avviato come segue:

```
java HelloWorld
```

Alcune cose importanti da notare sono:

- Dobbiamo fornire il nome della classe: non il nome del percorso per il file ".class" o il file ".java".
- Se la classe è dichiarata in un pacchetto (come la maggior parte delle classi Java), il nome della classe che forniamo al comando `java` deve essere il nome completo della classe. Ad esempio se `SomeClass` è dichiarato nel pacchetto `com.example` , il nome completo della `com.example.SomeClass` sarà `com.example.SomeClass` .

Specifica di un percorso di classe

A meno che non stiamo usando la sintassi del comando `java -jar` , il comando `java` cerca la classe da caricare cercando nel classpath; vedi The Classpath . Il comando precedente si basa sul percorso di classe predefinito che è (o include) la directory corrente. Possiamo essere più espliciti a riguardo specificando il classpath da usare usando l'opzione `-cp` .

```
java -cp . HelloWorld
```

Questo dice di fare la directory corrente (che è ciò che "." Si riferisce a) l'unica voce sul classpath.

Il `-cp` è un'opzione che viene elaborata dal comando `java` . Tutte le opzioni che sono intese per il comando `java` dovrebbero essere prima del nome della classe. Qualsiasi cosa dopo la classe verrà trattata come un argomento della riga di comando per l'applicazione Java e verrà passata all'applicazione nella `String[]` che viene passata al metodo `main` .

(Se non viene fornita l'opzione `-cp` , `java` utilizzerà il classpath fornito dalla `CLASSPATH` ambiente `CLASSPATH` . Se tale variabile non è impostata o è vuota, `java` utilizza "." Come percorso di classe predefinito.)

Classi di ingresso

Una classe entry-point Java ha un metodo `main` con la seguente firma e modificatori:

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

> Sidenote: a causa di come funzionano gli array, può anche essere `(String args[])`

Quando il comando `java` avvia la macchina virtuale, carica le classi del punto di ingresso specificate e prova a trovare il `main` . Se ha esito positivo, gli argomenti dalla riga di comando vengono convertiti in oggetti `String` Java e assemblati in una matrice. Se `main` viene invocato in questo modo, l'array *non* sarà `null` e non conterrà alcuna voce `null` .

Un metodo di classe punto di ingresso valido deve eseguire quanto segue:

- Essere nominato `main` (sensibile al maiuscolo / minuscolo)
- Sii `public` e `static`
- Avere un tipo di reso `void`
- Avere un singolo argomento con un array `String[]` . L'argomento deve essere presente e non è consentito più di un argomento.
- Sii generico: i parametri di tipo non sono ammessi.
- Avere una classe chiusa non generica, di livello superiore (non nidificata o interna)

È normale dichiarare la classe come `public` ma non strettamente necessaria. Da Java 5 in poi, il tipo di argomento del metodo `main` potrebbe essere un varargs `String` invece di un array di stringhe. `main` può facoltativamente lanciare delle eccezioni, e il suo parametro può essere chiamato qualsiasi cosa, ma convenzionalmente è `args` .

Punti d'ingresso JavaFX

Da Java 8 in poi il comando `java` può anche avviare direttamente un'applicazione JavaFX. JavaFX è documentato nel tag JavaFX , ma un punto di accesso JavaFX deve eseguire quanto segue:

- Estendi `javafx.application.Application`
- Sii `public` e non `abstract`
- Non essere generico o annidato
- Avere un costruttore di no-args esplicito o implicito `public`

Risoluzione dei problemi del comando 'java'

Questo esempio copre errori comuni con l'uso del comando 'java'.

"Comando non trovato"

Se ricevi un messaggio di errore come:

```
java: command not found
```

quando si tenta di eseguire il comando `java` , ciò significa che non vi è alcun comando `java` sul

percorso di ricerca dei comandi della shell. La causa potrebbe essere:

- non hai affatto installato Java JRE o JDK,
- non hai aggiornato la variabile di ambiente `PATH` (correttamente) nel file di inizializzazione della shell, o
- non hai "originato" il file di inizializzazione pertinente nella shell corrente.

Fare riferimento a "Installazione di Java" per i passaggi che è necessario eseguire.

"Impossibile trovare o caricare la classe principale"

Questo messaggio di errore viene emesso dal comando `java` se non è stato possibile trovare / caricare la classe del punto di ingresso che è stata specificata. In termini generali, ci sono tre grandi ragioni per cui questo può accadere:

- Hai specificato una classe del punto di ingresso che non esiste.
- La classe esiste, ma l'hai specificata in modo errato.
- La classe esiste e l'hai specificata correttamente, ma Java non riesce a trovarla perché il classpath non è corretto.

Ecco una procedura per diagnosticare e risolvere il problema:

1. Scopri il nome completo della classe del punto di ingresso.

    - Se si dispone di codice sorgente per una classe, il nome completo è costituito dal nome del pacchetto e dal nome della classe semplice. L'istanza che la classe "Main" è dichiarata nel pacchetto "com.example.myapp" quindi il suo nome completo è "com.example.myapp.Main".
    - Se si dispone di un file di classe compilato, è possibile trovare il nome della classe eseguendo `javap` su di esso.
    - Se il file di classe si trova in una directory, è possibile dedurre il nome completo della classe dai nomi delle directory.
    - Se il file di classe si trova in un file JAR o ZIP, è possibile dedurre il nome completo della classe dal percorso del file nel file JAR o ZIP.

2. Guarda il messaggio di errore dal comando `java` . Il messaggio dovrebbe terminare con il nome completo della classe che `java` sta tentando di utilizzare.

    - Verifica che corrisponda esattamente al nome classe completo per la classe del punto di ingresso.
    - Non dovrebbe terminare con ".java" o ".class".
    - Non dovrebbe contenere barre o altri caratteri non legali in un identificativo Java [1] .
    - L'intelaiatura del nome dovrebbe corrispondere esattamente al nome completo della classe.

3. Se stai usando il nome di classe corretto, assicurati che la classe sia effettivamente sul classpath:

- Calcolare il nome del percorso a cui il nome della classe corrisponde; consulta Mapping dei nomi di classe ai nomi di percorso
- Calcola qual è il percorso di classe; vedere questo esempio: diversi modi per specificare il classpath
- Osservare ciascuno dei file JAR e ZIP sul classpath per vedere se contengono una classe con il nome di percorso richiesto.
- Guarda ogni directory per vedere se il percorso si risolve in un file all'interno della directory.

Se il controllo del classpath a mano non ha riscontrato il problema, è possibile aggiungere le opzioni `-Xdiag` e `-XshowSettings` . Il primo elenca tutte le classi caricate e quest'ultima stampa le impostazioni che includono il classpath effettivo per JVM.

Infine, ci sono alcune cause *oscure* per questo problema:

- Un file JAR eseguibile con un attributo `Main-Class` che specifica una classe che non esiste.
- Un file JAR eseguibile con un attributo `Class-Path` errato.
- Se si incasinano [2] opzioni prima del nome della classe, il comando `java` può tentare di interpretarne uno come nomeclassifica.
- Se qualcuno ha ignorato le regole di stile Java e ha utilizzato identificativi di pacchetto o classe che differiscono solo nel caso di lettere, e si sta eseguendo su una piattaforma che considera il caso di lettere nei nomi di file come non significativo.
- Problemi con gli omogei nei nomi delle classi nel codice o sulla riga di comando.

"Metodo principale non trovato nella classe <nome>"

Questo problema si verifica quando il comando `java` è in grado di trovare e caricare la classe che è stata nominata, ma non è in grado di trovare un metodo entry-point.

Ci sono tre possibili spiegazioni:

- Se si sta tentando di eseguire un file JAR eseguibile, il manifest di JAR presenta un attributo "Main-Class" non corretto che specifica una classe che non è una classe del punto di ingresso valida.
- Hai detto al comando `java` una classe che non è una classe entry point.
- La classe del punto di ingresso non è corretta; vedere le classi dei punti di ingresso per ulteriori informazioni.

Altre risorse

- Cosa significa "Impossibile trovare o caricare la classe principale"?
- http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html

---

1 - Da Java 8 e `java` successive, il comando `java` associa in modo utile un separatore di file ("/" o "") a un punto (".") . Tuttavia, questo comportamento non è documentato nelle pagine di manuale.

2 - Un caso davvero oscuro è se copi e incolli un comando da un documento formattato in cui l'editor di testo ha usato

un "trattino lungo" invece di un trattino normale.

Esecuzione di un'applicazione Java con dipendenze di libreria

Questa è una continuazione degli esempi "main class" e "eseguibili JAR" .

Le tipiche applicazioni Java sono costituite da un codice specifico dell'applicazione e da vari codici libreria riutilizzabili implementati o implementati da terze parti. Questi ultimi sono comunemente definiti come dipendenze delle librerie e sono generalmente pacchettizzati come file JAR.

Java è un linguaggio dinamicamente vincolato. Quando si esegue un'applicazione Java con dipendenze della libreria, la JVM deve sapere dove si trovano le dipendenze in modo che possa caricare le classi come richiesto. In generale, ci sono due modi per affrontare questo:

- L'applicazione e le sue dipendenze possono essere riconfezionate in un singolo file JAR che contiene tutte le classi e le risorse richieste.
- La JVM può sapere dove trovare i file JAR dipendenti tramite il percorso di classe di runtime.

Per un file JAR eseguibile, il percorso di classe runtime è specificato dall'attributo manifest "Class-Path". *(Nota editoriale: questo dovrebbe essere descritto in un argomento separato sul comando `jar` .)* In caso contrario, il percorso di classe runtime deve essere fornito utilizzando l'opzione `-cp` o utilizzando la `CLASSPATH` ambiente `CLASSPATH` .

Ad esempio, supponiamo di avere un'applicazione Java nel file "myApp.jar" la cui classe del punto di ingresso è `com.example.MyApp` . Supponiamo inoltre che l'applicazione dipenda dai file JAR della libreria "lib / library1.jar" e "lib / library2.jar". Potremmo avviare l'applicazione usando il comando `java` come segue in una riga di 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
```

(Su Windows, dovresti usare `;` invece di `:` come separatore del classpath, e dovresti impostare la variabile (locale) `CLASSPATH` usando `set` piuttosto che `export` .)

Mentre uno sviluppatore Java sarebbe a proprio agio con questo, non è "user friendly". Quindi è pratica comune scrivere un semplice script di shell (o un file batch di Windows) per nascondere i dettagli di cui l'utente non ha bisogno di essere informato. Ad esempio, se metti il seguente script di shell in un file chiamato "myApp", lo rendi eseguibile e lo metti in una directory sul percorso di ricerca del comando:

```
#!/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
```

allora potresti eseguirlo come segue:

```
$ myApp arg1 arg2 ...
```

Qualsiasi argomento sulla riga di comando verrà passato all'applicazione Java tramite l'espansione `"$@"` . (Puoi fare qualcosa di simile con un file batch di Windows, anche se la sintassi è diversa.)

Spazi e altri caratteri speciali negli argomenti

Prima di tutto, il problema di gestire gli spazi negli argomenti NON è in realtà un problema Java. Piuttosto, è un problema che deve essere gestito dalla shell dei comandi che si sta utilizzando quando si esegue un programma Java.

Ad esempio, supponiamo di avere il seguente semplice programma che stampa la dimensione di un file:

```
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());
        }
    }
}
```

Supponiamo ora di voler stampare la dimensione di un file il cui percorso ha spazi in esso; ad esempio `/home/steve/Test File.txt` . Se eseguiamo il comando in questo modo:

```
$ java PrintFileSizes /home/steve/Test File.txt
```

la shell non saprà che `/home/steve/Test File.txt` è in realtà un percorso. Invece, passerà 2 argomenti distinti all'applicazione Java, che tenterà di trovare le rispettive dimensioni dei file, e fallirà perché i file con quei percorsi (probabilmente) non esistono.

Soluzioni che utilizzano una shell POSIX

Shell POSIX includono `sh` come derivati e quali `bash` e `ksh` . Se stai usando una di queste shell, allora puoi risolvere il problema *citando* l'argomento.

```
$ java PrintFileSizes "/home/steve/Test File.txt"
```

Le doppie virgolette attorno al nome del percorso dicono alla shell che dovrebbe essere passata come un singolo argomento. Le virgolette saranno rimosse quando questo accadrà. Ci sono un paio di altri modi per farlo:

```
$ java PrintFileSizes '/home/steve/Test File.txt'
```

Le virgolette singole (diritte) sono trattate come virgolette, tranne che sopprimono anche varie espansioni all'interno dell'argomento.

```
$ java PrintFileSizes /home/steve/Test\ File.txt
```

Un backslash sfugge allo spazio seguente e fa sì che non venga interpretato come un separatore di argomenti.

Per una documentazione più completa, comprese le descrizioni di come trattare altri caratteri speciali negli argomenti, fare riferimento all'argomento di citazione nella documentazione di Bash .

Soluzione per Windows

Il problema fondamentale per Windows è che a livello di sistema operativo, gli argomenti vengono passati a un processo figlio come una singola stringa (origine). Ciò significa che la responsabilità finale di parsing (o ri-analisi) della riga di comando ricade sul programma o sulle sue librerie di runtime. C'è molta incongruenza.

Nel caso Java, per farla breve:

- Puoi mettere virgolette su un argomento in un comando `java` , e questo ti permetterà di passare argomenti con spazi al loro interno.
- Apparentemente, il comando `java` stesso sta analizzando la stringa di comando, e lo ottiene più o meno giusto
- Tuttavia, quando si tenta di combinare questo con l'uso di `SET` e la sostituzione delle variabili in un file batch, diventa davvero complicato se vengono rimosse le virgolette.
- La shell `cmd.exe` apparentemente ha altri meccanismi di escape; es. raddoppiando le virgolette e usando `^` escapes.

Per maggiori dettagli, fare riferimento alla documentazione del file batch .

Opzioni Java

Il comando `java` supporta una vasta gamma di opzioni:

- Tutte le opzioni iniziano con un trattino singolo o segno meno (`-`): la convenzione GNU / Linux di utilizzo `--` per le opzioni "lunghe" non è supportata.
- Le opzioni devono apparire prima dell'argomento `<classname>` o `-jar <jarfile>` per essere riconosciuti. Qualsiasi argomento dopo di essi verrà trattato come argomento da passare all'app Java in esecuzione.
- Le opzioni che non iniziano con `-X` o `-XX` sono opzioni standard. Puoi fare affidamento su

tutte le implementazioni Java [1] per supportare qualsiasi opzione standard.

- Le opzioni che iniziano con `-X` sono opzioni non standard e possono essere ritirate da una versione Java alla successiva.
- Le opzioni che iniziano con `-XX` sono opzioni avanzate e possono anche essere ritirate.

Impostazione delle proprietà del sistema con `-D`

L'opzione `-D<property>=<value>` viene utilizzata per impostare una proprietà nell'oggetto `Properties` del sistema. Questo parametro può essere ripetuto per impostare proprietà differenti.

Opzioni di memoria, Stack e Garbage Collector

Le opzioni principali per il controllo delle dimensioni di heap e stack sono documentate in Impostazione delle dimensioni di heap, PermGen e stack . *(Nota editoriale: le opzioni di Garbage Collector devono essere descritte nello stesso argomento).*

Abilitazione e disabilitazione delle asserzioni

Le opzioni `-ea` e `-da` rispettivamente abilitano e disabilitano il controllo delle `assert` Java:

- Il controllo di tutte le asserzioni è disabilitato per impostazione predefinita.
- L'opzione `-ea` consente di verificare tutte le asserzioni
- Il `-ea:<packagename>...` consente di verificare le asserzioni in un pacchetto *e tutti i sotto-*pacchetti.
- Il `-ea:<classname>...` abilita il controllo delle asserzioni in una classe.
- L'opzione `-da` disabilita il controllo di tutte le asserzioni
- Il `-da:<packagename>...` disabilita il controllo delle asserzioni in un pacchetto *e tutti i* pacchetti *secondari* .
- Il `-da:<classname>...` disabilita il controllo delle asserzioni in una classe.
- L'opzione `-esa` consente di verificare tutte le classi di sistema.
- L'opzione `-dsa` disabilita il controllo di tutte le classi di sistema.

Le opzioni possono essere combinate. Per esempio.

```
$ # 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
```

Si noti che l'abilitazione al controllo delle asserzioni è suscettibile di alterare il comportamento di una programmazione Java.

- È responsabile rendere l'applicazione più lenta in generale.
- Può far sì che i metodi specifici impieghino più tempo per essere eseguiti, il che potrebbe

cambiare il tempo dei thread in un'applicazione multi-thread.
- Può introdurre relazioni fortuite *prima di quelle* che possono causare la scomparsa delle anomalie della memoria.
- Una dichiarazione `assert` erroneamente implementata potrebbe avere effetti collaterali indesiderati.

Selezione del tipo di VM

I `-client` e `-server` opzioni consentono di scegliere tra due diverse forme di HotSpot VM:

- Il modulo "client" è ottimizzato per le applicazioni utente e offre un avvio più veloce.
- Il modulo "server" è ottimizzato per le applicazioni di lunga durata. Ci vuole più tempo per acquisire statistiche durante il "warm up" JVM, che consente al compilatore JIT di ottimizzare il codice nativo.

Per impostazione predefinita, la JVM verrà eseguita in modalità 64 bit, se possibile, in base alle funzionalità della piattaforma. Le opzioni `-d32` e `-d64` consentono di selezionare la modalità in modo esplicito.

---

1 - Controlla il manuale ufficiale per il comando `java` . A volte un'opzione *standard* è descritta come "soggetto a modifiche".

Leggi Il comando Java - 'java' e 'javaw' online: https://riptutorial.com/it/java/topic/5791/il-comando-java----java--e--javaw-

Capitolo 76: Il percorso di classe

introduzione

Il classpath elenca i luoghi in cui il runtime Java deve cercare classi e risorse. Il classpath viene anche utilizzato dal compilatore Java per trovare le dipendenze precedentemente compilate ed esterne.

Osservazioni

Caricamento della classe Java

La JVM (Java Virtual Machine) caricherà le classi come e quando sono richieste le classi (questo è chiamato lazy-loading). Le posizioni delle classi da utilizzare sono specificate in tre punti: -

1. Quelli richiesti dalla piattaforma Java vengono caricati per primi, come quelli nella libreria di classi Java e le sue dipendenze.
2. Le classi di estensione vengono caricate successivamente (cioè quelle in `jre/lib/ext/`)
3. Vengono quindi caricate le classi definite dall'utente tramite il classpath

Le classi vengono caricate usando classi che sono sottotipi di `java.lang.ClassLoader` . Questo è descritto più in dettaglio in questo argomento: Classloader .

classpath

Il classpath è un parametro utilizzato dalla JVM o dal compilatore che specifica le posizioni delle classi e dei pacchetti definiti dall'utente. Questo può essere impostato nella riga di comando come con la maggior parte di questi esempi o attraverso una variabile ambientale (`CLASSPATH`)

Examples

Diversi modi per specificare il classpath

Esistono tre modi per impostare il classpath.

1. Può essere impostato utilizzando la `CLASSPATH` ambiente `CLASSPATH` :

```
set CLASSPATH=...          # Windows and csh
export CLASSPATH=...       # Unix ksh/bash
```

2. Può essere impostato sulla riga di comando come segue

```
java -classpath ...
javac -classpath ...
```

   Si noti che l' `-classpath` (o `-cp`) ha la precedenza sulla `CLASSPATH` ambiente `CLASSPATH` .

3. Il percorso di classe per un file JAR eseguibile viene specificato utilizzando l'elemento `Class-Path` in `MANIFEST.MF` :

   ```
   Class-Path: jar1-name jar2-name directory-name/jar3-name
   ```

   Si noti che questo si applica solo quando il file JAR viene eseguito in questo modo:

   ```
   java -jar some.jar ...
   ```

   In questa modalità di esecuzione, l'opzione `-classpath` e la variabile di ambiente CLASSPATH verranno ignorate, anche se il file JAR non ha un elemento `Class-Path` .

Se non viene specificato alcun percorso di classe, il percorso di classe predefinito è il file JAR selezionato quando si utilizza `java -jar` o altrimenti la directory corrente.

Relazionato:

- https://docs.oracle.com/javase/tutorial/deployment/jar/downman.html
- http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html

Aggiunta di tutti i JAR in una directory al classpath

Se si desidera aggiungere tutti i JAR nella directory al classpath, è possibile farlo concisamente utilizzando la sintassi dei caratteri jolly di classpath; per esempio:

```
someFolder/*
```

Questo dice alla JVM di aggiungere tutti i file JAR e ZIP nella directory `someFolder` al classpath. Questa sintassi può essere utilizzata in un argomento `-cp` , in una `CLASSPATH` ambiente `CLASSPATH` o in un attributo `Class-Path` in un file manifest di un file JAR eseguibile. Vedere *Impostazione del percorso di classe:* Wildcard *del percorso di classe* per esempi e avvertimenti.

Gli appunti:

1. I caratteri jolly di Classpath sono stati introdotti per la prima volta in Java 6. Le versioni precedenti di Java non trattano "*" come carattere jolly.
2. Non puoi mettere altri personaggi prima o dopo " *"; es. "someFolder* / .jar" non è un jolly.
3. Un carattere jolly corrisponde solo ai file con il suffisso ".jar" o ".JAR". I file ZIP vengono ignorati, così come i file JAR con suffissi diversi.
4. Un carattere jolly corrisponde solo ai file JAR nella directory stessa, non nelle sue sottodirectory.
5. Quando un gruppo di file JAR è abbinato a una voce jolly, il loro ordine relativo sul classpath non viene specificato.

Sintassi del percorso del percorso di classe

Il classpath è una sequenza di voci che sono nomi di percorsi di directory, nomi di percorsi file JAR o ZIP o specifiche jolly JAR / ZIP.

- Per un classpath specificato sulla riga di comando (ad esempio `-classpath`) o come variabile di ambiente, le voci devono essere separate con `;` (punto e virgola) caratteri su Windows, o `:` (due punti) caratteri su altre piattaforme (Linux, UNIX, MacOSX e così via).

- Per l'elemento `Class-Path` in `MANIFEST.MF` di un file JAR, utilizzare un singolo spazio per separare le voci.

A volte è necessario incorporare uno spazio in una voce classpath

- Quando il classpath è specificato sulla riga di comando, si tratta semplicemente di utilizzare il quoting della shell appropriato. Per esempio:

  ```
  export CLASSPATH="/home/user/My JAR Files/foo.jar:second.jar"
  ```

  (I dettagli possono dipendere dalla shell dei comandi che usi).

- Quando il classpath viene specificato in un file JAR come file "MANIFEST.MF", è necessario utilizzare la codifica URL.

  ```
  Class-Path: /home/user/My%20JAR%20Files/foo.jar second.jar
  ```

Percorso di classe dinamico

A volte, l'aggiunta di tutti i JAR da una cartella non è sufficiente, ad esempio quando si dispone di codice nativo e occorre selezionare un sottoinsieme di JAR. In questo caso, sono necessari due metodi `main()` . Il primo costruisce un classloader e quindi usa questo classloader per chiamare il secondo `main()` .

Ecco un esempio che seleziona il JAR nativo SWT corretto per la piattaforma, aggiunge tutti i JAR dell'applicazione e quindi richiama il metodo `main()` reale: Creazione di un'applicazione SWT Java multipiattaforma

Carica una risorsa dal classpath

Può essere utile caricare una risorsa (immagine, file di testo, proprietà, KeyStore, ...) che è impacchettata all'interno di un JAR. A tale scopo, possiamo usare `Class` e `ClassLoader` .

Supponiamo di avere la seguente struttura di progetto:

```
program.jar
|
\-com
  \-project
    |
    |-file.txt
    \-Test.class
```

E vogliamo accedere ai contenuti di `file.txt` dalla classe `Test` . Possiamo farlo chiedendo al classloader:

```
InputStream is = Test.class.getClassLoader().getResourceAsStream("com/project/file.txt");
```

Usando il classloader, dobbiamo specificare il percorso completo della nostra risorsa (ciascun pacchetto).

In alternativa, possiamo chiedere direttamente all'oggetto della classe Test

```
InputStream is = Test.class.getResourceAsStream("file.txt");
```

Usando l'oggetto classe, il percorso è relativo alla classe stessa. Il nostro `Test.class` è nel pacchetto `com.project` , lo stesso di `file.txt` , non abbiamo bisogno di specificare alcun percorso.

Possiamo, tuttavia, utilizzare percorsi assoluti dall'oggetto classe, in questo modo:

```
 is = Test.class.getResourceAsStream("/com/project/file.txt");
```

Mappatura dei nomi di classe ai nomi di percorso

La toolchain Java standard (e strumenti di terze parti progettati per interagire con essi) hanno regole specifiche per mappare i nomi delle classi ai nomi dei percorsi dei file e altre risorse che li rappresentano.

Le mappature sono le seguenti

- Per le classi nel pacchetto predefinito, i nomi di percorso sono nomi di file semplici.
- Per le classi in un pacchetto denominato, i componenti del nome del pacchetto vengono mappati nelle directory.
- Per le classi nidificate e interne nominate, il componente nomefile viene formato unendo i nomi di classe con un carattere `$` .
- Per le classi interne anonime, i numeri vengono utilizzati al posto dei nomi.

Questo è illustrato nella seguente tabella:

Nome della classe	Percorso di origine	Classfile pathname
`SomeClass`	`SomeClass.java`	`SomeClass.class`
`com.example.SomeClass`	`com/example/SomeClass.java`	`com/example/SomeClass.class`
`SomeClass.Inner`	(in `SomeClass.java`)	`SomeClass$Inner.class`
`SomeClass` interni anon di `SomeClass`	(in `SomeClass.java`)	`SomeClass$1.class` , `SomeClass$2.class` , ecc

Cosa significa classpath: come funzionano le ricerche

Lo scopo del classpath è di dire a una JVM dove trovare classi e altre risorse. Il significato del classpath e del processo di ricerca sono intrecciati.

Il classpath è una forma di percorso di ricerca che specifica una sequenza di *posizioni* in cui cercare le risorse. In un classpath standard, queste posizioni sono o una directory nel file system host, un file JAR o un file ZIP. In ogni caso, la posizione è la radice di uno *spazio* dei *nomi* che verrà cercato.

La procedura standard per la ricerca di una classe sul classpath è la seguente:

1. Mappare il nome della classe in un percorso di file relativo relativo `RP` . Il mapping dei nomi di classi ai nomi di classe è descritto altrove.

2. Per ogni voce `E` nel classpath:

    - Se la voce è una directory del filesystem:
        - Risolvi `RP` rispetto a `E` per dare un `AP` assoluto.
        - Verifica se `AP` è un percorso per un file esistente.
        - Se sì, carica la classe da quel file
    - Se la voce è un file JAR o ZIP:
        - Cerca `RP` nell'indice del file JAR / ZIP.
        - Se la voce del file JAR / ZIP corrispondente esiste, caricare la classe da quella voce.

La procedura per cercare una risorsa sul classpath dipende dal fatto che il percorso della risorsa sia assoluto o relativo. Per un percorso di risorse assoluto, la procedura è come sopra. Per un percorso di risorse relativo risolto utilizzando `Class.getResource` o `Class.getResourceAsStream` , il percorso per il pacchetto di classi viene anteposto prima della ricerca.

(Nota: queste sono le procedure implementate dai classloader Java standard. Un classloader personalizzato potrebbe eseguire la ricerca in modo diverso).

Il percorso di classe di bootstrap

I normali classloader Java cercano prima le classi nel percorso di classe bootstrap, prima di cercare le estensioni e il classpath dell'applicazione. Per impostazione predefinita, il percorso di classe bootstrap è costituito dal file "rt.jar" e da altri file JAR importanti forniti dall'installazione di JRE. Questi forniscono tutte le classi nella libreria di classi Java SE standard, insieme a varie classi di implementazione "interne".

In circostanze normali, non è necessario preoccuparsi di questo. Per impostazione predefinita, comandi come `java` , `javac` e così via useranno le versioni appropriate delle librerie di runtime.

Molto raramente, è necessario sovrascrivere il normale comportamento del runtime Java utilizzando una versione alternativa di una classe nelle librerie standard. Ad esempio, nelle librerie di runtime potresti incontrare un bug "show stopper" che non è possibile aggirare in modo normale. In tale situazione, è possibile creare un file JAR contenente la classe modificata e quindi aggiungerlo al percorso di classe bootstrap che avvia la JVM.

Il comando `java` fornisce le seguenti opzioni `-X` per la modifica del percorso di classe bootstrap:

- `-Xbootclasspath:<path>` sostituisce il percorso di classe di avvio corrente con il percorso

fornito.

- `-Xbootclasspath/a:<path>` aggiunge il percorso fornito al percorso di classe di avvio corrente.
- `-Xbootclasspath/p:<path>` antepone il percorso fornito al percorso di classe di avvio corrente.

Si noti che quando si usano le opzioni bootclasspath per sostituire o sovrascrivere una classe Java (eccetera), si sta tecnicamente modificando Java. *Potrebbero esserci* implicazioni di licenza se poi distribuisci il tuo codice. (Fare riferimento ai termini e alle condizioni della licenza binaria Java ... e consultare un avvocato.)

Leggi Il percorso di classe online: https://riptutorial.com/it/java/topic/3720/il-percorso-di-classe

Capitolo 77: Implementazione Java

introduzione

Ci sono una varietà di tecnologie per "confezionare" applicazioni Java, webapp e così via, per l'implementazione sulla piattaforma su cui verranno eseguite. Si va dalla semplice libreria o file `JAR` eseguibili, ai file `WAR` e `EAR` , fino agli installer e agli eseguibili autonomi.

Osservazioni

Al livello più fondamentale, un programma Java può essere distribuito copiando una classe compilata (ad esempio un file ".class") o una struttura di directory contenente classi compilate. Tuttavia, Java viene normalmente distribuito in uno dei seguenti modi:

- Copiando un file JAR o una raccolta di file JAR nel sistema in cui verranno eseguiti; ad esempio utilizzando `javac` .
- Copiando o caricando un WAR, EAR o un file simile in un "servlet container" o "application server".
- Eseguendo un qualche tipo di programma di installazione dell'applicazione che automatizza quanto sopra. Il programma di installazione potrebbe anche installare un JRE incorporato.
- Inserendo i file JAR dell'applicazione su un server Web per consentire il loro avvio utilizzando Java WebStart.

L'esempio Creazione di file JAR, WAR ed EAR riepiloga i diversi modi per creare questi file.

Esistono numerosi strumenti di "generatore di installazione" open source e "commerciale" e "generatore EXE" per Java. Allo stesso modo, ci sono strumenti per offuscare i file di classe Java (per rendere più difficile la retroingegnerizzazione) e per aggiungere il controllo delle licenze di runtime. Questi sono tutti fuori portata per la documentazione "Linguaggio di programmazione Java".

Examples

Creare un JAR eseguibile dalla riga di comando

Per creare un jar, hai bisogno di uno o più file di classe. Questo dovrebbe avere un metodo principale se deve essere eseguito con un doppio clic.

Per questo esempio, useremo:

```
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);
    }

}
```

È stato nominato HelloWorld.java

Successivamente, vogliamo compilare questo programma.

Puoi usare qualsiasi programma che vuoi fare. Per eseguire dalla riga di comando, consultare la documentazione sulla compilazione e l'esecuzione del primo programma java.

Una volta che hai HelloWorld.class, crea una nuova cartella e chiamala come vuoi.

Crea un altro file chiamato manifest.txt e incollalo

```
Main-Class: HelloWorld
Class-Path: HelloWorld.jar
```

Inseriscilo nella stessa cartella con HelloWorld.class
Usa la riga di comando per creare la tua directory corrente (`cd C:\Your\Folder\Path\Here` on windows) la tua cartella.

Usa Terminale e cambia la directory nella directory (`cd /Users/user/Documents/Java/jarfolder` su Mac) della tua cartella

Al termine, digita `jar -cvfm HelloWorld.jar manifest.txt HelloWorld.class` e premi `jar -cvfm HelloWorld.jar manifest.txt HelloWorld.class` . Questo crea un file jar (nella cartella con manifest e HelloWorld.class) utilizzando i file .class specificati e denominati HelloWorld.jar. Vedi la sezione Sintassi per informazioni sulle opzioni (come -m e -v).
Dopo questi passaggi, vai nella tua directory con il file manifest e dovresti trovare HelloWorld.jar Cliccando su di esso dovrebbe visualizzare `Hello, World` in una casella di testo.

Creazione di file JAR, WAR ed EAR

I tipi di file JAR, WAR ed EAR sono fondamentalmente file ZIP con un file "manifest" e (per i file WAR e EAR) una particolare directory interna / struttura di file.

Il modo consigliato per creare questi file è utilizzare uno strumento di compilazione specifico di Java che "capisca" i requisiti per i rispettivi tipi di file. Se non si utilizza uno strumento di compilazione, IDE "export" è la prossima opzione da provare.

(*Nota editoriale: le descrizioni di come creare questi file sono nella migliore posizione nella documentazione per i rispettivi strumenti. Inseritele lì. Si prega di mostrare un po 'di autocontrollo e NON CUCCHIA le corno in questo esempio!*)

Creazione di file JAR e WAR usando Maven

Creare un JAR o WAR usando Maven è semplicemente una questione di mettere l'elemento `<packaging>` corretto nel file POM; per esempio,

```
<packaging>jar</packaging>
```

o

```
<packaging>war</packaging>
```

Per ulteriori dettagli. Maven può essere configurato per creare file JAR "eseguibili" aggiungendo le informazioni necessarie sulla classe del punto di ingresso e le dipendenze esterne come proprietà del plugin per il plugin maven jar. Esiste anche un plugin per creare file "uberJAR" che combinano un'applicazione e le sue dipendenze in un singolo file JAR.

Per ulteriori informazioni, consultare la documentazione di Maven (http://www.Scriptutorial.com/topic/898).

Creare file JAR, WAR e EAR usando Ant

Lo strumento di configurazione Ant ha "compiti" separati per la costruzione di JAR, WAR ed EAR. Per ulteriori informazioni, consultare la documentazione di Ant (http://www.Scriptutorial.com/topic/4223).

Creazione di file JAR, WAR ed EAR utilizzando un IDE

I tre IDE Java più popolari hanno tutti il supporto integrato per la creazione di file di distribuzione. La funzionalità è spesso descritta come "esportazione".

- Eclipse - http://www.Scriptutorial.com/topic/1143
- NetBeans - http://www.Scriptutorial.com/topic/5438
- Intellij-IDEA - Esportazione

Creazione di file JAR, WAR ed EAR utilizzando il comando `jar` .

È anche possibile creare questi file "a mano" usando il comando `jar` . Si tratta semplicemente di assemblare un albero di file con i file dei componenti corretti nella posizione corretta, creare un file manifest e avviare `jar` per creare il file JAR.

Fare riferimento al comando `jar` Argomento (Creazione e modifica di file JAR) per ulteriori informazioni

Introduzione a Java Web Start

Le Oracle Java Tutorials riassumono Web Start come segue:

> Il software Java Web Start offre la possibilità di avviare applicazioni complete con un solo clic. Gli utenti possono scaricare e avviare applicazioni, come un programma completo per fogli di calcolo o un client di chat via Internet, senza passare attraverso lunghe procedure di installazione.

Altri vantaggi di Java Web Start sono il supporto per il codice firmato e la dichiarazione esplicita delle dipendenze della piattaforma e il supporto per la memorizzazione nella cache del codice e la distribuzione degli aggiornamenti dell'applicazione.

Java Web Start è indicato anche come JavaWS e JAWS. Le principali fonti di informazione sono:

- The Java Tutorials - Lezione: Java Web Start
- Guida di avvio Web Java
- Domande frequenti su Java Web Start
- Specifica JNLP
- `javax.jnlp` Documentazione API
- Sito per sviluppatori Java Web Start

Prerequisiti

Ad un livello elevato, Web Start funziona distribuendo applicazioni Java compresse come file JAR da un server web remoto. I prerequisiti sono:

- Un'installazione Java preesistente (JRE o JDK) sul computer di destinazione in cui deve essere eseguita l'applicazione. È richiesto Java 1.2.2 o versione successiva:
    - Da Java 5.0 in poi, il supporto per Web Start è incluso in JRE / JDK.
    - Per le versioni precedenti, il supporto di Web Start viene installato separatamente.
    - L'infrastruttura Web Start include alcuni Javascript che possono essere inclusi in una pagina Web per aiutare l'utente a installare il software necessario.
- Il server Web che ospita il software deve essere accessibile al computer di destinazione.
- Se l'utente sta per avviare un'applicazione Web Start utilizzando un collegamento in una pagina Web, quindi:
    - hanno bisogno di un browser web compatibile e
    - per i browser moderni (sicuri), è necessario dir loro come dire al browser di consentire l'esecuzione di Java ... senza compromettere la sicurezza del browser.

Un esempio di file JNLP

L'esempio seguente ha lo scopo di illustrare le funzionalità di base di 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>
```

Come puoi vedere, un file JNLP basato su XML e le informazioni sono tutte contenute nell'elemento `<jnlp>` .

- L'attributo `spec` fornisce la versione della specifica JNPL a cui questo file è conforme.
- L'attributo `codebase` fornisce l'URL di base per la risoluzione degli URL relativi `href` nel resto del file.
- L'attributo `href` fornisce l'URL definitivo per questo file JNLP.
- L'elemento `<information>` contiene i metadati dell'applicazione, inclusi titolo, autori, descrizione e sito Web di assistenza.
- L'elemento `<resources>` descrive le dipendenze per l'applicazione, tra cui la versione Java, la piattaforma OS ei file JAR richiesti.
- L'elemento `<application-desc>` (o `<applet-desc>`) fornisce le informazioni necessarie per avviare l'applicazione.

Configurazione del server web

Il server web deve essere configurato per utilizzare `application/x-java-jnlp-file` come MIMEtype per i file `.jnlp` .

Il file JNLP ei file JAR dell'applicazione devono essere installati sul server Web in modo che siano disponibili utilizzando gli URL indicati dal file JNLP.

Abilitazione dell'avvio tramite una pagina Web

Se l'applicazione deve essere avviata tramite un collegamento Web, la pagina che contiene il collegamento deve essere creata sul server web.

- Se si può presumere che Java Web Start sia già installato sul computer dell'utente, la pagina

Web deve semplicemente contenere un collegamento per avviare l'applicazione. Per esempio.

```
<a href="https://www.example.com/demo_webstart.jnlp">Launch the application</a>
```

- In caso contrario, la pagina dovrebbe includere anche alcuni script per rilevare il tipo di browser che l'utente sta utilizzando e richiedere di scaricare e installare la versione richiesta di Java.

NOTA: è una cattiva idea incoraggiare gli utenti a incoraggiare l'installazione di Java in questo modo, o persino a abilitare Java nei loro browser Web in modo che l'avvio della pagina Web JNLP funzioni.

Avvio di applicazioni Web Start dalla riga di comando

Le istruzioni per avviare un'applicazione Web Start dalla riga di comando sono semplici. Supponendo che l'utente abbia installato Java 5.0 JRE o JDK, è sufficiente eseguire questo:

```
$ javaws <url>
```

dove `<url>` è l'URL per il file JNLP sul server remoto.

Creazione di un UberJAR per un'applicazione e le sue dipendenze

Un requisito comune per un'applicazione Java è che può essere distribuito copiando un singolo file. Per applicazioni semplici che dipendono solo dalle librerie di classi Java SE standard, questo requisito viene soddisfatto creando un file JAR contenente tutte le classi di applicazioni (compilate).

Le cose non sono così semplici se l'applicazione dipende da librerie di terze parti. Se si inseriscono semplicemente i file JAR di dipendenza all'interno di un JAR dell'applicazione, il programma di caricamento classi Java standard non sarà in grado di trovare le classi della libreria e l'applicazione non verrà avviata. Invece, è necessario creare un singolo file JAR che contenga le classi dell'applicazione e le risorse associate insieme alle classi di dipendenza e alle risorse. Questi devono essere organizzati come un singolo spazio dei nomi per il classloader da cercare.

Il file JAR di questo tipo viene spesso definito UberJAR.

Creare un UberJAR usando il comando "jar"

La procedura per creare un UberJAR è semplice. (Userò i comandi di Linux per semplicità: i comandi dovrebbero essere identici per Mac OS e simili per Windows).

1. Creare una directory temporanea e cambiarne la directory.

```
$ mkdir tempDir
```

```
$ cd tempDir
```

2. Per ogni file JAR dipendente, *nell'ordine inverso* che devono apparire sul classpath dell'applicazione, è stato utilizzato il comando `jar` per decomprimere il JAR nella directory temporanea.

   ```
   $ jar -xf <path/to/file.jar>
   ```

   Facendo questo per più file JAR si *sovrappone il* contenuto dei JAR.

3. Copia le classi dell'applicazione dall'albero di creazione nella directory temporanea

   ```
   $ cp -r path/to/classes .
   ```

4. Crea UberJAR dai contenuti della directory temporanea:

   ```
   $ jar -cf ../myApplication.jar
   ```

   Se si sta creando un file JAR eseguibile, includere un MANIFEST.MF appropriato come descritto qui.

Creare un UberJAR usando Maven

Se il tuo progetto è stato creato usando Maven, puoi creare un UberJAR usando i plugin "maven-assembly" o "maven-shade". Vedere l'argomento Assembly Maven (nella documentazione di Maven) per i dettagli.

I vantaggi e gli svantaggi di UberJARs

Alcuni dei vantaggi degli UberJAR sono evidenti:

- Un UberJAR è facile da distribuire.
- Non è possibile rompere le dipendenze della libreria per un UberJAR, poiché le librerie sono autonome.

Inoltre, se si utilizza uno strumento appropriato per creare UberJAR, si avrà la possibilità di escludere le classi di libreria che non vengono utilizzate dal file JAR. Tuttavia, questo è tipicamente fatto dall'analisi statica delle classi. Se l'applicazione utilizza la riflessione, l'elaborazione delle annotazioni e tecniche simili, è necessario fare attenzione che le classi non siano escluse in modo errato.

Gli UberJAR hanno anche alcuni svantaggi:

- Se hai molti UberJAR con le stesse dipendenze, ognuno conterrà una copia delle dipendenze.

- Alcune librerie open source hanno licenze che *possono* precludere [1] il loro utilizzo in un UberJAR.

---

1 - Alcune licenze di librerie open source consentono di utilizzare solo la libreria dell'utente finale che è in grado di sostituire una versione della libreria con un'altra. Gli UberJAR possono rendere difficile la sostituzione delle dipendenze della versione.

Leggi Implementazione Java online: https://riptutorial.com/it/java/topic/6840/implementazione-java

Capitolo 78: Implementazioni del sistema di plugin Java

Osservazioni

Se si utilizza un IDE e / o un sistema di compilazione, è molto più semplice impostare questo tipo di progetto. Crei un modulo applicativo principale, quindi un modulo API, quindi crei un modulo plugin e lo faccia dipendere dal modulo API o da entrambi. Successivamente, si configura dove devono essere posizionate le risorse del progetto, nel nostro caso i jar del plugin compilati possono essere inviati direttamente alla directory "plugins", evitando così di fare movimenti manuali.

Examples

Utilizzando URLClassLoader

Esistono diversi modi per implementare un sistema di plugin per un'applicazione Java. Uno dei più semplici è usare *URLClassLoader* . L'esempio seguente comporterà un po 'di codice JavaFX.

Supponiamo di avere un modulo di un'applicazione principale. Questo modulo dovrebbe caricare plugin in forma di Jars dalla cartella 'plugins'. Codice iniziale:

```
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);
    }
}
```

Quindi, creiamo un'interfaccia che rappresenterà un plugin.

```
package main;

public interface Plugin
```

```
{
    default void initialize()
    {
        System.out.println("Initialized "+this.getClass().getName());
    }
    default String name(){return getClass().getSimpleName();}
}
```

Vogliamo caricare classi che implementano questa interfaccia, quindi prima dobbiamo filtrare i file che hanno l'estensione '.jar':

```
File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
```

Se ci sono file, dobbiamo creare raccolte di URL e nomi di classi:

```
    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);
        }

    }
```

Aggiungiamo un HashSet statico a *MainApplication* che manterrà i plugin caricati:

```
static HashSet<Plugin> plugins=new HashSet<>();
```

Successivamente, istanziamo un *URLClassLoader* e *iteriamo* su nomi di classi, istanziando classi che implementano l'interfaccia *Plugin* :

```
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();}
});
```

Quindi, possiamo chiamare i metodi del plugin, ad esempio, per inizializzarli:

```
if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded plugins:"));
    plugins.forEach(plugin -> {
        plugin.initialize();
        loadedPlugins.getChildren().add(new Label(plugin.name()));
});
```

Il codice finale di *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);
    }
}
```

Creiamo due plugin. Ovviamente, la fonte del plugin dovrebbe essere in un modulo separato.

```
package plugins;

import main.Plugin;

public class FirstPlugin implements Plugin
{
    //this plugin has default behaviour
}
```

Secondo plugin:

```
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"));
    }
}
```

Questi plugin devono essere impacchettati in giare standard: questo processo dipende dal tuo IDE o da altri strumenti.

Quando i Jars verranno inseriti direttamente nei "plug-in", *MainApplication* li rileverà e *istanzerà le* classi appropriate.

Leggi Implementazioni del sistema di plugin Java online:
https://riptutorial.com/it/java/topic/7160/implementazioni-del-sistema-di-plugin-java

Capitolo 79: Imposta

Examples

Dichiarazione di un hashset con valori

Puoi creare una nuova classe che eredita da HashSet:

```
Set<String> h = new HashSet<String>() {{
    add("a");
    add("b");
}};
```

Una soluzione di linea:

```
Set<String> h = new HashSet<String>(Arrays.asList("a", "b"));
```

Utilizzando guava:

```
Sets.newHashSet("a", "b", "c")
```

Usando i flussi:

```
Set<String> set3 = Stream.of("a", "b", "c").collect(toSet());
```

Tipi e utilizzo degli insiemi

In generale, gli insiemi sono un tipo di raccolta che memorizza valori univoci. L'univocità è determinata dai metodi `equals()` e `hashCode()` .

L'ordinamento è determinato dal tipo di set.

`HashSet` - Ordinamento casuale

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` - Ordine di inserzione

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` - Per `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"]
```

Inizializzazione

Un Set è una Collezione che non può contenere elementi duplicati. Modella l'astrazione dell'insieme matematico.

`Set` ha la sua implementazione in varie classi come `HashSet` , `TreeSet` , `LinkedHashSet` .

Per esempio:

HashSet:

```
Set<T> set = new HashSet<T>();
```

Qui `T` può essere `String` , `Integer` o qualsiasi altro **oggetto** . **HashSet** consente una rapida ricerca di O (1) ma non ordina i dati aggiunti e perde l'ordine di inserimento degli articoli.

TreeSet:

Memorizza i dati in modo ordinato sacrificando una certa velocità per le operazioni di base che richiedono O (lg (n)). Non mantiene l'ordine di inserzione degli articoli.

```
TreeSet<T> sortedSet = new TreeSet<T>();
```

LinkedHashSet:

Si tratta di un'implementazione di elenchi collegati di `HashSet` Una volta può scorrere gli articoli nell'ordine in cui sono stati aggiunti. L'ordinamento non è fornito per il suo contenuto. Vengono fornite le operazioni di base O (1), tuttavia è presente un costo maggiore rispetto a `HashSet` nel mantenimento dell'elenco dei collegamenti di backup.

```
LinkedHashSet<T> linkedhashset = new LinkedHashSet<T>();
```

Nozioni di base di Set

Cos'è un set?

Un set è una struttura di dati che contiene un insieme di elementi con un'importante proprietà che nessun elemento nel set è uguale.

Tipi di set:

1. **HashSet:** un set supportato da una tabella hash (in realtà un'istanza di HashMap)
2. **HashSet collegato:** un set supportato dalla tabella hash e dall'elenco collegato, con un ordine di iterazione prevedibile
3. **TreeSet:** un'implementazione NavigableSet basata su una TreeMap.

Creare un set

```
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
```

Aggiunta di elementi a un set

Gli elementi possono essere aggiunti a un set usando il metodo `add()`

```
 set.add(12); //  - Adds element 12 to the set
 set.add(13); //  - Adds element 13 to the set
```

Il nostro set dopo aver eseguito questo metodo:

```
set = [12,13]
```

Elimina tutti gli elementi di un Set

```
set.clear();  //Removes all objects from the collection.
```

Dopo questo set sarà:

```
set = []
```

Controlla se un elemento fa parte del Set

L'esistenza di un elemento nel set può essere verificata usando il metodo `contains()`

```
set.contains(0);  //Returns true if a specified object is an element within the set.
```

Uscita: `False`

Controlla se un Set è vuoto

`isEmpty()` metodo `isEmpty()` può essere usato per verificare se un Set è vuoto.

```
set.isEmpty();  //Returns true if the set has no elements
```

Uscita: vero

Rimuovi un elemento dal Set

```
 set.remove(0); // Removes first occurrence of a specified object from the collection
```

Controlla la dimensione del set

```
set.size(); //Returns the number of elements in the collection
```

Uscita: 0

Crea una lista da un Set esistente

Usando una nuova lista

```
List<String> list = new ArrayList<String>(listOfElements);
```

Utilizzando il metodo List.addAll ()

```
    Set<String> set = new HashSet<String>();
    set.add("foo");
    set.add("boo");

    List<String> list = new ArrayList<String>();
    list.addAll(set);
```

Utilizzo dell'API Java 8 Steam

```
List<String> list = set.stream().collect(Collectors.toList());
```

Eliminare i duplicati usando Set

Supponiamo di avere `elements` raccolta e di voler creare un'altra raccolta contenente gli stessi elementi ma **eliminando** tutti i **duplicati** :

```
Collection<Type> noDuplicates = new HashSet<Type>(elements);
```

Esempio :

```
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);
```

Uscita :

```
noDuplicates = [Marco, Emily, John, Jenny]
```

Leggi Imposta online: https://riptutorial.com/it/java/topic/3102/imposta

Capitolo 80: incapsulamento

introduzione

Immagina di avere una classe con alcune variabili piuttosto importanti e che siano state impostate (da altri programmatori dal loro codice) a valori inaccettabili. Il loro codice ha causato errori nel codice. Come soluzione, in OOP, si consente di modificare lo stato di un oggetto (memorizzato nelle sue variabili) solo tramite i metodi. Nascondere lo stato di un oggetto e fornire tutte le interazioni attraverso i metodi di un oggetto è noto come Incapsulamento dei dati.

Osservazioni

È molto più semplice iniziare a contrassegnare una variabile `private` e esporla se necessario piuttosto che nascondere una variabile già `public` .

Esiste un'eccezione in cui l'incapsulamento potrebbe non essere vantaggioso: strutture di dati "stupide" (classi il cui unico scopo è quello di contenere variabili).

```
public class DumbData {
    public String name;
    public int timeStamp;
    public int value;
}
```

In questo caso, l'interfaccia della classe è i dati che contiene.

Si noti che le variabili contrassegnate come `final` possono essere contrassegnate come `public` senza violare l'incapsulamento perché non possono essere modificate dopo essere state impostate.

Examples

Incapsulamento per mantenere invarianti

Ci sono due parti di una classe: l'interfaccia e l'implementazione.

L'interfaccia è la funzionalità esposta della classe. I suoi metodi e variabili pubblici sono parte dell'interfaccia.

L'implementazione è il funzionamento interno di una classe. Altre classi non dovrebbero avere bisogno di conoscere l'implementazione di una classe.

L'incapsulamento si riferisce alla pratica di nascondere l'implementazione di una classe da qualsiasi utente di quella classe. Ciò consente alla classe di formulare ipotesi sul suo stato interno.

Ad esempio, prendi questa classe che rappresenta un angolo:

```
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(){}
}
```

Questa classe si basa su **un'ipotesi di** base (o *invariante*): **angleInDegrees e angleInRadians sono sempre sincronizzati** . Se i membri della classe fossero pubblici, non ci sarebbe alcuna garanzia che le due rappresentazioni degli angoli siano correlate.

Incapsulamento per ridurre l'accoppiamento

Incapsulamento consente di apportare modifiche interne a una classe senza influire sul codice che chiama la classe. Questo riduce l' *accoppiamento* , o quanto una determinata classe si basa sull'implementazione di un'altra classe.

Ad esempio, cambiamo l'implementazione della classe Angle dall'esempio precedente:

```
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(){}
}
```

L'implementazione di questa classe è cambiata in modo che memorizzi solo una rappresentazione dell'angolo e calcola l'altro angolo quando necessario.

Tuttavia, **l'implementazione è cambiata, ma l'interfaccia no** . Se una classe chiamante si affidava all'accesso al metodo angleInRadians, avrebbe dovuto essere modificata per utilizzare la nuova versione di `Angle` . Le classi di chiamata non dovrebbero preoccuparsi della rappresentazione interna di una classe.

Leggi incapsulamento online: https://riptutorial.com/it/java/topic/1295/incapsulamento

Capitolo 81: InputStreams e OutputStreams

Sintassi

- int read (byte [] b) genera IOException

Osservazioni

Nota che il più delle volte NON usi direttamente `InputStream` ma usi `BufferedStream` s, o simili. Questo perché `InputStream` legge dall'origine ogni volta che viene chiamato il metodo di lettura. Ciò può causare un utilizzo significativo della CPU in switch di contesto all'interno e all'esterno del kernel.

Examples

Lettura di InputStream in una stringa

A volte potresti voler leggere l'input da byte in una stringa. Per fare ciò è necessario trovare qualcosa che converta tra `byte` e i "Codepoint" UTF-16 "nativi Java" utilizzati come `char` . Questo è fatto con un `InputStreamReader` .

Per accelerare un po 'il processo, è "normale" allocare un buffer, in modo da non avere un sovraccarico eccessivo durante la lettura da Input.

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();
}
```

Trasformare questo esempio in codice Java 6 (e inferiore) -compatibile è lasciato fuori come esercizio per il lettore.

Scrittura di byte su un OutputStream

Scrittura di byte su un `OutputStream` un byte alla volta

```
OutputStream stream = object.getOutputStream();

byte b = 0x00;
stream.write( b );
```

Scrivere un array di byte

```
byte[] bytes = new byte[] { 0x00, 0x00 };

stream.write( bytes );
```

Scrivere una sezione di un array di byte

```
int offset = 1;
int length = 2;
byte[] bytes = new byte[] { 0xFF, 0x00, 0x00, 0xFF };

stream.write( bytes, offset, length );
```

Flussi di chiusura

La maggior parte dei flussi deve essere chiusa quando hai finito con loro, altrimenti potresti introdurre una perdita di memoria o lasciare un file aperto. È importante che gli stream siano chiusi anche se viene lanciata un'eccezione.

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
}
```

Ricorda: try-with-resources garantisce che le risorse siano state chiuse quando il blocco è stato interrotto, sia che ciò avvenga con il normale flusso di controllo o a causa di un'eccezione.

Java SE 6

A volte, try-with-resources non è un'opzione, o forse stai supportando la versione precedente di Java 6 o precedente. In questo caso, la corretta movimentazione è di usare un `finally` blocco:

```
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...
    }
}
```

Nota che chiudere un flusso wrapper chiuderà anche il suo stream sottostante. Ciò significa che non puoi eseguire il wrapping di un flusso, chiudere il wrapper e continuare a utilizzare il flusso originale.

Copia del flusso di input sul flusso di uscita

Questa funzione copia i dati tra due flussi:

```
void copy(InputStream in, OutputStream out) throws IOException {
    byte[] buffer = new byte[8192];
    while ((bytesRead = in.read(buffer)) > 0) {
        out.write(buffer, 0, bytesRead);
    }
}
```

Esempio -

```
 // reading from System.in and writing to System.out
copy(System.in, System.out);
```

Racchiudere flussi di input / output

`OutputStream` e `InputStream` hanno molte classi diverse, ognuna con una funzionalità unica. Con il wrapping di un flusso attorno ad un altro, si ottiene la funzionalità di entrambi i flussi.

Puoi avvolgere un flusso qualsiasi numero di volte, prendi nota dell'ordine.

Combinazioni utili

Scrivere caratteri su un file mentre si utilizza un buffer

```
File myFile = new File("targetFile.txt");
PrintWriter writer = new PrintWriter(new BufferedOutputStream(new FileOutputStream(myFile)));
```

Comprimere e crittografare i dati prima di scrivere su un file mentre si utilizza un buffer

```
Cipher cipher = ... // Initialize cipher
File myFile = new File("targetFile.enc");
BufferedOutputStream outputStream = new BufferedOutputStream(new DeflaterOutputStream(new
CipherOutputStream(new FileOutputStream(myFile), cipher)));
```

Elenco di wrapper di flusso input / output

involucro	Descrizione
BufferedOutputStream / BufferedInputStream	Mentre `OutputStream` scrive i dati un byte alla volta, `BufferedOutputStream` scrive i dati in blocchi. Questo riduce il numero di chiamate di sistema, migliorando così le prestazioni.
DeflaterOutputStream / DeflaterInputStream	Esegue la compressione dei dati.
InflaterOutputStream / InflaterInputStream	Esegue la decompressione dei dati.
CipherOutputStream / CipherInputStream	Crittografa / decrittografa i dati.
DigestOutputStream / DigestInputStream	Genera Message Digest per verificare l'integrità dei dati.
CheckedOutputStream / CheckedInputStream	Genera un Somma di controllo. CheckSum è una versione più banale di Message Digest.
DataOutputStream / DataInputStream	Consente la scrittura di tipi di dati e stringhe primitivi. Significa per scrivere byte. Piattaforma indipendente.
PrintStream	Consente la scrittura di tipi di dati e stringhe primitivi. Significa per scrivere byte. Piattaforma dipendente
OutputStreamWriter	Converte un OutputStream in un writer. Un OutputStream tratta i byte mentre i writer si occupano di caratteri
PrintWriter	Chiama automaticamente OutputStreamWriter. Consente la scrittura di tipi di dati e stringhe primitivi. Rigorosamente per scrivere caratteri e meglio per scrivere personaggi

Esempio di 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+"-");
    }
  }
}
```

Leggi InputStreams e OutputStreams online: https://riptutorial.com/it/java/topic/110/inputstreams-e-outputstreams

Capitolo 82: Insidie di Java - Nulls e NullPointerException

Osservazioni

Il valore `null` è il valore predefinito per un valore non inizializzato di un campo il cui tipo è un tipo di riferimento.

`NullPointerException` (o NPE) è l'eccezione che viene generata quando si tenta di eseguire un'operazione inappropriata sul riferimento all'oggetto `null` . Tali operazioni includono:

- chiamando un metodo di istanza su un oggetto di destinazione `null` ,
- accedere a un campo di un oggetto obiettivo `null` ,
- tentando di indicizzare un oggetto array `null` o accedere alla sua lunghezza,
- usando un riferimento oggetto `null` come mutex in un blocco `synchronized` ,
- casting di un oggetto `null` riferimento,
- unboxing di un riferimento a oggetti `null` e
- lancio di un riferimento a oggetto `null` .

Le cause principali più comuni per gli NPE:

- dimenticando di inizializzare un campo con un tipo di riferimento,
- dimenticando di inizializzare elementi di una matrice di un tipo di riferimento, o
- non testare i risultati di determinati metodi API che vengono *specificati* come restituire `null` in determinate circostanze.

Esempi di metodi comunemente usati che restituiscono `null` includono:

- Il metodo `get(key)` nell'API di `Map` restituirà un valore `null` se lo chiami con una chiave che non ha una mappatura.
- I `getResource(path)` e `getResourceAsStream(path)` nelle API `ClassLoader` e `Class` restituiranno `null` se non è possibile trovare la risorsa.
- Il metodo `get()` nell'API di `Reference` restituirà `null` se il garbage collector ha cancellato il riferimento.
- Vari metodi `getXxxx` nelle API servlet Java EE restituiscono `null` se si tenta di recuperare un parametro di richiesta inesistente, un attributo di sessione o sessione e così via.

Esistono strategie per evitare NPE indesiderati, come test espliciti per `null` o usando "Yoda Notation", ma queste strategie spesso hanno il risultato indesiderato di *nascondere i* problemi nel codice che dovrebbero essere corretti.

Examples

Pitfall - L'uso non necessario di Primitive Wrapper può portare a

NullPointerExceptions

A volte, i programmatori che sono nuovi Java useranno i tipi primitivi e i wrapper in modo intercambiabile. Questo può portare a problemi. Considera questo esempio:

```
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
```

La `MyRecord` classe `MyRecord` [1] si basa `MyRecord` predefinita per inizializzare i valori sui suoi campi. Quindi, quando creiamo un `new` record, i campi `a` e `b` saranno impostati a zero ei campi `c` e `d` saranno impostati su `null` .

Quando proviamo a utilizzare i campi inizializzati predefiniti, vediamo che i campi `int` funzionano sempre, ma i campi `Integer` funzionano in alcuni casi e non in altri. Nello specifico, nel caso in cui non riesca (con `d`), ciò che accade è che l'espressione sul lato destro tenta di annullare la selezione di un riferimento `null` , e questo è il `NullPointerException` per cui viene generata l' `NullPointerException` .

Ci sono un paio di modi per osservare questo:

- Se i campi `c` e `d` devono essere wrapper primitivi, allora non dovremmo fare affidamento sull'inizializzazione di default, o dovremmo testare per `null` . Per ex è l'approccio corretto a *meno* che *non* vi sia un significato definito per i campi nello stato `null` .
- Se i campi non hanno bisogno di essere involucri primitivi, allora è un errore renderli involucri primitivi. Oltre a questo problema, i wrapper primitivi hanno overhead aggiuntivi rispetto ai tipi primitivi.

La lezione qui è di non usare tipi di wrapper primitivi a meno che tu non ne abbia davvero bisogno.

---

1 - Questa classe non è un esempio di buona pratica di codifica. Ad esempio, una classe ben progettata non avrebbe campi pubblici. Tuttavia, questo non è il punto di questo esempio.

Pitfall - Uso di null per rappresentare una matrice o una raccolta vuota

Alcuni programmatori pensano che sia una buona idea risparmiare spazio usando un `null` per rappresentare una matrice o una collezione vuota. Anche se è vero che puoi risparmiare una piccola quantità di spazio, il rovescio della medaglia è che rende il tuo codice più complicato e più fragile. Confronta queste due versioni di un metodo per sommare un array:

La prima versione è come si codificherà normalmente il metodo:

```
/**
 * 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 seconda versione è il modo in cui è necessario codificare il metodo se si ha l'abitudine di utilizzare `null` per rappresentare una matrice vuota.

```
/**
 * 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;
}
```

Come puoi vedere, il codice è un po 'più complicato. Questo è direttamente attribuibile alla decisione di utilizzare `null` in questo modo.

Considerare ora se questa matrice che potrebbe essere un `null` viene utilizzata in molti posti. In ogni luogo in cui lo si utilizza, è necessario considerare se è necessario eseguire il test per `null` . Se si perde un test `null` che deve essere presente, si rischia una `NullPointerException` . Quindi, la strategia di usare `null` in questo modo porta a rendere la tua applicazione più fragile; cioè più vulnerabile alle conseguenze degli errori del programmatore.

La lezione qui è usare matrici vuote e liste vuote quando questo è ciò che intendi.

```
int[] values = new int[0];                      // always empty
List<Integer> list = new ArrayList();           // initially empty
List<Integer> list = Collections.emptyList();  // always empty
```

L'overhead dello spazio è piccolo e ci sono altri modi per ridurlo al minimo se questa è una cosa utile da fare.

Pitfall - "Making good" null inaspettati

Su StackOverflow, spesso vediamo un codice come questo in Answers:

```
public String joinStrings(String a, String b) {
    if (a == null) {
        a = "";
    }
    if (b == null) {
        b = "";
    }
    return a + ": " + b;
}
```

Spesso, questo è accompagnato da un'asserzione che è "best practice" per testare `null` come questo per evitare `NullPointerException` .

È la migliore pratica? In breve: No.

Ci sono alcune ipotesi sottostanti che devono essere messe in discussione prima che possiamo dire se è una buona idea farlo nelle nostre `joinStrings` :

Che cosa significa "a" o "b" essere nulli?

Un valore di `String` può essere zero o più caratteri, quindi abbiamo già un modo per rappresentare una stringa vuota. `null` significa qualcosa di diverso da `""` ? Se no, allora è problematico avere due modi per rappresentare una stringa vuota.

Il null proviene da una variabile non inizializzata?

Un `null` può provenire da un campo non inizializzato o da un elemento di matrice non inizializzata. Il valore potrebbe non essere inizializzato dal design o per errore. Se è stato per caso, questo è un bug.

Il null rappresenta un "non so" o "un valore mancante"?

A volte un `null` può avere un significato genuino; per esempio che il valore reale di una variabile è sconosciuto o non disponibile o "opzionale". In Java 8, la classe `Optional` fornisce un modo migliore di esprimerlo.

Se questo è un bug (o un errore di progettazione) dovremmo "fare del bene"?

Un'interpretazione del codice è che stiamo "facendo del bene" un `null` inaspettato usando una stringa vuota al suo posto. È la strategia corretta? Sarebbe meglio lasciare che `NullPointerException` verifichi, quindi catturare l'eccezione più in alto nello stack e registrarlo come un bug?

Il problema di "fare del bene" è che è in grado di nascondere il problema o renderlo più difficile da diagnosticare.

Questo è efficiente / buono per la qualità del codice?

Se l'approccio del "buon funzionamento" viene utilizzato in modo coerente, il tuo codice conterrà molti test nulli "difensivi". Questo renderà più lungo e difficile da leggere. Inoltre, tutto questo test e "fare il bravo" è suscettibile di influire sulle prestazioni della vostra applicazione.

In sintesi

Se `null` è un valore significativo, il test per il caso `null` è l'approccio corretto. Il corollario è che se un valore `null` è significativo, questo dovrebbe essere chiaramente documentato nei javadoc di tutti i metodi che accettano il valore `null` o lo restituiscono.

In caso contrario, si tratta di un'idea migliore per il trattamento di un inaspettato `null` come un errore di programmazione, e lasciare che il `NullPointerException` accadere in modo che lo sviluppatore viene a sapere che c'è un problema nel codice.

Pitfall - Restituisce null invece di generare un'eccezione

Alcuni programmatori Java hanno un'avversione generale a lanciare o propagare eccezioni. Questo porta al codice come il seguente:

```
public Reader getReader(String pathname) {
    try {
        return new BufferedReader(FileReader(pathname));
    } catch (IOException ex) {
        System.out.println("Open failed: " + ex.getMessage());
        return null;
    }
```

}

Quindi qual è il problema?

Il problema è che `getReader` restituisce un valore `null` come valore speciale per indicare che il `Reader` non può essere aperto. Ora il valore restituito deve essere testato per vedere se è `null` prima che venga utilizzato. Se il test viene `NullPointerException` , il risultato sarà `NullPointerException` .

Ci sono in realtà tre problemi qui:

1. L' `IOException` stato catturato troppo presto.
2. La struttura di questo codice significa che esiste il rischio di perdita di una risorsa.
3. È stato utilizzato un valore `null` quindi è stato restituito perché nessun `Reader` "reale" era disponibile per il reso.

In effetti, supponendo che l'eccezione abbia dovuto essere catturata in anticipo in questo modo, c'erano un paio di alternative per restituire `null` :

1. Sarebbe possibile implementare una classe `NullReader` ; ad esempio quello in cui le

operazioni dell'API si comportano come se il lettore fosse già nella posizione di "fine del file".
2. Con Java 8, sarebbe possibile dichiarare `getReader` come restituire un `Optional<Reader>` .

Pitfall - Non verificare se un flusso I / O non è nemmeno inizializzato quando si chiude

Per evitare perdite di memoria, non si dovrebbe dimenticare di chiudere un flusso di input o un flusso di output il cui lavoro è stato eseguito. Questo di solito è fatto con una frase `try` - `catch` - `finally` senza 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();
    }
}
```

Mentre il codice sopra potrebbe sembrare innocente, ha un difetto che può rendere impossibile il debugging. Se la riga dove `out` è inizializzata (`out = new FileOutputStream(filename)`) genera un'eccezione, quindi `out` sarà `null` quando `out.close()` viene eseguito, risultando in una brutta `NullPointerException` !

Per evitare ciò, assicurati semplicemente che lo stream non sia `null` prima di provare a chiuderlo.

```
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 approccio ancora migliore è quello di `try` -con-risorse, dal momento che sarà chiudere automaticamente il flusso con una probabilità da 0 a gettare un NPE, senza la necessità di un `finally` bloccare.

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    try (FileOutputStream out = new FileOutputStream(filename)) {
        for(; count > 0; count--)
            out.write(0);
    }
}
```

Pitfall - Usando la "notazione Yoda" per evitare NullPointerException

Un sacco di codice di esempio pubblicato su StackOverflow include snippet come questo:

```
if ("A".equals(someString)) {
    // do something
}
```

Ciò "previene" o "evita" una possibile `NullPointerException` nel caso in cui `someString` sia `null` . Inoltre, è discutibile che

```
    "A".equals(someString)
```

è meglio di:

```
    someString != null && someString.equals("A")
```

(È più conciso, e in alcune circostanze potrebbe essere più efficiente. Tuttavia, come discuteremo di seguito, la concisione potrebbe essere negativa.)

Tuttavia, la vera trappola consiste nell'usare il test Yoda **per evitare `NullPointerExceptions`** come una questione di abitudine.

Quando scrivi `"A".equals(someString)` realtà stai "facendo del bene" nel caso in cui `someString` sia `null` . Ma come un altro esempio (Pitfall - "Making good" null inaspettati) spiega, "fare buoni" valori `null` può essere dannoso per una serie di motivi.

Ciò significa che le condizioni Yoda non sono "best practice" [1] . A meno che non sia previsto il valore `null` , è meglio consentire l' `NullPointerException` modo da ottenere un errore di test dell'unità (o una segnalazione di errore). Ciò consente di trovare e correggere il bug che ha causato l'apparizione del `null` inaspettato / indesiderato.

Le condizioni Yoda devono essere utilizzate solo nei casi in cui è *previsto* il valore `null` perché l'oggetto che si sta provando proviene da un'API *documentata* che restituisce un valore `null` . E, discutibilmente, potrebbe essere meglio usare uno dei modi meno carini per esprimere il test, perché questo aiuta a mettere in evidenza il test `null` a qualcuno che sta rivedendo il tuo codice.

---

1 - Secondo Wikipedia : *"Le migliori pratiche di codifica sono un insieme di regole informali che la comunità di sviluppo del software ha imparato nel tempo e che possono aiutare a migliorare la qualità del software."* . L'uso della notazione Yoda non raggiunge questo risultato. In molte situazioni, peggiora il codice.

Leggi Insidie di Java - Nulls e NullPointerException online:
https://riptutorial.com/it/java/topic/5680/insidie---di-java---nulls-e-nullpointerexception

Capitolo 83: Insidie di Java - Problemi di prestazioni

introduzione

Questo argomento descrive una serie di "insidie" (ad esempio errori commessi dai programmatori java principianti) che si riferiscono alle prestazioni dell'applicazione Java.

Osservazioni

Questo argomento descrive alcune pratiche di codifica Java "micro" che sono inefficienti. Nella maggior parte dei casi, le inefficienze sono relativamente piccole, ma vale comunque la pena di evitarle.

Examples

Pitfall - I costi generali di creazione dei messaggi di log

`TRACE` livelli di log `TRACE` e `DEBUG` sono lì per essere in grado di trasmettere dettagli elevati sul funzionamento del codice dato in fase di esecuzione. In genere, è consigliabile impostare il livello di registro sopra questi, tuttavia è necessario prestare attenzione affinché tali istruzioni non influiscano sulle prestazioni anche se apparentemente "disattivate".

Considera questa dichiarazione di registro:

```
// Processing a request of some kind, logging the parameters
LOG.debug("Request coming from " + myInetAddress.toString()
          + " parameters: " + Arrays.toString(veryLongParamArray));
```

Anche quando il livello di registro è impostato su `INFO` , gli argomenti passati a `debug()` verranno valutati in ogni esecuzione della riga. Questo rende inutilmente il consumo su diversi fronti:

- `String` stringhe: verranno create più istanze `String`
- `InetAddress` potrebbe persino eseguire una ricerca DNS.
- `veryLongParamArray` potrebbe essere molto lungo: la creazione di una stringa che consuma memoria richiede tempo

Soluzione

La maggior parte della struttura di logging fornisce mezzi per creare messaggi di log usando stringhe fisse e riferimenti a oggetti. Il messaggio di registro verrà valutato solo se il messaggio viene effettivamente registrato. Esempio:

```
// No toString() evaluation, no string concatenation if debug is disabled
LOG.debug("Request coming from {} parameters: {}", myInetAddress, parameters));
```

Funziona molto bene finché tutti i parametri possono essere convertiti in stringhe usando String.valueOf (Object) . Se il compendio dei messaggi di registro è più complesso, il livello di registro può essere controllato prima della registrazione:

```
if (LOG.isDebugEnabled()) {
    // Argument expression evaluated only when DEBUG is enabled
    LOG.debug("Request coming from {}, parameters: {}", myInetAddress,
              Arrays.toString(veryLongParamArray);
}
```

Qui, `LOG.debug()` con il costoso `Arrays.toString(Obect[])` viene elaborato solo quando `DEBUG` è effettivamente abilitato.

Pitfall - La concatenazione di stringhe in un loop non viene ridimensionata

Considera il seguente codice come illustrazione:

```
public String joinWords(List<String> words) {
    String message = "";
    for (String word : words) {
        message = message + " " + word;
    }
    return message;
}
```

Sfortunatamente questo codice è inefficiente se l'elenco delle `words` è lungo. La radice del problema è questa affermazione:

```
message = message + " " + word;
```

Per ciascuna iterazione del ciclo, questa istruzione crea una nuova stringa di `message` contenente una copia di tutti i caratteri nella stringa del `message` originale con caratteri aggiuntivi aggiunti ad essa. Questo genera un sacco di stringhe temporanee e fa molte copie.

Quando analizziamo `joinWords` , supponendo che ci siano N parole con una lunghezza media di M, scopriamo che le stringhe temporanee di O (N) sono create e che i caratteri O (MN^2) verranno copiati nel processo. Il componente N^2 è particolarmente preoccupante.

L'approccio consigliato per questo tipo di problema [1] consiste nell'utilizzare un oggetto `StringBuilder` anziché una concatenazione di stringhe come segue:

```
public String joinWords2(List<String> words) {
    StringBuilder message = new StringBuilder();
    for (String word : words) {
        message.append(" ").append(word);
    }
    return message.toString();
}
```

L'analisi di `joinWords2` deve tenere conto delle spese generali di "crescita" dell'array di supporto `StringBuilder` che contiene i caratteri del builder. Tuttavia, si scopre che il numero di nuovi oggetti creati è O (logN) e che il numero di caratteri copiati è O (MN) caratteri. Quest'ultimo include caratteri copiati nella chiamata finale a `toString()` .

(Potrebbe essere possibile ottimizzarlo ulteriormente, creando `StringBuilder` con la capacità corretta per iniziare. Tuttavia, la complessità complessiva rimane la stessa.)

Tornando al metodo `joinWords` originale, si scopre che la dichiarazione critica verrà ottimizzata da un tipico compilatore Java per qualcosa di simile a questo:

```
StringBuilder tmp = new StringBuilder();
tmp.append(message).append(" ").append(word);
message = tmp.toString();
```

Tuttavia, il compilatore Java non "solleva" il `StringBuilder` dal ciclo, come abbiamo fatto a mano nel codice per `joinWords2` .

Riferimento:

- "L'operatore Java's String '+' è lento?"

---

1 - In Java 8 e `Joiner` successive, la classe `Joiner` può essere utilizzata per risolvere questo particolare problema. Tuttavia, questo non è ciò di cui si *suppone* questo esempio.

Trappola - L'uso di "nuovo" per creare istanze di wrapper primitive è inefficiente

Il linguaggio Java ti permette di usare `new` per creare istanze `Integer` , `Boolean` e così via, ma generalmente è una cattiva idea. È preferibile utilizzare l'autoboxing (Java 5 e `valueOf` successive) o il metodo `valueOf` .

```
Integer i1 = new Integer(1);      // BAD
Integer i2 = 2;                   // BEST (autoboxing)
Integer i3 = Integer.valueOf(3);  // OK
```

La ragione per cui l'utilizzo del `new Integer(int)` esplicito è una cattiva idea è che crea un nuovo oggetto (se non ottimizzato dal compilatore JIT). Al contrario, quando vengono utilizzati il box automatico o una chiamata `valueOf` esplicita, il runtime Java tenterà di riutilizzare un oggetto `Integer` da una cache di oggetti preesistenti. Ogni volta che il runtime ha una cache "hit", evita di creare un oggetto. Ciò consente anche di risparmiare memoria heap e di ridurre i costi generali del GC causati dall'abbandono degli oggetti.

Gli appunti:

1. Nelle recenti implementazioni Java, l'autoboxing è implementato chiamando `valueOf` e ci sono cache per `Boolean` , `Byte` , `Short` , `Integer` , `Long` e `Character` .
2. Il comportamento di caching per i tipi integrali è richiesto dalla specifica del linguaggio Java.

Trappola: chiamare "nuova stringa (stringa)" è inefficiente

L'uso di una `new String(String)` per duplicare una stringa è inefficiente e quasi sempre non necessario.

- Gli oggetti stringa sono immutabili, quindi non è necessario copiarli per proteggerli dalle modifiche.
- In alcune versioni precedenti di Java, gli oggetti `String` possono condividere array di backup con altri oggetti `String` . In queste versioni, è possibile perdere memoria creando una sottostringa (piccola) di una stringa (grande) e conservandola. Tuttavia, da Java 7 in poi, gli array di backing delle `String` non sono condivisi.

In assenza di vantaggi tangibili, chiamare la `new String(String)` è semplicemente uno spreco:

- Fare la copia richiede tempo CPU.
- La copia utilizza più memoria che aumenta il footprint memorum dell'applicazione e / o aumenta i costi generali del GC.
- Operazioni come `equals(Object)` e `hashCode()` possono essere più lente se gli oggetti String vengono copiati.

Pitfall - Calling System.gc () è inefficiente

È (quasi sempre) una cattiva idea chiamare `System.gc()` .

Javadoc per il metodo `gc()` specifica quanto segue:

> "Chiamare il metodo `gc` suggerisce che la Java Virtual Machine spenda gli sforzi per riciclare oggetti inutilizzati al fine di rendere disponibile la memoria attualmente occupata per il riutilizzo rapido. Quando il controllo ritorna dalla chiamata al metodo, la Java Virtual Machine ha fatto il massimo sforzo per reclamare spazio da tutti gli oggetti scartati. "

Ci sono un paio di punti importanti che si possono trarre da questo:

1. L'uso della parola "suggerisce" piuttosto che (dire) "dice" significa che la JVM è libera di ignorare il suggerimento. Il comportamento JVM predefinito (versioni recenti) deve seguire il suggerimento, ma questo può essere sovrascritto impostando `-XX:+DisableExplicitGC` quando si avvia JVM.

2. L'espressione "il miglior tentativo di recuperare spazio da tutti gli oggetti scartati" implica che chiamare `gc` inneschi una garbage collection "completa".

Quindi, perché chiamare `System.gc()` una cattiva idea?

Innanzitutto, eseguire una garbage collection completa è costoso. Un GC completo comporta la visita e la "marcatura" di ogni oggetto che è ancora raggiungibile; vale a dire ogni oggetto che non è spazzatura. Se si attiva questo quando non c'è molta immondizia da raccogliere, allora il GC fa un sacco di lavoro con relativamente poco beneficio.

In secondo luogo, una garbage collection completa è suscettibile di disturbare le proprietà "locality" degli oggetti che non vengono raccolti. Gli oggetti allocati dallo stesso thread all'incirca nello stesso momento tendono ad essere allocati vicini in memoria. Questo è buono. È probabile che gli oggetti assegnati nello stesso momento siano correlati; vale a dire riferimento l'un l'altro. Se l'applicazione utilizza tali riferimenti, è probabile che l'accesso alla memoria risulti più veloce a causa di vari effetti di memorizzazione nella cache e nelle pagine. Sfortunatamente, una raccolta completa dei rifiuti tende a spostare gli oggetti in modo che gli oggetti che erano una volta vicini siano ora più distanti.

In terzo luogo, l'esecuzione di una garbage collection completa può mettere in pausa l'applicazione fino al completamento della raccolta. Mentre questo sta accadendo, la tua applicazione non risponderà.

In effetti, la strategia migliore è lasciare che la JVM decida quando eseguire il GC e quale tipo di raccolta eseguire. Se non interferisci, la JVM sceglierà un tipo di tempo e di raccolta che ottimizzi il throughput o minimizzi i tempi di pausa del GC.

All'inizio abbiamo detto "... (quasi sempre) una cattiva idea ...". In effetti ci sono un paio di scenari in cui *potrebbe* essere una buona idea:

1. Se si sta implementando un test unitario per un codice che è sensibile alla garbage collection (ad es. Qualcosa che riguarda i finalizzatori o i riferimenti debole / soft / phantom `System.gc()` potrebbe essere necessario chiamare `System.gc()` .
2. In alcune applicazioni interattive, ci possono essere dei momenti particolari in cui l'utente non si cura se c'è una pausa nella raccolta dei dati inutili. Un esempio è un gioco in cui ci sono pause naturali nel "gioco"; ad esempio quando si carica un nuovo livello.

Trappola: l'uso eccessivo di tipi di wrapper primitivi è inefficiente

Considera questi due pezzi di codice:

```
int a = 1000;
int b = a + 1;
```

e

```
Integer a = 1000;
Integer b = a + 1;
```

Domanda: quale versione è più efficiente?

Risposta: Le due versioni sembrano quasi identiche, ma la prima versione è molto più efficiente della seconda.

La seconda versione usa una rappresentazione per i numeri che usano più spazio, e si basa sul box automatico e sull'automodifica dietro le quinte. In effetti la seconda versione è direttamente equivalente al seguente codice:

```
Integer a = Integer.valueOf(1000);               // box 1000
Integer b = Integer.valueOf(a.intValue() + 1);   // unbox 1000, add 1, box 1001
```

Confrontando questo con l'altra versione che usa `int` , ci sono chiaramente tre chiamate di metodo extra quando viene usato l' `Integer` . Nel caso di `valueOf` , le chiamate creeranno e inizializzeranno un nuovo oggetto `Integer` . È probabile che tutto questo lavoro di boxing e unboxing in più renderà la seconda versione un ordine di grandezza più lenta della prima.

Oltre a ciò, la seconda versione alloca gli oggetti nell'heap in ogni `valueOf` chiamata. Mentre l'utilizzo dello spazio è specifico della piattaforma, è probabile che si trovi nella regione di 16 byte per ogni oggetto `Integer` . Al contrario, la versione `int` bisogno di zero spazio su heap, assumendo che `a` e `b` siano variabili locali.

Un altro grande motivo per cui le primitive sono più veloci rispetto al loro equivalente in scatola è il modo in cui i rispettivi tipi di array sono disposti in memoria.

Se prendi `int[]` e `Integer[]` come esempio, nel caso di un `int[]` i *valori* `int` sono disposti in modo contiguo nella memoria. Ma nel caso di un `Integer[]` non sono i valori che sono disposti, ma i riferimenti (puntatori) agli oggetti `Integer` , che a loro volta contengono i valori `int` effettivi.

Oltre ad essere un ulteriore livello di riferimento, questo può essere un grande serbatoio quando si tratta di localizzare la cache quando si esegue un'iterazione sui valori. Nel caso di un `int[]` la CPU potrebbe recuperare tutti i valori dell'array, nella sua cache in una volta, perché sono contigui in memoria. Ma nel caso di un `Integer[]` la CPU deve potenzialmente eseguire un recupero di memoria aggiuntiva per ciascun elemento, poiché l'array contiene solo riferimenti ai valori effettivi.

In breve, l'uso di tipi di wrapper primitivi è relativamente costoso sia nella CPU che nelle risorse di memoria. Usarli inutilmente è efficiente.

Pitfall - Iterare le chiavi di una mappa può essere inefficiente

Il seguente codice di esempio è più lento di quanto deve essere:

```
Map<String, String> map = new HashMap<>();
for (String key : map.keySet()) {
    String value = map.get(key);
    // Do something with key and value
}
```

Questo perché richiede una ricerca della mappa `get()` metodo `get()`) per ogni chiave nella mappa. Questa ricerca potrebbe non essere efficiente (in una HashMap, implica chiamare `hashCode` sulla chiave, quindi cercare il bucket corretto nelle strutture di dati interne e talvolta persino chiamare `equals`). Su una mappa di grandi dimensioni, questo potrebbe non essere un overhead banale.

Il modo corretto per evitare ciò è di ripetere le voci della mappa, che sono dettagliate nell'argomento Raccolte

Pitfall - Usare size () per verificare se una collezione è vuota è inefficiente.

Java Collections Framework fornisce due metodi correlati per tutti gli oggetti `Collection` :

- `size()` restituisce il numero di voci in una `Collection` , e
- `isEmpty()` metodo `isEmpty()` restituisce true se (e solo se) la `Collection` è vuota.

Entrambi i metodi possono essere utilizzati per testare il vuoto di raccolta. Per esempio:

```
Collection<String> strings = new ArrayList<>();
boolean isEmpty_wrong = strings.size() == 0; // Avoid this
boolean isEmpty = strings.isEmpty();         // Best
```

Sebbene questi approcci abbiano lo stesso aspetto, alcune implementazioni di raccolta non memorizzano le dimensioni. Per una tale raccolta, l'implementazione di `size()` deve calcolare la dimensione ogni volta che viene chiamata. Per esempio:

- Una semplice lista di classi collegate (ma non `java.util.LinkedList`) potrebbe dover attraversare la lista per contare gli elementi.
- La classe `ConcurrentHashMap` deve sommare le voci in tutti i "segmenti" della mappa.
- Un'implementazione lenta di una raccolta potrebbe dover realizzare l'intera collezione in memoria per contare gli elementi.

Al contrario, un metodo `isEmpty()` deve solo verificare se c'è *almeno un* elemento nella collezione. Questo non implica il conteggio degli elementi.

Mentre `size() == 0` non è sempre meno efficiente che `isEmpty()` , è concepibile che un correttamente attuato `isEmpty()` per essere meno efficiente di `size() == 0` . Quindi `isEmpty()` è preferito.

Trappola: problemi di efficienza con espressioni regolari

La corrispondenza delle espressioni regolari è uno strumento potente (in Java e in altri contesti) ma presenta alcuni inconvenienti. Una di queste espressioni regolari tende ad essere piuttosto costosa.

Le istanze Pattern and Matcher devono essere riutilizzate

Considera il seguente esempio:

```
/**
 * 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;
}
```

Questo codice è corretto, ma è inefficiente. Il problema è nella chiamata alle `matches(...)` . Sotto il cofano, `s.matches("[A-Za-z0-9]*")` è equivalente a questo:

```
Pattern.matches(s, "[A-Za-z0-9]*")
```

che è a sua volta equivalente a

```
Pattern.compile("[A-Za-z0-9]*").matcher(s).matches()
```

La `Pattern.compile("[A-Za-z0-9]*")` analizza l'espressione regolare, la analizza e costruisce un oggetto `Pattern` che contiene la struttura dati che verrà utilizzata dal motore regex. Questo è un calcolo non banale. Quindi viene creato un oggetto `Matcher` per racchiudere l'argomento `s` . Infine chiamiamo `match()` per fare la corrispondenza del pattern attuale.

Il problema è che questo lavoro viene ripetuto per ogni iterazione del ciclo. La soluzione è ristrutturare il codice come segue:

```
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;
}
```

Si noti che javadoc per gli stati `Pattern` :

> Le istanze di questa classe sono immutabili e sono sicure per l'utilizzo da più thread simultanei. Le istanze della classe `Matcher` non sono sicure per tale uso.

Non usare match () quando dovresti usare find ()

Supponiamo di voler verificare se una stringa `s` contiene tre o più cifre in una riga. Puoi esprimerlo in vari modi, tra cui:

```
  if (s.matches(".*[0-9]{3}.*")) {
      System.out.println("matches");
  }
```

o

```
 if (Pattern.compile("[0-9]{3}").matcher(s).find()) {
     System.out.println("matches");
 }
```

Il primo è più conciso, ma è anche probabile che sia meno efficiente. A prima vista, la prima versione cercherà di abbinare l'intera stringa al modello. Inoltre, dato che ". *" È un pattern "goloso", è probabile che il pattern matcher faccia avanzare "impazientemente" la fine della stringa, e backtrack fino a quando non trova una corrispondenza.

Al contrario, la seconda versione cercherà da sinistra a destra e interromperà la ricerca non appena trova le 3 cifre di seguito.

Utilizzare alternative più efficienti alle espressioni regolari

Le espressioni regolari sono uno strumento potente, ma non dovrebbero essere il tuo unico strumento. Molte attività possono essere svolte in modo più efficiente in altri modi. Per esempio:

```
Pattern.compile("ABC").matcher(s).find()
```

fa la stessa cosa di:

```
s.contains("ABC")
```

tranne che quest'ultimo è molto più efficiente. (Anche se è possibile ammortizzare il costo della compilazione dell'espressione regolare).

Spesso, la forma non regex è più complicata. Ad esempio, il test eseguito da `matches()` chiama il metodo `allAlplanumeric` precedente può essere riscritto come:

```
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;
}
```

Ora questo è più codice rispetto all'utilizzo di un `Matcher` , ma sarà anche molto più veloce.

Catastrophic Backtracking

(Questo è potenzialmente un problema con tutte le implementazioni delle espressioni regolari, ma lo menzioneremo qui perché è un trabocchetto per l'utilizzo di `Pattern` .)

Considera questo esempio (forzato):

```
Pattern pat = Pattern.compile("(A+)+B");
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAB").matches());
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAC").matches());
```

La prima chiamata `println` verrà stampata rapidamente `true` . Il secondo stamperà `false` . Infine. Infatti, se sperimentate il codice sopra, vedrete che ogni volta che aggiungete un `A` prima del `C` , il tempo impiegato raddoppierà.

Questo comportamento è un esempio di *backtracking catastrofico* . Il motore di corrispondenza dei modelli che implementa la corrispondenza delle espressioni regolari sta tentando inutilmente tutti i *possibili* modi in cui il modello *potrebbe* corrispondere.

Vediamo cosa significa in realtà `(A+)+B` Superficialmente, sembra dire "uno o più personaggi `A` seguiti da un valore `B` ", ma in realtà dice uno o più gruppi, ognuno dei quali è costituito da uno o più caratteri `A` Quindi, ad esempio:

- 'AB' corrisponde solo in un modo: '(A) B'
- 'AAB' corrisponde a due modi: '(AA) B' o '(A) (A) B`
- 'AAAB' corrisponde a quattro modi: '(AAA) B' o '(AA) (A) B `or '(A)(AA)B` o '(A) (A) (A) B`
- e così via

In altre parole, il numero di corrispondenze possibili è 2 ^{N} dove N è il numero di caratteri `A`

L'esempio sopra è chiaramente artificioso, ma i modelli che esibiscono questo tipo di caratteristiche di performance (cioè `O(2^N)` o `O(N^K)` per un `K` grande) sorgono frequentemente quando si usano espressioni regolari sconsiderate. Ci sono alcuni rimedi standard:

- Evitare di annidare i pattern ripetuti all'interno di altri pattern ripetitivi.
- Evita di usare troppi pattern ripetuti.
- Utilizzare la ripetizione senza retromarcia come appropriato.
- Non utilizzare espressioni regex per attività di analisi complicate. (Scrivi invece un parser adeguato).

Infine, fai attenzione alle situazioni in cui un utente o un client API può fornire una stringa regex con caratteristiche patologiche. Ciò può portare a "denial of service" accidentale o intenzionale.

Riferimenti:

- Il tag Regular Expressions , in particolare http://www.Scriptutorial.com/regex/topic/259/getting-started-with-regular-expressions/977/backtracking#t=20161001033913136116 3 e http://www.riptutorial.com/regex / topic / 259 / getting-iniziato-con-Regular-espressioni / 4527 / quando-si-deve-non-uso-regolari espressioni # t = 20161001033959356491 3
- "Regex Performance" di Jeff Atwood.
- "Come uccidere Java con un'espressione regolare" di Andreas Haufler.

Pitfall - Le stringhe Internazionali in modo che tu possa usare == è una cattiva idea

Quando alcuni programmatori vedono questo consiglio:

> "Testare le stringhe usando `==` non è corretto (a meno che le stringhe non siano internate)"

la loro reazione iniziale è alle stringhe interne in modo che possano usare `==` . (Dopotutto `==` è più veloce di chiamare `String.equals(...)` , non è vero).

Questo è l'approccio sbagliato, da un certo numero di prospettive:

Fragilità

Prima di tutto, puoi usare tranquillamente `==` se sai che *tutti* gli oggetti `String` che stai testando sono stati internati. Il JLS garantisce che i valori letterali stringa nel codice sorgente siano stati internati. Tuttavia, nessuna delle API Java SE standard garantisce di restituire stringhe internate, a parte `String.intern(String)` stesso. Se manchi solo una fonte di oggetti `String` che non sono stati internati, la tua applicazione sarà inaffidabile. Questa inaffidabilità si manifesterà come falsi negativi piuttosto che come eccezioni che potrebbero renderlo più difficile da rilevare.

Costi dell'utilizzo di 'intern ()'

Sotto il cofano, interning funziona mantenendo una tabella hash che contiene oggetti `String` precedentemente internati. Viene utilizzato un tipo di meccanismo di riferimento debole in modo che la tabella hash interna non diventi una perdita di archiviazione. Mentre la tabella hash è implementato in codice nativo (a differenza `HashMap` , `HashTable` e così via), i `intern` chiamate sono ancora relativamente costosi in termini di CPU e memoria utilizzata.

Questo costo deve essere confrontato con il risparmio che otterremo usando `==` invece di `equals` . In realtà, non stiamo andando in pareggio, a meno che ogni stringa internata venga confrontata con altre stringhe "un paio di volte".

(A parte: le poche situazioni in cui vale la pena internazionalizzare tendono a ridurre la memoria del footprint di un'applicazione in cui le stesse stringhe ricorrono molte volte *e* quelle stringhe hanno una lunga durata.)

L'impatto sulla raccolta dei rifiuti

Oltre ai costi diretti della CPU e della memoria sopra descritti, le stringhe interne influiscono sulle prestazioni del garbage collector.

Per le versioni di Java precedenti a Java 7, le stringhe internate vengono conservate nello spazio "PermGen" che viene raccolto di rado. Se è necessario raccogliere PermGen, questo (in genere) attiva una garbage collection completa. Se lo spazio PermGen si riempie completamente, la JVM si arresta in modo anomalo, anche se c'era spazio libero negli spazi heap normali.

In Java 7, il pool di stringhe è stato spostato da "PermGen" nell'heap normale. Tuttavia, la tabella hash sarà ancora una struttura di dati di lunga durata, che farà sì che le stringhe internamente

siano di lunga durata. (Anche se gli oggetti stringa interni sono stati allocati nello spazio Eden, molto probabilmente verrebbero promossi prima di essere raccolti).

Pertanto, in tutti i casi, l'internatura di una stringa prolungherà la sua durata rispetto a una stringa ordinaria. Ciò aumenterà i costi generali di raccolta dei dati obsoleti nel corso della durata della JVM.

Il secondo problema è che la tabella hash deve utilizzare un meccanismo di riferimento debole di qualche tipo per impedire che la stringa internamente perdi memoria. Ma un tale meccanismo è più lavoro per il garbage collector.

Queste spese generali di raccolta dei dati inutili sono difficili da quantificare, ma non c'è dubbio che esistano. Se usi `intern` , potrebbero essere significativi.

La dimensione hashtable del pool di stringhe

Secondo questa fonte , da Java 6 in poi, il pool di stringhe viene implementato come tabella hash di dimensioni fisse con catene per gestire stringhe che eseguono lo hash nello stesso bucket. Nelle prime versioni di Java 6, la tabella hash aveva una dimensione costante (cablata). Un parametro di ottimizzazione (`-XX:StringTableSize`) è stato aggiunto come aggiornamento mid-life a Java 6. Quindi, in un aggiornamento di metà vita di Java 7, la dimensione predefinita del pool è stata modificata da `1009` a `60013` .

La linea di fondo è che se si intende utilizzare `intern` internamente nel proprio codice, è *consigliabile* scegliere una versione di Java in cui la dimensione della tabella hash può essere regolata e assicurarsi di regolarne la dimensione in modo appropriato. In caso contrario, le prestazioni di `intern` rischiano di peggiorare man mano che la piscina si ingrandisce.

Interning come potenziale negazione del vettore di servizio

L'algoritmo di hashcode per le stringhe è ben noto. Se le stringhe interne fornite da utenti malintenzionati o applicazioni, questo potrebbe essere utilizzato come parte di un attacco DoS (denial of service). Se l'agente malintenzionato dispone che tutte le stringhe che fornisce abbiano lo stesso codice hash, ciò potrebbe comportare una tabella hash sbilanciata e prestazioni `O(N)` per `intern` ... dove `N` è il numero di stringhe collise.

(Esistono metodi più semplici / più efficaci per lanciare un attacco DoS contro un servizio, tuttavia questo vettore potrebbe essere utilizzato se l'obiettivo dell'attacco DoS è quello di violare la sicurezza o di eludere le difese DoS di prima linea.)

Trappola - Le piccole letture / scritture sui flussi non bufferizzati sono inefficienti

Considera il seguente codice per copiare un file in un altro:

```
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);
           }
        }
    }
}
```

(Abbiamo deliberato di omettere il normale controllo degli argomenti, la segnalazione degli errori e così via perché non sono pertinenti al *punto* di questo esempio).

Se compili il codice sopra e lo usi per copiare un file enorme, noterai che è molto lento. In effetti, sarà inferiore di almeno un paio di ordini di grandezza rispetto alle utility di copia di file OS standard.

(*Aggiungi misurazioni di prestazioni effettive qui!*)

Il motivo principale per cui l'esempio precedente è lento (nel caso di file di grandi dimensioni) è che sta eseguendo letture a un byte e scritture a byte singolo su flussi di byte senza buffer. Il modo semplice per migliorare le prestazioni è quello di avvolgere gli stream con flussi bufferizzati. Per esempio:

```
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);
           }
        }
    }
}
```

Queste piccole modifiche miglioreranno la velocità di copia dei dati di *almeno* un paio di ordini di grandezza, a seconda dei vari fattori legati alla piattaforma. I wrapper di flusso bufferizzati causano la lettura e la scrittura dei dati in blocchi più grandi. Le istanze hanno entrambi buffer implementati come array di byte.

- Con `is` , i dati vengono letti dal file nel buffer pochi kilobyte alla volta. Quando viene chiamato `read()` , l'implementazione tipicamente restituisce un byte dal buffer. Legge solo dal flusso di input sottostante se il buffer è stato svuotato.
- Il comportamento per `os` è analogo. Chiama su `os.write(int)` scrivere byte singoli nel buffer.

I dati vengono scritti nel flusso di output solo quando il buffer è pieno o quando l' `os` viene svuotato o chiuso.

Che dire dei flussi basati sui personaggi?

Come dovresti sapere, Java I / O fornisce diverse API per leggere e scrivere dati binari e di testo.

- `InputStream` e `OutputStream` sono le API di base per I / O binari basati sul flusso
- `Reader` e `Writer` sono le API di base per l'I / O di testo basato sul flusso.

Per l'I / O di testo, `BufferedReader` e `BufferedWriter` sono gli equivalenti per `BufferedInputStream` e `BufferedOutputStream` .

Perché i flussi bufferizzati fanno la differenza?

La vera ragione per cui gli stream bufferizzati aiutano le prestazioni è il modo in cui un'applicazione comunica con il sistema operativo:

- Il metodo Java in un'applicazione Java o le chiamate di procedure native nelle librerie di runtime native della JVM sono veloci. Solitamente richiedono un paio di istruzioni della macchina e hanno un impatto minimo sulle prestazioni.

- Al contrario, le chiamate di runtime JVM al sistema operativo non sono veloci. Coinvolgono qualcosa conosciuto come "syscall". Il modello tipico per un syscall è il seguente:

    1. Metti gli argomenti di syscall in registri.
    2. Esegui un'istruzione trap SYSENTER.
    3. Il gestore trap passa allo stato privilegiato e modifica i mapping della memoria virtuale. Quindi invia al codice per gestire lo specifico syscall.
    4. Il gestore syscall controlla gli argomenti, facendo attenzione che non gli venga detto di accedere alla memoria che il processo utente non dovrebbe vedere.
    5. Il lavoro specifico di syscall viene eseguito. Nel caso di una `read` syscall, ciò potrebbe comportare:
        1. controllando che ci siano dati da leggere nella posizione corrente del descrittore del file
        2. chiamando il gestore del file system per recuperare i dati richiesti dal disco (o ovunque sia archiviato) nella cache del buffer,
        3. copia dei dati dalla cache del buffer all'indirizzo fornito da JVM
        4. regolazione della posizione del descrittore del file pointstream
    6. Ritorna da syscall. Ciò comporta di nuovo la modifica dei mapping delle macchine virtuali e il passaggio dallo stato privilegiato.

Come puoi immaginare, l'esecuzione di un singolo syscall può contenere migliaia di istruzioni. Conservativamente, *almeno* due ordini di grandezza più lunghi di una normale chiamata di metodo. (Probabilmente tre o più.)

Detto questo, la ragione per cui i flussi bufferizzati fanno una grande differenza è che riducono

drasticamente il numero di syscall. Invece di eseguire una syscall per ogni chiamata `read()` , il flusso di input memorizzato nel buffer legge una grande quantità di dati in un buffer come richiesto. La maggior parte delle chiamate `read()` sul flusso bufferizzato esegue alcuni semplici controlli e restituisce un `byte` letto in precedenza. Ragionamento analogo si applica al caso del flusso di output e anche ai casi del flusso di caratteri.

(Alcune persone pensano che le prestazioni di I / O bufferizzate derivino dalla mancata corrispondenza tra la dimensione della richiesta di lettura e le dimensioni di un blocco del disco, la latenza di rotazione del disco e cose del genere.In realtà, un sistema operativo moderno utilizza una serie di strategie per garantire che il in *genere l'* applicazione non ha bisogno di attendere il disco. Questa non è la vera spiegazione.)

I flussi bufferizzati sono sempre una vittoria?

Non sempre. I flussi bufferizzati sono sicuramente una vittoria se la tua applicazione farà molte letture o scritture "piccole". Tuttavia, se l'applicazione deve eseguire solo letture o scritture di grandi dimensioni su / da un `byte[]` grande `byte[]` o `char[]` , i flussi memorizzati nel buffer non offrono vantaggi reali. In effetti potrebbe anche esserci una (piccola) penalità per le prestazioni.

È questo il modo più veloce per copiare un file in Java?

No, non lo è. Quando si usano le API basate sul flusso di Java per copiare un file, si incorre nel costo di almeno una copia extra dei dati da memoria a memoria. È possibile evitare questo se si utilizzano NIO `ByteBuffer` e le API del `Channel` . (*Aggiungi un link ad un esempio separato qui.*)

Leggi Insidie di Java - Problemi di prestazioni online:
https://riptutorial.com/it/java/topic/5455/insidie---di-java---problemi-di-prestazioni

Capitolo 84: Insidie di Java - Thread e concorrenza

Examples

Trabocchetto: uso errato di wait () / notify ()

I metodi `object.wait()` , `object.notify()` e `object.notifyAll()` sono pensati per essere usati in un modo molto specifico. (vedi http://stackoverflow.com/documentation/java/5409/wait-notify#t=20160811161648303307)

Il problema "Lost Notification"

Un errore principiante comune è quello di chiamare incondizionatamente `object.wait()`

```
private final Object lock = new Object();

public void myConsumer() {
    synchronized (lock) {
        lock.wait();     // DON'T DO THIS!!
    }
    doSomething();
}
```

Il motivo per cui questo è sbagliato è che dipende da qualche altro thread per chiamare `lock.notify()` o `lock.notifyAll()` , ma nulla garantisce che l'altro thread non abbia effettuato quella chiamata *prima* del thread del consumatore chiamato `lock.wait()` .

`lock.notify()` e `lock.notifyAll()` non fanno nulla se qualche altro thread non sta *già* aspettando la notifica. Il thread che chiama `myConsumer()` in questo esempio si bloccherà per sempre se è troppo tardi per ricevere la notifica.

Il bug "Illegal Monitor State"

Se si chiama `wait()` o `notify()` su un oggetto senza tenere il blocco, la JVM genererà `IllegalMonitorStateException` .

```
public void myConsumer() {
    lock.wait();      // throws exception
    consume();
}

public void myProducer() {
    produce();
    lock.notify();    // throws exception
}
```

(Il design per `wait()` / `notify()` richiede che il blocco sia trattenuto perché è necessario per evitare condizioni di gara sistemiche.Se fosse possibile chiamare `wait()` o `notify()` senza bloccare, sarebbe impossibile implementare il caso d'uso principale per questi primitivi: aspettare che si verifichi una condizione).

L'attesa / notifica è troppo bassa

Il modo *migliore* per evitare problemi con `wait()` e `notify()` è di non usarli. La maggior parte dei problemi di sincronizzazione può essere risolta utilizzando gli oggetti di sincronizzazione di livello superiore (code, barriere, semafori, ecc.) Disponibili nel pacchetto `java.utils.concurrent` .

Pitfall - Estensione di 'java.lang.Thread'

Javadoc per la classe `Thread` mostra due modi per definire e utilizzare un thread:

Utilizzando una classe thread personalizzata:

```
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();
```

(Fonte: `java.lang.Thread` javadoc .)

L'approccio alla classe del thread personalizzato funziona, ma presenta alcuni problemi:

1. È scomodo utilizzare `PrimeThread` in un contesto che utilizza un pool di thread classico, un executor o il framework ForkJoin. (Non è impossibile, perché `PrimeThread` implementa indirettamente `Runnable` , ma l'uso di una classe `Thread` personalizzata come `Runnable` è certamente goffo e potrebbe non essere fattibile ... a seconda di altri aspetti della classe.)

2. Vi sono più opportunità per errori in altri metodi. Ad esempio, se hai dichiarato un `PrimeThread.start()` senza delegare a `Thread.start()` , si otterrebbe un "thread" eseguito sul thread corrente.

L'approccio di mettere la logica del thread in un `Runnable` evita questi problemi. Infatti, se si utilizza una classe anonima (Java 1.1 in poi) per implementare il `Runnable` il risultato è più sintetico e più leggibile rispetto agli esempi precedenti.

```
final long minPrime = ...
new Thread(new Runnable() {
    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}.start();
```

Con un'espressione lambda (Java 8 in poi), l'esempio sopra sarebbe diventato ancora più elegante:

```
final long minPrime = ...
new Thread(() -> {
   // compute primes larger than minPrime
    . . .
}).start();
```

Pitfall: troppi thread rendono l'applicazione più lenta.

Un sacco di persone che sono nuove nel multi-threading pensano che l'utilizzo di thread faccia automaticamente andare più veloce un'applicazione. In realtà, è molto più complicato di così. Ma una cosa che possiamo affermare con certezza è che per qualsiasi computer esiste un limite al numero di thread che possono essere eseguiti contemporaneamente:

- Un computer ha un numero fisso di *core* (o *hyperthreads*).
- Un thread Java deve essere *programmato* su un core o hyperthread per poter essere eseguito.
- Se esistono più thread Java eseguibili rispetto ai core / hyperthread (disponibili), alcuni di essi devono attendere.

Questo ci dice che la semplice creazione di un numero sempre maggiore di thread Java *non può* rendere l'applicazione sempre più veloce. Ma ci sono anche altre considerazioni:

- Ogni thread richiede un'area di memoria fuori dallo heap per il suo stack di thread. La tipica dimensione di stack di thread (predefinita) è 512Kbytes o 1Mbytes. Se si dispone di un numero significativo di thread, l'utilizzo della memoria può essere significativo.

- Ogni thread attivo farà riferimento a un numero di oggetti nell'heap. Ciò aumenta il working set di oggetti *raggiungibili* , che ha un impatto sulla garbage collection e sull'utilizzo della memoria fisica.
- I costi generali del passaggio da un thread all'altro non sono banali. Di solito comporta un passaggio nello spazio del kernel del sistema operativo per prendere una decisione sulla schedulazione del thread.
- I sovraccarichi di sincronizzazione dei thread e di segnalazione inter-thread (ad esempio wait (), notify () / notifyAll) *possono essere* significativi.

A seconda dei dettagli dell'applicazione, questi fattori generalmente indicano che esiste un "punto debole" per il numero di thread. Oltre a ciò, l'aggiunta di più thread offre un miglioramento minimo delle prestazioni e può peggiorare le prestazioni.

Se l'applicazione viene creata per ogni nuova attività, un aumento imprevisto del carico di lavoro (ad esempio un alto tasso di richieste) può comportare un comportamento catastrofico.

Un modo migliore per gestire questo è utilizzare il pool di thread limitato di cui è possibile controllare le dimensioni (staticamente o dinamicamente). Quando c'è troppo lavoro da fare, l'applicazione deve accodare le richieste. Se si utilizza un `ExecutorService` , si prenderà cura della gestione del pool di thread e dell'accodamento delle attività.

Pitfall - La creazione del thread è relativamente costosa

Considera questi due micro-benchmark:

Il primo benchmark crea semplicemente, avvia e unisce i thread. `Runnable` del thread non funziona.

```
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
```

In un tipico PC moderno che esegue Linux con Java 8 u101 a 64 bit, questo benchmark mostra un tempo medio impiegato per creare, avviare e unire thread compreso tra 33,6 e 33,9 microsecondi.

Il secondo benchmark fa l'equivalente al primo ma utilizza un `ExecutorService` per inviare compiti e un `Future` rendere alla fine del compito.

```
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
```

Come puoi vedere, le medie sono comprese tra 5,3 e 5,6 microsecondi.

Mentre i tempi effettivi dipenderanno da una varietà di fattori, la differenza tra questi due risultati è significativa. È chiaramente più veloce utilizzare un pool di thread per riciclare i thread piuttosto che creare nuovi thread.

Trappola: le variabili condivise richiedono una sincronizzazione adeguata

Considera questo esempio:

```
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.
    }
}
```

Lo scopo di questo programma è quello di avviare un thread, lasciarlo girare per 1000 millisecondi e quindi farlo arrestare impostando il flag `stop` .

Funzionerà come previsto?

Forse sì forse no.

Un'applicazione non si ferma necessariamente quando viene restituito il metodo `main` . Se è stato creato un altro thread e tale thread non è stato contrassegnato come thread daemon, l'applicazione continuerà ad essere eseguita dopo che il thread principale è terminato. In questo esempio, ciò significa che l'applicazione continuerà a essere in esecuzione fino alla fine del thread secondario. Questo dovrebbe accadere quando `tt.stop` è impostato su `true` .

Ma questo in realtà non è strettamente vero. In realtà, il thread figlio si fermerà dopo aver *osservato* `stop` con il valore `true` . Succederà? Forse sì forse no.

La specifica del linguaggio Java *garantisce* che le letture e le scritture di memoria eseguite in un thread siano visibili a quel thread, secondo l'ordine delle istruzioni nel codice sorgente. Tuttavia, in generale, questo NON è garantito quando un thread scrive e un altro thread (successivamente) legge. Per ottenere una visibilità garantita, deve esserci una catena di successi *, prima delle* relazioni tra una scrittura e una successiva. Nell'esempio sopra, non esiste una catena di questo tipo per l'aggiornamento al flag di `stop` , e quindi non è garantito che il thread secondario vedrà il cambio di `stop` su `true` .

(Nota per gli autori: ci dovrebbe essere un argomento separato sul modello di memoria Java per entrare nei dettagli tecnici profondi).

Come risolviamo il problema?

In questo caso, ci sono due semplici modi per garantire che l'aggiornamento di `stop` sia visibile:

1. Dichiara che `stop` è `volatile` ; vale a dire

   ```
   private volatile boolean stop = false;
   ```

   Per una variabile `volatile` , il JLS specifica che esiste una relazione *before-before* tra una scrittura di un thread e una successiva in un secondo thread.

2. Utilizzare un mutex per sincronizzare come segue:

```
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.
        }
    }
}
```

Oltre a garantire l'esistenza dell'esclusione reciproca, il JLS specifica che esiste una relazione *before-before* tra il rilascio di un mutex in un thread e l'ottenimento dello stesso mutex in un secondo thread.

Ma non è un compito atomico?

Sì!

Tuttavia, questo fatto non significa che gli effetti dell'aggiornamento saranno visibili simultaneamente a tutti i thread. Solo una catena adeguata di relazioni *-prima-prima* lo garantirà.

Perché lo hanno fatto?

I programmatori che eseguono la programmazione multi-thread in Java per la prima volta scoprono che il modello di memoria è impegnativo. I programmi si comportano in modo non intuitivo perché la naturale aspettativa è che le scritture siano visibili in modo uniforme. Quindi, perché i designer Java progettano il modello di memoria in questo modo.

In realtà si tratta di un compromesso tra prestazioni e facilità d'uso (per il programmatore).

Una moderna architettura di computer è composta da più processori (core) con set di registri individuali. La memoria principale è accessibile a tutti i processori oa gruppi di processori. Un'altra proprietà dell'hardware dei computer moderni è che l'accesso ai registri è in genere un ordine di grandezza più veloce all'accesso rispetto all'accesso alla memoria principale. Con l'aumentare del numero di core, è facile vedere che leggere e scrivere sulla memoria principale può diventare il collo di bottiglia principale delle prestazioni del sistema.

Questa mancata corrispondenza viene affrontata implementando uno o più livelli di memorizzazione nella cache della memoria tra i core del processore e la memoria principale. Ogni core ha accesso alle celle di memoria tramite la sua cache. Normalmente, una lettura della memoria principale si verifica solo quando c'è un errore di cache e una scrittura della memoria principale si verifica solo quando una riga della cache deve essere scaricata. Per un'applicazione in cui il set di memoria funzionante di ciascun core si adatta alla sua cache, la velocità di core non è più limitata dalla velocità di memoria / larghezza di banda principale.

Ma questo ci dà un nuovo problema quando più core leggono e scrivono variabili condivise. L'ultima versione di una variabile può trovarsi nella cache di un core. A meno che quel core non svuota la linea della cache nella memoria principale, e altri core invalidi la copia cache delle versioni precedenti, alcuni di essi potrebbero vedere versioni stabili della variabile. Ma se le cache sono state scaricate in memoria ogni volta che c'è una scrittura cache ("nel caso in cui ci fosse una lettura da un altro core) che consumerebbe inutilmente la larghezza di banda della memoria principale.

La soluzione standard utilizzata a livello di set di istruzioni hardware consiste nel fornire istruzioni per l'invalidazione della cache e il write-through della cache e lasciare al compilatore la decisione su quando utilizzarli.

Ritorno a Java. il modello di memoria è progettato in modo che i compilatori Java non siano tenuti a inviare istruzioni di invalidazione della cache e write-through laddove non sono realmente necessarie. Il presupposto è che il programmatore utilizzerà un meccanismo di sincronizzazione appropriato (es. Mutex primitivi, classi di concorrenza `volatile` , di livello superiore e così via) per indicare che ha bisogno di visibilità della memoria. In assenza di una relazione di *happen-before* , i compilatori Java sono liberi di *presupporre* che non siano richieste operazioni di cache (o simili).

Ciò ha notevoli vantaggi in termini di prestazioni per le applicazioni multi-thread, ma il lato negativo è che la scrittura di applicazioni multi-thread corrette non è una questione semplice. Il programmatore *non* deve capire quello che lui o lei sta facendo.

Perché non posso riprodurre questo?

Esistono diverse ragioni per cui problemi di questo tipo sono difficili da riprodurre:

1. Come spiegato sopra, la conseguenza di non gestire correttamente i problemi di visibilità della memoria è in *genere* che l'applicazione compilata non gestisce correttamente le cache di memoria. Tuttavia, come accennato sopra, le cache di memoria spesso si svuotano comunque.

2. Quando si modifica la piattaforma hardware, le caratteristiche delle cache di memoria potrebbero cambiare. Ciò può comportare un comportamento diverso se l'applicazione non si sincronizza correttamente.

3. Potresti star osservando gli effetti della sincronizzazione *fortuita* . Ad esempio, se si aggiungono traceprints, in genere avviene una sincronizzazione dietro le quinte nei flussi I / O che provoca il flush della cache. Quindi l'aggiunta di traceprints *spesso* causa un comportamento diverso dell'applicazione.

4. L'esecuzione di un'applicazione in un debugger fa in modo che venga compilata in modo diverso dal compilatore JIT. Punti di rottura e single stepping esacerbano questo. Questi effetti cambieranno spesso il comportamento di un'applicazione.

Queste cose rendono i bachi dovuti a una sincronizzazione inadeguata particolarmente difficili da risolvere.

Leggi Insidie di Java - Thread e concorrenza online:
https://riptutorial.com/it/java/topic/5567/insidie---di-java---thread-e-concorrenza

Capitolo 85: Insidie di Java - Utilizzo delle eccezioni

introduzione

Diverse attività linguistiche di programmazione Java potrebbero condurre un programma a generare risultati errati nonostante siano stati compilati correttamente. Lo scopo principale di questo argomento è elencare le **insidie più** comuni relative alla **gestione delle eccezioni** e proporre il modo corretto per evitare tali insidie.

Examples

Pitfall - Eccezioni ignoranti o di schiacciamento

Questo esempio riguarda deliberatamente ignorare o "schiacciare" le eccezioni. O per essere più precisi, si tratta di come catturare e gestire un'eccezione in un modo che la ignora. Tuttavia, prima di descrivere come farlo, dovremmo innanzitutto sottolineare che le eccezioni dello schiacciamento non sono generalmente il modo corretto per gestirle.

Le eccezioni vengono solitamente generate (da qualcosa) per notificare ad altre parti del programma che si è verificato un evento significativo (cioè "eccezionale"). Generalmente (sebbene non sempre) un'eccezione significa che qualcosa è andato storto. Se si codifica il programma per eliminare l'eccezione, è probabile che il problema riappaia in un'altra forma. Per peggiorare le cose, quando si schiaccia l'eccezione, si eliminano le informazioni nell'oggetto eccezione e la relativa traccia dello stack associata. È probabile che sia più difficile capire quale fosse la fonte originale del problema.

In pratica, lo schiacciamento delle eccezioni si verifica spesso quando si utilizza la funzione di correzione automatica dell'IDE per "correggere" un errore di compilazione causato da un'eccezione non gestita. Ad esempio, potresti vedere un codice come questo:

```
try {
    inputStream = new FileInputStream("someFile");
} catch (IOException e) {
    /* add exception handling code here */
}
```

Chiaramente, il programmatore ha accettato il suggerimento dell'IDE di far sparire l'errore di compilazione, ma il suggerimento era inappropriato. (Se il file aperto non funziona, probabilmente il programma dovrebbe fare qualcosa al riguardo: con la "correzione" sopra, il programma potrebbe fallire in un secondo momento, ad es. Con `NullPointerException` perché `inputStream` ora è `null`).

Detto questo, ecco un esempio di deliberatamente schiacciare un'eccezione. (Ai fini della discussione, supponiamo di aver determinato che un'interruzione mentre si mostra il selfie è

innocua.) Il commento dice al lettore che abbiamo schiacciato deliberatamente l'eccezione, e perché lo abbiamo fatto.

```
try {
    selfie.show();
} catch (InterruptedException e) {
    // It doesn't matter if showing the selfie is interrupted.
}
```

Un altro modo convenzionale per evidenziare che stiamo *deliberatamente* schiacciando un'eccezione senza dire perché è quello di indicare questo con il nome della variabile di eccezione, come questo:

```
try {
    selfie.show();
} catch (InterruptedException ignored) {  }
```

Alcuni IDE (come IntelliJ IDEA) non visualizzeranno un avviso sul blocco catch vuoto se il nome della variabile è impostato su `ignored` .

Pitfall - Catching Throwable, Exception, Error o RuntimeException

Uno schema di pensiero comune per i programmatori Java inesperti è che le eccezioni sono "un problema" o "un peso" e il modo migliore per affrontarle è catturarle tutte [1] il prima possibile. Questo porta a codice come questo:

```
....
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (Exception ex) {
    System.out.println("Could not open file " + fileName);
}
```

Il codice sopra riportato ha un difetto significativo. La `catch` è in realtà andando a prendere più eccezioni che il programmatore si aspetta. Supponiamo che il valore di `fileName` sia `null` , a causa di un bug altrove nell'applicazione. Ciò farà sì che il costruttore `FileInputStream` lanci una `NullPointerException` . Il gestore prenderà questo e riferirà all'utente:

```
    Could not open file null
```

che è inutile e confuso. Peggio ancora, supponiamo che sia stato il codice "process the input" che ha lanciato l'eccezione inaspettata (selezionata o deselezionata!). Ora l'utente riceverà il messaggio fuorviante per un problema che non si è verificato durante l'apertura del file e potrebbe non essere affatto correlato all'I / O.

La radice del problema è che il programmatore ha codificato un gestore per `Exception` . Questo è quasi sempre un errore:

- L' `Exception` cattura rileverà tutte le eccezioni controllate e anche la maggior parte delle

eccezioni non controllate.
- Catching `RuntimeException` catturerà la maggior parte delle eccezioni non controllate.
- `Error` cattura rileverà eccezioni non controllate che segnalano errori interni JVM. Questi errori non sono generalmente recuperabili e non dovrebbero essere scoperti.
- Catching `Throwable` catturerà tutte le possibili eccezioni.

Il problema con l'acquisizione di un insieme troppo ampio di eccezioni è che il gestore in genere non è in grado di gestirli tutti in modo appropriato. Nel caso `Exception` e così via, è difficile per il programmatore prevedere cosa *potrebbe* essere catturato; cioè cosa aspettarsi.

In generale, la soluzione corretta è quello di affrontare le eccezioni che *vengono* gettati. Ad esempio, puoi prenderli e gestirli in situ:

```
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (FileNotFoundException ex) {
    System.out.println("Could not open file " + fileName);
}
```

oppure puoi dichiararli come `thrown` dal metodo di inclusione.

Ci sono pochissime situazioni in cui è opportuno catturare l' `Exception` . L'unico che si presenta comunemente è qualcosa del genere:

```
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);
    }
}
```

Qui vogliamo sinceramente occuparci di tutte le eccezioni, quindi prendere `Exception` (o anche `Throwable`) è corretto.

---

1 - Conosciuto anche come Pokemon Exception Handling .

Trappola - Lancio Throwable, Exception, Error o RuntimeException

Anche se la cattura delle `Exception` `Throwable` , `Exception` , `Error` e `RuntimeException` è buona, lanciarle è ancora peggio.

Il problema di base è che quando l'applicazione deve gestire le eccezioni, la presenza delle eccezioni di primo livello rende difficile discriminare tra diverse condizioni di errore. Per esempio

```
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
}
```

Il problema è che, poiché abbiamo lanciato un'istanza `Exception` , siamo costretti a prenderlo. Tuttavia, come descritto in un altro esempio, la cattura di `Exception` è errata. In questa situazione, diventa difficile distinguere tra il caso "attesa" di `Exception` che viene generata se `somethingBad` è `true` , e il caso inaspettato in cui abbiamo effettivamente rilevare un'eccezione incontrollato come `NullPointerException` .

Se è consentita la propagazione dell'eccezione di primo livello, ci imbattiamo in altri problemi:

- Ora dobbiamo ricordare tutti i diversi motivi per cui abbiamo lanciato il livello più alto e discriminato / gestito.
- Nel caso di `Exception` e `Throwable` dobbiamo anche aggiungere queste eccezioni alla clausola dei metodi `throws` se vogliamo che l'eccezione si propaghi. Questo è problematico, come descritto di seguito.

In breve, non gettare queste eccezioni. Getta un'eccezione più specifica che descrive più da vicino l'evento eccezionale che è accaduto. Se necessario, definire e utilizzare una classe di eccezioni personalizzata.

Dichiarare Throwable o Exception nei "lanci" di un metodo è problematico.

Si è tentati di sostituire una lunga lista di eccezioni generate in un metodo di `throws` clausola con `Exception` o anche `Throwable. Questa è una cattiva idea:

1. Costringe il chiamante a gestire (o propagare) l' `Exception` .
2. Non possiamo più fare affidamento sul compilatore per dirci quali eccezioni controllate specifiche devono essere gestite.
3. Gestire correttamente l' `Exception` è difficile. È difficile sapere quali sono le reali eccezioni che possono essere scoperte e se non sai cosa potrebbe essere catturato, è difficile sapere quale strategia di ripristino sia appropriata.
4. Handling `Throwable` è ancora più difficile, dal momento che ora devi affrontare anche i potenziali fallimenti che non dovrebbero mai essere recuperati.

Questo consiglio significa che alcuni altri modelli dovrebbero essere evitati. Per esempio:

```
try {
    doSomething();
```

```
} catch (Exception ex) {
    report(ex);
    throw ex;
}
```

Quanto sopra tenta di registrare tutte le eccezioni mentre passano, senza gestirle definitivamente. Sfortunatamente, prima di Java 7, il `throw ex;` dichiarazione ha indotto il compilatore a pensare che qualsiasi `Exception` potesse essere lanciata. Questo potrebbe costringerti a dichiarare il metodo di inclusione come `throws Exception` . Da Java 7 in poi, il compilatore sa che l'insieme di eccezioni che potrebbero essere (ri-generate) è più piccolo.

Pitfall - Catching InterruptedException

Come già sottolineato in altre insidie, prendendo tutte le eccezioni usando

```
try {
    // Some code
} catch (Exception) {
    // Some error handling
}
```

Viene fornito con molti problemi diversi. Ma un problema perticolare è che può portare a deadlock in quanto interrompe il sistema di interrupt durante la scrittura di applicazioni multi-thread.

Se inizi una discussione, di solito devi anche essere in grado di fermarla bruscamente per vari motivi.

```
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
```

`t.interrupt()` genererà un InterruptedException in quel thread, che è inteso per arrestare il thread. Ma cosa succede se il thread ha bisogno di ripulire alcune risorse prima che si fermi completamente? Per questo può prendere l'InterruptedException e fare un po 'di pulizia.

```
 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();
        }
    }
}
```

Ma se hai un'espressione catch-all nel tuo codice, anche l'InterruptedException verrà catturata da esso e l'interruzione non continuerà. Che in questo caso potrebbe portare a un deadlock come thread genitore attende indefinitamente per questo thead per interrompere 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();
        }
    }
}
```

Quindi è meglio catturare le eccezioni individualmente, ma se insisti a usare un catch-all, prendi almeno l'InterruptedException individualmente in anticipo.

```
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 - Utilizzo delle eccezioni per il normale controllo del flusso

C'è un mantra che alcuni esperti di Java sono soliti recitare:

> "Le eccezioni dovrebbero essere utilizzate solo per casi eccezionali."

(Ad esempio: http://programmers.stackexchange.com/questions/184654)

L'essenza di questo è che è una cattiva idea (in Java) utilizzare le eccezioni e la gestione delle eccezioni per implementare il normale controllo del flusso. Ad esempio, confronta questi due modi di trattare un parametro che potrebbe essere nullo.

```
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 "";
    }
}
```

In questo esempio, siamo (in base alla progettazione) trattando il caso in cui la `word` è `null` come se fosse una parola vuota. Le due versioni si occupano di `null` usando convenzionale *se ... else* e o *try ... catch* . Come dovremmo decidere quale versione è migliore?

Il primo criterio è la leggibilità. Sebbene la leggibilità sia difficile da quantificare oggettivamente, la maggior parte dei programmatori concorda sul fatto che il significato essenziale della prima versione sia più facile da comprendere. In effetti, per capire veramente il secondo modulo, è necessario capire che una `NullPointerException` non può essere lanciata dai metodi `Math.min` o `String.substring` .

Il secondo criterio è l'efficienza. Nelle versioni di Java precedenti a Java 8, la seconda versione è significativamente (ordini di grandezza) più lenta della prima versione. In particolare, la costruzione di un oggetto di eccezione comporta l'acquisizione e la registrazione degli stackframes, nel caso sia necessario lo stacktrace.

D'altra parte, ci sono molte situazioni in cui l'uso delle eccezioni è più leggibile, più efficiente e (a volte) più corretto dell'uso del codice condizionale per gestire eventi "eccezionali". In effetti, ci sono rare situazioni in cui è necessario utilizzarle per eventi "non eccezionali"; cioè eventi che si

verificano relativamente frequentemente. Per quest'ultimo, vale la pena esaminare i modi per ridurre i costi generali della creazione di oggetti di eccezione.

Pitfall - Stacktraces eccessivi o inappropriati

Una delle cose più fastidiose che i programmatori possono fare è distribuire le chiamate a `printStackTrace()` attraverso il loro codice.

Il problema è che `printStackTrace()` sta per scrivere lo stacktrace sullo standard output.

- Per un'applicazione destinata agli utenti finali che non sono programmatori Java, uno stacktrace non è informativo al meglio e allarmante nel peggiore dei casi.
- Per un'applicazione lato server, è probabile che nessuno guarderà lo standard output.

Un'idea migliore è quella di non chiamare direttamente `printStackTrace` , o se lo si chiama, farlo in modo che la traccia dello stack sia scritta in un file di registro o in un file di errore piuttosto che nella console dell'utente finale.

Un modo per farlo consiste nell'utilizzare un framework di registrazione e passare l'oggetto di eccezione come parametro dell'evento di registro. Tuttavia, anche la registrazione dell'eccezione può essere dannosa se eseguita in modo non appropriato. Considera quanto segue:

```
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;
    }
}
```

Se l'eccezione è lanciata in `method2` , è probabile che si vedano due copie dello stesso stacktrace nel file di log, corrispondente allo stesso errore.

In breve, registra l'eccezione o rilancia ulteriormente (eventualmente racchiusa in un'altra eccezione). Non fare entrambe le cose.

Pitfall - Direttamente sottoclassando `Throwable`

`Throwable` ha due sottoclassi dirette, `Exception` ed `Error` . Sebbene sia possibile creare una nuova classe che estenda `Throwable` direttamente, ciò è sconsigliabile dal momento che molte

applicazioni presuppongono solo `Exception` ed `Error` .

Più `Throwable` , non vi è alcun vantaggio pratico per la sottoclasse diretta di `Throwable` , poiché la classe risultante è, in effetti, semplicemente un'eccezione controllata. Subclassing `Exception` invece produrrà lo stesso comportamento, ma renderà più chiaro il tuo intento.

Leggi Insidie di Java - Utilizzo delle eccezioni online:
https://riptutorial.com/it/java/topic/5381/insidie---di-java---utilizzo-delle-eccezioni

Capitolo 86: Insidie Java - Sintassi della lingua

introduzione

Diverse attività linguistiche di programmazione Java potrebbero condurre un programma a generare risultati errati nonostante siano stati compilati correttamente. Lo scopo principale di questo argomento è elencare le insidie più comuni con le loro cause e proporre il modo corretto per evitare di cadere in tali problemi.

Osservazioni

Questo argomento riguarda aspetti specifici della sintassi del linguaggio Java che sono soggetti a errori o che non dovrebbero essere utilizzati in determinati modi.

Examples

Pitfall - Ignorare la visibilità del metodo

Persino gli esperti sviluppatori Java tendono a pensare che Java abbia solo tre modificatori di protezione. La lingua ha in realtà quattro! Il livello di visibilità **privato** (ovvero predefinito) del **pacchetto** è spesso dimenticato.

Dovresti prestare attenzione a quali metodi rendi pubblici. I metodi pubblici in un'applicazione sono l'API visibile dell'applicazione. Questo dovrebbe essere il più piccolo e compatto possibile, specialmente se si sta scrivendo una libreria riutilizzabile (vedere anche il principio SOLID). È importante considerare allo stesso modo la visibilità di tutti i metodi e utilizzare solo l'accesso privato protetto o pacchetto se appropriato.

Quando si dichiarano metodi che devono essere **privati** come pubblici, si espongono i dettagli di implementazione interna della classe.

Un corollario di questo è che tu collaudi solo i metodi pubblici della tua classe - in effetti puoi **solo** testare metodi pubblici. È una cattiva pratica aumentare la visibilità dei metodi privati solo per poter eseguire test unitari contro questi metodi. Il test di metodi pubblici che chiamano i metodi con una visibilità più restrittiva dovrebbe essere sufficiente per testare un'intera API. Non devi **mai** espandere la tua API con altri metodi pubblici solo per consentire il test delle unità.

Pitfall - Manca una 'pausa' in un caso 'interruttore'

Questi problemi di Java possono essere molto imbarazzanti e talvolta rimangono sconosciuti fino a quando non vengono eseguiti in produzione. Il comportamento fallito nelle istruzioni switch è spesso utile; tuttavia, mancare una parola chiave "break" quando tale comportamento non è desiderato può portare a risultati disastrosi. Se hai dimenticato di mettere una "interruzione" in

"caso 0" nell'esempio di codice qui sotto, il programma scriverà "Zero" seguito da "Uno", poiché il flusso di controllo qui dentro passerà attraverso l'intera istruzione "switch" fino a raggiunge una "pausa". Per esempio:

```
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");
        }
}
```

Nella maggior parte dei casi, la soluzione più pulita sarebbe utilizzare le interfacce e spostare il codice con un comportamento specifico in implementazioni separate (*composizione sull'ereditarietà*)

Se una dichiarazione di commutazione è inevitabile, si consiglia di documentare le scoperte "previste" se si verificano. In questo modo mostri agli altri sviluppatori che sei consapevole della rottura mancante e che questo è un comportamento previsto.

```
switch(caseIndex) {
    [...]
    case 2:
        System.out.println("Two");
        // fallthrough
    default:
        System.out.println("Default");
```

Trappola - punto e virgola fuori luogo e parentesi graffe mancanti

Questo è un errore che crea vera confusione per i principianti di Java, almeno per la prima volta che lo fanno. Invece di scrivere questo:

```
if (feeling == HAPPY)
    System.out.println("Smile");
else
    System.out.println("Frown");
```

accidentalmente scrivono questo:

```
if (feeling == HAPPY);
    System.out.println("Smile");
else
    System.out.println("Frown");
```

e sono perplessi quando il compilatore Java dice loro che il `else` è fuori posto. Il compilatore Java interpreta quanto sopra come segue:

```
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");
```

In altri casi, non ci saranno errori di compilazione, ma il codice non farà ciò che il programmatore intende fare. Per esempio:

```
for (int i = 0; i < 5; i++);
    System.out.println("Hello");
```

stampa solo "Hello" una volta. Ancora una volta, il punto e virgola spuria significa che il corpo del ciclo `for` è un'istruzione vuota. Ciò significa che la chiamata `println` che segue è incondizionata.

Un'altra variante:

```
for (int i = 0; i < 5; i++);
    System.out.println("The number is " + i);
```

Questo darà un errore "Impossibile trovare il simbolo" per `i` . La presenza del punto e virgola spuria significa che la chiamata `println` sta tentando di utilizzare `i` al di fuori del suo ambito.

In questi esempi, c'è una soluzione semplice: basta eliminare il punto e virgola spuria. Tuttavia, ci sono alcune lezioni più profonde da trarre da questi esempi:

1. Il punto e virgola in Java non è "rumore sintattico". La presenza o l'assenza di un punto e virgola può modificare il significato del tuo programma. Non aggiungerli solo alla fine di ogni riga.

2. Non fidarti del rientro del codice. Nel linguaggio Java, gli spazi bianchi extra all'inizio di una riga vengono ignorati dal compilatore.

3. Utilizzare un penetratore automatico. Tutti gli IDE e molti semplici editor di testo comprendono come indentare correttamente il codice Java.

4. Questa è la lezione più importante. Segui le ultime linee guida in stile Java e metti le parentesi attorno alle istruzioni "then" e "else" e all'istruzione body di un loop. La parentesi aperta (`{`) non dovrebbe essere su una nuova riga.

Se il programmatore ha seguito le regole di stile, l'esempio `if` con un punto e virgola fuori posto sarebbe simile a questo:

```
if (feeling == HAPPY); {
    System.out.println("Smile");
} else {
```

```
    System.out.println("Frown");
}
```

Sembra strano per un occhio esperto. Se indentri automaticamente tale codice, probabilmente sarà simile a questo:

```
if (feeling == HAPPY); {
                           System.out.println("Smile");
                       } else {
                           System.out.println("Frown");
                       }
```

che dovrebbe distinguersi come sbagliato anche per un principiante.

Trappola - Lasciando fuori parentesi: i problemi "dangling if" e "dangling else"

L'ultima versione della guida di stile Oracle Java impone che le istruzioni "then" e "else" in un'istruzione `if` vengano sempre racchiuse tra "parentesi graffe" o "parentesi graffe". Regole simili si applicano ai corpi di varie dichiarazioni di loop.

```
if (a) {            // <- open brace
    doSomething();
    doSomeMore();
}                   // <- close brace
```

Questo non è in realtà richiesto dalla sintassi del linguaggio Java. In effetti, se la parte "allora" di un'istruzione `if` è una singola istruzione, è legale escludere le parentesi

```
if (a)
    doSomething();
```

o anche

```
if (a) doSomething();
```

Tuttavia ci sono pericoli nell'ignorare le regole di stile Java e tralasciare le parentesi. In particolare, si aumenta in modo significativo il rischio che il codice con rientri errati venga interpretato erroneamente.

Il problema "dangling if":

Considera il codice di esempio riportato sopra, riscritto senza parentesi.

```
if (a)
   doSomething();
   doSomeMore();
```

Questo codice *sembra dire* che le chiamate a `doSomething` e `doSomeMore` si verificheranno entrambe

se e solo se `a` è `true` . In effetti, il codice è rientrato in modo errato. La specifica del linguaggio Java che la chiamata `doSomeMore()` è un'istruzione separata che segue l'istruzione `if` . La rientranza corretta è la seguente:

```
if (a)
   doSomething();
doSomeMore();
```

Il problema "dangling else"

Un secondo problema appare quando aggiungiamo `else` al mix. Considera il seguente esempio con parentesi mancanti.

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
else
   doZ();
```

Il codice sopra *sembra dire* che `doZ` sarà chiamato quando `a` è `false` . In realtà, il rientro è errato ancora una volta. Il rientro corretto per il codice è:

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
   else
      doZ();
```

Se il codice è stato scritto in base alle regole di stile Java, sarebbe in effetti simile a questo:

```
if (a) {
   if (b) {
      doX();
   } else if (c) {
      doY();
   } else {
      doZ();
   }
}
```

Per illustrare il motivo per cui è meglio, supponiamo di aver accidentalmente indentato il codice. Potresti finire con qualcosa del genere:

```
if (a) {                          if (a) {
   if (b) {                           if (b) {
      doX();                             doX();
   } else if (c) {                    } else if (c) {
      doY();                             doY();
} else {                          } else {
```

```
   doZ();                              doZ();
}                                      }
}                                   }
```

Ma in entrambi i casi, il codice errato "sembra sbagliato" per l'occhio di un esperto programmatore Java.

Trappola - Sovraccarico invece di scavalcare

Considera il seguente esempio:

```
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();
    }
}
```

Questo codice non si comporterà come previsto. Il problema è che i metodi `equals` e `hashcode` per `Person` non sovrascrivono i metodi standard definiti da `Object` .

- Il metodo `equals` ha la firma sbagliata. Dovrebbe essere dichiarato come `equals(Object)` non `equals(String)` .
- Il metodo `hashcode` ha il nome sbagliato. Dovrebbe essere `hashCode()` (notare la maiuscola **C**).

Questi errori significano che abbiamo dichiarato sovraccarichi accidentali, e questi non saranno usati se la `Person` viene usata in un contesto polimorfico.

Tuttavia, c'è un modo semplice per gestire questo (da Java 5 in poi). Usa l'annotazione `@Override` ogni volta che *intendi che il* tuo metodo sia un override:

Java SE 5

```
public final class Person {
    ...

    @Override
```

```
    public boolean equals(String other) {
        ....
    }

    @Override
    public hashcode() {
        ....
    }
}
```

Quando aggiungiamo un `@Override` un'annotazione a una dichiarazione di metodo, il compilatore verificherà che il metodo *non* sovrascrivere (o implementare) un metodo dichiarato in una superclasse o interfaccia. Quindi nell'esempio sopra, il compilatore ci darà due errori di compilazione, che dovrebbero essere sufficienti per avvisarci dell'errore.

Pitfall - Ottali letterali

Considera il seguente frammento di codice:

```
// 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 Java potrebbe essere sorpreso di sapere che il programma di cui sopra stampa la risposta sbagliata. In realtà stampa la somma dei numeri da 1 a 8.

Il motivo è che un intero letterale che inizia con la cifra zero ('0') viene interpretato dal compilatore Java come un letterale ottale, non come un valore letterale decimale come ci si potrebbe aspettare. Quindi, `010` è il numero ottale 10, che è 8 in decimale.

Pitfall - Dichiarare classi con lo stesso nome delle classi standard

A volte, i programmatori che sono nuovi in Java commettono l'errore di definire una classe con un nome uguale a una classe ampiamente utilizzata. Per esempio:

```
package com.example;

/**
 * My string utilities
 */
public class String {
    ....
}
```

Poi si chiedono perché ottengono errori imprevisti. Per esempio:

```
package com.example;

public class Test {
```

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

Se si compila e quindi si tenta di eseguire le classi precedenti, si otterrà un errore:

```
$ 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
```

Qualcuno che guarda il codice per la classe `Test` vedrebbe la dichiarazione di `main` e guarderà la sua firma e si chiederà di cosa si lamenta il comando `java` . Ma in realtà, il comando `java` sta dicendo la verità.

Quando dichiariamo una versione di `String` nello stesso pacchetto di `Test` , questa versione ha la precedenza sull'importazione automatica di `java.lang.String` . Pertanto, la firma del metodo `Test.main` è in realtà

```
void main(com.example.String[] args)
```

invece di

```
void main(java.lang.String[] args)
```

e il `java` comando *non* riconoscerà come metodo entry point.

Lezione: non definire classi con lo stesso nome delle classi esistenti in `java.lang` o altre classi comunemente utilizzate nella libreria Java SE. Se lo fai, ti stai preparando per tutti i tipi di errori oscuri.

Pitfall - Usando '==' per testare un booleano

A volte un nuovo programmatore Java scriverà un codice come questo:

```
public void check(boolean ok) {
    if (ok == true) {           // Note 'ok == true'
        System.out.println("It is OK");
    }
}
```

Un programmatore esperto vedrebbe che è goffo e desidera riscriverlo come:

```
public void check(boolean ok) {
    if (ok) {
       System.out.println("It is OK");
    }
}
```

Tuttavia, c'è più sbagliato con `ok == true` rispetto alla semplice goffaggine. Considera questa variazione:

```
public void check(boolean ok) {
    if (ok = true) {           // Oooops!
        System.out.println("It is OK");
    }
}
```

Qui il programmatore ha scritto male `==` as `=` ... e ora il codice ha un bug sottile. L'espressione `x = true` assegna incondizionatamente `true` a `x` e quindi valuta `true` . In altre parole, il metodo di `check` ora stamperà "It is OK" indipendentemente dal parametro.

La lezione qui è di uscire dall'abitudine di usare `== false` e `== true` . Oltre ad essere prolissi, rendono la tua codifica più incline agli errori.

Nota: una possibile alternativa a `ok == true` che evita il trabocchetto consiste nell'utilizzare le condizioni Yoda ; vale a dire mettere il letterale sul lato sinistro dell'operatore relazionale, come in `true == ok` . Funziona, ma la maggior parte dei programmatori probabilmente concorderebbe sul fatto che le condizioni di Yoda sembrano strane. Certamente `ok` (o `!ok`) è più conciso e più naturale.

Pitfall: le importazioni con caratteri jolly possono rendere fragile il tuo codice

Considera il seguente esempio parziale:

```
import com.example.somelib.*;
import com.acme.otherlib.*;

public class Test {
    private Context x = new Context();   // from com.example.somelib
    ...
}
```

Supponiamo che quando hai sviluppato per la prima volta il codice contro la versione 1.0 di `somelib` e la versione 1.0 di `otherlib` . Quindi, in un secondo momento, è necessario aggiornare le dipendenze a versioni successive e si decide di utilizzare `otherlib` versione 2.0. Supponiamo inoltre che una delle modifiche apportate ad `otherlib` tra 1.0 e 2.0 sia stata l'aggiunta di una classe `Context` .

Ora quando ricomponi `Test` , riceverai un errore di compilazione che ti dice che `Context` è un'importazione ambigua.

Se hai familiarità con il codebase, probabilmente questo è solo un piccolo inconveniente. In caso contrario, hai del lavoro da fare per affrontare questo problema, qui e potenzialmente altrove.

Il problema qui è l'importazione di caratteri jolly. Da un lato, l'uso di caratteri jolly può rendere le tue lezioni di qualche riga più corte. D'altro canto:

- Modifiche verso l'alto compatibili con altre parti del codice base, con librerie standard Java o con librerie di terze parti possono portare a errori di compilazione.
- La leggibilità soffre. A meno che non si stia utilizzando un IDE, è difficile determinare quale delle importazioni di caratteri jolly stia utilizzando una classe denominata.

La lezione è che è una cattiva idea utilizzare le importazioni di caratteri jolly nel codice che deve essere longevo. Le importazioni specifiche (non jolly) non sono molto difficili da mantenere se si utilizza un IDE e lo sforzo è utile.

Pitfall: utilizzo di 'assert' per argomento o validazione dell'input dell'utente

Una domanda che occasionalmente su StackOverflow è se sia appropriato utilizzare `assert` per convalidare gli argomenti forniti a un metodo o anche gli input forniti dall'utente.

La risposta semplice è che non è appropriato.

Migliori alternative includono:

- Lanciare un IllegalArgumentException usando un codice personalizzato.
- Utilizzo dei metodi `Preconditions` disponibili nella libreria Google Guava.
- Utilizzo dei metodi `Validate` disponibili nella libreria di Apache Commons Lang3.

Questo è ciò che la specifica del linguaggio Java (JLS 14.10, per Java 8) consiglia su questo argomento:

> In genere, il controllo delle asserzioni è abilitato durante lo sviluppo e il test del programma e disabilitato per la distribuzione, per migliorare le prestazioni.
>
> Poiché le asserzioni possono essere disabilitate, i programmi non devono presumere che le espressioni contenute nelle asserzioni saranno valutate. Pertanto, queste espressioni booleane dovrebbero generalmente essere prive di effetti collaterali. La valutazione di tale espressione booleana non dovrebbe influire su nessuno stato visibile dopo aver completato la valutazione. Non è illegale che un'espressione booleana contenuta in un'asserzione abbia un effetto collaterale, ma in genere è inappropriata, in quanto potrebbe causare una variazione del comportamento del programma a seconda che le asserzioni fossero abilitate o disabilitate.
>
> Alla luce di ciò, le asserzioni non dovrebbero essere usate per il controllo degli argomenti nei metodi pubblici. Il controllo degli argomenti è in genere parte del contratto di un metodo e questo contratto deve essere mantenuto se le asserzioni sono abilitate o disabilitate.
>
> Un problema secondario con l'utilizzo delle asserzioni per il controllo degli argomenti è che gli argomenti errati dovrebbero comportare un'eccezione run-time appropriata (come `IllegalArgumentException` , `ArrayIndexOutOfBoundsException` o `NullPointerException`). Un errore di asserzione non genererà un'eccezione appropriata. Ancora una volta, non è illegale usare asserzioni per il controllo degli argomenti sui metodi pubblici, ma è generalmente inappropriato. È inteso che `AssertionError` non venga mai rilevato, ma è

possibile farlo, quindi le regole per le istruzioni try dovrebbero trattare le asserzioni che appaiono in un blocco try analogamente al trattamento corrente delle istruzioni throw.

Trappola di oggetti Null Auto-Unboxing in Primitive

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

        // example:
        Boolean ignore = null;
        if (ignore == false) {
            System.out.println("Do not ignore!");
        }
    }
}
```

Il trabocchetto qui è che `null` è paragonato a `false` . Dal momento che stiamo confrontando un `boolean` primitivo con un `Boolean` , Java tenta di *unbox* `Object` `Boolean` in un equivalente primitivo, pronto per il confronto. Tuttavia, poiché tale valore è `null` , viene generata una `NullPointerException` .

Java non è in grado di confrontare i tipi primitivi con valori `null` , il che causa una `NullPointerException` in fase di runtime. Considera il caso primitivo della condizione `false == null` ; questo genererebbe un errore `incomparable types: int and <null>` *tempo di compilazione* `incomparable types: int and <null>` .

Leggi Insidie Java - Sintassi della lingua online: https://riptutorial.com/it/java/topic/5382/insidie---java---sintassi-della-lingua

Capitolo 87: Installazione di Java (Standard Edition)

introduzione

Questa pagina di documentazione dà accesso alle istruzioni per l'installazione di `java standard edition` su `Windows` , `Linux` e `macOS` .

Examples

Impostazione% PATH% e% JAVA_HOME% dopo l'installazione su Windows

ipotesi:

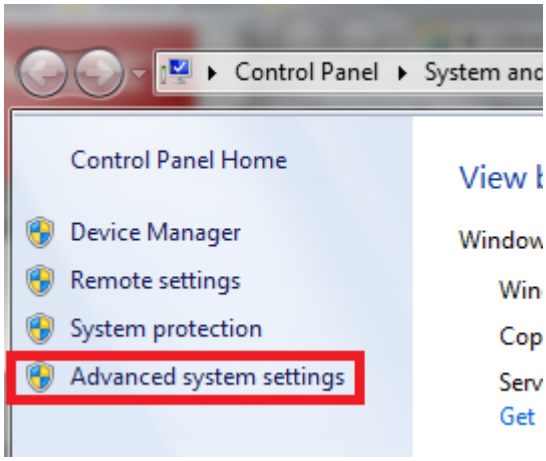
- È stato installato un JDK Oracle.
- Il JDK è stato installato nella directory predefinita.

Passaggi di installazione

1. Apri Windows Explorer.

2. Nel pannello di navigazione, fare clic con il pulsante destro del mouse su *Questo PC* (o *Computer* per versioni precedenti di Windows). C'è un modo più breve senza usare l'explorer nelle effettive versioni di Windows: basta premere `Win` + `Pause`

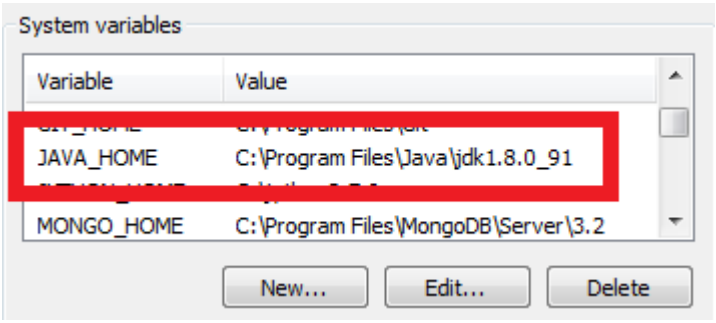
3. Nella finestra del Pannello di controllo appena aperta, fai clic con il pulsante sinistro del mouse su *Impostazioni di sistema avanzate* che dovrebbero trovarsi nell'angolo in alto a sinistra. Questo aprirà la finestra *Proprietà* del *sistema* .

   In alternativa, digita `SystemPropertiesAdvanced` (senza distinzione tra maiuscole e minuscole) in *Run* (`Win` + `R`), quindi premi `Invio` .

4. Nella scheda *Avanzate* di *Proprietà* del *sistema* selezionare il pulsante `Variabili d'ambiente ...` nell'angolo in basso a destra della finestra.

5. Aggiungi una **nuova variabile di sistema** facendo clic sul pulsante `Nuovo ...` in *Variabili di sistema* con il nome `JAVA_HOME` e il cui valore è il percorso della directory in cui è stato installato JDK. Dopo aver inserito questi valori, premere `OK` .



6. Scorri verso il basso l'elenco delle *variabili* di *sistema* e seleziona la variabile `Path` .

7. **ATTENZIONE:** Windows si basa su `Path` per trovare programmi importanti. Se tutto o parte di esso viene rimosso, Windows potrebbe non essere in grado di funzionare correttamente. Deve essere modificato per consentire a Windows di eseguire il JDK. Con questo in mente, fai clic sul pulsante "Modifica ..." con la variabile `Path` selezionata. Aggiungi `%JAVA_HOME%\bin;` all'inizio della variabile `Path` .

È meglio accodare all'inizio della riga perché il software di Oracle utilizzato per registrare la propria versione di Java in `Path` : in questo modo la versione verrà ignorata se si verifica dopo la dichiarazione di Oracle.

Controlla il tuo lavoro

1. Aprire il prompt dei comandi facendo clic su Start, quindi digitando `cmd` e premendo `Enter` .
2. Immettere `javac -version` nel prompt. Se ha avuto successo, la versione di JDK verrà stampata sullo schermo.

Nota: se devi riprovare, chiudi il messaggio prima di controllare il tuo lavoro. Questo costringerà Windows a ottenere la nuova versione di `Path` .

Selezione di una versione di Java SE appropriata

Ci sono state molte versioni di Java dal rilascio originale di Java 1.0 nel 1995. (Fare riferimento alla cronologia delle versioni di Java per un sommario.) Tuttavia la maggior parte delle versioni ha superato le date ufficiali di fine vita. Ciò significa che il fornitore (in genere Oracle ora) ha cessato il nuovo sviluppo per il rilascio e non fornisce più patch pubbliche / gratuite per eventuali bug o problemi di sicurezza. (Le versioni di patch private sono generalmente disponibili per le persone / organizzazioni con un contratto di supporto, contattare l'ufficio vendite del fornitore.)

In generale, la versione Java SE raccomandata per l'uso sarà l'ultimo aggiornamento per l'ultima

versione pubblica. Attualmente, questo significa l'ultima versione disponibile di Java 8. Java 9 è previsto per il rilascio pubblico nel 2017. (Java 7 ha superato End of Life e l'ultima versione pubblica è stata rilasciata ad aprile 2015. Java 7 e versioni precedenti non sono raccomandati).

Questa raccomandazione si applica a tutti i nuovi sviluppi Java e a chiunque stia imparando Java. Si applica anche alle persone che vogliono solo eseguire il software Java fornito da una terza parte. In generale, il codice Java ben scritto funzionerà su una versione più recente di Java. (Ma controlla le note di rilascio del software e contatta l'autore / fornitore / fornitore in caso di dubbi.)

Se si sta lavorando su una base di codice Java precedente, si consiglia di assicurarsi che il codice venga eseguito sull'ultima versione di Java. Decidere quando iniziare a utilizzare le funzionalità delle nuove versioni di Java è più difficile, in quanto ciò avrà un impatto sulla capacità di supportare i clienti che non sono in grado o non desiderano l'installazione di Java.

Rilascio Java e denominazione delle versioni

La denominazione delle versioni Java è un po 'confusa. Esistono in realtà due sistemi di denominazione e numerazione, come mostrato in questa tabella:

Versione JDK	Nome di 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 rebranded 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 (non ancora rilasciato)

1 - Sembra che Oracle intenda interrompere la precedente pratica di utilizzare uno schema di "numero di versione semantico" nelle stringhe di versione di Java. Resta da vedere se seguiranno questo.

La "SE" nei nomi di marketing si riferisce alla Standard Edition. Questa è la versione di base per l'esecuzione di Java su molti laptop, PC e server (ad eccezione di Android).

Ci sono altre due edizioni ufficiali di Java: "Java ME" è la Micro Edition e "Java EE" è l'Enterprise Edition. L'aroma Android di Java è anche significativamente diverso da Java SE. Java ME, Java EE e Android Java non rientrano nell'ambito di questo argomento.

Il numero di versione completo per una versione Java è simile al seguente:

```
1.8.0_101-b13
```

Questo dice JDK 1.8.0, Update 101, Build # 13. Oracle si riferisce a questo nelle note di rilascio come:

```
Java™ SE Development Kit 8, Update 101 (JDK 8u101)
```

Il numero di aggiornamento è importante: Oracle rilascia regolarmente aggiornamenti a una versione principale con patch di sicurezza, correzioni di bug e (in alcuni casi) nuove funzionalità. Il numero di build è in genere irrilevante. Si noti che Java 8 e Java 1.8 si *riferiscono alla stessa cosa* ; Java 8 è solo il nome "marketing" per Java 1.8.

Cosa mi serve per lo sviluppo Java

Un'installazione JDK e un editor di testo sono il minimo indispensabile per lo sviluppo Java. (È bello avere un editor di testo che possa fare l'evidenziazione della sintassi Java, ma puoi fare a meno).

Tuttavia, per un lavoro di sviluppo serio, si consiglia di utilizzare anche quanto segue:

- Un IDE Java come Eclipse, Intellij IDEA o NetBeans
- Uno strumento di compilazione Java come Ant, Gradle o Maven
- Un sistema di controllo della versione per la gestione della base di codice (con backup appropriati e replica off-site)
- Strumenti di test e strumenti CI (integrazione continua)

Installazione di un JDK Java su Linux

Utilizzando il gestore di pacchetti

Le versioni JDK e / o JRE per OpenJDK o Oracle possono essere installate utilizzando il gestore pacchetti sulla maggior parte della distribuzione Linux mainstream. (Le scelte a tua disposizione dipenderanno dalla distribuzione.)

Come regola generale, la procedura è di aprire la finestra del terminale ed eseguire i comandi mostrati di seguito. (Si presume che tu abbia un accesso sufficiente per eseguire comandi come utente "root" ... che è ciò che fa il comando `sudo` . Se non lo fai, per favore parla con gli amministratori del tuo sistema.)

L'uso del gestore di pacchetti è consigliato perché (in genere) rende più semplice mantenere aggiornata l'installazione di Java.

distribuzioni Linux basate su Debian `apt-get` (Ubuntu, ecc.)

Le seguenti istruzioni installeranno Oracle Java 8:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Nota: per impostare automaticamente le variabili di ambiente Java 8, è possibile installare il seguente pacchetto:

```
$ sudo apt-get install oracle-java8-set-default
```

Creare un file `.deb`

Se preferisci creare il file `.deb` da solo dal file `.tar.gz` scaricato da Oracle, `./<jdk>.tar.gz` come segue (supponendo che tu abbia scaricato il file `.tar.gz` in `./<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 : ciò presuppone che l'input sia fornito come un file ".tar.gz".

`slackpkg` , distribuzioni Linux basate su Slackware

```
sudo slapt-get install default-jdk
```

`yum` , RedHat, CentOS, ecc

```
sudo yum install java-1.8.0-openjdk-devel.x86_64
```

`dnf` , Fedora

Nelle recenti versioni di Fedora, `yum` è stato sostituito da `dnf` .

```
sudo dnf install java-1.8.0-openjdk-devel.x86_64
```

Nelle recenti versioni di Fedora, non ci sono pacchetti per l'installazione di Java 7 e precedenti.

`pacman` , distribuzioni Linux basate su Arch

```
sudo pacman -S jdk8-openjdk
```

L'uso di `sudo` non è richiesto se sei in esecuzione come utente root.

Gentoo Linux

La guida di Gentoo Java è gestita dal team di Gentoo Java e mantiene una pagina wiki aggiornata che include i pacchetti di portage corretti e le flag USE necessarie.

Installazione di Oracle JDK su Red Hat, CentOS, Fedora

Installazione di JDK da un file `tar.gz` JDK o JRE Oracle.

1. Scarica il file dell'archivio Oracle appropriato ("tar.gz") per la versione desiderata dal sito di download di Oracle Java .
2. Cambia la directory nel punto in cui vuoi installare l'installazione;
3. Decomprimere il file di archivio; per esempio

```
tar xzvf jdk-8u67-linux-x64.tar.gz
```

Installazione da un file RPM Java Oracle.

1. Recupera il file RPM richiesto per la versione desiderata dal sito di download di Oracle Java .
2. Installa usando il comando `rpm` . Per esempio:

```
$ sudo rpm -ivh jdk-8u67-linux-x644.rpm
```

Installazione di un JDK o JRE Java su Windows

Solo JDK e JRE Oracle sono disponibili per piattaforme Windows. La procedura di installazione è semplice:

1. Visita la pagina Download di Oracle Java:
2. Fare clic sul pulsante JDK, sul pulsante JRE o sul pulsante Server JRE. Nota che per sviluppare usando Java hai bisogno di JDK. Per conoscere la differenza tra JDK e JRE, vedere qui
3. Scorri verso il basso fino alla versione che desideri scaricare. (In generale, è consigliato il più recente.)
4. Seleziona il pulsante di opzione "Accetta contratto di licenza".
5. Scarica il programma di installazione Windows x86 (32 bit) o Windows x64 (64 bit).
6. Esegui il programma di installazione ... nel modo normale per la tua versione di Windows.

Un modo alternativo per installare Java su Windows utilizzando il prompt dei comandi è utilizzare Chocolately:

1. Installa Chocolately da https://chocolatey.org/
2. Apri un'istanza di cmd, ad esempio premi `Win` + `R` e poi digita "cmd" nella finestra "Esegui" seguita da un invio.
3. Nell'istanza di cmd, eseguire il seguente comando per scaricare e installare un JDK Java 8:

```
C:\> choco install jdk8
```

Salire e correre con versioni portatili

Ci sono casi in cui si potrebbe voler installare JDK / JRE su un sistema con privilegi limitati come una VM o si potrebbe voler installare e utilizzare più versioni o architetture (x64 / x86) di JDK / JRE. I passaggi rimangono gli stessi fino al punto in cui si scarica l'installer (.EXE). I passaggi successivi sono i seguenti (i passaggi sono applicabili per JDK / JRE 7 e versioni successive, per le versioni precedenti sono leggermente diversi nei nomi di cartelle e file):

1. Spostare il file in una posizione appropriata in cui si desidera che i file binari Java risiedano in modo permanente.

2. Installa 7-Zip o la sua versione portatile se disponi di privilegi limitati.

3. Con 7-Zip, estrai i file dal programma di installazione di Java EXE nella posizione.

4. Apri il prompt dei comandi tenendo `Shift` e il `Shift Right-Click` nella cartella in explorer o naviga verso quella posizione da qualsiasi posizione.

5. Passare alla cartella appena creata. Supponiamo che il nome della cartella sia `jdk-7u25-windows-x64` . Quindi scrivi `cd jdk-7u25-windows-x64` . Quindi, digita i seguenti comandi nell'ordine:

   ```
   cd .rsrc\JAVA_CAB10

   extrac32 111
   ```

6. Questo creerà un file `tools.zip` in quella posizione. Estrai il file `tools.zip` con 7-Zip in modo che i file al suo interno siano ora creati sotto gli `tools` nella stessa directory.

7. Ora esegui questi comandi sul prompt dei comandi già aperto:

   ```
   cd tools

   for /r %x in (*.pack) do .\bin\unpack200 -r "%x" "%~dx%~px%~nx.jar"
   ```

8. Attendere il completamento del comando. Copia il contenuto degli `tools` nella posizione in cui desideri che i tuoi binari siano.

In questo modo, è possibile installare qualsiasi versione di JDK / JRE che è necessario installare contemporaneamente.

Post originale: http://stackoverflow.com/a/6571736/1448252

Installazione di un JDK Java su macOS

Oracle Java 7 e Java 8

Java 7 e Java 8 per macOS sono disponibili da Oracle. Questa pagina Oracle risponde a molte domande su Java per Mac. Si noti che Java 7 prima del 7u25 è stato disabilitato da Apple per motivi di sicurezza.

In generale, Oracle Java (versione 7 e successive) richiede un Mac basato su Intel con macOS

10.7.3 o versioni successive.

Installazione di Oracle Java

I programmi di installazione Java 7 e 8 JDK e JRE per macOS possono essere scaricati dal sito Web di Oracle:

- Download di Java 8 - Java SE
- Java 7 - Oracle Java Archive.

Dopo aver scaricato il pacchetto pertinente, fare doppio clic sul pacchetto e seguire la normale procedura di installazione. Un JDK dovrebbe essere installato qui:

```
/Library/Java/JavaVirtualMachines/<version>.jdk/Contents/Home
```

dove corrisponde alla versione installata.

Commutazione della riga di comando

Quando Java è installato, la versione installata viene automaticamente impostata come predefinita. Per passare da una modalità all'altra, utilizzare:

```
export JAVA_HOME=/usr/libexec/java_home -v 1.6 #Or 1.7 or 1.8
```

Le seguenti funzioni possono essere aggiunte a `~/.bash_profile` (se si utilizza la shell Bash predefinita) per facilità d'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 su macOS

Nelle versioni precedenti di macOS (10.11 El Capitan e precedenti), il rilascio di Apple 6 di Apple è

preinstallato. Se installato, può essere trovato in questa posizione:

```
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

Si noti che Java 6 ha superato la fine del proprio ciclo di vita molto tempo fa, quindi è consigliabile eseguire l'aggiornamento a una versione più recente. Sono disponibili ulteriori informazioni sulla reinstallazione di Apple Java 6 sul sito Web Oracle.

Configurare e cambiare le versioni di Java su Linux usando alternative

Utilizzo di alternative

Molte distribuzioni Linux usano il comando delle `alternatives` per passare tra diverse versioni di un comando. È possibile utilizzarlo per passare tra diverse versioni di Java installate su una macchina.

1. In una shell di comandi, imposta $ JDK sul percorso di un JDK appena installato; per esempio

   ```
   $ JDK=/Data/jdk1.8.0_67
   ```

2. Utilizzare le `alternatives --install` per aggiungere i comandi nell'SDK Java alle alternative:

   ```
   $ 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
   ```

   E così via.

Ora puoi passare da una versione all'altra di un comando Java come segue:

```
$ 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
$
```

Per ulteriori informazioni sull'uso di `alternatives` , fare riferimento alla voce di manuale alternative (8) .

Installazioni basate su Arch

Le installazioni basate su Arch Linux arrivano con il comando `archlinux-java` per passare alle versioni java.

Elenco degli ambienti installati

```
$ archlinux-java status
Available Java environments:
  java-7-openjdk (default)
  java-8-openjdk/jre
```

Cambiare ambiente corrente

```
# archlinux-java set <JAVA_ENV_NAME>
```

Per esempio:

```
# archlinux-java set java-8-openjdk/jre
```

Maggiori informazioni possono essere trovate su Arch Linux Wiki

Controllo e configurazione post-installazione su Linux

Dopo aver installato un SDK Java, è consigliabile verificare che sia pronto per l'uso. Puoi farlo eseguendo questi due comandi, usando il tuo normale account utente:

```
$ java -version
$ javac -version
```

Questi comandi stampano le informazioni sulla versione per JRE e JDK (rispettivamente) che si trovano nel percorso di ricerca dei comandi della shell. Cerca la stringa di versione JDK / JRE.

- Se uno dei due comandi non riesce, dicendo "comando non trovato", il JRE o JDK non si trova affatto nel percorso di ricerca; vai a **Configurazione del PERCORSO direttamente** sotto.
- Se uno dei due comandi sopra mostra una stringa di versione diversa da quella che ti aspettavi, allora il tuo percorso di ricerca o il sistema delle "alternative" devono essere aggiustati; vai a **controllare le alternative**
- Se vengono visualizzate le stringhe della versione corretta, hai quasi finito; saltare a **Controllare JAVA_HOME**

Configurazione del PERCORSO direttamente

Se al momento non ci sono `java` o `javac` nel percorso di ricerca, la soluzione semplice è aggiungerla al tuo percorso di ricerca.

Per prima cosa, trova dove hai installato Java; vedi **dove è stato installato Java?** sotto se hai dei

dubbi.

Quindi, assumendo che `bash` sia la shell dei comandi, usa un editor di testo per aggiungere le seguenti righe alla fine di `~/.bash_profile` o `~/.bashrc` (Se usi Bash come shell).

```
JAVA_HOME=<installation directory>
PATH=$JAVA_HOME/bin:$PATH

export JAVA_HOME
export PATH
```

... sostituendo `<installation directory>` con il percorso per la directory di installazione Java. Si noti che quanto sopra presuppone che la directory di installazione contenga una directory `bin` e che la directory `bin` contenga i comandi `java` e `javac` che si sta tentando di utilizzare.

Quindi, fonte il file che hai appena modificato, in modo che le variabili di ambiente per la tua shell corrente vengano aggiornate.

```
$ source ~/.bash_profile
```

Quindi, ripetere i controlli della versione di `java` e `javac` . Se ci sono ancora problemi, usa `which java` e `which javac` per verificare che tu abbia aggiornato correttamente le variabili d'ambiente.

Infine, disconnettiti e accedi di nuovo in modo che le variabili di ambiente aggiornate ptopagate su tutte le tue shell. Ora dovresti aver finito.

Controllo delle alternative

Se `java -version` o `javac -version` funzionavano ma davano un numero di versione inaspettato, è necessario controllare da dove provengono i comandi. Usa `which` e `ls -l` per scoprirlo come segue:

```
$ ls -l `which java`
```

Se l'output è simile a questo,:

```
lrwxrwxrwx. 1 root root 22 Jul 30 22:18 /usr/bin/java -> /etc/alternatives/java
```

quindi viene utilizzata la commutazione della versione `alternatives` . È necessario decidere se continuare a utilizzarlo o semplicemente sovrascriverlo impostando direttamente il `PATH` .

- Configurare e cambiare versioni Java su Linux usando alternative
- Vedi "Configurazione del PERCORSO direttamente" sopra.

Dove è stato installato Java?

Java può essere installato in una varietà di luoghi, a seconda del metodo di installazione.

- Gli RPM Oracle inseriscono l'installazione Java in "/ usr / java".
- Su Fedora, il percorso predefinito è "/ usr / lib / jvm".
- Se Java è stato installato manualmente dai file ZIP o JAR, l'installazione potrebbe essere ovunque.

Se stai trovando difficilmente la directory di installazione, ti suggeriamo di utilizzare `find` (o `slocate`) per trovare il comando. Per esempio:

```
$ find / -name java -type f 2> /dev/null
```

Questo ti dà il percorso per tutti i file chiamati `java` sul tuo sistema. (Il reindirizzamento dell'errore standard a "/ dev / null" sopprime i messaggi su file e directory a cui non si può accedere).

Installare oracle java su Linux con l'ultimo file tar

Seguire i passaggi seguenti per installare Oracle JDK dall'ultimo file tar:

1. Scarica l'ultimo file tar da qui - L'ultima versione aggiornata è Java SE Development Kit 8u112.

2. Hai bisogno di privilegi sudo:

   ```
   sudo su
   ```

3. Creare una directory per l'installazione di jdk:

   ```
   mkdir /opt/jdk
   ```

4. Estratto tar scaricato in esso:

   ```
   tar -zxf jdk-8u5-linux-x64.tar.gz -C /opt/jdk
   ```

5. Verifica se i file sono stati estratti:

   ```
   ls /opt/jdk
   ```

6. Impostazione di Oracle JDK come JVM predefinita:

   ```
   update-alternatives --install /usr/bin/java java /opt/jdk/jdk1.8.0_05/bin/java 100
   ```

   e

   ```
   update-alternatives --install /usr/bin/javac javac /opt/jdk/jdk1.8.0_05/bin/javac 100
   ```

7. Controlla la versione di Java:

   java -version

Uscita prevista:

```
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)
```

Leggi Installazione di Java (Standard Edition) online:
https://riptutorial.com/it/java/topic/4754/installazione-di-java--standard-edition-

Capitolo 88: interfacce

introduzione

Un'interfaccia è un tipo di riferimento, simile a una classe, che può essere dichiarata tramite `interface` parola chiave. Le interfacce possono contenere solo costanti, firme dei metodi, metodi predefiniti, metodi statici e tipi nidificati. I corpi dei metodi esistono solo per metodi predefiniti e metodi statici. Come le classi astratte, le interfacce non possono essere istanziate: possono essere implementate solo da classi o estese da altre interfacce. L'interfaccia è un modo comune per ottenere la completa astrazione in Java.

Sintassi

- interfaccia pubblica Foo {void foo (); / * altri metodi * /}
- l'interfaccia pubblica Foo1 estende Foo {void bar (); / * altri metodi * /}
- classe pubblica Foo2 implementa Foo, Foo1 {/ * implementazione di Foo e Foo1 * /}

Examples

Dichiarazione e implementazione di un'interfaccia

Dichiarazione di un'interfaccia che utilizza la parola chiave `interface` :

```
public interface Animal {
    String getSound(); // Interface methods are public by default
}
```

Sostituisci annotazione

```
@Override
public String getSound() {
    // Code goes here...
}
```

Questo costringe il compilatore a controllare che stiamo eseguendo l'override e impedisce al programma di definire un nuovo metodo o rovinare la firma del metodo.

Le interfacce sono implementate usando la parola chiave `implements` .

```
public class Cat implements Animal {

    @Override
    public String getSound() {
        return "meow";
    }
}
```

```
public class Dog implements Animal {

    @Override
    public String getSound() {
        return "woof";
    }
}
```

Nell'esempio, le classi `Cat` e `Dog` **devono** definire il metodo `getSound()` poiché i metodi di un'interfaccia sono intrinsecamente astratti (ad eccezione dei metodi predefiniti).

Utilizzando le interfacce

```
Animal cat = new Cat();
Animal dog = new Dog();

System.out.println(cat.getSound()); // prints "meow"
System.out.println(dog.getSound()); // prints "woof"
```

Implementazione di più interfacce

Una classe Java può implementare più interfacce.

```
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");
    }
}
```

Si noti come la classe `Cat` **deve** implementare i metodi `abstract` ereditati in entrambe le interfacce. Inoltre, nota come una classe può praticamente implementare tutte le interfacce necessarie (c'è un limite di **65.535 a** causa della Limitazione 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. Tutte le variabili dichiarate in un'interfaccia sono `public static final`
2. Tutti i metodi dichiarati in un'interfaccia metodi sono `public abstract` (Questa dichiarazione è valida solo tramite Java 7. Da Java 8, è possibile avere metodi in un'interfaccia, che non devono essere astratti, tali metodi sono noti come metodi predefiniti)
3. Le interfacce non possono essere dichiarate `final`
4. Se più di una interfaccia dichiara un metodo che ha la stessa firma, allora efficacemente viene trattato come un solo metodo e non è possibile distinguere da quale metodo di interfaccia è implementato
5. Un file **InterfaceName.class** corrispondente verrà generato per ogni interfaccia, dopo la compilazione

Estendere un'interfaccia

Un'interfaccia può estendere un'altra interfaccia tramite la parola chiave `extends` .

```
public interface BasicResourceService {
    Resource getResource();
}

public interface ExtendedResourceService extends BasicResourceService {
    void updateResource(Resource resource);
}
```

Ora una classe che implementa `ExtendedResourceService` dovrà implementare sia `getResource()` che `updateResource()` .

Estensione di più interfacce

A differenza delle classi, la parola chiave `extends` può essere utilizzata per estendere più interfacce (separate da virgole) consentendo combinazioni di interfacce in una nuova interfaccia

```
public interface BasicResourceService {
    Resource getResource();
}

public interface AlternateResourceService {
    Resource getAlternateResource();
}

public interface ExtendedResourceService extends BasicResourceService,
AlternateResourceService {
    Resource updateResource(Resource resource);
}
```

In questo caso, una classe che implementa `ExtendedResourceService` dovrà implementare `getResource()` , `getAlternateResource()` e `updateResource()` .

Utilizzo delle interfacce con Generics

Supponiamo che tu voglia definire un'interfaccia che consenta la pubblicazione / il consumo di dati da e verso diversi tipi di canali (ad es. AMQP, JMS, ecc.), Ma tu vuoi essere in grado di cambiare i dettagli di implementazione ...

Definiamo un'interfaccia IO di base che può essere riutilizzata in più implementazioni:

```
public interface IO<IncomingType, OutgoingType> {

    void publish(OutgoingType data);
    IncomingType consume();
    IncomingType RPCSubmit(OutgoingType data);

}
```

Ora posso creare un'istanza di questa interfaccia, ma dal momento che non abbiamo implementazioni predefinite per questi metodi, avrà bisogno di un'implementazione quando la istanziamo:

```
    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
```

Possiamo anche fare qualcosa di più utile con quell'interfaccia, diciamo che vogliamo usarlo per avvolgere alcune funzioni base di 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);
    }

}
```

Diciamo che voglio usare questa interfaccia IO ora come un modo per contare le visite al mio sito web dal mio ultimo riavvio del sistema e quindi essere in grado di visualizzare il numero totale di visite - puoi fare qualcosa del genere:

```
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);
    }

}
```

Ora usiamo il 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
```

Quando si implementano più interfacce, non è possibile implementare la stessa interfaccia due volte. Questo vale anche per le interfacce generiche. Pertanto, il seguente codice non è valido e genererà un errore di compilazione:

```
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); }
}
```

Utilità delle interfacce

Le interfacce possono essere estremamente utili in molti casi. Ad esempio, supponiamo di avere una lista di animali e di voler scorrere l'elenco, ognuno dei quali stampa il suono che produce.

```
{cat, dog, bird}
```

Un modo per farlo sarebbe utilizzare le interfacce. Ciò consentirebbe lo stesso metodo per essere chiamato su tutte le classi

```
public interface Animal {
    public String getSound();
}
```

Qualsiasi classe che `implements Animal` deve avere anche un metodo `getSound()` , tuttavia possono avere implementazioni diverse

```
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";
    }
}
```

Ora abbiamo tre diverse classi, ognuna delle quali ha un metodo `getSound()` . Poiché tutte queste

classi `implement` l'interfaccia `Animal` , che dichiara il metodo `getSound()` , qualsiasi istanza di un `Animal` può avere `getSound()` chiamato su di esso

```
Animal dog = new Dog();
Animal cat = new Cat();
Animal bird = new Bird();

dog.getSound(); // "Woof"
cat.getSound(); // "Meow"
bird.getSound(); // "Chirp"
```

Poiché ognuno di questi è un `Animal` , potremmo persino inserire gli animali in un elenco, scorrerli tra loro e stampare i loro suoni

```
Animal[] animals = { new Dog(), new Cat(), new Bird() };
for (Animal animal : animals) {
    System.out.println(animal.getSound());
}
```

Poiché l'ordine dell'array è `Dog` , `Cat` e quindi `Bird` , *"Woof Meow Chirp"* verrà stampato sulla console.

Le interfacce possono anche essere utilizzate come valore di ritorno per le funzioni. Ad esempio, restituendo un `Dog` se l'input è *"dog"* , `Cat` se l'input è *"cat"* e `Bird` se è *"bird"* , e quindi si può stampare il suono di quell'animale 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
```

Le interfacce sono anche utili per l'estensibilità, perché se si desidera aggiungere un nuovo tipo di `Animal` , non è necessario modificare nulla con le operazioni eseguite su di esse.

Implementazione di interfacce in una classe astratta

Un metodo definito in `interface` è di default `public abstract` . Quando una `abstract class` implementa `interface` , i metodi definiti `interface` non devono essere implementati dalla `abstract class` . Questo perché una `class` dichiarata `abstract` può contenere dichiarazioni di metodi astratti. È quindi responsabilità della prima sottoclasse concreta implementare qualsiasi metodo `abstract` ereditato da qualsiasi interfaccia e / o `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.");
    }
}
```

Da Java 8 in poi è possibile che `interface` dichiari implementazioni `default` di metodi, il che significa che il metodo non sarà `abstract` , quindi qualsiasi sottoclasse concreta non sarà forzata ad implementare il metodo ma erediterà l'implementazione `default` meno che non venga sovrascritta.

Metodi predefiniti

Introdotto in Java 8, i metodi predefiniti sono un modo per specificare un'implementazione all'interno di un'interfaccia. Questo potrebbe essere usato per evitare la tipica classe "Base" o "Abstract" fornendo un'implementazione parziale di un'interfaccia e limitando la gerarchia delle sottoclassi.

Implementazione del modello di osservatore

Ad esempio, è possibile implementare il pattern Observer-Listener direttamente nell'interfaccia, fornendo maggiore flessibilità alle classi di implementazione.

```
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);
        }
    }
}
```

Ora, qualsiasi classe può essere resa "Osservabile" semplicemente implementando l'interfaccia Observable, pur essendo libera di far parte di una diversa gerarchia di classi.

```
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

Il compilatore in Java 8 è a conoscenza del problema dei rombi causato da una classe che

implementa interfacce contenenti un metodo con la stessa firma.

Per risolverlo, una classe di implementazione deve sovrascrivere il metodo condiviso e fornire la propria implementazione.

```
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"
    }
}
```

C'è ancora il problema di avere metodi con lo stesso nome e parametri con diversi tipi di ritorno, che non verranno compilati.

Utilizzare i metodi predefiniti per risolvere i problemi di compatibilità

Le implementazioni del metodo predefinito sono molto utili se un metodo viene aggiunto a un'interfaccia in un sistema esistente in cui le interfacce vengono utilizzate da più classi.

Per evitare di suddividere l'intero sistema, è possibile fornire un'implementazione di metodo predefinita quando si aggiunge un metodo a un'interfaccia. In questo modo, il sistema sarà ancora compilato e le implementazioni effettive possono essere fatte passo dopo passo.

Per ulteriori informazioni, vedere l'argomento Metodi predefiniti .

Modificatori nelle interfacce

L'Oracle Java Style Guide afferma:

> I modificatori non dovrebbero essere scritti quando sono impliciti.

(Vedi Modificatori in Oracle Official Code Standard per il contesto e un link al documento Oracle effettivo.)

Questo orientamento di stile si applica in particolare alle interfacce. Consideriamo il seguente frammento di codice:

```
interface I {
    public static final int VARIABLE = 0;

    public abstract void method();

    public static void staticMethod() { ... }
    public default void defaultMethod() { ... }
}
```

---

variabili

Tutte le variabili di interfaccia sono *costanti* implicite con modificatori `public` impliciti (accessibili a tutti), `static` (accessibili tramite nome interfaccia) e `final` (devono essere inizializzati durante la dichiarazione):

```
~~public static final~~ int VARIABLE = 0;
```

---

metodi

1. Tutti i metodi che *non prevedono l'implementazione* sono implicitamente `public` e `abstract` .

```
~~public abstract~~ void method();
```

Java SE 8

2. Tutti i metodi con modificatore `static` o `default` *devono fornire l'implementazione* e sono implicitamente `public` .

```
~~public~~ static void staticMethod() { ... }
```

Dopo aver applicato tutte le modifiche precedenti, otterremo quanto segue:

```
interface I {
    int VARIABLE = 0;

    void method();
```

```
    static void staticMethod() { ... }
    default void defaultMethod() { ... }
}
```

Rafforza i parametri del tipo limitato

I parametri di tipo limitato consentono di impostare restrizioni sugli argomenti di tipo generico:

```
class SomeClass {

}

class Demo<T extends SomeClass> {

}
```

Ma un parametro di tipo può collegarsi solo a un singolo tipo di classe.

Un tipo di interfaccia può essere associato a un tipo che ha già un legame. Questo si ottiene usando il simbolo `&` :

```
interface SomeInterface {

}

class GenericClass<T extends SomeClass & SomeInterface> {

}
```

Ciò rafforza il legame, potenzialmente richiedendo argomenti di tipo per derivare da più tipi.

Più tipi di interfaccia possono essere associati a un parametro di tipo:

```
class Demo<T extends SomeClass & FirstInterface & SecondInterface> {

}
```

Ma dovrebbe essere usato con cautela. I collegamenti di interfacce multiple sono di solito un segno di odore di codice , suggerendo che dovrebbe essere creato un nuovo tipo che funge da adattatore per gli altri tipi:

```
interface NewInterface extends FirstInterface, SecondInterface {

}

class Demo<T extends SomeClass & NewInterface> {

}
```

Leggi interfacce online: https://riptutorial.com/it/java/topic/102/interfacce

Capitolo 89: Interfacce funzionali

introduzione

In Java 8+, un'interfaccia *funzionale* è un'interfaccia che ha solo un metodo astratto (a parte i metodi di Object). Vedi JLS §9.8. Interfacce funzionali

Examples

Elenco delle interfacce funzionali della libreria Java Runtime standard mediante firma

Tipi di parametri	Tipo di reso	Interfaccia
()	vuoto	Runnable
()	T	Fornitore
()	booleano	BooleanSupplier
()	int	IntSupplier
()	lungo	LongSupplier
()	Doppio	DoubleSupplier
(T)	vuoto	Consumatori <T>
(T)	T	UnaryOperator <T>
(T)	R	Funzione <T, R>
(T)	booleano	Predicate <T>
(T)	int	ToIntFunction <T>
(T)	lungo	ToLongFunction <T>
(T)	Doppio	ToDoubleFunction <T>
(T, T)	T	BinaryOperator <T>
(T, U)	vuoto	BiConsumer <T, U>
(T, U)	R	BiFunction <T, U, R>
(T, U)	booleano	BiPredicate <T, U>

Tipi di parametri	Tipo di reso	Interfaccia
(T, U)	int	ToIntBiFunction <T, U>
(T, U)	lungo	ToLongBiFunction <T, U>
(T, U)	Doppio	ToDoubleBiFunction <T, U>
(T, int)	vuoto	ObjIntConsumer <T>
(T, lungo)	vuoto	ObjLongConsumer <T>
(T, doppio)	vuoto	ObjDoubleConsumer <T>
(Int)	vuoto	IntConsumer
(Int)	R	IntFunction <R>
(Int)	booleano	IntPredicate
(Int)	int	IntUnaryOperator
(Int)	lungo	IntToLongFunction
(Int)	Doppio	IntToDoubleFunction
(int, int)	int	IntBinaryOperator
(lungo)	vuoto	LongConsumer
(lungo)	R	LongFunction <R>
(lungo)	booleano	LongPredicate
(lungo)	int	LongToIntFunction
(lungo)	lungo	LongUnaryOperator
(lungo)	Doppio	LongToDoubleFunction
(lungo lungo)	lungo	LongBinaryOperator
(Doppio)	vuoto	DoubleConsumer
(Doppio)	R	DoubleFunction <R>
(Doppio)	booleano	DoublePredicate
(Doppio)	int	DoubleToIntFunction
(Doppio)	lungo	DoubleToLongFunction

Tipi di parametri	Tipo di reso	Interfaccia
(Doppio)	Doppio	DoubleUnaryOperator
(doppio, doppio)	Doppio	DoubleBinaryOperator

Leggi Interfacce funzionali online: https://riptutorial.com/it/java/topic/10001/interfacce-funzionali

Capitolo 90: Interfaccia Dequeue

introduzione

Un Deque è una collezione lineare che supporta l'inserimento e la rimozione degli elementi ad entrambe le estremità.

Il nome deque è l'abbreviazione di "coda doppia" e viene solitamente pronunciato "mazzo".

La maggior parte delle implementazioni di Deque non pone limiti fissi sul numero di elementi che possono contenere, ma questa interfaccia supporta deques a capacità limitata e senza limiti di dimensione fissa.

L'interfaccia di Deque è un tipo di dati astratto più ricco di Stack e Queue perché implementa contemporaneamente stack e code allo stesso tempo

Osservazioni

I generici possono essere usati con Deque.

```
Deque<Object> deque = new LinkedList<Object>();
```

Quando una deque viene utilizzata come coda, vengono visualizzati i risultati del comportamento FIFO (First-In-First-Out).

Deques può anche essere utilizzato come stack LIFO (Last-In-First-Out).

Per ulteriori informazioni sui metodi, consultare questa documentazione.

Examples

Aggiungere elementi 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");
```

Rimozione di elementi da 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();
```

Recupero di elementi senza rimozione

```
//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();
```

Iterare attraverso 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;
}
```

Leggi Interfaccia Dequeue online: https://riptutorial.com/it/java/topic/10156/interfaccia-dequeue

Capitolo 91: Invio di metodi dinamici

introduzione

Cos'è il metodo dinamico di spedizione?

Dynamic Method Dispatch è un processo in cui la chiamata a un metodo sovrascritto viene risolta in fase di esecuzione anziché in fase di compilazione. Quando un metodo sottoposto a override viene chiamato da un riferimento, Java determina quale versione di quel metodo eseguire in base al tipo di oggetto a cui si riferisce. Questo è anche noto come polimorfismo di runtime.

Vedremo questo attraverso un esempio.

Osservazioni

- Associazione dinamica = rilegatura tardiva
- Le classi astratte non possono essere istanziate, ma possono essere sottoclassate (base per una classe figlio)
- Un metodo astratto è un metodo dichiarato senza implementazione
- La classe astratta può contenere una combinazione di metodi dichiarati con o senza un'implementazione
- Quando una classe astratta è sottoclassata, la sottoclasse di solito fornisce implementazioni per tutti i metodi astratti nella sua classe genitore. Tuttavia, se così non fosse, anche la sottoclasse deve essere dichiarata astratta
- La spedizione del metodo dinamico è un meccanismo mediante il quale una chiamata a un metodo sovrascritto viene risolta in fase di runtime. Ecco come java implementa il polimorfismo di runtime.
- Upcasting: trasmettere un sottotipo ad un supertipo, verso l'alto nell'albero ereditario.
- Polimorfismo di runtime = Polymorphism dinamico

Examples

Invio di metodi dinamici - Codice di esempio

Classe astratta:

```
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");
    }
}
```

Sovrascrittura del metodo astratto in classe secondaria:

```
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
```

Il binding dinamico / tardivo porta alla spedizione del metodo dinamico:

```
//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();

    }
}
```

Risultato:

```
Scan for Internet Security
Detect Virus & Malware
```

Upcasting:

```
objVS = new ClsFreeVersion();
objVS = new ClsPaidVersion();
objVS = new ClsTrialVersion()
```

Leggi Invio di metodi dinamici online: https://riptutorial.com/it/java/topic/9204/invio-di-metodi-dinamici

Capitolo 92: Iteratore e Iterable

introduzione

`java.util.Iterator` è l'interfaccia standard Java SE per oggetti che implementano il modello di progettazione Iterator. L'interfaccia `java.lang.Iterable` è per gli oggetti che possono *fornire* un iteratore.

Osservazioni

È possibile iterare su un array usando il ciclo `for` -each, sebbene gli array java non implementino Iterable; l'iterazione viene eseguita da JVM utilizzando un indice non accessibile in background.

Examples

Uso di loop Iterable in for

Le classi che implementano l'interfaccia `Iterable<>` possono essere utilizzate in cicli `for` . Questo è in realtà solo zucchero sintattico per ottenere un iteratore dall'oggetto e utilizzarlo per ottenere tutti gli elementi in sequenza; rende il codice più chiaro, più veloce da scrivere e meno incline agli errori.

```
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);
        }
    }
}
```

Utilizzando l'iteratore raw

L'uso del ciclo foreach (o "extended for loop") è semplice, a volte è utile utilizzare direttamente l'iteratore. Ad esempio, se desideri generare un gruppo di valori separati da virgola, ma non vuoi che l'ultimo elemento abbia una virgola:

```
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(",");
    }
}
```

Questo è molto più facile e più chiaro di avere una variabile `isLastEntry` o fare calcoli con l'indice di loop.

Crea il tuo Iterable.

Per creare il tuo Iterable come con qualsiasi interfaccia, devi semplicemente implementare i metodi astratti nell'interfaccia. Per `Iterable` c'è solo uno che è chiamato `iterator()` . Ma il suo tipo di ritorno `Iterator` è di per sé un'interfaccia con tre metodi astratti. Puoi restituire un iteratore associato ad alcune raccolte o creare la tua implementazione personalizzata:

```
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");
            }
        };
    }
}
```

Usare:

```
public static void main(String[] args) {
    for(char c : new Alphabet()) {
        System.out.println("c = " + c);
    }
}
```

Il nuovo `Iterator` dovrebbe avere uno stato che punta al primo elemento, ogni chiamata al successivo aggiorna il suo stato in modo che punti a quello successivo. `hasNext()` controlla se l'iteratore è alla fine. Se l'iteratore fosse collegato a una raccolta modificabile, il metodo `remove()` facoltativo dell'iteratore potrebbe essere implementato per rimuovere l'elemento attualmente indirizzato dalla raccolta sottostante.

Rimozione di elementi mediante un iteratore

Il metodo `Iterator.remove()` è un metodo facoltativo che rimuove l'elemento restituito dalla precedente chiamata a `Iterator.next()` . Ad esempio, il codice seguente popola un elenco di stringhe e quindi rimuove tutte le stringhe vuote.

```
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());
```

Produzione :

```
Old Size : 5
New Size : 3
```

Nota che il codice sopra è il modo sicuro per rimuovere elementi mentre si itera una raccolta tipica. Se invece, provi a rimuovere elementi da una raccolta come questa:

```
for (String el: names) {
    if (el.equals("")) {
        names.remove(el); // WRONG!
    }
}
```

una raccolta tipica (come `ArrayList`) che fornisce agli iteratori la semantica dell'iteratore di *fail veloce* genererà una `ConcurrentModificationException` .

Il metodo `remove()` può solo chiamare (una volta) dopo una chiamata `next()` . Se viene chiamato prima di chiamare `next()` o se viene chiamato due volte dopo una chiamata `next()` , allora la chiamata `remove()` genererà un `IllegalStateException` .

L'operazione di `remove` è descritta come un'operazione *opzionale* ; cioè non tutti gli iteratori lo permetteranno. Esempi in cui non è supportato includono iteratori per raccolte immutabili, viste di sola lettura di raccolte o raccolte di dimensioni fisse. Se `remove()` viene chiamato quando l'iteratore

non supporta la rimozione, genererà un `UnsupportedOperationException` .

Leggi Iteratore e Iterable online: https://riptutorial.com/it/java/topic/172/iteratore-e-iterable

Capitolo 93: Java Native Interface

Parametri

Parametro	Dettagli
MEnv	Puntatore all'ambiente JNI
jobject	L'oggetto che ha invocato il metodo `native` non `static`
jclass	La classe che ha invocato il metodo `native static`

Osservazioni

L'impostazione di JNI richiede sia un compilatore Java che nativo. A seconda dell'IDE e del sistema operativo, sono necessarie alcune impostazioni. Una guida per Eclipse può essere trovata qui . Un tutorial completo può essere trovato qui .

Questi sono i passaggi per configurare il collegamento Java-C ++ su windows:

- Compilare i file di origine Java (`.java`) in classi (`.class`) usando `javac` .
- Crea `.h` intestazione (`.h`) dalle classi Java che contengono metodi `native` usando `javah` . Questi file "istruiscono" il codice nativo su quali metodi è responsabile dell'implementazione.
- Includere i file di intestazione (`#include`) nei file di origine C ++ (`.cpp`) che implementano i metodi `native` .
- Compilare i file di origine C ++ e creare una libreria (`.dll`). Questa libreria contiene l'implementazione del codice nativo.
- Specificare il percorso della libreria (`-Djava.library.path`) e caricarlo nel file di origine Java (`System.loadLibrary(...)`).

Callback (Chiamando i metodi Java dal codice nativo) richiede di specificare un descrittore del metodo. Se il descrittore non è corretto, si verifica un errore di runtime. Per questo `javap -s` è utile avere i descrittori creati per noi, questo può essere fatto con `javap -s` .

Examples

Chiamare i metodi C ++ da Java

I metodi statici e membri in Java possono essere contrassegnati come *nativi* per indicare che la loro implementazione si trova in un file di libreria condiviso. Al momento dell'esecuzione di un metodo nativo, la JVM cerca una funzione corrispondente nelle librerie caricate (vedi Caricamento librerie native), usando un semplice schema di mangling dei nomi, esegue la conversione degli argomenti e l'impostazione dello stack, quindi passa il controllo al codice nativo.

Codice 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);
    }
}
```

Codice C ++

I file di intestazione contenenti dichiarazioni di funzioni native devono essere generati utilizzando lo strumento `javah` sulle classi di destinazione. Eseguendo il seguente comando nella directory di build:

```
javah -o com_example_jni_JNIJava.hpp com.example.jni.JNIJava
```

... produce il seguente file di intestazione (*commenti ridotti per brevità*):

```
// 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
```

Ecco un esempio di implementazione:

```
// 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;
}
```

Produzione

L'esecuzione della classe di esempio sopra produce l'output seguente:

> Invocato C ++ 'printString' da Java
> Ottenuto risultato da 'media' C ++: 3.4

Chiamare i metodi Java da C ++ (callback)

La chiamata di un metodo Java da codice nativo è un processo in due fasi:

1. ottenere un puntatore del metodo con la funzione JNI `GetMethodID` , utilizzando il nome del metodo e il descrittore;
2. chiama una delle funzioni `Call*Method` elencate qui .

Codice 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);
    }
}
```

Codice 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);
}
```

Produzione

Ottenuto float da C ++: 5.221
Got int da C ++: 17

Ottenere il descrittore

I descrittori (o *le firme dei tipi interni*) sono ottenuti utilizzando il programma **javap** nel file `.class` compilato. Ecco l'output di `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
}
```

Caricamento delle librerie native

L'idioma comune per caricare i file di libreria condivisi in Java è il seguente:

```
public class ClassWithNativeMethods {
    static {
        System.loadLibrary("Example");
    }

    public native void someNativeMethod(String arg);
    ...
```

Le chiamate a `System.loadLibrary` sono quasi sempre statiche in modo da verificarsi durante il caricamento della classe, assicurando che nessun metodo nativo possa essere eseguito prima che la libreria condivisa sia stata caricata. Tuttavia è possibile:

```
public class ClassWithNativeMethods {
    // Call this before using any native method
    public static void prepareNativeMethods() {
        System.loadLibrary("Example");
    }

    ...
```

Ciò consente di posticipare il caricamento della libreria condivisa finché necessario, ma richiede un'attenzione particolare per evitare `java.lang.UnsatisfiedLinkError` s.

Ricerca file di destinazione

I file della libreria condivisa vengono cercati nei percorsi definiti dalla proprietà di sistema `java.library.path` , che può essere sovrascritta utilizzando l'argomento `-Djava.library.path=` JVM in fase di runtime:

```
java -Djava.library.path=path/to/lib/:path/to/other/lib MainClassWithNativeMethods
```

Attenzione ai separatori dei percorsi di sistema: ad esempio, Windows utilizza ; invece di :

Notare che `System.loadLibrary` risolve i nomi dei file di libreria in modo dipendente dalla piattaforma: lo snippet di codice sopra prevede un file denominato `libExample.so` su Linux e `Example.dll` su Windows.

Un'alternativa a `System.loadLibrary` è `System.load(String)` , che prende il percorso completo di un file di libreria condivisa, aggirando la ricerca `java.library.path` :

```
public class ClassWithNativeMethods {
    static {
        System.load("/path/to/lib/libExample.so");
    }

    ...
```

Leggi Java Native Interface online: https://riptutorial.com/it/java/topic/168/java-native-interface

Capitolo 94: Java Performance Tuning

Examples

Approccio generale

Internet è pieno di suggerimenti per il miglioramento delle prestazioni dei programmi Java. Forse il consiglio numero uno è la consapevolezza. Questo significa:

- Identificare possibili problemi di prestazioni e colli di bottiglia.
- Utilizzare strumenti di analisi e test.
- Conoscere buone pratiche e cattive pratiche.

Il primo punto dovrebbe essere fatto durante la fase di progettazione se si parla di un nuovo sistema o modulo. Se si parla di codice legacy, gli strumenti di analisi e testing entrano in scena. Lo strumento più basilare per analizzare le tue prestazioni JVM è JVisualVM, che è incluso nel JDK.

Il terzo punto riguarda principalmente l'esperienza e la ricerca approfondita e, ovviamente, i suggerimenti grezzi che verranno visualizzati in questa pagina e altri, come questo .

Riduzione della quantità di stringhe

In Java, è troppo "facile" creare molte istanze String che non sono necessarie. Questo e altri motivi potrebbero far sì che il tuo programma abbia molte stringhe che il GC è impegnato a ripulire.

Alcuni modi in cui potresti creare istanze String:

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

O peggio, in un ciclo o ricorsione:

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

Il problema è che ogni `+` crea una nuova stringa (di solito, poiché i nuovi compilatori ottimizzano alcuni casi). Una possibile ottimizzazione può essere fatta usando `StringBuilder` o `StringBuffer` :

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

Se si costruiscono spesso stringhe lunghe (ad esempio SQL), utilizzare un'API di costruzione di

stringhe.

Altre cose da considerare:

- Ridurre l'uso di `replace` , `substring` ecc.
- Evitare `String.toArray()` , specialmente nel codice di accesso frequente.
- Le stampe di registro che sono destinate a essere filtrate (a causa del livello di registro per esempio) non dovrebbero essere generate (il livello di registro deve essere controllato in anticipo).
- Usa librerie come questa se necessario.
- StringBuilder è migliore se la variabile viene utilizzata in modo non condiviso (attraverso thread).

Un approccio evidence-based all'ottimizzazione delle prestazioni di Java

Donald Knuth è spesso citato come dicendo questo:

> "I programmatori sprecano enormi quantità di tempo a pensare, o preoccuparsi, la velocità di parti non critiche dei loro programmi, e questi tentativi di efficienza in realtà avere un forte impatto negativo quando il debug e la manutenzione sono considerati. *Dobbiamo dimenticare le piccole efficienze, dicono di Il 97% delle volte* : l'ottimizzazione prematura è la radice di tutti i mali, tuttavia non dovremmo perdere le nostre opportunità in quel 3% critico ".

fonte

Tenendo presente questo consiglio, ecco la procedura consigliata per l'ottimizzazione dei programmi:

1. Prima di tutto, progetta e codifica il tuo programma o libreria con particolare attenzione alla semplicità e alla correttezza. Per cominciare, non spendere molto per le prestazioni.

2. Portalo a uno stato di lavoro e (idealmente) sviluppa test unitari per le parti chiave del codice base.

3. Sviluppare un benchmark delle prestazioni a livello di applicazione. Il benchmark dovrebbe coprire gli aspetti critici delle prestazioni della vostra applicazione e dovrebbe eseguire una serie di attività tipiche di come l'applicazione verrà utilizzata nella produzione.

4. Misura le prestazioni.

5. Confronta le prestazioni misurate con i tuoi criteri per la velocità con cui l'applicazione deve essere. (Evita criteri non realistici, irraggiungibili o non quantificabili come "il più velocemente possibile".)

6. Se hai soddisfatto i criteri, FERMA. Il lavoro è finito (Qualsiasi ulteriore sforzo è probabilmente una perdita di tempo.)

7. Profili l'applicazione mentre sta eseguendo il tuo benchmark delle prestazioni.

8. Esamina i risultati del profilo e scegli i "hotspot di prestazione" più grandi (non ottimizzati); cioè sezioni del codice in cui l'applicazione sembra passare più tempo.

9. Analizza la sezione del codice hotspot per cercare di capire perché è un collo di bottiglia e pensa a un modo per renderlo più veloce.

10. Implementalo come una proposta di modifica del codice, test e debug.

11. Rieseguire il benchmark per vedere se il cambio di codice ha migliorato le prestazioni:

    - Se Sì, quindi tornare al passaggio 4.
    - Se No, abbandonare la modifica e tornare al punto 9. Se non si stanno facendo progressi, selezionare un punto di attivazione diverso per l'attenzione.

Alla fine arriverete a un punto in cui l'applicazione è abbastanza veloce o avete preso in considerazione tutti gli hotspot significativi. A questo punto devi fermare questo approccio. Se una sezione di codice sta consumando (diciamo) l'1% del tempo complessivo, allora anche un miglioramento del 50% renderà l'applicazione solo dello 0,5% più veloce nel complesso.

Chiaramente, c'è un punto oltre il quale l'ottimizzazione dell'hotspot è uno spreco di energie. Se arrivi a quel punto, devi adottare un approccio più radicale. Per esempio:

- Guarda la complessità algoritmica dei tuoi algoritmi core.
- Se l'applicazione sta spendendo un sacco di tempo per la garbage collection, cerca modi per ridurre la velocità di creazione dell'oggetto.
- Se le parti chiave dell'applicazione sono ad uso intensivo della CPU e single-thread, cerca opportunità per il parallelismo.
- Se l'applicazione è già multi-thread, cerca i colli di bottiglia della concorrenza.

Ma ovunque sia possibile, affidati a strumenti e misure piuttosto che all'istinto per indirizzare i tuoi sforzi di ottimizzazione.

Leggi Java Performance Tuning online: https://riptutorial.com/it/java/topic/4160/java-performance-tuning

Capitolo 95: Java Virtual Machine (JVM)

Examples

Queste sono le basi.

JVM è una **macchina di calcolo astratta** o una **macchina virtuale** che risiede nella RAM. Ha un ambiente di esecuzione indipendente dalla piattaforma che interpreta il bytecode Java nel codice macchina nativo. (Javac è Java Compiler che compila il codice Java in Bytecode)

Il programma Java verrà eseguito all'interno della JVM, che verrà quindi mappato sulla macchina fisica sottostante. È uno strumento di programmazione in JDK.

(Il `Byte code` è un codice indipendente dalla piattaforma che viene eseguito su ogni piattaforma e il `Machine code` è un codice specifico della piattaforma che viene eseguito solo su una piattaforma specifica come windows o linux, ma dipende dall'esecuzione.)

Alcuni dei componenti: -

- Class Loder - carica il file .class nella RAM.
- Bytecode verificatore: controlla se ci sono violazioni di restrizioni di accesso nel tuo codice.
- Motore di esecuzione: converte il codice byte in codice macchina eseguibile.
- JIT (just in time) - JIT fa parte di JVM che ha contribuito a migliorare le prestazioni di JVM. Compilerà o convertirà dinamicamente bytecode java in codice macchina nativo durante il tempo di esecuzione.

(Modificato)

Leggi Java Virtual Machine (JVM) online: https://riptutorial.com/it/java/topic/8110/java-virtual-machine--jvm-

Capitolo 96: JavaBean

introduzione

JavaBeans (TM) è un modello per la progettazione di API classe Java che consente l'utilizzo di istanze (bean) in vari contesti e l'utilizzo di vari strumenti *senza* scrivere codice Java in modo esplicito. I pattern sono costituiti da convenzioni per la definizione di getter e setter per le *proprietà* , per la definizione dei costruttori e per la definizione delle API listener di eventi.

Sintassi

- **Regole di denominazione delle proprietà JavaBean**
- Se la proprietà non è un valore booleano, è necessario ottenere il prefisso del metodo getter. Ad esempio, getSize () è un nome getter JavaBeans valido per una proprietà denominata "size". Ricorda che non è necessario avere una variabile di nome size. Il nome della proprietà viene dedotto dai getter e setter, non da eventuali variabili della classe. Ciò che ritorni da getSize () dipende da te.
- Se la proprietà è un valore booleano, il prefisso del metodo getter è get o is. Ad esempio, getStopped () o isStopped () sono entrambi nomi JavaBeans validi per una proprietà booleana.
- Il prefisso del metodo setter deve essere impostato. Ad esempio, setSize () è il nome JavaBean valido per una proprietà denominata size.
- Per completare il nome di un metodo getter o setter, modificare la prima lettera del nome della proprietà in maiuscolo e quindi aggiungerla al prefisso appropriato (get, is o set).
- Le firme del metodo di incastonatura devono essere contrassegnate come pubbliche, con un tipo di reso vuoto e un argomento che rappresenta il tipo di proprietà.
- Le firme del metodo Getter devono essere contrassegnate come pubbliche, non richiedono argomenti e hanno un tipo restituito che corrisponda al tipo di argomento del metodo setter per quella proprietà.
- **Regole di denominazione dei listener JavaBean**
- I nomi dei metodi listener utilizzati per "registrare" un listener con un'origine evento devono utilizzare il prefisso add, seguito dal tipo listener. Ad esempio, addActionListener () è un nome valido per un metodo che un'origine evento dovrà consentire ad altri di registrarsi per eventi Action.
- I nomi dei metodi listener utilizzati per rimuovere ("annullare la registrazione") un listener devono utilizzare il prefisso remove, seguito dal tipo listener (utilizzando le stesse regole del metodo di aggiunta della registrazione).
- Il tipo di listener da aggiungere o rimuovere deve essere passato come argomento del metodo.
- I nomi dei metodi listener devono terminare con la parola "Listener".

Osservazioni

Affinché una classe sia un Java Bean deve seguire questo standard - in sintesi:

- Tutte le sue proprietà devono essere private e accessibili solo attraverso getter e setter.
- Deve avere un costruttore pubblico senza argomenti.
- Deve implementare l'interfaccia `java.io.Serializable` .

Examples

Java Bean di base

```
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;
    }
}
```

Leggi JavaBean online: https://riptutorial.com/it/java/topic/8157/javabean

Capitolo 97: JAXB

introduzione

JAXB o Java Architecture per XML Binding (JAXB) è un framework software che consente agli sviluppatori Java di associare classi Java a rappresentazioni XML. Questa pagina introdurrà i lettori a JAXB usando esempi dettagliati sulle sue funzioni fornite principalmente per il marshalling e l'un-marshaling di oggetti Java nel formato xml e viceversa.

Sintassi

- JAXB.marshall (oggetto, fileObjOfXML);
- Object obj = JAXB.unmarshall (fileObjOfXML, className);

Parametri

Parametro	Dettagli
fileObjOfXML	Oggetto `File` di un file XML
nome della classe	Nome di una classe con estensione `.class`

Osservazioni

Utilizzando lo strumento XJC disponibile nel JDK, il codice java per una struttura xml descritta in uno schema xml (file `.xsd`) può essere generato automaticamente, vedere l' argomento XJC .

Examples

Scrivere un file XML (eseguire il marshalling di un oggetto)

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

Usando l'annotazione `XMLRootElement` , possiamo contrassegnare una classe come elemento radice di un file 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()` è usato per scrivere il contenuto dell'oggetto in un file XML. Qui `user` oggetto `user` e un nuovo oggetto `File` vengono passati come argomenti al `marshal()` .

In caso di esecuzione riuscita, viene creato un file XML denominato `UserDetails.xml` nel percorso classe con il contenuto sottostante.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

Lettura di un file XML (annullamento della memoria)

Leggere un file XML denominato `UserDetails.xml` con il contenuto sottostante

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

Abbiamo bisogno di una classe POJO denominata `User.java` come di seguito

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

Qui abbiamo creato le variabili e il nome della classe in base ai nodi XML. Per `XmlRootElement` ,

usiamo l'annotazione `XmlRootElement` sulla classe.

```
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!");
        }
    }
}
```

Qui viene usato il metodo `unmarshal()` per analizzare il file XML. Prende il nome del file XML e il tipo di classe come due argomenti. Quindi possiamo usare i metodi getter dell'oggetto per stampare i dati.

Utilizzando XmlAdapter per generare il formato xml desiderato

Quando il formato XML desiderato differisce dal modello a oggetti Java, è possibile utilizzare un'implementazione XmlAdapter per trasformare l'oggetto modello in oggetto formato xml e viceversa. Questo esempio mostra come inserire il valore di un campo in un attributo di un elemento con il nome 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);
    }
}
```

Configurazione automatica del mapping XML di campi / proprietà (@XmlAccessorType)

Annotation `@XmlAccessorType` determina se i campi / proprietà verranno serializzati automaticamente in XML. Nota: le annotazioni `@XmlElement` campi e sui metodi `@XmlElement` , `@XmlAttribute` o `@XmlTransient` hanno la precedenza sulle impostazioni predefinite.

```
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
```

Produzione

```
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>
```

Configurazione del mapping XML di campo / proprietà manuale

Annotazioni `@XmlElement` , `@XmlAttribute` o `@XmlTransient` e altro nel pacchetto `javax.xml.bind.annotation` consentono al programmatore di specificare quale e in che modo i campi o le proprietà marcati devono essere serializzati.

```
@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!");
        }
    }
}
```

Specifica di un'istanza XmlAdapter per (ri) utilizzare i dati esistenti

A volte devono essere utilizzate specifiche istanze di dati. La ricreazione non è desiderata e il riferimento a dati `static` avrebbe un odore di codice.

È possibile specificare un `XmlAdapter` esempio `Unmarshaller` dovrebbe usare, che permette all'utente di utilizzare `XmlAdapter` s senza costruttore a zero arg e / o trasmettere dati alla scheda.

Esempio

Classe utente

La seguente classe contiene un nome e un'immagine dell'utente.

```
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);
    }

}
```

Adattatore

Per evitare di creare la stessa immagine in memoria due volte (oltre a scaricare nuovamente i dati), l'adattatore memorizza le immagini in una mappa.

Java SE 7

Per codice Java 7 valido sostituire il metodo `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();
    }

}
```

XML di esempio

I seguenti 2 xmls sono per *Jon Skeet* e la sua controparte earth 2, che hanno entrambi lo stesso aspetto e quindi usano lo stesso 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 l'adattatore

```
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());
```

Associazione di uno spazio dei nomi XML a una classe Java serializzabile.

Questo è un esempio di un file `package-info.java` che associa uno spazio dei nomi XML a una classe Java serializzabile. Questo dovrebbe essere collocato nello stesso pacchetto delle classi Java che dovrebbero essere serializzate usando lo spazio dei nomi.

```
/**
 * 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;
```

Utilizzando XmlAdapter per tagliare la stringa.

```
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();
    }
}
```

E in package-info.java aggiungi la seguente dichiarazione.

```
@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;
```

Leggi JAXB online: https://riptutorial.com/it/java/topic/147/jaxb

Capitolo 98: JAX-WS

Examples

Autenticazione di base

Il modo di fare una chiamata JAX-WS con l'autenticazione di base è un po 'non ovvio.

Ecco un esempio in cui `Service` è la rappresentazione della classe di servizio e `Port` è la porta di servizio a cui si desidera accedere.

```
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();
```

Leggi JAX-WS online: https://riptutorial.com/it/java/topic/4105/jax-ws

Capitolo 99: JMX

introduzione

La tecnologia JMX fornisce gli strumenti per la creazione di soluzioni distribuite, basate sul Web, modulari e dinamiche per la gestione e il monitoraggio di dispositivi, applicazioni e reti basate sui servizi. In base alla progettazione, questo standard è adatto per adattare i sistemi legacy, implementare nuove soluzioni di gestione e monitoraggio e collegarsi a quelli del futuro.

Examples

Semplice esempio con Platform MBean Server

Diciamo che abbiamo un server che registra i nuovi utenti e li saluta con un messaggio. E vogliamo monitorare questo server e modificare alcuni dei suoi parametri.

Innanzitutto, abbiamo bisogno di un'interfaccia con i nostri metodi di monitoraggio e controllo

```
public interface UserCounterMBean {
    long getSleepTime();

    void setSleepTime(long sleepTime);

    int getUserCount();

    void setUserCount(int userCount);

    String getGreetingString();

    void setGreetingString(String greetingString);

    void stop();
}
```

E qualche semplice implementazione che ci permetterà di vedere come funziona e come la influenziamo

```
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);
```

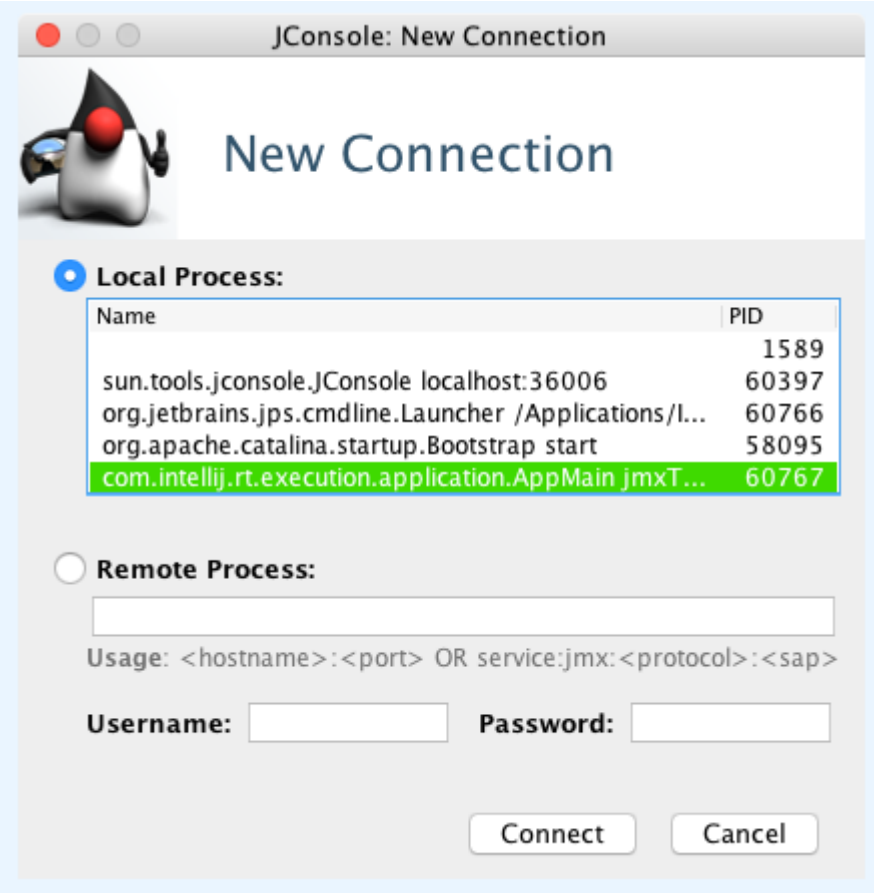
```
    }

    @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) {
            }
        }
    }
}
```

Per un semplice esempio con gestione locale o remota, dobbiamo registrare il nostro MBean:

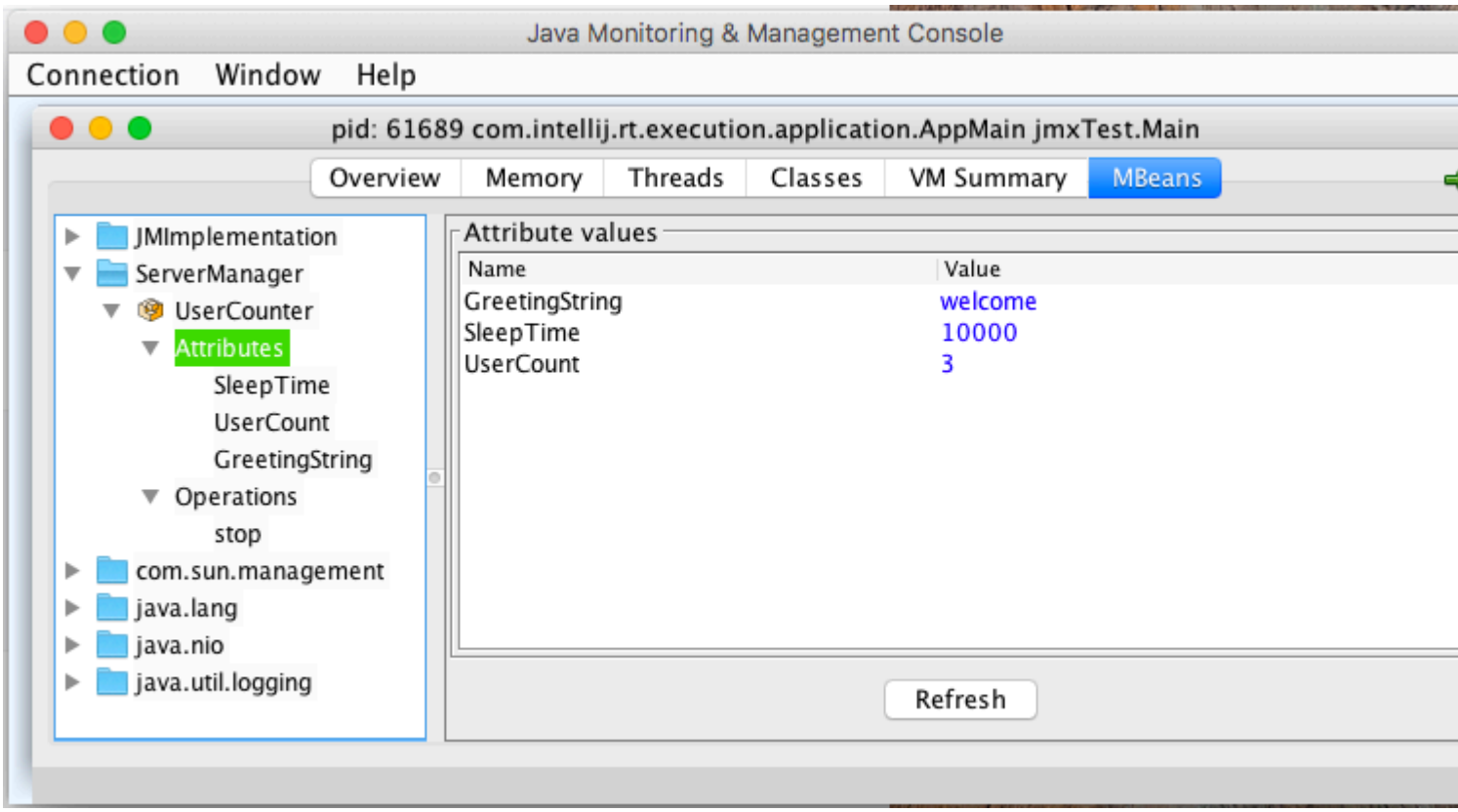
```
import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;

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();
    }
}
```

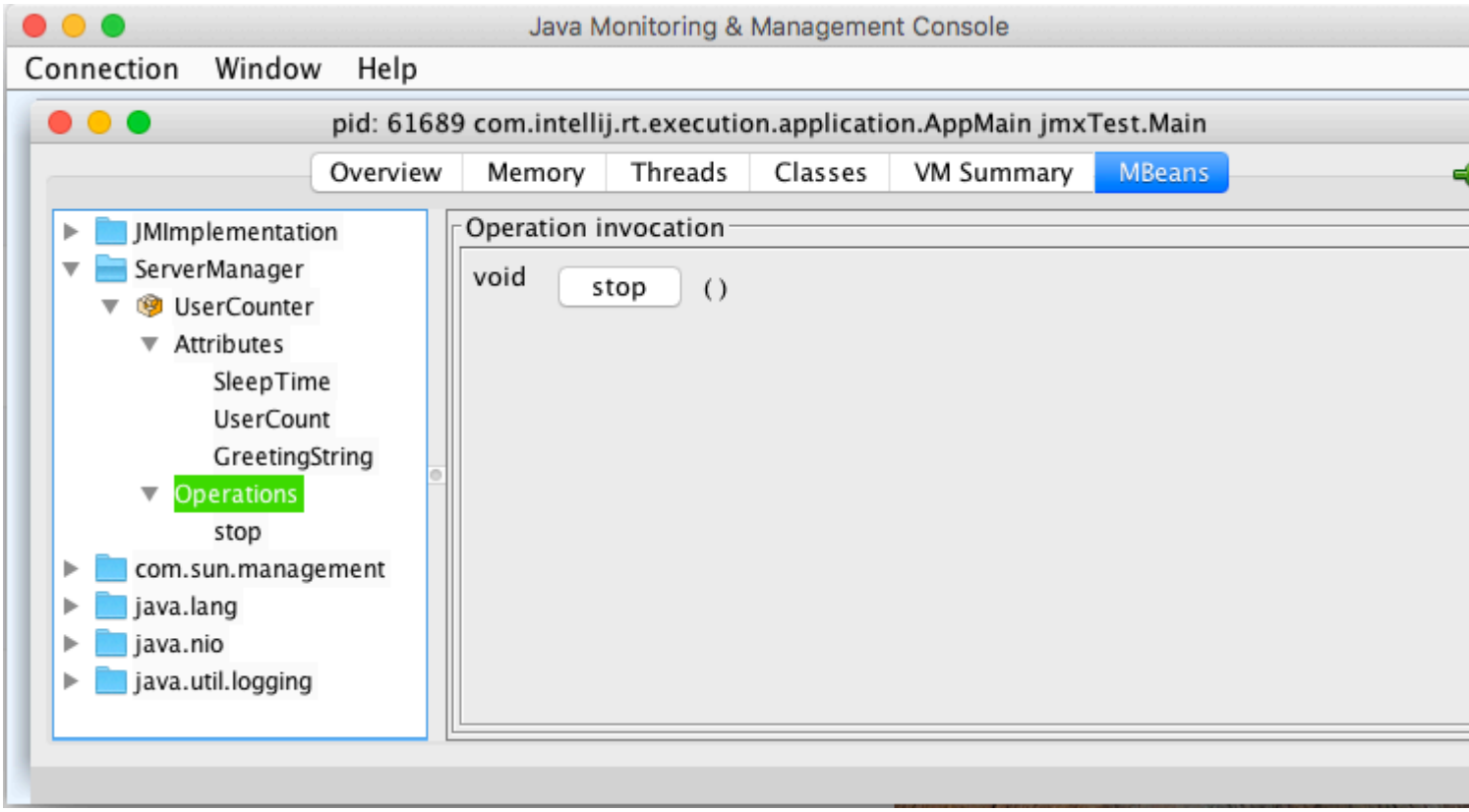
Dopodiché possiamo eseguire la nostra applicazione e connetterci tramite jConsole, che può essere trovata nella directory `$JAVA_HOME/bin` . Innanzitutto, dobbiamo trovare il nostro processo Java locale con la nostra applicazione



quindi passa alla scheda MBeans e trova l'MBean che abbiamo usato nella nostra classe Main come `ObjectName` (nell'esempio sopra è `ServerManager`). Nella sezione `Attributes` possiamo vedere gli attributi. Se hai specificato solo il metodo get, l'attributo sarà leggibile ma non scrivibile. Se hai specificato entrambi i metodi get e set, l'attributo sarebbe leggibile e scrivibile.



I metodi specificati possono essere richiamati nella sezione `Operations` .



Se vuoi essere in grado di utilizzare la gestione remota, avrai bisogno di parametri JVM aggiuntivi, come:

```
-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
```

Questi parametri sono disponibili nel Capitolo 2 delle guide JMX . Dopodiché sarai in grado di connetterti alla tua applicazione tramite jConsole in remoto con `jconsole host:port` o specificando `host:port` o `service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi` nella jConsole GUI.

Link utili:

- Guide JMX
- Best practice JMX

Leggi JMX online: https://riptutorial.com/it/java/topic/9278/jmx

Capitolo 100: JNDI

Examples

RMI attraverso JNDI

Questo esempio mostra come funziona JNDI in RMI. Ha due ruoli:

- per fornire al server un'API di binding / unbind / rebind al registro RMI
- per fornire al client un'API di ricerca / elenco per il registro RMI.

Il registro RMI è parte di RMI, non JNDI.

Per semplificare, useremo `java.rmi.registry.CreateRegistry()` per creare il registro RMI.

1. Server.java (il server 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 (il client 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 (classe server RMI)

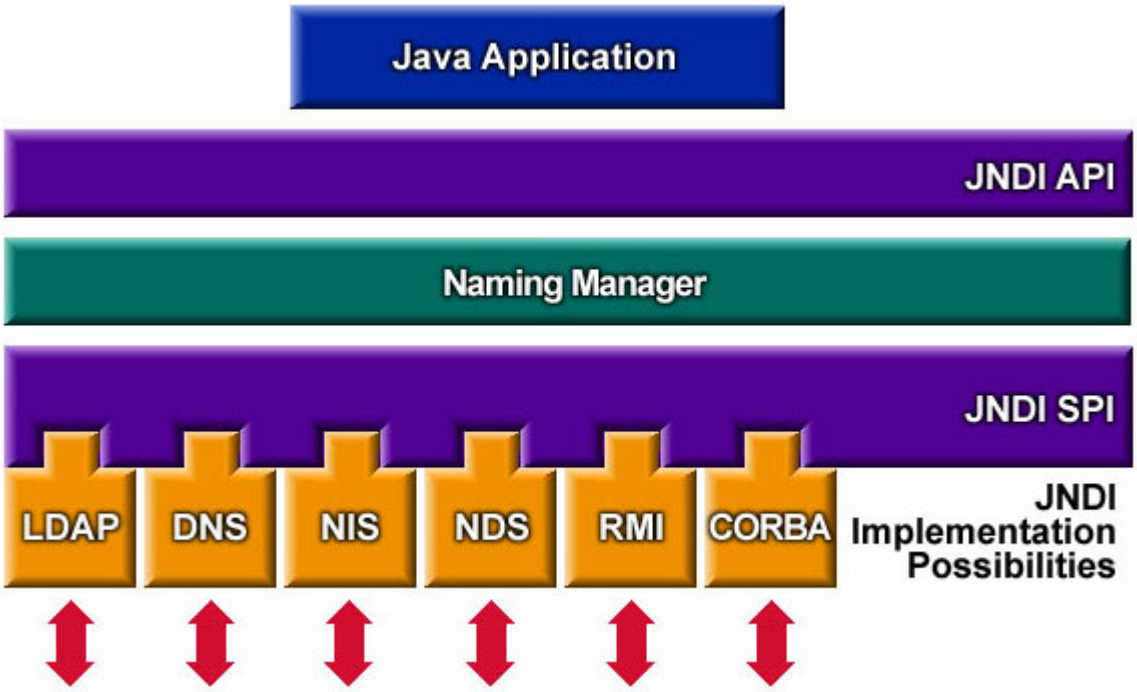
```
package com.neohope.jndi.test;

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;
    }
}
```

Come eseguire l'eaxmple:

1. costruire e avviare il server
2. costruire e avviare il client

Introdurre



JNDI (Java Naming and Directory Interface) è un'API Java per un servizio di directory che consente ai client software Java di rilevare e cercare dati e oggetti tramite un nome. È progettato per essere indipendente da qualsiasi specifica implementazione di servizi di denominazione o directory.

L'architettura JNDI è costituita da **un'API** (Application Programming Interface) e da una **SPI** (Service Provider Interface). Le applicazioni Java utilizzano questa API per accedere a una varietà di servizi di denominazione e directory. L'SPI consente di collegare in modo trasparente un'ampia gamma di servizi di denominazione e directory, consentendo all'applicazione Java che utilizza l'API della tecnologia JNDI di accedere ai propri servizi.

Come puoi vedere dall'immagine qui sopra, JNDI supporta LDAP, DNS, NIS, NDS, RMI e

CORBA. Certo, puoi estenderlo.

Come funziona

In questo esempio, Java RMI utilizza l'API JNDI per cercare oggetti in una rete. Se vuoi cercare un oggetto, hai bisogno di almeno due informazioni:

- Dove trovare l'oggetto

Il registro RMI gestisce i collegamenti dei nomi, ti dice dove trovare l'oggetto.

- Il nome dell'oggetto

Qual è il nome di un oggetto? Di solito è una stringa, può anche essere un oggetto che implementa l'interfaccia Nome.

Passo dopo passo

1. Per prima cosa è necessario un registro, che gestisca il binding del nome. In questo esempio, usiamo `java.rmi.registry.LocateRegistry` .

```
//This will start a registry on localhost, port 1234
registry = LocateRegistry.createRegistry(1234);
```

2. Sia il client che il server hanno bisogno di un contesto. Il server usa il contesto per associare il nome e l'oggetto. Il client usa il contesto per cercare il nome e ottenere l'oggetto.

```
//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. Il server associa il nome e l'oggetto

```
//The jndi name is "/neohope/jndi/test01"
bindJNDI("/neohope/jndi/test01", msg);
```

4. Il client cerca l'oggetto con il nome "/ neohope / jndi / test01"

```
//look up the object by name "java:com/neohope/jndi/test01"
NeoMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
```

5. Ora il client può usare l'oggetto

6. Al termine del server, è necessario pulire.

```
ctx.unbind("/neohope/jndi/test01");
ctx.close();
```

Leggi JNDI online: https://riptutorial.com/it/java/topic/5720/jndi

Capitolo 101: JShell

introduzione

JShell è un REPL interattivo per Java aggiunto in JDK 9. Consente agli sviluppatori di valutare istantaneamente espressioni, classi di test e sperimentare con il linguaggio Java. L'accesso anticipato per jdk 9 può essere ottenuto da: http://jdk.java.net/9/

Sintassi

- $ jshell - Avvia il REPL di JShell
- jshell> / <comando>: esegue un comando JShell specificato
- jshell> / exit - Esci da JShell
- jshell> / help - Visualizza un elenco di comandi di JShell
- jshell> <java_expression> - Valuta la specifica espressione Java (punto e virgola opzionale)
- jshell> / vars OR / metodi OR / types - Visualizza un elenco di variabili, metodi o classi, rispettivamente.
- jshell> / open <file> - legge un file come input per la shell
- jshell> / edit <identificatore> - modifica uno snippet nell'editor di set
- jshell> / set editor <comando>: imposta il comando da utilizzare per modificare i frammenti usando / modifica
- jshell> / drop <identificatore>: elimina uno snippet
- jshell> / reset - Reimposta la JVM ed elimina tutti i frammenti

Osservazioni

JShell richiede Java JDK 9, che attualmente (marzo 2017) può essere scaricato come istantanee di accesso anticipato da jdk9.java.net . Se, quando tenti di eseguire il comando `jshell` , ricevi un errore che inizia con `Unable to locate an executable` , assicurati che `JAVA_HOME` sia impostato correttamente.

Importa predefinite

I seguenti pacchetti vengono importati automaticamente all'avvio di 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

Entrare e uscire da JShell

Avvio di JShell

Prima di provare ad avviare JShell, assicurati che la tua variabile di ambiente `JAVA_HOME` punti ad un'installazione di JDK 9. Per avviare JShell, eseguire il seguente comando:

```
$ jshell
```

Se tutto va bene, dovresti vedere un prompt di `jshell>` .

Chiusura di JShell

Per uscire da JShell, eseguire il seguente comando dal prompt di JShell:

```
jshell> /exit
```

espressioni

All'interno di JShell, puoi valutare le espressioni Java, con o senza punto e virgola. Questi possono variare da espressioni di base e dichiarazioni a quelli più complessi:

```
jshell> 4+2
jshell> System.out.printf("I am %d years old.\n", 421)
```

Anche i loop e i condizionali vanno bene:

```
jshell> for (int i = 0; i<3; i++) {
   ...> System.out.println(i);
   ...> }
```

È importante notare che le **espressioni all'interno dei blocchi devono avere il punto e virgola!**

variabili

È possibile dichiarare variabili locali all'interno di JShell:

```
jshell> String s = "hi"
jshell> int i = s.length
```

Tieni presente che le variabili possono essere ridichiarate con tipi diversi; questo è perfettamente valido in JShell:

```
jshell> String var = "hi"
```

```
jshell> int var = 3
```

Per visualizzare un elenco di variabili, immettere `/vars` al prompt di JShell.

Metodi e classi

È possibile definire metodi e classi all'interno di JShell:

```
jshell> void speak() {
   ...> System.out.println("hello");
   ...> }

jshell> class MyClass {
   ...> void doNothing() {}
   ...> }
```

Non sono necessari modificatori di accesso. Come con altri blocchi, è richiesto il punto e virgola all'interno dei corpi del metodo. Tieni presente che, come per le variabili, è possibile ridefinire metodi e classi. Per visualizzare un elenco di metodi o classi, immettere `/methods` o `/types` al prompt di JShell, rispettivamente.

Frammenti di modifica

L'unità di base del codice utilizzata da JShell è lo **snippet** o la **voce di origine** . Ogni volta che dichiari una variabile locale o definisci un metodo o una classe locale, crei uno snippet il cui nome è l'identificatore della variabile / metodo / classe. In qualsiasi momento puoi modificare uno snippet che hai creato con il comando `/edit` . Ad esempio, supponiamo di aver creato la classe `Foo` con una sola `bar` , metodo:

```
jshell> class Foo {
   ...> void bar() {
   ...> }
   ...> }
```

Ora, voglio riempire il corpo del mio metodo. Invece di riscrivere l'intera classe, posso modificarlo:

```
jshell> /edit Foo
```

Per impostazione predefinita, un editor oscillante apparirà con le funzionalità di base possibili. Tuttavia è possibile modificare l'editor utilizzato da JShell:

```
jshell> /set editor emacs
jshell> /set editor vi
jshell> /set editor nano
jshell> /set editor -default
```

Nota che se **la nuova versione dello snippet contiene errori di sintassi, potrebbe non essere salvata.** Allo stesso modo, uno snippet viene creato solo se la dichiarazione / definizione originale è sintatticamente corretta; il seguente non funziona:

```
jshell> String st = String 3
//error omitted
jshell> /edit st
|  No such snippet: st
```

Tuttavia, gli snippet possono essere compilati e quindi modificabili nonostante alcuni errori in fase di compilazione, come i tipi non corrispondenti: i seguenti lavori:

```
jshell> int i = "hello"
//error omitted
jshell> /edit i
```

Infine, gli snippet possono essere cancellati usando il 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)
|
```

Per eliminare tutti i frammenti, quindi resettare lo stato della JVM, usa `\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
|  ^
```

Leggi JShell online: https://riptutorial.com/it/java/topic/9511/jshell

Capitolo 102: JSON in Java

introduzione

JSON (JavaScript Object Notation) è un formato di scambio di dati leggero, basato su testo e indipendente dalla lingua, che è facile da leggere e scrivere per gli utenti e le macchine. JSON può rappresentare due tipi strutturati: oggetti e matrici. JSON viene spesso utilizzato nelle applicazioni Ajax, nelle configurazioni, nei database e nei servizi Web RESTful. L'API Java per JSON Processing fornisce API portatili per analizzare, generare, trasformare e interrogare JSON.

Osservazioni

Questo esempio si concentra sull'analisi e sulla creazione di JSON in Java utilizzando varie librerie come la libreria **Google Gson** , il Jackson Object Mapper e altri ..

Gli esempi che utilizzano altre librerie possono essere trovati qui: Come analizzare JSON in Java

Examples

Codifica dei dati come JSON

Se è necessario creare un oggetto `JSONObject` e inserire dati, considerare il seguente esempio:

```
// 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"]
   }
*/
```

Decodifica dei dati JSON

Se è necessario ottenere dati da un oggetto `JSONObject` , considerare il seguente esempio:

```
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);
```

optXXX vs getXXX metodi

`JSONObject` e `JSONArray` hanno alcuni metodi che sono molto utili mentre si tratta di una possibilità che un valore che stai cercando di ottenere non esiste o è di un altro 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"
```

Le stesse regole si applicano ai metodi `getXXX` / `optXXX` di `JSONArray` .

Object To JSON (Gson Library)

Supponiamo che tu abbia una classe chiamata `Person` con un solo `name`

```
private class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Codice:

```
Gson g = new Gson();

Person person = new Person("John");
System.out.println(g.toJson(person)); // {"name":"John"}
```

Ovviamente il vaso Gson deve trovarsi sul classpath.

JSON To Object (Gson Library)

Supponiamo che tu abbia una classe chiamata `Person` con un solo `name`

```
private class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Codice:

```
Gson gson = new Gson();
String json = "{\"name\": \"John\"}";

Person person = gson.fromJson(json, Person.class);
System.out.println(person.name); //John
```

Devi avere la libreria gson nel tuo classpath.

Estrai singolo elemento da 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
```

Utilizzando Jackson Object Mapper

Modello di Pojo

```
public class Model {
    private String firstName;
    private String lastName;
    private int age;
    /* Getters and setters not shown for brevity */
}
```

Esempio: stringa su oggetto

```
Model outputObject = objectMapper.readValue(
     "{\"firstName\":\"John\",\"lastName\":\"Doe\",\"age\":23}",
     Model.class);
System.out.println(outputObject.getFirstName());
//result: John
```

Esempio: oggetto da stringa

```
String jsonString = objectMapper.writeValueAsString(inputObject));
//result: {"firstName":"John","lastName":"Doe","age":23}
```

Dettagli

Importazione necessaria:

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

Dipendenza da Maven: jackson-databind

`ObjectMapper`

```
//creating one
ObjectMapper objectMapper = new ObjectMapper();
```

- `ObjectMapper` è thread-safe
- consigliato: avere un'istanza statica condivisa

deserializzazione:

```
<T> T readValue(String content, Class<T> valueType)
```

- `valueType` deve essere specificato - il ritorno sarà di questo tipo
- Genera
    - `IOException` : in caso di problemi I / O di basso livello

- `JsonParseException` - se l'input sottostante contiene contenuti non validi
- `JsonMappingException` - se la struttura JSON di input non corrisponde alla struttura dell'oggetto

Esempio di utilizzo (jsonString è la stringa di input):

```
Model fromJson = objectMapper.readValue(jsonString, Model.class);
```

Metodo per la serializzazione:

String writeValueAsString (Valore oggetto)

- Genera
  - `JsonProcessingException` in caso di errore
  - Nota: prima della versione 2.1, la clausola throws includeva IOException; 2.1 rimosso.

JSON Iteration

`JSONObject` su proprietà `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` su valori di `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: metodi di concatenamento

È possibile utilizzare il metodo di concatenamento mentre si lavora con `JSONObject` e `JSONArray` .

Esempio di 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

Se è necessario aggiungere una proprietà con un valore `null` , è necessario utilizzare il `JSONObject.NULL` finale statico predefinito e non il riferimento `null` standard di Java.

`JSONObject.NULL` è un valore sentinella utilizzato per definire esplicitamente una proprietà con un valore vuoto.

```
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
```

Che è una **chiara violazione** del contratto `Java.equals()` :

> Per qualsiasi valore di riferimento non nullo x, x.equals (null) deve restituire false

Elenco da JsonArray a Java (Gson Library)

Ecco un semplice JsonArray che vorresti convertire in una `ArrayList` Java:

```
{
    "list": [
                "Test_String_1",
                "Test_String_2"
            ]
}
```

Ora passa l''elenco' di `JsonArray` al seguente metodo che restituisce una corrispondente `ArrayList` Java:

```
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;
}
```

È necessario aggiungere la seguente dipendenza `POM.xml` file `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 dovresti avere il jar `com.google.code.gson:gson:jar:<version>` nel classpath.

Deserializzare la raccolta JSON nella collezione di oggetti usando Jackson

Supponi di avere una `Person` classe pojo

```
public class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

E tu vuoi analizzarlo in un array JSON o in una mappa di oggetti Person. A causa della cancellazione dei tipi non è possibile costruire classi di `List<Person>` e `Map<String, Person>` direttamente in runtime *(e quindi usarle per deserializzare JSON)* . Per superare questa limitazione, Jackson offre due approcci: `TypeFactory` e `TypeReference` .

TypeFactory

L'approccio qui utilizzato è quello di utilizzare una fabbrica (e la sua funzione di utilità statica) per creare il tuo tipo per te. I parametri necessari sono la raccolta che si desidera utilizzare (elenco, set, ecc.) E la classe che si desidera archiviare in quella raccolta.

TypeReference

L'approccio al tipo di riferimento sembra più semplice perché consente di risparmiare un po 'di digitazione e sembra più pulito. TypeReference accetta un parametro type, in cui si passa il tipo di `List<Person>` desiderato `List<Person>` . Basta istanziare questo oggetto TypeReference e usarlo come contenitore del tipo.

Ora vediamo come deserializzare realmente il tuo JSON in un oggetto Java. Se il tuo JSON è formattato come array, puoi deserializzare come elenco. Se esiste una struttura nidificata più complessa, è necessario deserializzare su una mappa. Vedremo esempi di entrambi.

Deserializzazione dell'array JSON

```
String jsonString = "[{\"name\": \"Alice\"}, {\"name\": \"Bob\"}]"
```

Approccio TypeFactory

```
CollectionType listType =
    factory.constructCollectionType(List.class, Person.class);
List<Preson> list = mapper.readValue(jsonString, listType);
```

Tipo Approccio di riferimento

```
TypeReference<Person> listType = new TypeReference<List<Person>>() {};
List<Person> list = mapper.readValue(jsonString, listType);
```

Deserializzazione della mappa JSON

```
String jsonString = "{\"0\": {\"name\": \"Alice\"}, \"1\": {\"name\": \"Bob\"}}"
```

Approccio TypeFactory

```
CollectionType mapType =
    factory.constructMapLikeType(Map.class, String.class, Person.class);
List<Person> list = mapper.readValue(jsonString, mapType);
```

Tipo Approccio di riferimento

```
TypeReference<Person> mapType = new TypeReference<Map<String, Person>>() {};
Map<String, Person> list = mapper.readValue(jsonString, mapType);
```

Dettagli

Importazione utilizzata:

```
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.type.CollectionType;
```

Istanze utilizzate:

```
ObjectMapper mapper = new ObjectMapper();
TypeFactory factory = mapper.getTypeFactory();
```

Nota

Mentre `TypeReference` approccio `TypeReference` può sembrare migliore ha diversi inconvenienti:

1. `TypeReference` dovrebbe essere istanziato usando una classe anonima

2. Dovresti fornire generica esplicita

Non riuscendo a farlo potrebbe portare alla perdita dell'argomento di tipo generico che porterà a un errore di deserializzazione.

Leggi JSON in Java online: https://riptutorial.com/it/java/topic/840/json-in-java

Capitolo 103: JVM Tool Interface

Osservazioni

JVM TM Tool Interface

Versione 1.2

http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

Examples

Iterare su oggetti raggiungibili dall'oggetto (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);
            }
        }
    }
}
```

Ottieni l'ambiente JVMTI

Dentro il metodo Agent_OnLoad:

```
jvmtiEnv* jvmti;
/* Get JVMTI environment */
vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
```

Esempio di inizializzazione all'interno del metodo 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;
}
```

Leggi JVM Tool Interface online: https://riptutorial.com/it/java/topic/3316/jvm-tool-interface

Capitolo 104: La classe java.util.Objects

Examples

Uso di base per il controllo nullo dell'oggetto

Per il metodo di controllo nullo

```
Object nullableObject = methodReturnObject();
if (Objects.isNull(nullableObject)) {
    return;
}
```

Per il metodo di controllo non nullo

```
Object nullableObject = methodReturnObject();
if (Objects.nonNull(nullableObject)) {
    return;
}
```

Il riferimento al metodo Objects.nonNull () utilizza nello stream api

Nel vecchio modo per la raccolta di assegni nulli

```
List<Object> someObjects = methodGetList();
for (Object obj : someObjects) {
    if (obj == null) {
        continue;
    }
    doSomething(obj);
}
```

Con il metodo `Objects.nonNull` e l'API Java8 Stream, possiamo fare quanto sopra in questo modo:

```
List<Object> someObjects = methodGetList();
someObjects.stream()
           .filter(Objects::nonNull)
           .forEach(this::doSomething);
```

Leggi La classe java.util.Objects online: https://riptutorial.com/it/java/topic/5768/la-classe-java-util-objects

Capitolo 105: Lambda Expressions

introduzione

Le espressioni Lambda forniscono un modo chiaro e conciso per implementare un'interfaccia a metodo singolo usando un'espressione. Ti permettono di ridurre la quantità di codice che devi creare e mantenere. Sebbene simili alle classi anonime, non hanno informazioni di tipo da soli. Il tipo di inferenza deve accadere.

I riferimenti al metodo implementano interfacce funzionali usando i metodi esistenti piuttosto che le espressioni. Appartengono anche alla famiglia Lambda.

Sintassi

- () -> {restituisce espressione; } // Zero-arity con il corpo della funzione per restituire un valore.
- () -> espressione // Abbreviazione per la dichiarazione di cui sopra; non esiste un punto e virgola per le espressioni.
- () -> {function-body} // Effetto collaterale nell'espressione lambda per eseguire operazioni.
- parameterName -> expression // Espressione lambda one-arity. Nelle espressioni lambda con un solo argomento, la parentesi può essere rimossa.
- (Digitare parameterName, Type secondParameterName, ...) -> expression // lambda che valuta un'espressione con i parametri elencati a sinistra
- (parameterName, secondParameterName, ...) -> expression // Stenografia che rimuove i tipi di parametri per i nomi dei parametri. Può essere utilizzato solo in contesti che possono essere dedotti dal compilatore in cui le dimensioni dell'elenco dei parametri corrispondono a una (e solo una) della dimensione delle interfacce funzionali previste.

Examples

Utilizzo delle espressioni Lambda per ordinare una raccolta

Liste di ordinamento

Prima di Java 8, era necessario implementare l'interfaccia `java.util.Comparator` con una classe anonima (o named) durante l'ordinamento di 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 partire da Java 8, la classe anonima può essere sostituita con un'espressione lambda. Si noti che i tipi per i parametri `p1` e `p2` possono essere omessi, in quanto il compilatore li dedurrà automaticamente:

```
Collections.sort(
    people,
    (p1, p2) -> p1.getFirstName().compareTo(p2.getFirstName())
);
```

L'esempio può essere semplificato utilizzando `Comparator.comparing` e i riferimenti ai metodi espressi utilizzando il simbolo `::` (doppio punto).

```
Collections.sort(
    people,
    Comparator.comparing(Person::getFirstName)
);
```

Un'importazione statica ci consente di esprimerlo in modo più conciso, ma è discutibile se questo migliora la leggibilità complessiva:

```
import static java.util.Collections.sort;
import static java.util.Comparator.comparing;
//...
sort(people, comparing(Person::getFirstName));
```

I comparatori costruiti in questo modo possono anche essere concatenati insieme. Ad esempio, dopo aver confrontato le persone con il loro nome, se ci sono persone con lo stesso nome, il metodo `thenComparing` con anche il confronto per cognome:

```
sort(people, comparing(Person::getFirstName).thenComparing(Person::getLastName));
```

---

1 - Si noti che Collections.sort (...) funziona solo su raccolte che sono sottotipi di `List` . Le API `Set` e `Collection` non implicano alcun ordine degli elementi.

Ordinare le mappe

È possibile ordinare le voci di una `HashMap` base al valore in modo simile. (Si noti che una `LinkedHashMap` deve essere utilizzata come destinazione. Le chiavi in una `HashMap` ordinaria non sono ordinate.)

```
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));
```


Interfacce funzionali

Lambdas può operare solo su un'interfaccia funzionale, che è un'interfaccia con un solo metodo astratto. Le interfacce funzionali possono avere un numero qualsiasi di metodi `default` o `static` . (Per questo motivo, a volte vengono definiti Interfacce di metodo astratto singolo o Interfacce 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();
}
```

Quando si dichiara un'interfaccia funzionale, è possibile aggiungere l'annotazione `@FunctionalInterface` . Ciò non ha alcun effetto speciale, ma verrà generato un errore del compilatore se questa annotazione viene applicata a un'interfaccia che non è funzionale, in modo da ricordare che l'interfaccia non deve essere modificata.

```
@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
}
```

Viceversa, questa **non** è un'interfaccia funzionale, poiché ha più di **un** metodo **astratto** :

```
interface BadFoo {
    void bar();
    void quux(); // <-- Second method prevents lambda: which one should
                 // be considered as lambda?
}
```

Anche questa **non è** un'interfaccia funzionale, in quanto non ha alcun metodo:

```
interface BlankFoo2 { }
```

Prendi nota di quanto segue. Supponiamo di avere

```
interface Parent { public int parentMethod(); }
```

e

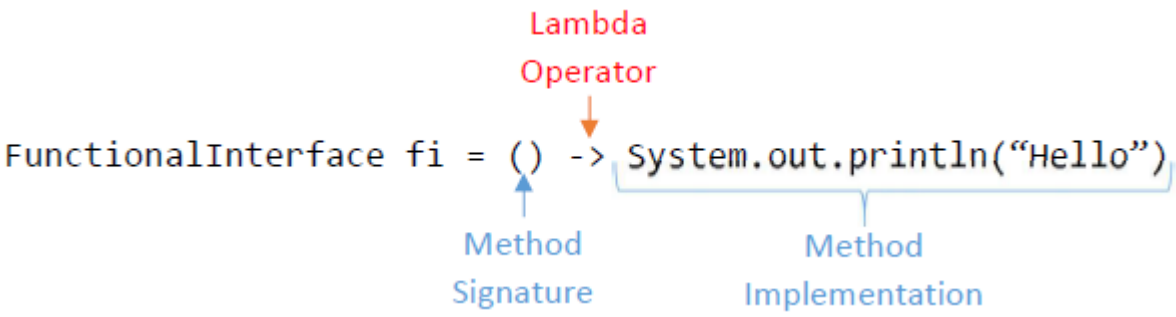
```
interface Child extends Parent { public int ChildMethod(); }
```

Quindi `Child` **non può** essere un'interfaccia funzionale poiché ha due metodi specificati.

Java 8 fornisce anche un certo numero di interfacce funzionali basate su modelli generici nel pacchetto `java.util.function` . Ad esempio, l'interfaccia incorporata `Predicate<T>` esegue il wrapping di un singolo metodo che immette un valore di tipo `T` e restituisce un valore `boolean` .

Lambda Expressions

La struttura di base di un'espressione Lambda è:



`fi` manterrà quindi un'istanza singleton di una classe, simile a una classe anonima, che implementa `FunctionalInterface` e in cui la definizione del metodo uno è `{ System.out.println("Hello"); }` . In altre parole, quanto sopra è principalmente equivalente a:

```
FunctionalInterface fi = new FunctionalInterface() {
    @Override
    public void theOneMethod() {
        System.out.println("Hello");
    }
};
```

Il lambda è solo "per lo più equivalente" alla classe anonima perché in un lambda, il significato di espressioni come `this` , `super` o `toString()` riferimento alla classe in cui si svolge l'assegnazione, non all'oggetto appena creato.

Non è possibile specificare il nome del metodo quando si utilizza un lambda, ma non è necessario, perché un'interfaccia funzionale deve avere solo un metodo astratto, quindi Java lo sostituisce.

Nei casi in cui il tipo di lambda non è certo, (ad esempio metodi sovraccaricati) è possibile aggiungere un cast al lambda per dire al compilatore quale dovrebbe essere il suo tipo, in questo modo:

```
Object fooHolder = (Foo1) () -> System.out.println("Hello");
System.out.println(fooHolder instanceof Foo1); // returns true
```

Se il metodo singolo dell'interfaccia funzionale prende i parametri, i nomi formali locali di questi dovrebbero apparire tra le parentesi del lambda. Non è necessario dichiarare il tipo del parametro o return in quanto questi sono presi dall'interfaccia (sebbene non sia un errore dichiarare i tipi di parametro se lo si desidera). Quindi, questi due esempi sono equivalenti:

```
Foo2 longFoo = new Foo2() {
    @Override
    public int bar(boolean baz) {
        return baz ? 1 : 0;
    }
};
Foo2 shortFoo = (x) -> { return x ? 1 : 0; };
```

Le parentesi intorno all'argomento possono essere omesse se la funzione ha solo un argomento:

```
Foo2 np = x -> { return x ? 1 : 0; }; // okay
Foo3 np2 = x, y -> x.toString() + y // not okay
```

Ritorni impliciti

Se il codice inserito all'interno di una lambda è *un'espressione* Java anziché *un'istruzione* , viene considerato come un metodo che restituisce il valore dell'espressione. Quindi, i due seguenti sono equivalenti:

```
IntUnaryOperator addOneShort = (x) -> (x + 1);
IntUnaryOperator addOneLong = (x) -> { return (x + 1); };
```

Accesso alle variabili locali (chiusure di valore)

Poiché lambdas è una sintassi sintattica per le classi anonime, esse seguono le stesse regole per accedere alle variabili locali nell'ambito di inclusione; le variabili devono essere considerate come `final` e non modificate all'interno della 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
}
```

Se è necessario avvolgere una variabile variabile in questo modo, deve essere utilizzato un oggetto regolare che conserva una copia della variabile. Maggiori informazioni in Java Chiusure con espressioni lambda.

Accettare Lambdas

Poiché un lambda è un'implementazione di un'interfaccia, non occorre fare nulla di speciale per far sì che un metodo accetti un lambda: qualsiasi funzione che prende un'interfaccia funzionale può anche accettare un lambda.

```
public void passMeALambda(Foo1 f) {
    f.bar();
}
passMeALambda(() -> System.out.println("Lambda called"));
```

Il tipo di espressione lambda

Un'espressione lambda, di per sé, non ha un tipo specifico. Se è vero che i tipi e il numero di parametri, insieme al tipo di un valore di ritorno può trasmettere alcune informazioni di tipo, tali informazioni vincoleranno solo i tipi a cui può essere assegnato. Lambda riceve un tipo quando viene assegnato a un tipo di interfaccia funzionale in uno dei seguenti modi:

- Assegnazione diretta a un tipo funzionale, ad es. `myPredicate = s -> s.isEmpty()`
- Passandolo come parametro che ha un tipo funzionale, ad esempio `stream.filter(s -> s.isEmpty())`
- Restituirlo da una funzione che restituisce un tipo funzionale, ad esempio `return s -> s.isEmpty()`
- Trasmetterlo a un tipo funzionale, ad es. `(Predicate<String>) s -> s.isEmpty()`

Fino a quando non viene effettuata alcuna assegnazione a un tipo funzionale, la lambda non ha un tipo definito. Per illustrare, considera l'espressione lambda `o -> o.isEmpty()` . La stessa espressione lambda può essere assegnata a molti tipi funzionali diversi:

```
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();
```

Ora che sono assegnati, gli esempi mostrati sono di tipi completamente diversi anche se le espressioni lambda erano uguali e non possono essere assegnate l'una all'altra.

Riferimenti al metodo

I riferimenti al metodo consentono metodi predefiniti statici o di istanza che aderiscono a un'interfaccia funzionale compatibile da passare come argomenti anziché un'espressione lambda anonima.

Supponiamo di avere un modello:

```
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();
```

Riferimento al metodo di istanza (a un'istanza arbitraria)

```
people.stream().map(Person::getName)
```

Il lambda equivalente:

```
people.stream().map(person -> person.getName())
```

In questo esempio, viene passato un riferimento al metodo di istanza `getName()` di tipo `Person` . Poiché è noto che è del tipo di raccolta, verrà richiamato il metodo sull'istanza (noto in seguito).

Riferimento al metodo di istanza (a un'istanza specifica)

```
people.forEach(System.out::println);
```

Poiché `System.out` è un'istanza di `PrintStream` , un riferimento al metodo a questa specifica istanza viene passato come argomento.

Il lambda equivalente:

```
people.forEach(person -> System.out.println(person));
```

Riferimento al metodo statico

Anche per la trasformazione degli stream possiamo applicare riferimenti a metodi statici:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
numbers.stream().map(String::valueOf)
```

Questo esempio passa un riferimento al metodo statico `valueOf()` sul tipo `String` . Pertanto, l'oggetto istanza nella raccolta viene passato come argomento a `valueOf()` .

Il lambda equivalente:

```
 numbers.stream().map(num -> String.valueOf(num))
```

Riferimento a un costruttore

```
List<String> strings = Arrays.asList("1", "2", "3");
strings.stream().map(Integer::new)
```

Leggi Raccogli elementi di un flusso in una raccolta per vedere come raccogliere elementi da raccogliere.

Qui viene utilizzato il costruttore di argomento String singolo del tipo `Integer` , per costruire un intero dato la stringa fornita come argomento. In questo caso, finché la stringa rappresenta un numero, lo stream verrà mappato su Integers. Il lambda equivalente:

```
strings.stream().map(s -> new Integer(s));
```

Cheat-sheet

Metodo di riferimento Formato	Codice	Lambda equivalente
Metodo statico	`TypeName::method`	`(args) -> TypeName.method(args)`
Metodo non statico (su istanza *)	`instance::method`	`(args) -> instance.method(args)`
Metodo non statico (nessuna istanza)	`TypeName::method`	`(instance, args) -> instance.method(args)`
Costruttore **	`TypeName::new`	`(args) -> new TypeName(args)`

Metodo di riferimento Formato	Codice	Lambda equivalente
Costruttore di array	`TypeName[]::new`	`(int size) -> new TypeName[size]`

* `instance` può essere qualsiasi espressione che valuta un riferimento a un'istanza, ad esempio `getInstance()::method` , `this::method`

** Se `TypeName` è una classe interna non statica, il riferimento del costruttore è valido solo nell'ambito di un'istanza di classe esterna

Implementazione di più interfacce

A volte potresti voler avere un'espressione lambda che implementa più di un'interfaccia. Questo è utile soprattutto con le interfacce marker (come java.io.Serializable) poiché non aggiungono metodi astratti.

Ad esempio, si desidera creare un `TreeSet` di `TreeSet` con un `Comparator` personalizzato, quindi serializzarlo e inviarlo tramite la rete. L'approccio banale:

```
TreeSet<Long> ts = new TreeSet<>((x, y) -> Long.compare(y, x));
```

non funziona poiché il lambda per il comparatore non implementa `Serializable` . È possibile risolvere questo problema utilizzando i tipi di intersezione e specificando esplicitamente che questo lambda deve essere serializzabile:

```
TreeSet<Long> ts = new TreeSet<>(
    (Comparator<Long> & Serializable) (x, y) -> Long.compare(y, x));
```

Se si utilizzano spesso tipi di intersezione (ad esempio, se si utilizza un framework come Apache Spark in cui quasi tutto deve essere serializzabile), è possibile creare interfacce vuote e utilizzarle invece nel codice:

```
public interface SerializableComparator extends Comparator<Long>, Serializable {}

public class CustomTreeSet {
  public CustomTreeSet(SerializableComparator comparator) {}
}
```

In questo modo hai la certezza che il comparatore passato sarà serializzabile.

Lambdas ed Execute-around Pattern

Esistono diversi buoni esempi di utilizzo di lambda come FunctionalInterface in scenari semplici. Un caso d'uso abbastanza comune che può essere migliorato da lambda è quello che viene chiamato il modello Execute-Around. In questo modello, hai un set di codice standard di setup / teardown che è necessario per più scenari che circondano il codice caso d'uso specifico. Alcuni esempi comuni di questo sono file io, database io, blocchi try / catch.

```
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();
    }
}
```

Quindi chiamare questo metodo con un lambda potrebbe essere simile a:

```
public static void updateMyDAO(MyVO vo) throws DatabaseException {
    doProcessing((Connection conn) -> MyDAO.update(conn, ObjectMapper.map(vo)));
}
```

Questo non è limitato alle operazioni di I / O. Può essere applicato a qualsiasi scenario in cui attività simili di impostazione / eliminazione siano applicabili con variazioni minori. Il principale vantaggio di questo Pattern è il riutilizzo del codice e l'applicazione di DRY (Do not Repeat Yourself).

Usando espressione lambda con la tua interfaccia funzionale

Lambdas ha lo scopo di fornire un codice di implementazione inline per le interfacce a singolo metodo e la capacità di passarle come abbiamo fatto con le variabili normali. Li chiamiamo interfaccia funzionale.

Ad esempio, scrivendo un Runnable in una classe anonima e iniziando una discussione assomiglia a:

```
//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();
```

Ora, in linea con sopra, diciamo che hai qualche interfaccia personalizzata:

```
interface TwoArgInterface {
    int operate(int a, int b);
}
```

Come usi lambda per dare l'implementazione di questa interfaccia nel tuo codice? Come nell'esempio Runnable mostrato sopra. Vedi il programma del driver qui sotto:

```
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` restituisce solo il lambda, non il metodo esterno

Il metodo di `return` ritorna solo dal lambda, non dal metodo esterno.

Fai attenzione che questo è *diverso* da Scala e 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!
  });
}
```

Questo può portare a un comportamento imprevisto durante il tentativo di scrivere propri costrutti del linguaggio, come in costrutti built come ad esempio `for` i cicli `return` comporta in modo diverso:

```
void demo2() {
  for (int i = 0; i < 3; i++) {
    System.out.println(i);
    return; // Return from 'demo2' entirely
  }
}
```

In Scala e Kotlin, `demo` e `demo2` avrebbero entrambi stampato solo `0` . Ma questo *non* è più coerente. L'approccio Java è coerente con il refactoring e l'uso delle classi - il `return` nel codice nella parte superiore, e il codice sottostante si comporta allo stesso modo:

```
void demo3() {
  threeTimes(new MyIntConsumer());
}

class MyIntConsumer implements IntConsumer {
  public void accept(int i) {
```

```
    System.out.println(i);
    return;
  }
}
```

Pertanto, il `return` Java è più coerente con i metodi di classe e il refactoring, ma meno con i builtin `for` e `while` , rimangono speciali.

Per questo motivo, i seguenti due sono equivalenti in 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);
```

Inoltre, l'utilizzo di try-with-resources è sicuro in 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.
  });
}
```

stamperà `before accept() after close()` . Nella semantica di Scala e Kotlin, il try-with-resources non sarebbe stato chiuso, ma sarebbe stampato solo `before accept()` .

Chiusure Java con espressioni lambda.

Una chiusura lambda viene creata quando un'espressione lambda fa riferimento alle variabili di un ambito che racchiude (globale o locale). Le regole per farlo sono le stesse di quelle per i metodi inline e le classi anonime.

Le variabili locali da un ambito che viene utilizzato all'interno di un lambda devono essere `final` . Con Java 8 (la prima versione che supporta lambda), non è necessario *dichiararli* `final` nel contesto esterno, ma devono essere trattati in questo modo. Per esempio:

```
int n = 0; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
```

```
    int i = n;
    // do something
};
```

Questo è legale finché il valore della variabile `n` non è cambiato. Se provi a cambiare la variabile, all'interno o all'esterno di lambda, otterrai il seguente errore di compilazione:

> "Le variabili locali referenziate da un'espressione lambda devono essere *definitive* o *effettivamente definitive* ".

Per esempio:

```
int n = 0;
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
n++; // Will generate an error.
```

Se è necessario utilizzare una variabile variabile all'interno di una lambda, l'approccio normale è quello di dichiarare una copia `final` della variabile e utilizzare la copia. Per esempio

```
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, il corpo della lambda non vede le modifiche alla variabile originale.

Si noti che Java non supporta le chiusure reali. Un lambda Java non può essere creato in un modo che gli consenta di vedere i cambiamenti nell'ambiente in cui è stato istanziato. Se vuoi implementare una chiusura che osserva o apporta modifiche al suo ambiente, devi simularlo usando una classe regolare. Per esempio:

```
// Does not compile ...
public IntUnaryOperator createAccumulator() {
    int value = 0;
    IntUnaryOperator accumulate = (x) -> { value += x; return value; };
    return accumulate;
}
```

L'esempio sopra non verrà compilato per le ragioni discusse in precedenza. Possiamo aggirare l'errore di compilazione come segue:

```
// Compiles, but is incorrect ...
public class AccumulatorGenerator {
    private int value = 0;

    public IntUnaryOperator createAccumulator() {
```

```
        IntUnaryOperator accumulate = (x) -> { value += x; return value; };
        return accumulate;
    }
}
```

Il problema è che questo rompe il contratto di design per l'interfaccia `IntUnaryOperator` che afferma che le istanze dovrebbero essere funzionali e senza stato. Se tale chiusura viene passata a funzioni integrate che accettano oggetti funzionali, è probabile che causi arresti anomali o comportamenti errati. Le chiusure che incapsulano lo stato mutabile dovrebbero essere implementate come classi regolari. Per esempio.

```
// Correct ...
public class Accumulator {
   private int value = 0;

   public int accumulate(int x) {
      value += x;
      return value;
   }
}
```

Lambda - Esempio di listener

Ascoltatore di classe anonima

Prima di Java 8, è molto comune che una classe anonima venga utilizzata per gestire l'evento click di un JButton, come mostrato nel codice seguente. Questo esempio mostra come implementare un listener anonimo nell'ambito di `btn.addActionListener` .

```
JButton btn = new JButton("My Button");
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button was pressed");
    }
});
```

Ascoltatore Lambda

Poiché l'interfaccia `ActionListener` definisce solo un metodo `actionPerformed()` , è un'interfaccia funzionale che significa che c'è un posto dove utilizzare le espressioni Lambda per sostituire il codice boilerplate. L'esempio sopra può essere riscritto usando espressioni Lambda come segue:

```
JButton btn = new JButton("My Button");
btn.addActionListener(e -> {
    System.out.println("Button was pressed");
});
```

Stile tradizionale in stile Lambda

Modo tradizionale

```
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));
    }
}
```

Stile Lambda

1. Rimuovi il nome della classe e il corpo dell'interfaccia funzionale.

```
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. Dichiarazione di tipo opzionale

```
MathOperation isEven = (num) -> {
    return num%2 == 0;
};
```

3. Parentesi opzionale attorno al parametro, se si tratta di un parametro singolo

```
MathOperation isEven = num -> {
    return num%2 == 0;
};
```

4. Parentesi graffe facoltative, se c'è una sola linea nel corpo della funzione
5. Parola chiave di restituzione facoltativa, se esiste una sola riga nel corpo della funzione

```
MathOperation isEven = num -> num%2 == 0;
```

Lambda e utilizzo della memoria

Dal momento che i lambda Java sono chiusure, possono "catturare" i valori delle variabili

nell'accluso ambito lessicale. Mentre non tutti i lambda catturano qualcosa - i lambda semplici come `s -> s.length()` non catturano nulla e sono chiamati *apolidi* - i lambda di cattura richiedono un oggetto temporaneo per contenere le variabili catturate. In questo snippet di codice, lambda `() -> j` è un lambda di cattura e può causare l'allocazione di un oggetto quando viene valutato:

```
public static void main(String[] args) throws Exception {
    for (int i = 0; i < 1000000000; i++) {
        int j = i;
        doSomethingWithLambda(() -> j);
    }
}
```

Anche se potrebbe non essere immediatamente evidente dal momento che la `new` parola chiave non appare in nessun punto dello snippet, questo codice è suscettibile di creare 1.000.000.000 di oggetti separati per rappresentare le istanze dell'espressione `() -> j` lambda. Tuttavia, va anche notato che le versioni future di Java [1] potrebbero essere in grado di ottimizzarlo in modo che *durante il runtime* le istanze lambda venissero riutilizzate o rappresentate in altro modo.

---

1 - Ad esempio, Java 9 introduce una fase facoltativa di "collegamento" alla sequenza di build Java che fornirà l'opportunità di eseguire ottimizzazioni globali come questa.

Usando espressioni lambda e predicati per ottenere un determinato valore (s) da un elenco

A partire da Java 8, è possibile utilizzare espressioni lambda e predicati.

Esempio: utilizzare espressioni lambda e un predicato per ottenere un determinato valore da un elenco. In questo esempio ogni persona verrà stampata con il fatto se ha 18 anni o meno.

Classe di appartenenza:

```
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; }
}
```

L'interfaccia integrata Predicato dai pacchetti java.util.function.Predicate è un'interfaccia funzionale con un metodo di `boolean test(T t)` .

Esempio di utilizzo:

```
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.");
            }
        }
    }
}
```

La `print(personList, p -> p.getAge() >= 18);` il metodo utilizza un'espressione lambda (poiché il predicato viene utilizzato come parametro) in cui è possibile definire l'espressione necessaria. Il metodo di verifica del correttore verifica se questa espressione è corretta o meno: `checker.test(person)` .

Puoi facilmente cambiarlo in qualcos'altro, ad esempio per `print(personList, p -> p.getName().startsWith("J"));` . Questo controllerà se il nome della persona inizia con una "J".

Leggi Lambda Expressions online: https://riptutorial.com/it/java/topic/91/lambda-expressions

Capitolo 106: letterali

introduzione

Un letterale Java è un elemento sintattico (cioè qualcosa che si trova nel *codice sorgente* di un programma Java) che rappresenta un valore. Gli esempi sono `1` , `0.333F` , `false` , `'X'` e `"Hello world\n"` .

Examples

Letterali esadecimali, ottali e binari

Un numero `hexadecimal` è un valore in base-16. Ci sono 16 cifre, `0-9` e le lettere `AF` (caso non importa). `AF` rappresenta `10-16` .

Un numero `octal` è un valore in base-8 e utilizza le cifre `0-7` .

Un numero `binary` è un valore in base-2 e utilizza le cifre `0` e `1` .

Tutti questi numeri hanno lo stesso valore, `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
```

Si noti che la sintassi binaria letterale è stata introdotta in Java 7.

Il letterale ottale può facilmente essere una trappola per errori semantici. Se definisci uno `'0'` ai tuoi valori decimali, otterrai il valore errato:

```
int a = 0100;        // Instead of 100, a == 64
```

Usando il trattino basso per migliorare la leggibilità

Da Java 7 è stato possibile utilizzare uno o più caratteri di sottolineatura (_) per separare gruppi di cifre in un numero letterale primitivo per migliorarne la leggibilità.

Ad esempio, queste due dichiarazioni sono equivalenti:

Java SE 7

```
int i1 = 123456;
int i2 = 123_456;
System.out.println(i1 == i2); // true
```

Questo può essere applicato a tutti i numeri letterali primitivi come mostrato di seguito:

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;
```

Questo funziona anche usando i prefissi per basi binarie, ottali ed esadecimali:

Java SE 7

```
short binary= 0b0_1_0_1;
int octal = 07_7_7_7_7_7_7_0;
long hexBytes = 0xFF_EC_DE_5E;
```

Ci sono alcune regole sui caratteri di sottolineatura che **vietano il** loro posizionamento nei seguenti luoghi:

- All'inizio o alla fine di un numero (ad es. `_123` o `123_` *non* sono validi)
- Adiacente a un punto decimale in un letterale in virgola mobile (ad es. `1._23` o `1_.23` *non* sono validi)
- Prima di un suffisso F o L (ad es. `1.23_F` o `9999999_L` *non* sono validi)
- Nelle posizioni in cui è prevista una stringa di cifre (ad es. `0_xFFFF` *non* è valido)

Fuga di sequenze in letterali

Stringhe e caratteri letterali forniscono un meccanismo di escape che consente di esprimere codici di caratteri che altrimenti non sarebbero consentiti nel letterale. Una sequenza di escape consiste in un carattere barra rovesciata (`\`) seguito da uno o più altri caratteri. Le stesse sequenze sono valide in entrambi i caratteri e in stringhe letterali.

L'insieme completo di sequenze di escape è il seguente:

Sequenza di fuga	Senso
`\\`	Indica un carattere barra rovesciata (`\`)
`\'`	Denota un carattere a virgoletta singola (`'`)
`\"`	Denota un carattere a doppia virgola (`"`)
`\n`	Indica un carattere di avanzamento riga (`LF`)
`\r`	Indica un carattere di ritorno a `CR` (`CR`)
`\t`	Denota un carattere di tabulazione orizzontale (`HT`)
`\f`	Indica un carattere di avanzamento modulo (`FF`)
`\b`	Denota un carattere backspace (`BS`)

Sequenza di fuga	Senso
`\<octal>`	Denota un codice di carattere compreso tra 0 e 255.

Il `<octal>` in quanto sopra è costituito da una, due o tre cifre ottali (da '0' a '7') che rappresentano un numero compreso tra 0 e 255 (decimale).

Si noti che una barra rovesciata seguita da qualsiasi altro carattere è una sequenza di escape non valida. Le sequenze di escape non valide vengono trattate come errori di compilazione dal JLS.

Riferimento:

- JLS 3.10.6. Sequenze di escape per caratteri e stringhe letterali

Escape Unicode

Oltre alle sequenze di escape di stringhe e caratteri descritte sopra, Java ha un meccanismo di escape di Unicode più generale, come definito in JLS 3.3. Unicode Escapes . Una escape Unicode ha la seguente sintassi:

```
'\' 'u' <hex-digit> <hex-digit> <hex-digit> <hex-digit>
```

dove `<hex-digit>` è uno tra `'0'` , `'1'` , `'2'` , `'3'` , `'4'` , `'5'` , `'6'` , `'7'` , `'8'` , `'9'` , `'a'` , `'b'` , `'c'` , `'d'` , `'e'` , `'f'` , `'A'` , `'B'` , `'C'` , `'D'` , `'E'` , `'F'` .

Un'uscita Unicode viene mappata dal compilatore Java a un carattere (in senso stretto *un'unità di codice* Unicode a 16 bit) e può essere utilizzata ovunque nel codice sorgente in cui il carattere mappato è valido. È comunemente usato in caratteri e stringhe letterali quando è necessario rappresentare un carattere non ASCII in un letterale.

Escaping in regex

TBD

Letterali decimali interi

I valori letterali interi forniscono valori che possono essere utilizzati laddove è necessaria un'istanza `byte` , `short` , `int` , `long` o `char` . (Questo esempio si concentra sulle semplici forme decimali.Altri esempi spiegano come eseguire letterali in ottale, esadecimale e binario e l'uso di caratteri di sottolineatura per migliorare la leggibilità.)

Letterali interi ordinari

La forma più semplice e più comune di valori letterali interi è un valore letterale intero decimale. Per esempio:

```
 0       // The decimal number zero       (type 'int')
```

```
1       // The decimal number one        (type 'int')
42      // The decimal number forty two  (type 'int')
```

Devi stare attento con gli zeri iniziali. Uno zero iniziale causa un'interpretazione letterale integer da interpretare come *ottale* non decimale.

```
077    // This literal actually means 7 x 8 + 7 ... or 63 decimal!
```

I valori letterali interi non sono firmati. Se vedete qualcosa di simile a `-10` o `+10` , queste sono in realtà *espressioni* utilizzando i unari `-` e unari `+` operatori.

L'intervallo di valori letterali interi di questo modulo ha un tipo intrinseco di `int` e deve rientrare nell'intervallo compreso tra zero e 2^{31} o 2.147.483.648.

Notare che 2^{31} è `1` maggiore di `Integer.MAX_VALUE` . Letterali da 0 fino alla `2147483647` può essere utilizzato ovunque, ma è un errore di compilazione di utilizzare `2147483648` senza un unario precedente `-` operatore. (In altre parole, è riservato per esprimere il valore di `Integer.MIN_VALUE` .)

```
int max = 2147483647;     // OK
int min = -2147483648;    // OK
int tooBig = 2147483648;  // ERROR
```

Letterali interi lunghi

I letterali di tipo `long` vengono espressi aggiungendo un suffisso `L` Per esempio:

```
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
```

Nota che la distinzione tra `int` e letterali `long` è significativa in altri posti. Per esempio

```
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.
```

Riferimento: JLS 3.10.1 - Integer Literals

Letterali booleani

I letterali booleani sono il più semplice dei letterali nel linguaggio di programmazione Java. I due possibili valori `boolean` sono rappresentati dai letterali `true` e `false` . Questi sono case-sensitive. Per esempio:

```
boolean flag = true;    // using the 'true' literal
flag = false;           // using the 'false' literal
```

Stringhe letterali

Le stringhe letterali forniscono il modo più conveniente per rappresentare i valori stringa nel codice sorgente Java. Una stringa letterale consiste di:

- Un carattere di apertura doppia citazione (`"`).
- Zero o più altri caratteri che non sono né una virgoletta o un carattere di interruzione di riga. (Un carattere barra rovesciata (`\`) altera il significato dei caratteri successivi, vedi Escape sequenze in letterali .)
- Un carattere di doppia citazione di chiusura.

Per esempio:

```
"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
```

Si noti che una singola stringa letterale non può estendersi su più righe di codice sorgente. Si tratta di un errore di compilazione per un'interruzione di riga (o la fine del file di origine) che si verifica prima della doppia virgola di chiusura di un letterale. Per esempio:

```
"Jello world    // Compilation error (at the end of the line!)
```

Corde lunghe

Se hai bisogno di una stringa troppo lunga per adattarsi a una linea, il modo convenzionale per esprimerlo è dividerla in più letterali e utilizzare l'operatore di concatenazione (`+`) per unire i pezzi. Per esempio

```
String typingPractice = "The quick brown fox " +
                        "jumped over " +
                        "the lazy dog"
```

Un'espressione come la precedente consistente in string letterali e `+` soddisfa i requisiti per essere un'espressione costante . Ciò significa che l'espressione verrà valutata dal compilatore e rappresentata in fase di esecuzione da un singolo oggetto `String` .

Interning di stringhe letterali

Quando il file di classe contenente stringhe letterali viene caricato da JVM, gli oggetti `String` corrispondenti vengono *internati* dal sistema di runtime. Ciò significa che una stringa letterale utilizzata in più classi non occupa più spazio rispetto a se fosse utilizzata in una classe.

Per ulteriori informazioni sull'internamento e sul pool di stringhe, fare riferimento al pool di stringhe e all'heap storage esempio nell'argomento Strings.

Il letterale Null

Il valore letterale Null (scritto come `null`) rappresenta l'unico valore del tipo null. Ecco alcuni esempi

```
MyClass object = null;
MyClass[] objects = new MyClass[]{new MyClass(), null, new MyClass()};

myMethod(null);

if (objects != null) {
     // Do something
}
```

Il tipo null è piuttosto inusuale. Non ha nome, quindi non puoi esprimerlo nel codice sorgente Java. (E non ha nemmeno una rappresentazione runtime.)

L'unico scopo del tipo nullo è di essere il tipo di `null` . È un'assegnazione compatibile con tutti i tipi di riferimento e può essere digitata per qualsiasi tipo di riferimento. (In quest'ultimo caso, il cast non comporta un controllo del tipo di runtime.)

Infine, `null` ha la proprietà che `null instanceof <SomeReferenceType>` valuterà in `false` , indipendentemente dal tipo.

Letterali in virgola mobile

I valori letterali in virgola mobile forniscono valori che possono essere utilizzati laddove è necessario un `float` o una `double` istanza. Esistono tre tipi di letterale in virgola mobile.

- Semplici forme decimali
- Forme decimali ridimensionate
- Forme esadecimali

(Le regole di sintassi JLS combinano le due forme decimali in un unico modulo. Li trattiamo separatamente per facilitare la spiegazione.)

Esistono tipi letterali distinti per `float` e `double` literals, espressi usando i suffissi. Le varie forme usano le lettere per esprimere cose diverse. Queste lettere non fanno distinzione tra maiuscole e minuscole.

Semplici forme decimali

La forma più semplice di letterale in virgola mobile consiste di una o più cifre decimali e un punto decimale (`.`) E un suffisso opzionale (`f` , `F` , `d` o `D`). Il suffisso opzionale consente di specificare che il valore letterale è un valore `float` (`f` o `F`) o `double` (`d` o `D`). L'impostazione predefinita (quando non viene specificato alcun suffisso) è `double` .

Per esempio

```
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)
```

Infatti, le cifre decimali seguite da un suffisso sono anche letterali in virgola mobile.

```
1F      // means the same thing as 1.0F
```

Il significato di un valore letterale decimale è il numero in virgola mobile IEEE *più vicino* al numero reale matematico di precisione infinita indicato dalla forma decimale in virgola mobile. Questo valore concettuale viene convertito in rappresentazione in virgola mobile binaria IEEE usando *round to nearest* . (La semantica precisa della conversione decimale è specificata in `Double.valueOf(String)` per `Double.valueOf(String)` e `Float.valueOf(String)` , tenendo presente che ci sono differenze nelle sintassi dei numeri.)

Forme decimali ridimensionate

Le forme decimali in scala consistono in semplici decimali con una parte esponenziale introdotta da una `E` o `e` , seguita da un intero con segno. La parte esponente è una mano corta per moltiplicare la forma decimale di un potere di dieci, come mostrato negli esempi qui sotto. Esiste anche un suffisso opzionale per distinguere i letterali `float` e `double` . Ecco alcuni esempi:

```
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)
```

La dimensione di un letterale è limitata dalla rappresentazione (`float` o `double`). È un errore di compilazione se il fattore di scala genera un valore troppo grande o troppo piccolo.

Forme esadecimali

A partire da Java 6, è possibile esprimere valori letterali in virgola mobile in formato esadecimale. La forma esadecimale ha una sintassi analoga alle forme decimali semplici e in scala con le seguenti differenze:

1. Ogni letterale in virgola mobile esadecimale inizia con uno zero (`0`) e quindi con una `x` o una `X`
2. Le cifre del numero (ma *non* la parte esponenziale!) Includono anche le cifre esadecimali da `a` a `f` e i loro equivalenti maiuscoli.
3. L'esponente è *obbligatorio* e viene introdotto dalla lettera `p` (o `P`) invece di una `e` o `E` L'esponente rappresenta un fattore di scala che è una potenza di 2 invece di una potenza di 10.

Ecco alcuni esempi:

```
0x0.0p0f    // this is zero expressed in hexadecimal form (`float`)
0xff.0p19   // this is 255.0 x 2^19 (`double`)
```

Consiglio: poiché le forme esadecimali a virgola mobile non sono familiari alla maggior parte dei programmatori Java, è consigliabile utilizzarle con parsimonia.

sottolineatura

A partire da Java 7, i caratteri di sottolineatura sono consentiti all'interno delle stringhe di cifre in tutte e tre le forme di letterale in virgola mobile. Questo vale anche per le parti "esponenti". Vedi Usare caratteri di sottolineatura per migliorare la leggibilità .

Casi speciali

È un errore di compilazione se un letterale in virgola mobile denota un numero troppo grande o troppo piccolo per rappresentare nella rappresentazione selezionata; vale a dire se il numero sarebbe traboccare a + INF o -INF, o underflow a 0.0. Tuttavia, è legale per un valore letterale rappresentare un numero denormalizzato diverso da zero.

La sintassi letterale in virgola mobile non fornisce rappresentazioni letterali per valori speciali IEEE 754 come i valori INF e NaN. Se è necessario esprimerli nel codice sorgente, il metodo consigliato è utilizzare le costanti definite da `java.lang.Float` e `java.lang.Double` ; ad esempio, `Float.NaN` , `Float.NEGATIVE_INFINITY` e `Float.POSITIVE_INFINITY` .

Caratteri letterali

I caratteri letterali forniscono il modo più conveniente per esprimere valori di `char` nel codice sorgente Java. Un carattere letterale consiste di:

- Un carattere di apertura singola (`'`).
- Una rappresentazione di un personaggio. Questa rappresentazione non può essere una virgoletta singola o un carattere di interruzione di riga, ma può essere una sequenza di escape introdotta da un carattere barra rovesciata (`\`); vedi Escape sequenze in letterali .
- Un carattere di chiusura singola (`'`) di chiusura.

Per esempio:

```
char a = 'a';
char doubleQuote = '"';
char singleQuote = '\'';
```

Un'interruzione di riga in un carattere letterale è un errore di compilazione:

```
char newline = '
// Compilation error in previous line
char newLine = '\n';  // Correct
```

Leggi letterali online: https://riptutorial.com/it/java/topic/8250/letterali

Capitolo 107: Lettori e scrittori

introduzione

Lettori e Scrittori e le rispettive sottoclassi forniscono I / O semplice per i dati basati su testo / carattere.

Examples

BufferedReader

introduzione

La classe `BufferedReader` è un wrapper per altre classi di `Reader` che ha due scopi principali:

1. `BufferedReader` fornisce il buffering per il `Reader` avvolto. Ciò consente a un'applicazione di leggere i caratteri uno alla volta senza inutili sovraccarichi I / O.

2. Un `BufferedReader` fornisce funzionalità per leggere il testo una riga alla volta.

Nozioni di base sull'utilizzo di BufferedReader

Il modello normale per l'utilizzo di `BufferedReader` è il seguente. Innanzitutto, ottieni il `Reader` cui vuoi leggere i caratteri. Successivamente si crea un'istanza di `BufferedReader` `Reader` che avvolge il `Reader` . Quindi leggi i dati dei personaggi. Finalmente chiudi `BufferedReader` che chiude il Reader `avvolto. Per esempio:

```
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);
}
```

È possibile applicare questo modello a qualsiasi `Reader`

Gli appunti:

1. Abbiamo utilizzato Java 7 (o versioni successive) *try-with-resources* per garantire che il lettore sottostante sia sempre chiuso. Ciò evita una potenziale perdita di risorse. Nelle versioni precedenti di Java, si chiudeva esplicitamente `BufferedReader` in un blocco `finally` .

2. Il codice all'interno del blocco `try` è praticamente identico a quello che `FileReader` se leggiamo direttamente da `FileReader` . In effetti, un `BufferedReader` funziona esattamente come il `Reader` che si avvolge si comporterebbe. La differenza è che *questa* versione è molto più efficiente.

La dimensione del buffer BufferedReader

Il metodo BufferedReader.readLine ()

Esempio: lettura di tutte le righe di un file in una lista

Ciò avviene inserendo ogni riga in un file e aggiungendola in un `List<String>` . L'elenco viene quindi restituito:

```
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 fornisce un modo più conciso per farlo usando il metodo `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();
}
```

Esempio di StringWriter

La classe Java StringWriter è un flusso di caratteri che raccoglie l'output dal buffer di stringa, che può essere utilizzato per costruire una stringa.

La classe StringWriter estende la classe Writer.

Nella classe StringWriter, le risorse di sistema come i socket di rete ei file non vengono utilizzati, pertanto la chiusura di StringWriter non è necessaria.

```
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();
    }
}
```

L'esempio sopra riportato ci aiuta a conoscere un semplice esempio di StringWriter che utilizza BufferedReader per leggere i dati del file dallo stream.

Leggi Lettori e scrittori online: https://riptutorial.com/it/java/topic/10618/lettori-e-scrittori

Capitolo 108: LinkedHashMap

introduzione

La classe LinkedHashMap è la tabella hash e l'implementazione dell'elenco collegato dell'interfaccia della mappa, con un ordine di iterazione prevedibile. Eredita la classe HashMap e implementa l'interfaccia della mappa.

I punti importanti sulla classe Java LinkedHashMap sono: A LinkedHashMap contiene valori basati sulla chiave. Contiene solo elementi unici. Può avere una chiave nullo e più valori nulli. È come se HashMap mantenga invece l'ordine di inserimento.

Examples

Classe Java LinkedHashMap

Punti chiave:-

- È l'implementazione della tabella hash e dell'elenco collegato dell'interfaccia della mappa, con un ordine di iterazione prevedibile.
- eredita la classe HashMap e implementa l'interfaccia Mappa.
- contiene valori basati sulla chiave.
- solo elementi unici.
- può avere una chiave nulla e più valori nulli.
- lo stesso di HashMap invece mantiene l'ordine di inserimento.

Metodi: -

- void clear ().
- boolean containsKey (chiave dell'oggetto).
- Oggetto get (chiave dell'oggetto).
- protected boolean removeEldestEntry (Map.Entry primogenito)

Esempio :-

```
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"));
    }
```

Leggi LinkedHashMap online: https://riptutorial.com/it/java/topic/10750/linkedhashmap

Capitolo 109: Lista vs SET

introduzione

Quali sono le differenze tra la raccolta List e Set al livello superiore e Come scegliere quando utilizzare List in java e quando utilizzare Set in Java

Examples

Lista vs Set

```
import java.util.ArrayList;
```

import java.util.HashSet; import java.util.List; import java.util.Set;

public class SetAndListExample {public static void main (String [] args) {System.out.println ("Elenco di esempio ....."); Elenco lista = 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);
    }
}
```

}

Esempio di lista di uscite ..... 1 2 3 4 1 Imposta esempio 3 2 10 5 4

Leggi Lista vs SET online: https://riptutorial.com/it/java/topic/10125/lista-vs-set

Capitolo 110: Localizzazione e internazionalizzazione

Osservazioni

Java è dotato di un meccanismo potente e flessibile per localizzare le tue applicazioni, ma è anche facile da usare impropriamente e finire con un programma che ignora o manipola le impostazioni locali dell'utente, e quindi come si aspettano che il tuo programma si comporti.

I tuoi utenti si aspetteranno di vedere i dati localizzati nei formati a cui sono abituati, e il tentativo di supportarli manualmente è un errore. Qui c'è solo un piccolo esempio dei diversi modi in cui gli utenti si aspettano di vedere il contenuto che si potrebbe presumere sia "sempre" visualizzato in un certo modo:

	Date	Numeri	Moneta locale	Moneta straniera	distanze
Brasile					
Cina					
Egitto					
Messico	20/3/16	1.234,56	$ 1,000.50	1.000,50 USD	
UK	20/3/16	1,234.56	£ 1,000.50		100 km
Stati Uniti d'America	3/20/16	1,234.56	$ 1,000.50	1,000.50 MXN	60 miglia

Risorse generali

- Wikipedia: internazionalizzazione e localizzazione

Risorse Java

- Tutorial Java: internazionalizzazione
- Oracle: internazionalizzazione: comprensione delle impostazioni internazionali nella piattaforma Java
- JavaDoc: `Locale`

Examples

Date formattate automaticamente usando "locale"

`SimpleDateFormatter` è ottimo in un pizzico, ma come suggerisce il nome, non scala bene.

Se digiti `"MM/dd/yyyy"` su tutta la tua applicazione, i tuoi utenti internazionali non saranno felici.

Lascia che Java faccia il lavoro per te

Utilizzare i metodi `static` in `DateFormat` per recuperare la formattazione corretta per l'utente. Per un'applicazione desktop (dove ti affidi alle impostazioni internazionali predefinite), chiama semplicemente:

```
String localizedDate = DateFormat.getDateInstance(style).format(date);
```

Dove lo `style` è una delle costanti di formattazione (`FULL` , `LONG` , `MEDIUM` , `SHORT` , ecc.) Specificate in `DateFormat` .

Per un'applicazione lato server in cui l'utente specifica la propria locale come parte della richiesta, è necessario passarla esplicitamente a `getDateInstance()` :

```
String localizedDate =
    DateFormat.getDateInstance(style, request.getLocale()).format(date);
```

Confronto di stringhe

Confronta due stringhe che ignorano il caso:

```
"School".equalsIgnoreCase("school"); // true
```

Non usare

```
text1.toLowerCase().equals(text2.toLowerCase());
```

Le lingue hanno regole diverse per la conversione di lettere maiuscole e minuscole. Un "I" verrebbe convertito in "i" in inglese. Ma in turco un 'io' diventa un 'io'. Se devi usare `toLowerCase()` usa il sovraccarico che si aspetta un `Locale` : `String.toLowerCase(Locale)` .

Confrontando due stringhe ignorando le differenze minori:

```
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.PRIMARY);
collator.equals("Gärten", "gaerten"); // returns true
```

Ordina le stringhe rispettando l'ordine della lingua naturale, ignorando il caso (usa la chiave di confronto per:

```
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"}
```

località

La classe `java.util.Locale` viene utilizzata per rappresentare una regione "geografica, politica o culturale" per localizzare un determinato testo, numero, data o operazione. Un oggetto Locale può quindi contenere un paese, una regione, una lingua e anche una variante di una lingua, ad esempio un dialetto parlato in una certa regione di un paese, o parlato in un paese diverso dal paese da cui proviene la lingua.

L'istanza Locale viene consegnata ai componenti che devono localizzare le loro azioni, sia che si tratti di convertire l'input, l'output o semplicemente di averne bisogno per le operazioni interne. La classe Locale non può effettuare internazionalizzazione o localizzazione da sola

linguaggio

La lingua deve essere un codice lingua ISO 639 a 2 o 3 caratteri o una sottotag della lingua registrata composta da un massimo di 8 caratteri. Nel caso in cui una lingua abbia un codice lingua a 2 e 3 caratteri, utilizzare il codice a 2 caratteri. Un elenco completo dei codici di lingua può essere trovato nel registro di sottotag della lingua IANA.

I codici lingua non fanno distinzione tra maiuscole e minuscole, ma la classe Locale utilizza sempre le versioni minuscole dei codici lingua

Creare un locale

La creazione di un'istanza `java.util.Locale` può essere eseguita in quattro modi diversi:

```
Locale constants
Locale constructors
Locale.Builder class
Locale.forLanguageTag factory method
```

Java ResourceBundle

Si crea un'istanza ResourceBundle in questo modo:

```
Locale locale = new Locale("en", "US");
ResourceBundle labels = ResourceBundle.getBundle("i18n.properties");
System.out.println(labels.getString("message"));
```

Considera che ho un file di proprietà `i18n.properties` :

```
message=This is locale
```

Produzione:

```
This is locale
```

Impostazione locale

Se si desidera riprodurre lo stato utilizzando altre lingue, è possibile utilizzare il metodo **`setDefault()`** . Il suo uso:

```
 setDefault(Locale.JAPANESE); //Set Japanese
```

Leggi Localizzazione e internazionalizzazione online:
https://riptutorial.com/it/java/topic/4086/localizzazione-e-internazionalizzazione

Capitolo 111: log4j / log4j2

introduzione

Apache Log4j è un'utilità di registrazione basata su Java, è uno dei numerosi framework di registrazione Java. Questo argomento mostra come configurare e configurare Log4j in Java con esempi dettagliati su tutti i suoi possibili aspetti di utilizzo.

Sintassi

- Logger.debug ("testo da registrare"); // Registrazione delle informazioni di debug
- Logger.info ("testo da registrare"); // Registrazione delle informazioni comuni
- Logger.error ("testo da registrare"); // Registrazione delle informazioni di errore
- Logger.warn ("testo da registrare"); // Avvisi di registrazione
- Logger.trace ("testo da registrare"); // Registrazione delle informazioni di traccia
- Logger.fatal ("testo da registrare"); // Registrazione degli errori fatali
- Utilizzo di Log4j2 con la registrazione dei parametri:
- Logger.debug ("Parametri di debug {} {} {}", param1, param2, param3); // Registrazione del debug con i parametri
- Logger.info ("Info params {} {} {}", param1, param2, param3); // Registrazione delle informazioni con parametri
- Logger.error ("Parametri di errore {} {} {}", param1, param2, param3); // Errore di registrazione con parametri
- Logger.warn ("Warn params {} {} {}", param1, param2, param3); // Registrazione degli avvisi con i parametri
- Logger.trace ("Parametri di traccia {} {} {}", param1, param2, param3); // Registrazione traccia con parametri
- Logger.fatal ("Parametri fatali {} {} {}", param1, param2, param3); // Registrazione fatale con parametri
- Logger.error ("Caught Exception:", ex); // Registrazione dell'eccezione con messaggio e stacktrace (verrà automaticamente aggiunta)

Osservazioni

Fine vita per Log4j 1 raggiunto

> Il 5 agosto 2015 il comitato di gestione dei progetti di servizi di registrazione ha annunciato che Log4j 1.x aveva raggiunto la fine della vita. Per il testo completo dell'annuncio, consultare il blog di Apache. **Gli utenti di Log4j 1 sono consigliati per l'aggiornamento ad Apache Log4j 2** .

Da: http://logging.apache.org/log4j/1.2/

Examples

Come ottenere Log4j

Versione corrente (log4j2)

Usando Maven:
Aggiungi la seguente dipendenza al tuo file `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>
```

Utilizzando 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>
```

Utilizzando 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'
}
```

Ottenere log4j 1.x

Nota: Log4j 1.x ha raggiunto la fine vita (EOL) (vedere Note).

Usando Maven:

Dichiarare questa dipendenza nel file `POM.xml` :

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
```

```
</dependency>
```

Utilizzando Ivy:

```
<dependency org="log4j" name="log4j" rev="1.2.17"/>
```

Usign Gradle:

```
compile group: 'log4j', name: 'log4j', version: '1.2.17'
```

Utilizzando Buildr:

```
'log4j:log4j:jar:1.2.17'
```

Aggiunta manuale nella compilazione del percorso:

Scarica dal progetto del sito web Log4j

Come usare Log4j nel codice Java

Innanzitutto occorre creare un oggetto `final static logger` :

```
final static Logger logger = Logger.getLogger(classname.class);
```

Quindi, chiama i metodi di registrazione:

```
//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);
```

Impostazione del file delle proprietà

Log4j ti dà la possibilità di registrare i dati in console e file contemporaneamente. Crea un file `log4j.properties` e inserisci questa configurazione di base:

```
# 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
```

Se stai usando Maven, metti questo file appropriato nel percorso:

```
/ProjectFolder/src/java/resources
```

File di configurazione di base log4j2.xml

```
<?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>
```

Questa è una configurazione base di log4j2.xml che ha un appender della console e un logger root. Il layout del modello specifica quale modello deve essere utilizzato per la registrazione delle istruzioni.
Per eseguire il debug del caricamento di log4j2.xml è possibile aggiungere lo `status = <WARN | DEBUG | ERROR | FATAL | TRACE | INFO>` dell'attributo `status = <WARN | DEBUG | ERROR | FATAL | TRACE | INFO>` nel tag di configurazione del tuo log4j2.xml.
È inoltre possibile aggiungere un intervallo di monitoraggio in modo che carichi nuovamente la configurazione dopo il periodo di intervallo specificato. L'intervallo del monitor può essere aggiunto al tag di configurazione come segue: `monitorInterval = 30` . Significa che la configurazione verrà caricata ogni 30 secondi.

Migrazione da log4j 1.x a 2.x

Se si desidera eseguire la migrazione da log4j 1.x esistente nel progetto a log4j 2.x, rimuovere tutte le dipendenze esistenti di log4j 1.x e aggiungere la seguente dipendenza:

Log4j 1.x Ponte API

Build Maven

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-1.2-api</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Edera Build

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-1.2-api" rev="2.6.2" />
</dependencies>
```

Gradle Build

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-1.2-api', version: '2.6.2'
}
```

Apache Commons Logging Bridge Se il progetto utilizza Apache Commons Logging che utilizza log4j 1.x e si desidera migrarlo a log4j 2.x, aggiungere le seguenti dipendenze:

Build Maven

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-jcl</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Edera Build

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-jcl" rev="2.6.2" />
</dependencies>
```

Gradle Build

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-jcl', version: '2.6.2'
}
```

Nota: non rimuovere alcuna dipendenza esistente dalla registrazione di Apache commons

Riferimento: https://logging.apache.org/log4j/2.x/maven-artifacts.html

Proprietà-File per accedere al DB

Per questo esempio, è necessario un driver JDBC compatibile con il sistema su cui è in esecuzione il database. Un opensource che consente di connettersi ai database DB2 su un sistema IBM i può essere trovato qui: JT400

Anche se questo esempio è specifico per DB2, funziona per quasi tutti gli altri sistemi se si scambia il driver e si adatta l'URL 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
```

Filtro Logoutput per livello (log4j 1.x)

È possibile utilizzare un filtro per registrare solo i messaggi "inferiori" rispetto al livello `ERROR` . **Ma il filtro non è supportato da PropertyConfigurator. Quindi è necessario passare alla configurazione XML per usarlo** . Vedi log4j-Wiki sui filtri .

Esempio "livello specifico"

```
<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>
```

Oppure "Gamma di livelli"

```
<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>
```

Leggi log4j / log4j2 online: https://riptutorial.com/it/java/topic/2472/log4j---log4j2

Capitolo 112: Manager della sicurezza

Examples

Abilitazione di SecurityManager

Java Virtual Machines (JVM) può essere eseguito con SecurityManager installato. SecurityManager governa ciò che il codice in esecuzione nella JVM è autorizzato a fare, in base a fattori quali il punto in cui è stato caricato il codice e quali certificati sono stati utilizzati per firmare il codice.

SecurityManager può essere installato impostando la proprietà di sistema java.security.manager sulla riga di comando all'avvio di JVM:

```
java -Djava.security.manager <main class name>
```

o programmaticamente dal codice Java:

```
System.setSecurityManager(new SecurityManager())
```

Lo standard Java SecurityManager concede le autorizzazioni sulla base di un criterio, definito in un file di criteri. Se non viene specificato alcun file di criteri, verrà utilizzato il file di criteri predefinito in `$JAVA_HOME/lib/security/java.policy` .

Classi di sandbox caricate da un ClassLoader

ClassLoader deve fornire un `ProtectionDomain` identifichi l'origine del codice:

```
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;
    }
}
```

`findClass` anziché `loadClass` il modello delegato viene mantenuto e PluginClassLoader eseguirà

prima una query sul classloader del sistema e del genitore per le definizioni di classe.

Crea un criterio:

```
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;
    }

}
```

Infine, imposta la politica e un SecurityManager (l'implementazione predefinita va bene):

```
    Policy.setPolicy(new PluginSecurityPolicy());
    System.setSecurityManager(new SecurityManager());
```

La politica di attuazione nega le regole

Occasionalmente è desiderabile *negare* una certa `Permission` ad alcuni `ProtectionDomain` , *indipendentemente* da qualsiasi altra autorizzazione che il dominio acquisisce. Questo esempio dimostra solo uno dei possibili approcci per soddisfare questo tipo di esigenza. Introduce una classe di autorizzazione "negativa", insieme a un wrapper che consente il riutilizzo del `Policy`

predefinito come repository di tali autorizzazioni.

Gli appunti:

- La sintassi del file delle politiche standard e il meccanismo per l'assegnazione dei permessi in generale rimangono inalterati. Ciò significa che *negare le* regole all'interno dei file delle politiche sono ancora espresse sotto forma di *sovvenzioni* .
- Il wrapper delle politiche ha lo scopo di incapsulare in modo specifico la `Policy` backup `Policy` file predefinita (presunta `com.sun.security.provider.PolicyFile`).
- Le autorizzazioni negate vengono elaborate solo come tali a livello di policy. Se assegnati staticamente a un dominio, per impostazione predefinita verranno trattati da quel dominio come normali autorizzazioni "positive".

La classe `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 = 47362516386980067 9L;

    /**
     * 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 classe `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();
    }

}
```

dimostrazione

```
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");
    }
```

```
}
```

Assegna alcune autorizzazioni:

```
grant codeBase "file:///path/to/classes/bin/-"
    permission java.util.PropertyPermission "*", "read,write";
    permission com.example.DeniedPermission "java.util.PropertyPermission:foo.bar:read";
};
```

Infine, esegui `Main` e guardalo fallire, a causa della regola "deny" (`DeniedPermission`) che sovrascrive la `grant` (la sua `PropertyPermission`). Si noti che una `setProperty("foo.baz", "xyz")` sarebbe invece riuscita, poiché l'autorizzazione negata copre solo l'azione "letta" e solo per la proprietà "foo.bar".

Leggi Manager della sicurezza online: https://riptutorial.com/it/java/topic/5712/manager-della-sicurezza

Capitolo 113: Manipolazione bit

Osservazioni

- A differenza di C / C ++, Java è completamente endian-neutral rispetto all'hardware della macchina sottostante. Di default non si ottiene un comportamento big o little endian; devi specificare esplicitamente quale comportamento vuoi.
- Il tipo di `byte` è firmato, con l'intervallo da -128 a +127. Per convertire un valore di byte nel suo equivalente senza segno, mascherarlo con 0xFF in questo modo: `(b & 0xFF)` .

Examples

Imballaggio / spacchettamento dei valori come frammenti di bit

È normale che le prestazioni della memoria comprimano più valori in un singolo valore primitivo. Questo può essere utile per passare varie informazioni in una singola variabile.

Ad esempio, uno può imballare 3 byte - come il codice colore in RGB - in un singolo int.

Imballaggio dei valori

```
// 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;
```

Disimballaggio dei valori

```
// 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)
};
```

Controllo, impostazione, cancellazione e commutazione di singoli bit. Utilizzo lungo come bit mask

Supponendo di voler modificare il bit `n` di un intero primitivo, `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 come bit mask:

```
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));
    }
}
```

stampe

```
FIRST_BIT: true
SECOND_BIT: false
THIRD_BIT: true
FOURTh_BIT: false
FIFTH_BIT: true
BIT_55: true
```

che corrisponde a quello maschera abbiamo passato come `checkBitMask` parametro: `FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55` .

Esprimendo la potenza di 2

Per esprimere la potenza di 2 (2 ^ n) di numeri interi, si può usare un'operazione bitshift che consente di specificare esplicitamente il `n` .

La sintassi è fondamentalmente:

```
int pow2 = 1<<n;
```

Esempi:

```
int twoExp4 = 1<<4; //2^4
int twoExp5 = 1<<5; //2^5
int twoExp6 = 1<<6; //2^6
...
int twoExp31 = 1<<31; //2^31
```

Ciò è particolarmente utile quando si definiscono valori costanti che dovrebbero renderlo evidente, che viene utilizzata una potenza di 2, invece di utilizzare valori esadecimali o decimali.

```
int twoExp4 = 0x10; //hexadecimal
int twoExp5 = 0x20; //hexadecimal
int twoExp6 = 64; //decimal
...
int twoExp31 = -2147483648; //is that a power of 2?
```

Un semplice metodo per calcolare la potenza int di 2 sarebbe

```
int pow2(int exp){
    return 1<<exp;
}
```

Controllare se un numero è una potenza di 2

Se un numero intero `x` è una potenza di 2, viene impostato solo un bit, mentre dopo `x-1` tutti i bit vengono impostati. Ad esempio: `4` è `100` e `3` è `011` come numero binario, che soddisfa la condizione summenzionata. Zero non è una potenza di 2 e deve essere controllato esplicitamente.

```
boolean isPowerOfTwo(int x)
{
    return (x != 0) && ((x & (x - 1)) == 0);
}
```

Utilizzo per spostamento a sinistra e a destra

Supponiamo di avere tre tipi di permessi, **READ** , **WRITE** ed **EXECUTE** . Ogni permesso può variare da 0 a 7. (Supponiamo che il sistema di numeri a 4 bit)

> RESOURCE = READ WRITE EXECUTE (numero di 12 bit)
>
> RESOURCE = 0100 0110 0101 = 4 6 5 (numero di 12 bit)

Come possiamo ottenere le autorizzazioni (12 bit), impostate sopra (numero di 12 bit)?

> 0100 0110 0101

0000 0000 0111 (&)

0000 0000 0101 = 5

Quindi, questo è il modo in cui possiamo ottenere le autorizzazioni **EXECUTE** di **RESOURCE** . Ora, cosa succede se vogliamo ottenere i permessi di **LETTURA** della **RISORSA** ?

0100 0110 0101

0111 0000 0000 (&)

0100 0000 0000 = 1024

Destra? Probabilmente stai assumendo questo? Ma, le autorizzazioni sono risultate in 1024. Vogliamo ottenere solo i permessi di LETTURA per la risorsa. Non preoccuparti, è per questo che abbiamo avuto gli operatori di turno. Se vediamo, i permessi di lettura sono 8 bit dietro il risultato effettivo, quindi se si applica un operatore di shift, che porterà i permessi di lettura all'estrema destra del risultato? Cosa succede se facciamo:

0100 0000 0000 >> 8 => 0000 0000 0100 (Perché è un numero positivo sostituito quindi da 0, se non ti interessa il segno, usa l'operatore di spostamento a destra senza segno)

Ora abbiamo i permessi di **LETTURA** che sono 4.

Ora, per esempio, ci sono le autorizzazioni **READ** , **WRITE** , **EXECUTE** per una **RISORSA** , cosa possiamo fare per ottenere i permessi per questa **RISORSA** ?

Prendiamo innanzitutto l'esempio delle autorizzazioni binarie. (Sempre ipotizzando un sistema a 4 bit)

LEGGI = 0001

WRITE = 0100

ESEGUI = 0110

Se stai pensando che faremo semplicemente:

`READ | WRITE | EXECUTE` , hai ragione, ma non esattamente. Vedi, cosa succederà se eseguiremo READ | SCRIVI | ESEGUIRE

0001 | 0100 | 0110 => 0111

Ma le autorizzazioni vengono effettivamente rappresentate (nel nostro esempio) come 0001 0100 0110

Quindi, per fare ciò, sappiamo che **READ** è posizionato 8 bit dietro, **WRITE** è posizionato 4 bit dietro e **PERMISSIONS** è posizionato all'ultimo. Il sistema numerico utilizzato per le autorizzazioni **RESOURCE** è in realtà 12 bit (nel nostro esempio). Può (sarà) diverso nei diversi sistemi.

(LEGGI << 8) | (SCRIVIA << 4) | (ESEGUIRE)

0000 0000 0001 << 8 (LEGGI)

0001 0000 0000 (spostamento a sinistra di 8 bit)

0000 0000 0100 << 4 (WRITE)

0000 0100 0000 (spostamento a sinistra di 4 bit)

0000 0000 0001 (ESEGUI)

Ora se aggiungiamo i risultati di uno spostamento sopra, sarà qualcosa di simile;

0001 0000 0000 (LEGGI)

0000 0100 0000 (WRITE)

0000 0000 0001 (ESEGUI)

0001 0100 0001 (AUTORIZZAZIONI)

classe java.util.BitSet

Dalla versione 1.7 esiste una classe java.util.BitSet che fornisce un'interfaccia di manipolazione e archiviazione di bit semplice e intuitiva:

```
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` e `Serializable` , e sotto la cappa tutti i valori di bit sono memorizzati nel campo `long[] words` , che si espande automaticamente.

Supporta anche operazioni logiche con set completo `and` , `or` , `xor` e `andNot` :

```
bitSet.and(new BitSet(8));
bitSet.or(new BitSet(8));
bitSet.xor(new BitSet(8));
bitSet.andNot(new BitSet(8));
```

Mai firmato e non firmato

In Java, tutti i numeri primitivi sono firmati. Ad esempio, un int rappresenta sempre i valori da [-2 ^ 31 - 1, 2 ^ 31], mantenendo il primo bit per firmare il valore - 1 per il valore negativo, 0 per il positivo.

Gli operatori di spostamento di base `>>` e `<<` sono operatori firmati. Conserveranno il segno del valore.

Ma è comune per i programmatori usare i numeri per memorizzare *valori non firmati* . Per un int, significa spostare l'intervallo su [0, 2 ^ 32 - 1], per avere il doppio del valore di un int firmato.

Per quegli utenti esperti, il bit per il segno non ha significato. Ecco perché Java ha aggiunto `>>>` , un operatore di spostamento a sinistra, ignorando quel bit di segno.

```
               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)
```

Perché non c'è `<<<` ?

Questo deriva dalla definizione del giusto turno. Mentre riempie i posti svuotati a sinistra, non ci sono decisioni da prendere riguardo al bit del segno. Di conseguenza, non sono necessari 2 operatori diversi.

Vedi questa domanda per una risposta più dettagliata.

Leggi Manipolazione bit online: https://riptutorial.com/it/java/topic/1177/manipolazione-bit

Capitolo 114: Mappa Enum

introduzione

La classe Java EnumMap è l'implementazione della mappa specializzata per le chiavi enum. Sopporta le classi Enum e AbstractMap.

i parametri per la classe java.util.EnumMap.

K: È il tipo di chiavi gestito da questa mappa. V: è il tipo di valori mappati.

Examples

Esempio di libro di enum mappa

```
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);

        }
}
}
```

Leggi Mappa Enum online: https://riptutorial.com/it/java/topic/10158/mappa-enum

Capitolo 115: Mappe

introduzione

L' interfaccia java.util.Map rappresenta una mappatura tra le chiavi e i loro valori. Una mappa non può contenere chiavi duplicate; e ogni chiave può mappare al massimo un valore.

Poiché `Map` è un'interfaccia, è necessario creare un'istanza concreta di tale interfaccia per utilizzarla; ci sono diverse implementazioni di `Map` e usate principalmente `java.util.HashMap` e `java.util.TreeMap`

Osservazioni

Una *mappa* è un oggetto che memorizza le *chiavi* con un *valore* associato per ogni chiave. Una chiave e il suo valore sono talvolta denominati *coppia chiave / valore* o una *voce* . Le mappe in genere forniscono queste funzionalità:

- I dati vengono memorizzati nella mappa in coppie chiave / valore.
- La mappa può contenere solo una voce per una particolare chiave. Se una mappa contiene una voce con una chiave particolare e si tenta di memorizzare una seconda voce con la stessa chiave, la seconda voce sostituirà la prima. In altre parole, questo cambierà il valore associato alla chiave.
- Le mappe forniscono operazioni veloci per verificare se esiste una chiave nella mappa, per recuperare il valore associato a una chiave e per rimuovere una coppia chiave / valore.

L'implementazione della mappa più comunemente utilizzata è HashMap . Funziona bene con chiavi che sono stringhe o numeri.

Mappe semplici come HashMap non sono ordinate. L'iterazione delle coppie chiave / valore può restituire singole voci in qualsiasi ordine. Se è necessario scorrere le voci della mappa in modo controllato, è necessario considerare quanto segue:

- Mappe ordinate come TreeMap eseguiranno iterazioni attraverso le chiavi nel loro ordine naturale (o in un ordine che puoi specificare, fornendo un comparatore). Ad esempio, una mappa ordinata utilizzando i numeri come chiavi dovrebbe scorrere le sue voci in ordine numerico.

- LinkedHashMap consente di iterare le voci nello stesso ordine in cui sono state inserite nella mappa o dall'ordine di accesso più recente.

Examples

Aggiungi un elemento

1. aggiunta

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
System.out.println(map.get(1));
```

Uscita: `First element.`

2. Oltrepassare

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
map.put(1, "New element.");
System.out.println(map.get(1));
```

Uscita: `New element.`

`HashMap` è usato come esempio. Possono essere utilizzate anche altre implementazioni che implementano l'interfaccia `Map` .

Aggiungi più oggetti

Possiamo usare `V put(K key,V value)` :

> Associa il valore specificato con la chiave specificata in questa mappa (operazione opzionale). Se la mappa in precedenza conteneva un mapping per la chiave, il vecchio valore viene sostituito dal valore specificato.

```
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());
```

Produzione:

```
3
```

Per aggiungere molti elementi puoi usare una classe interna come questa:

```
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");
}};
```

Tieni presente che la creazione di una classe interna anonima non è sempre efficiente e può portare a perdite di memoria, pertanto, quando possibile, utilizzare invece un blocco di inizializzazione:

```
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");
}
```

L'esempio sopra rende la mappa statica. Può anche essere usato in un contesto non statico rimuovendo tutte le occorrenze di `static` .

Oltre a ciò la maggior parte delle implementazioni supporta `putAll` , che può aggiungere tutte le voci in una mappa a un'altra in questo modo:

```
another.putAll(one);
```

Utilizzo dei metodi predefiniti di Map da Java 8

Esempi di utilizzo di metodi predefiniti introdotti in Java 8 nell'interfaccia Mappa

1. Utilizzando **getOrDefault**

Restituisce il valore mappato alla chiave, o se la chiave non è presente, restituisce il valore predefinito

```
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**

Permette di eseguire l'operazione specificata nell "'azione" su ogni voce della mappa

```
  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. Utilizzando **replaceAll**

Sostituirà con nuovo valore solo se è presente la chiave

```
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. Utilizzando **putIfAbsent**

La coppia valore-chiave viene aggiunta alla mappa, se la chiave non è presente o mappata su null

```
    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. Usando **rimuovere**

   Rimuove la chiave solo se è associata al valore specificato

   ```
   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 **sostituire**

   Se la chiave è presente, il valore viene sostituito dal nuovo valore. Se la chiave non è presente, non fa nulla.

```
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. Utilizzo di **computeIfAbsent**

Questo metodo aggiunge una voce nella mappa. la chiave è specificata nella funzione e il valore è il risultato dell'applicazione della funzione di mappatura

```
    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. Utilizzando **computeIfPresent**

Questo metodo aggiunge una voce o modifica una voce esistente nella mappa. Non fa nulla se non è presente una voce con quella chiave

```
    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. Utilizzando il **calcolo**

Questo metodo sostituisce il valore di una chiave con il valore appena calcolato

```
    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. Utilizzando l' **unione**

Aggiunge la coppia chiave-valore alla mappa, se la chiave non è presente o il valore per la chiave è null Sostituisce il valore con il valore appena calcolato, se la chiave è presente La chiave viene rimossa dalla mappa, se il nuovo valore calcolato è nullo

```
    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}
```

Cancella la mappa

```
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
```

Iterare attraverso il contenuto di una mappa

Maps fornisce metodi che consentono di accedere a chiavi, valori o coppie chiave-valore della mappa come raccolte. Puoi scorrere queste raccolte. Data la seguente mappa per esempio:

```
Map<String, Integer> repMap = new HashMap<>();
repMap.put("Jon Skeet", 927_654);
repMap.put("BalusC", 708_826);
repMap.put("Darin Dimitrov", 715_567);
```

Iterazione tramite chiavi della mappa:

```
for (String key : repMap.keySet()) {
    System.out.println(key);
}
```

stampe:

Darin Dimitrov
Jon Skeet
BalusC

`keySet()` fornisce le chiavi della mappa come un `Set` . `Set` viene utilizzato poiché le chiavi non possono contenere valori duplicati. L'iterazione attraverso l'insieme produce ogni chiave a turno. HashMaps non è ordinato, quindi in questo esempio le chiavi possono essere restituite in qualsiasi ordine.

Iterazione dei valori della mappa:

```
for (Integer value : repMap.values()) {
    System.out.println(value);
}
```

stampe:

715.567
927654
708.826

`values()` restituisce i valori della mappa come una `Collection` . L'iterazione attraverso la raccolta produce a turno ogni valore. Di nuovo, i valori possono essere restituiti in qualsiasi ordine.

Iterazione tra chiavi e valori insieme

```
for (Map.Entry<String, Integer> entry : repMap.entrySet()) {
    System.out.printf("%s = %d\n", entry.getKey(), entry.getValue());
}
```

stampe:

```
Darin Dimitrov = 715567
Jon Skeet = 927654
BalusC = 708826
```

`entrySet()` restituisce una raccolta di oggetti `Map.Entry` . Map.Entry dà accesso alla chiave e al valore per ogni voce.

Unione, combinazione e composizione di Maps

Usa `putAll` per inserire tutti i membri di una mappa in un'altra. Le chiavi già presenti nella mappa avranno i loro valori corrispondenti sovrascritti.

```
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)
```

Questo produce il seguente mapping in `numbers` :

```
"One" -> 1
"Two" -> 2
"Three" -> 4 //old value 3 was overwritten by new value 4
```

Se si desidera combinare i valori anziché sovrascriverli, è possibile utilizzare `Map.merge` , aggiunto in Java 8, che utilizza un `BiFunction` fornito `BiFunction` per unire i valori per le chiavi duplicate. `merge` funziona su singole chiavi e valori, quindi dovrai utilizzare un loop o `Map.forEach` . Qui concateniamo stringhe per chiavi duplicate:

```
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));
```

Se si desidera applicare il vincolo non ci sono chiavi duplicate, è possibile utilizzare una funzione di unione che genera un `AssertionError` :

```
mapA.forEach((k, v) ->
    mapB.merge(k, v, (v1, v2) ->
```

```
        {throw new AssertionError("duplicate values for key: "+k);}));
```

Componi Map <X, Y> e Map <Y, Z> per ottenere la mappa <X, Z>

Se si desidera comporre due mapping, è possibile farlo come segue

```
    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)));
```

Questo produce la seguente mappatura

```
    "key1" -> 1.0
    "key2" -> 2.0
    "key3" -> 3.0
```

Controlla se la chiave esiste

```
Map<String, String> num = new HashMap<>();
num.put("one", "first");

if (num.containsKey("one")) {
    System.out.println(num.get("one")); // => first
}
```

Le mappe possono contenere valori nulli

Per le mappe, bisogna carrefull non confondere "contenente una chiave" con "avere un valore". Ad esempio, `HashMap` può contenere null, il che significa che il seguente comportamento è perfettamente normale:

```
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
}
```

Più formalmente, non vi è alcuna garanzia che `map.contains(key) <=> map.get(key)!=null`

Iterazione delle voci della mappa in modo efficiente

Questa sezione fornisce codice e benchmark per dieci implementazioni di esempio uniche che eseguono l'iterazione sulle voci di `Map<Integer, Integer>` e generano la somma dei valori `Integer` . Tutti gli esempi hanno una complessità algoritmica di `O(n)` , tuttavia i benchmark sono ancora utili per fornire informazioni su quali implementazioni sono più efficienti in un ambiente "reale".

1. Implementazione tramite **`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. Implementazione usando **`for`** con **`Map.Entry`**

```
for (Map.Entry<Integer, Integer> pair : map.entrySet()) {
    sum += pair.getKey() + pair.getValue();
}
```

3. Implementazione con **`Map.forEach`** (Java 8+)

```
map.forEach((k, v) -> sum[0] += k + v);
```

4. Implementazione utilizzando **`Map.keySet`** con **`for`**

```
for (Integer key : map.keySet()) {
    sum += key + map.get(key);
}
```

5. Implementazione tramite **`Map.keySet`** con **`Iterator`**

```
Iterator<Integer> it = map.keySet().iterator();
while (it.hasNext()) {
    Integer key = it.next();
    sum += key + map.get(key);
}
```

6. Implementazione usando **`for`** con **`Iterator`** e **`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. Implementazione tramite **`Stream.forEach`** (Java 8+)

```
map.entrySet().stream().forEach(e -> sum += e.getKey() + e.getValue());
```

8. Implementazione con `Stream.forEach` con `Stream.parallel` (Java 8+)

```
map.entrySet()
   .stream()
   .parallel()
   .forEach(e -> sum += e.getKey() + e.getValue());
```

9. Implementazione tramite `IterableMap` da Apache Collections

```
MapIterator<Integer, Integer> mit = iterableMap.mapIterator();
while (mit.hasNext()) {
    sum += mit.next() + it.getValue();
}
```

10. Implementazione con `MutableMap` da collezioni Eclipse

```
mutableMap.forEachKeyValue((key, value) -> {
    sum += key + value;
});
```

Test delle prestazioni (*codice disponibile su Github*)
Ambiente di test: Windows 8.1 a 64 bit, Intel i7-4790 3.60 GHz, 16 GB

1. Prestazioni medie di 10 test (100 elementi) Migliore: 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
```

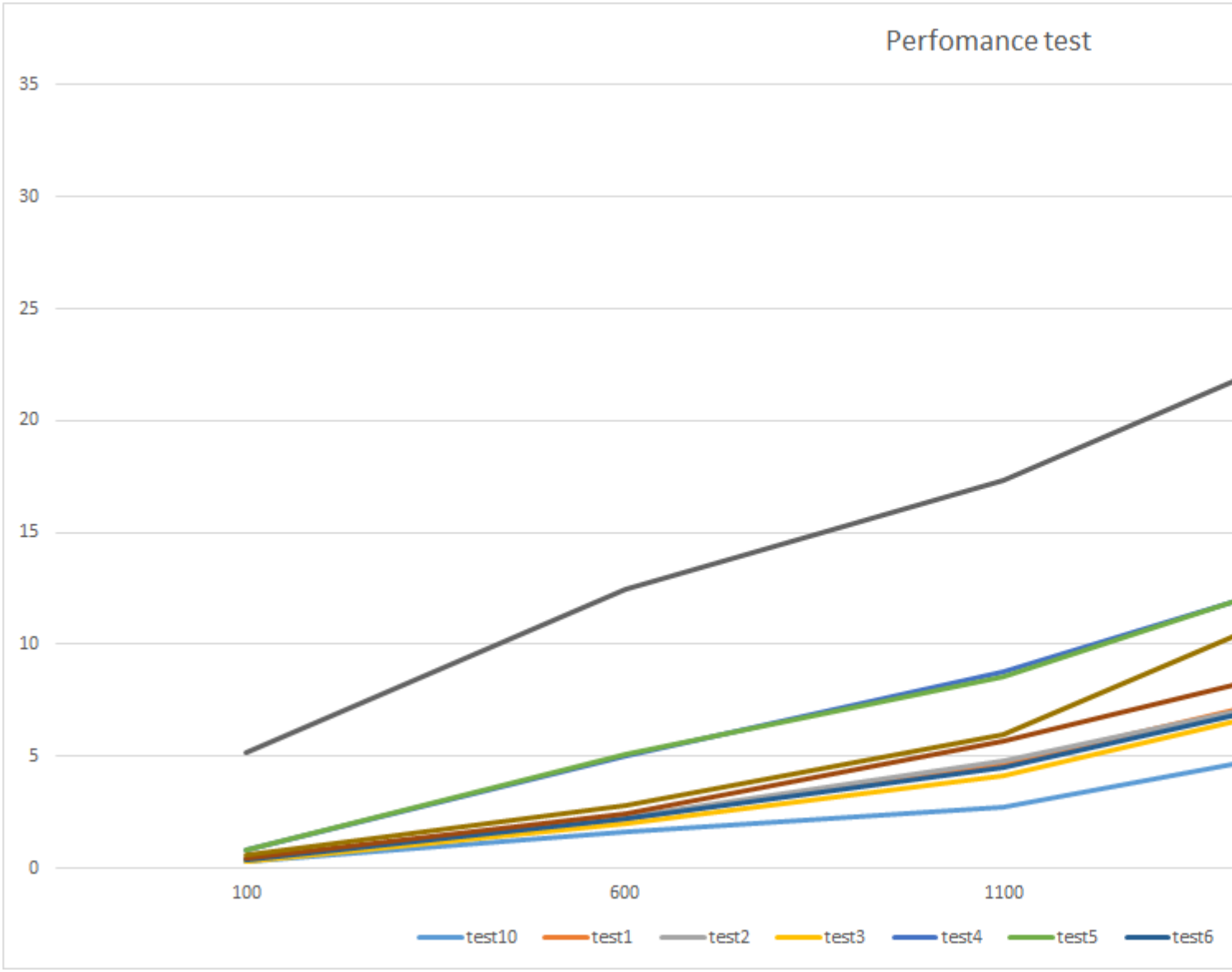
2. Prestazioni medie di 10 test (10000 elementi) Migliore: 37,606 ± 0,790 µs / op

```
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. Prestazioni medie di 10 test (100.000 elementi) Migliore: 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. Un confronto delle variazioni delle prestazioni rispetto alle dimensioni della mappa




```
   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
```

Usa oggetto personalizzato come chiave

Prima di utilizzare il proprio oggetto come chiave, è necessario sovrascrivere il metodo hashCode () e equals () dell'oggetto.

Nel caso semplice avresti qualcosa come:

```
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` deciderà a quale hash bucket appartiene la chiave e `equals` a decidere quale oggetto all'interno di quel bucket hash.

Senza questo metodo, il riferimento dell'oggetto verrà utilizzato per il confronto sopra che non funzionerà a meno che non si usi sempre lo stesso riferimento all'oggetto.

Utilizzo di HashMap

HashMap è un'implementazione dell'interfaccia Map che fornisce una struttura dati per archiviare i dati in coppie valore-chiave.

1. Dichiarazione di HashMap

```
Map<KeyType, ValueType> myMap = new HashMap<KeyType, ValueType>();
```

KeyType e ValueType devono essere tipi validi in Java, come ad esempio: String, Integer, Float o

qualsiasi classe personalizzata come Employee, Student ecc.

Ad esempio: `Map<String,Integer> myMap = new HashMap<String,Integer>();`

2. Mettere i valori in HashMap.

Per mettere un valore nella HashMap, dobbiamo chiamare `put` metodo sull'oggetto HashMap passando la chiave e il valore come parametri.

```
myMap.put("key1", 1);
myMap.put("key2", 2);
```

Se chiami il metodo put con la chiave che già esiste nella mappa, il metodo sostituirà il suo valore e restituirà il vecchio valore.

3. Ottenere valori da HashMap.

Per ottenere il valore da una HashMap devi chiamare il metodo `get` , passando la chiave come parametro.

```
myMap.get("key1");    //return 1 (class Integer)
```

Se passi una chiave che non esiste in HashMap, questo metodo restituirà `null`

4. Controllare se la chiave è nella mappa o no.

```
myMap.containsKey(varKey);
```

5. Controlla se il valore è nella mappa o no.

```
myMap.containsValue(varValue);
```

I metodi precedenti restituiranno un valore `boolean` vero o falso se la chiave, il valore esiste nella Mappa o no.

Creazione e inizializzazione di mappe

introduzione

`Maps` memorizza coppie chiave / valore, in cui ogni chiave ha un valore associato. Data una chiave particolare, la mappa può cercare il valore associato molto rapidamente.

`Maps` , noto anche come array associato, è un oggetto che memorizza i dati sotto forma di chiavi e valori. In Java, le mappe sono rappresentate tramite l'interfaccia Mappa che non è un'estensione dell'interfaccia di raccolta.

- Modo 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);
```

- Via 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);
```

- Strada 3: -

```
Map<String, Object> map = new HashMap<String, Object>(){{
    put("name", "A");
    put("address", "Malviya-Nagar");
    put("city", "Jaipur");
}};
System.out.println(map);
```

- Strada 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);
```

- Strada 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);
```

- Cammino 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");
}
```

- Way 7: - Creazione di una mappa di valore chiave immutabile.

```
    //Immutable single key-value map
    Map<String, String> singletonMap = Collections.singletonMap("key", "value");
```

Si noti che **è impossibile modificare tale mappa** .

Qualsiasi tentativo di modificare la mappa comporterà il lancio di 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
```

Leggi Mappe online: https://riptutorial.com/it/java/topic/105/mappe

Capitolo 116: Metodi di classe oggetto e costruttore

introduzione

Questa pagina di documentazione è per mostrare dettagli con esempio su classi Java costruttori e circa i metodi della classe Object che vengono automaticamente ereditate dalla superclasse `Object` di qualsiasi classe appena creata.

Sintassi

- classe nativa finale pubblica <?> getClass ()
- public final native void notify ()
- public final nativo vuoto notifyAll ()
- attesa finale nativa finale del pubblico (timeout lungo) genera InterruptedException
- public final void wait () genera InterruptedException
- attesa finale vuota pubblica (long timeout, int nanos) genera InterruptedException
- public native int hashCode ()
- public boolean equals (Object obj)
- public String toString ()
- protetto oggetto nativo clone () genera CloneNotSupportedException
- protected void finalize () lancia Throwable

Examples

metodo toString ()

Il metodo `toString()` viene utilizzato per creare una rappresentazione `String` di un oggetto utilizzando il contenuto dell'oggetto. Questo metodo dovrebbe essere sovrascritto durante la scrittura della classe. `toString()` viene chiamato implicitamente quando un oggetto è concatenato a una stringa come in `"hello " + anObject` .

Considera quanto segue:

```
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"
    }
}
```

Qui `toString()` dalla classe `Object` viene sovrascritto nella classe `User` per fornire dati significativi sull'oggetto durante la stampa.

Quando si utilizza `println()` , viene chiamato implicitamente il metodo `toString()` dell'oggetto. Pertanto, queste dichiarazioni fanno la stessa cosa:

```
System.out.println(user); // toString() is implicitly called on `user`
System.out.println(user.toString());
```

Se `toString()` non è sovrascritto nella suddetta classe `User` , `System.out.println(user)` può restituire `User@659e0bfd` o una stringa simile con quasi nessuna informazione utile eccetto il nome della classe. Questo perché la chiamata utilizzerà l'implementazione `toString()` della classe Java `Object` base che non conosce nulla sulla struttura della classe `User` o sulle regole aziendali. Se si desidera modificare questa funzionalità nella classe, è sufficiente sovrascrivere il metodo.

equals () metodo

TL; DR

`==` verifica l'uguaglianza di riferimento (indipendentemente dal fatto che siano lo ***stesso oggetto***)

`.equals()` verifica l'uguaglianza dei valori (indipendentemente dal fatto che siano ***logicamente "uguali"***)

`equals()` è un metodo utilizzato per confrontare due oggetti per l'uguaglianza. L'implementazione predefinita del metodo `equals()` nella classe `Object` restituisce `true` se e solo se entrambi i riferimenti puntano alla stessa istanza. Si comporta quindi come il confronto di `==` .

```
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
    }
```

```
}
```

Anche se `foo1` e `foo2` sono creati con gli stessi campi, stanno puntando a due oggetti diversi in memoria. L'implementazione di default `equals()` quindi è considerata `false` .

Per confrontare il contenuto di un oggetto per l'uguaglianza, `equals()` deve essere sovrascritto.

```
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
    }
}
```

Qui il metodo `equals()` over `equals()` sovrascritto decide che gli oggetti sono uguali se i loro campi sono uguali.

Si noti che anche il metodo `hashCode()` è stato sovrascritto. Il contratto per tale metodo afferma che quando due oggetti sono uguali, i loro valori hash devono essere uguali. Ecco perché bisogna quasi sempre sostituire `hashCode()` ed `equals()` insieme.

Presta particolare attenzione al tipo di argomento del metodo `equals` . È `Object obj` , non `Foo obj` .

Se metti quest'ultimo nel tuo metodo, questo non è un override del metodo `equals` .

Quando scrivi la tua classe, dovrai scrivere una logica simile quando esegui l'override di `equals()` e `hashCode()` . La maggior parte degli IDE può generare automaticamente questo per te.

Un esempio di un'implementazione `equals()` può essere trovato nella classe `String` , che fa parte dell'API Java principale. Piuttosto che confrontare i puntatori, la classe `String` confronta il contenuto della `String` .

Java SE 7

Java 1.7 ha introdotto la classe `java.util.Objects` che fornisce un metodo di convenienza, `equals` , che confronta due riferimenti potenzialmente `null` , quindi può essere utilizzato per semplificare le implementazioni del metodo `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);
}
```

Confronto di classe

Poiché il metodo equals può essere eseguito su qualsiasi oggetto, una delle prime cose che il metodo spesso esegue (dopo aver verificato il `null`) consiste nel verificare se la classe dell'oggetto confrontato corrisponde alla classe corrente.

```
@Override
public boolean equals(Object obj) {
    //...check for null
    if (getClass() != obj.getClass()) {
        return false;
    }
    //...compare fields
}
```

Questo è tipicamente fatto come sopra confrontando gli oggetti di classe. Tuttavia, ciò può fallire in alcuni casi speciali che potrebbero non essere ovvi. Ad esempio, alcuni framework generano proxy dinamici di classi e questi proxy dinamici sono in realtà una classe diversa. Ecco un esempio 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 meccanismo per ovviare a questa limitazione è confrontare le classi usando `instanceof`

```
@Override
public final boolean equals(Object obj) {
    if (!(obj instanceof Foo)) {
        return false;
    }
    //...compare fields
}
```

Tuttavia, ci sono alcune insidie che devono essere evitate quando si usa `instanceof` . Dato che Foo potrebbe potenzialmente avere altre sottoclassi e quelle sottoclassi potrebbero sovrascrivere `equals()` potresti entrare in un caso in cui un `Foo` è uguale a `FooSubclass` ma `FooSubclass` non è uguale a `Foo` .

```
Foo foo = new Foo(7);
FooSubclass fooSubclass = new FooSubclass(7, false);
foo.equals(fooSubclass) //true
fooSubclass.equals(foo) //false
```

Ciò viola le proprietà di simmetria e transitività e quindi è un'implementazione non valida del metodo `equals()` . Di conseguenza, quando si usa `instanceof` , una buona pratica consiste nel rendere `final` metodo `equals()` (come nell'esempio precedente). Ciò garantirà che nessuna sottoclasse sovrascrive `equals()` e viola le ipotesi chiave.

metodo hashCode ()

Quando una classe Java sovrascrive il metodo `equals` , dovrebbe sovrascrivere anche il metodo `hashCode` . Come definito nel contratto del metodo :

- Ogni volta che viene invocato sullo stesso oggetto più di una volta durante l'esecuzione di un'applicazione Java, il metodo `hashCode` deve restituire costantemente lo stesso numero intero, a condizione che non vengano modificate le informazioni utilizzate nei confronti degli uguali sull'oggetto. Questo numero intero non deve rimanere coerente da un'esecuzione di un'applicazione a un'altra esecuzione della stessa applicazione.
- Se due oggetti sono uguali secondo il metodo `equals(Object)` , quindi chiamare il metodo `hashCode` su ciascuno dei due oggetti deve produrre lo stesso risultato intero.
- Non è necessario che se due oggetti non sono uguali secondo il metodo `equals(Object)` , quindi chiamare il metodo `hashCode` su ciascuno dei due oggetti deve produrre risultati interi distinti. Tuttavia, il programmatore dovrebbe essere consapevole del fatto che la produzione di risultati interi distinti per oggetti non uguali può migliorare le prestazioni delle tabelle hash.

I codici hash vengono utilizzati nelle implementazioni hash come `HashMap` , `HashTable` e `HashSet` . Il

risultato della funzione `hashCode` determina il bucket in cui verrà inserito un oggetto. Queste implementazioni di hash sono più efficienti se l'implementazione `hashCode` fornita è buona. Una proprietà importante di una buona implementazione di `hashCode` è che la distribuzione dei valori `hashCode` è uniforme. In altre parole, esiste una piccola probabilità che numerose istanze vengano archiviate nello stesso bucket.

Un algoritmo per calcolare un valore di codice hash potrebbe essere simile al seguente:

```
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;
    }
}
```

Utilizzo di Arrays.hashCode () come scorciatoia

Java SE 1.2

In Java 1.2 e versioni successive, invece di sviluppare un algoritmo per calcolare un codice hash, è possibile generare uno utilizzando `java.util.Arrays#hashCode` fornendo una matrice Object o primitive contenente i valori del campo:

```
@Override
public int hashCode() {
    return Arrays.hashCode(new Object[] {field1, field2, field3});
}
```

Java SE 7

Java 1.7 ha introdotto la classe `java.util.Objects` che fornisce un metodo di convenienza, `hash(Object... objects)` , che calcola un codice hash basato sui valori degli oggetti forniti. Questo metodo funziona proprio come `java.util.Arrays#hashCode` .

```
@Override
public int hashCode() {
    return Objects.hash(field1, field2, field3);
}
```

Nota: questo approccio è inefficiente e produce oggetti spazzatura ogni volta che viene chiamato il metodo `hashCode()` personalizzato:

- Viene creato un `Object[]` temporaneo `Object[]` . (Nella versione `Objects.hash()` , la matrice viene creata dal meccanismo "varargs".)
- Se uno dei campi è di tipo primitivo, deve essere inserito in una scatola e ciò può creare più oggetti temporanei.
- La matrice deve essere popolata.
- L'array deve essere ripetuto dal metodo `Arrays.hashCode` o `Objects.hash` .
- Le chiamate a `Object.hashCode()` che `Arrays.hashCode` o `Objects.hash` devono rendere (probabilmente) non possono essere inline.

Memorizzazione nella cache interna dei codici hash

Poiché il calcolo del codice hash di un oggetto può essere costoso, può essere interessante memorizzare nella cache il valore del codice hash all'interno dell'oggetto la prima volta che viene calcolato. Per esempio

```
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;
    }
}
```

Questo approccio elimina il costo di (ripetutamente) il calcolo del codice hash contro il sovraccarico di un campo aggiuntivo per memorizzare il codice hash. Se ciò si ripaga, l'ottimizzazione delle prestazioni dipenderà dalla frequenza con cui un determinato oggetto viene sottoposto a hashing (cercato) e da altri fattori.

Noterete anche che se il vero codice hash di un `ImmutableArray` sembra essere pari a zero (una possibilità su 2^{32}), la cache è inefficace.

Infine, questo approccio è molto più difficile da implementare correttamente se l'oggetto che stiamo tritando è mutabile. Tuttavia, vi sono maggiori preoccupazioni se i codici hash cambiano; vedi il contratto di cui sopra.

metodi wait () e notify ()

`wait()` e `notify()` funzionano in tandem - quando un thread chiama `wait()` su un oggetto, quel thread bloccherà fino a quando un altro thread chiama `notify()` o `notifyAll()` su quello stesso oggetto.

(Vedi anche: wait () / notify ())

```
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!");
  }
}
```

Alcuni esempi di output:

```
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!
```

metodo getClass ()

Il metodo `getClass()` può essere utilizzato per trovare il tipo di classe runtime di un oggetto. Vedi l'esempio qui sotto:

```
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"
}
```

Il metodo `getClass()` restituirà il tipo di classe più specifico, motivo per cui quando viene chiamato `getClass()` su `anotherSpecificUser` , il valore restituito è `class SpecificUser` perché è inferiore nell'albero di eredità di `User` .

È interessante notare che, mentre il metodo `getClass` è dichiarato come:

```
public final native Class<?> getClass();
```

Il tipo statico effettivo restituito da una chiamata a `getClass` è di `Class<? extends T>` dove `T` è il tipo statico dell'oggetto su cui viene chiamato `getClass` .

cioè il seguente compila:

```
Class<? extends String> cls = "".getClass();
```

metodo clone ()

Il metodo `clone()` è usato per creare e restituire una copia di un oggetto. Questo metodo dovrebbe essere evitato in quanto è problematico e un costruttore di copia o qualche altro approccio per la copia dovrebbe essere usato a favore di `clone()` .

Per il metodo da utilizzare tutte le classi che chiamano il metodo devono implementare l'interfaccia `Cloneable` .

L'interfaccia `Cloneable` stessa è solo un'interfaccia di tag usata per cambiare il comportamento del metodo `native clone()` che controlla se la classe degli oggetti chiamanti implementa `Cloneable` . Se il chiamante non implementa questa interfaccia verrà lanciata una `CloneNotSupportedException` .

La classe `Object` non implementa questa interfaccia in modo che venga generata una `CloneNotSupportedException` se l'oggetto chiamante è di classe `Object` .

Perché un clone sia corretto dovrebbe essere indipendente dall'oggetto da cui viene clonato, quindi potrebbe essere necessario modificare l'oggetto prima che venga restituito. Ciò significa essenzialmente creare una "copia profonda" copiando anche uno qualsiasi degli oggetti *mutabili* che costituiscono la struttura interna dell'oggetto che viene clonato. Se questo non è implementato correttamente, l'oggetto clonato non sarà indipendente e avrà gli stessi riferimenti agli oggetti mutabili come l'oggetto da cui è stato clonato. Ciò comporterebbe un comportamento inconsistente in quanto eventuali modifiche a quelle in una influenzerebbero l'altra.

```
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);
        }
    }
}
```

finalize () metodo

Questo è un metodo *protetto* e *non statico* della classe `Object` . Questo metodo viene utilizzato per eseguire alcune operazioni finali o pulire le operazioni su un oggetto prima che venga rimosso dalla memoria.

Secondo il documento, questo metodo viene chiamato dal garbage collector su un

oggetto quando la garbage collection determina che non ci sono più riferimenti all'oggetto.

Ma non ci sono garanzie che il metodo `finalize()` venga chiamato se l'oggetto è ancora raggiungibile o non viene eseguito nessun Garbage Collector quando l'oggetto diventa idoneo. Ecco perché è meglio **non fare affidamento** su questo metodo.

Nelle librerie principali Java sono stati trovati alcuni esempi di utilizzo, ad esempio in `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();
    }
}
```

In questo caso è l'ultima possibilità di chiudere la risorsa se quella risorsa non è stata chiusa prima.

Generalmente è considerata una cattiva pratica usare il metodo `finalize()` in applicazioni di qualsiasi tipo e dovrebbe essere evitato.

I finalizzatori *non* sono pensati per liberare risorse (ad es. Chiudere i file). Il garbage collector viene chiamato quando (se!) Il sistema si scarica sullo spazio heap. Non si può fare affidamento su di esso per essere chiamato quando il sistema sta per esaurirsi sugli handle di file o, per qualsiasi altro motivo.

Il caso d'uso previsto per i finalizzatori è per un oggetto che sta per essere reclamato per notificare qualche altro oggetto sul suo imminente destino. A questo scopo esiste ora un meccanismo migliore --- la classe `java.lang.ref.WeakReference<T>` . Se pensi di aver bisogno di scrivere un metodo `finalize()` , allora dovresti esaminare se puoi risolvere lo stesso problema usando `WeakReference` . Se ciò non risolve il tuo problema, potrebbe essere necessario ripensare il tuo design a un livello più profondo.

Per ulteriori letture, ecco un articolo sul metodo `finalize()` libro "Effective Java" di Joshua Bloch.

Costruttore di oggetti

Tutti i costruttori in Java devono effettuare una chiamata al costruttore `Object` . Questo è fatto con la chiamata `super()` . Questa deve essere la prima riga in un costruttore. La ragione di ciò è che l'oggetto può effettivamente essere creato nell'heap prima che venga eseguita qualsiasi inizializzazione aggiuntiva.

Se non si specifica la chiamata a `super()` in un costruttore, il compilatore lo inserirà automaticamente.

Quindi tutti e tre questi esempi sono funzionalmente identici

con chiamata esplicita al costruttore `super()`

```
public class MyClass {

    public MyClass() {
        super();
    }
}
```

con chiamata implicita al costruttore `super()`

```
public class MyClass {

    public MyClass() {
        // empty
    }
}
```

con costruttore implicito

```
public class MyClass {

}
```

Che dire di Costruttore-Concatenamento?

È possibile chiamare altri costruttori come prima istruzione di un costruttore. Poiché sia la chiamata esplicita a un super costruttore che la chiamata a un altro costruttore devono essere entrambe le prime istruzioni, si escludono a vicenda.

```
public class MyClass {

    public MyClass(int size) {

        doSomethingWith(size);

    }

    public MyClass(Collection<?> initialValues) {

        this(initialValues.size());
        addInitialValues(initialValues);
    }
}
```

Chiamando la nuova `MyClass(Arrays.asList("a", "b", "c"))` chiamerà il secondo costruttore con l'argomento List, che a sua volta delegherà al primo costruttore (che delegherà implicitamente a `super()`) e quindi chiamare `addInitialValues(int size)` con la seconda dimensione dell'elenco. Questo viene utilizzato per ridurre la duplicazione del codice in cui più costruttori devono eseguire lo stesso lavoro.

Come posso chiamare un costruttore specifico?

Dato l'esempio sopra, si può chiamare `new MyClass("argument")` o `new MyClass("argument", 0)` . In altre parole, proprio come l' overloading dei metodi , basta chiamare il costruttore con i parametri necessari per il costruttore scelto.

Cosa succederà nel costruttore della classe Object?

Niente di più di quanto accadrebbe in una sottoclasse che ha un costruttore vuoto predefinito (meno la chiamata a `super()`).

Il costruttore vuoto predefinito può essere definito in modo esplicito, ma in caso contrario il compilatore lo inserirà finché non saranno già definiti altri costruttori.

Come viene creato un oggetto dal costruttore in Object?

La vera creazione di oggetti è verso la JVM. Ogni costruttore in Java appare come un metodo speciale denominato `<init>` che è responsabile per l'inizializzazione dell'istanza. Questo metodo `<init>` viene fornito dal compilatore e poiché `<init>` non è un identificatore valido in Java, non può essere utilizzato direttamente nella lingua.

In che modo JVM richiama questo metodo `<init>` ?

La JVM invocherà il metodo `<init>` usando l'istruzione `invokespecial` e può essere invocato solo su istanze di classe non inizializzate.

Per ulteriori informazioni, consulta le specifiche JVM e le specifiche del linguaggio Java:

- Metodi speciali (JVM) - JVMS - 2.9
- Costruttori - JLS - 8.8

Leggi Metodi di classe oggetto e costruttore online: https://riptutorial.com/it/java/topic/145/metodi-di-classe-oggetto-e-costruttore

Capitolo 117: Metodi di raccolta della fabbrica

introduzione

L'arrivo di Java 9 apporta molte nuove funzionalità all'API delle raccolte Java, una delle quali è costituita dai metodi di raccolta. Questi metodi consentono una facile inizializzazione delle collezioni **immutabili** , siano esse vuote o non vuote.

Si noti che questi metodi di fabbrica sono disponibili solo per le seguenti interfacce: `List<E>` , `Set<E>` e `Map<K, V>`

Sintassi

- `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)`

Parametri

Metodo w / Parametro	Descrizione
`List.of(E e)`	Un tipo generico che può essere una classe o un'interfaccia.
`Set.of(E e)`	Un tipo generico che può essere una classe o un'interfaccia.
`Map.of(K k, V v)`	Una coppia chiave-valore di tipi generici che possono essere una classe o un'interfaccia.
`Map.of(Map.Entry<? extends K, ? extends V> entry)`	Un'istanza di `Map.Entry` cui la sua chiave può essere `K` o uno dei suoi figli, e il suo valore può essere `V` o uno dei suoi figli.

Examples

Elenco Esempi di metodi di fabbrica

- `List<Integer> immutableEmptyList = List.of();`
  - Inizializza un `List<Integer>` vuoto e immutabile `List<Integer>` .
- `List<Integer> immutableList = List.of(1, 2, 3, 4, 5);`
  - Inizializza una `List<Integer>` immutabile `List<Integer>` con cinque elementi iniziali.
- `List<Integer> mutableList = new ArrayList<>(immutableList);`
  - Inizializza un `List<Integer>` mutabile `List<Integer>` da un `List<Integer>` immutabile `List<Integer>` .

Impostato Esempi di metodi di fabbrica

- `Set<Integer> immutableEmptySet = Set.of();`
  - Inizializza un `Set<Integer>` vuoto, immutabile.
- `Set<Integer> immutableSet = Set.of(1, 2, 3, 4, 5);`
  - Inizializza un `Set<Integer>` immutabile con cinque elementi iniziali.
- `Set<Integer> mutableSet = new HashSet<>(immutableSet);`
  - Inizializza un `Set<Integer>` mutabile `Set<Integer>` da un `Set<Integer>` immutabile `Set<Integer>` .

Carta geografica Esempi di metodi di fabbrica

- `Map<Integer, Integer> immutableEmptyMap = Map.of();`
  - Inizializza una mappa vuota, immutabile `Map<Integer, Integer>` .
- `Map<Integer, Integer> immutableMap = Map.of(1, 2, 3, 4);`
  - Inizializza una `Map<Integer, Integer>` immutabile `Map<Integer, Integer>` con due voci valore-chiave iniziali.
- `Map<Integer, Integer> immutableMap = Map.ofEntries(Map.entry(1, 2), Map.entry(3, 4));`
  - Inizializza una `Map<Integer, Integer>` immutabile `Map<Integer, Integer>` con due voci valore-chiave iniziali.
- `Map<Integer, Integer> mutableMap = new HashMap<>(immutableMap);`
  - Inizializza una `Map<Integer, Integer>` mutabile `Map<Integer, Integer>` da una `Map<Integer, Integer>` immutabile `Map<Integer, Integer>` .

Leggi Metodi di raccolta della fabbrica online: https://riptutorial.com/it/java/topic/9783/metodi-di-raccolta-della-fabbrica

Capitolo 118: Metodi predefiniti

introduzione

Il metodo predefinito introdotto in Java 8 consente agli sviluppatori di aggiungere nuovi metodi a un'interfaccia senza rompere le implementazioni esistenti di questa interfaccia. Fornisce flessibilità per consentire all'interfaccia di definire un'implementazione che verrà utilizzata come impostazione predefinita quando una classe che implementa tale interfaccia non riesce a fornire un'implementazione di tale metodo.

Sintassi

- public default void methodName () {/ * metodo body * /}

Osservazioni

Metodi predefiniti

- Può essere utilizzato all'interno di un'interfaccia per introdurre un comportamento senza forzare le sottoclassi esistenti per implementarlo.
- Può essere sovrascritto da sottoclassi o da una sotto-interfaccia.
- Non è consentito sovrascrivere i metodi nella classe java.lang.Object.
- Se una classe implementa più di un'interfaccia, eredita metodi predefiniti con firme di metodo identiche da ciascuno degli intefacenti, quindi deve eseguire l'override e fornire la propria interfaccia come se non fossero metodi predefiniti (come parte della risoluzione dell'ereditarietà multipla).
- Sebbene siano intesi a introdurre un comportamento senza rompere le implementazioni esistenti, le sottoclassi esistenti con un metodo statico con la stessa firma del metodo del metodo predefinito introdotto di recente verranno comunque interrotte. Tuttavia questo è vero anche in caso di introduzione di un metodo di istanza in una superclasse.

Metodi statici

- Può essere utilizzato all'interno di un'interfaccia, destinata principalmente a essere utilizzata come metodo di utilità per i metodi predefiniti.
- Non può essere sovrascritto da sottoclassi o da una sotto-interfaccia (è nascosta a loro). Tuttavia, come nel caso dei metodi statici, anche ora ogni classe o interfaccia può avere il suo.
- Non è consentito sovrascrivere i metodi di istanza nella classe java.lang.Object (come nel caso attuale delle sottoclassi).

Di seguito una tabella che riassume l'interazione tra sottoclasse e super-classe.

-	SUPER_CLASS-GRADO-METODO	SUPER_CLASS-STATIC-METODO
SUB_CLASS-GRADO-METODO	*le sostituzioni*	*genera-compiletime-errore*
SUB_CLASS-STATIC-METODO	*genera-compiletime-errore*	*nasconde*

Di seguito è riportata una tabella che riassume l'interazione tra interfaccia e classe di implementazione.

-	INTERFACCIA-default-METODO	INTERFACCIA-STATIC-METODO
IMPL_CLASS-GRADO-METODO	*le sostituzioni*	*nasconde*
IMPL_CLASS-STATIC-METODO	*genera-compiletime-errore*	*nasconde*

Riferimenti :

- http://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method
- https://docs.oracle.com/javase/tutorial/java/IandI/override.html

Examples

Utilizzo di base dei metodi predefiniti

```
/**
 * 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" );
    }
}
```

Le seguenti dichiarazioni

```
    new WithDefault().printString();
    new OverrideDefault().printString();
```

Produrrà questa uscita:

```
default implementation
overridden implementation
```

Accesso ad altri metodi di interfaccia nel metodo predefinito

Puoi anche accedere ad altri metodi di interfaccia dal tuo metodo predefinito.

```
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 seguente dichiarazione stamperà *3* :

```
System.out.println(new Sum().calculateSum());
```

I metodi predefiniti potrebbero essere utilizzati insieme ai metodi statici di interfaccia:

```
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 seguente dichiarazione stamperà anche 3:

```
System.out.println(new Sum().calculateSum());
```

Accesso ai metodi predefiniti sottoposti a override dalla classe di implementazione

Nelle classi, `super.foo()` apparirà solo nelle superclassi. Se si desidera chiamare un'implementazione predefinita da una superinterfaccia, è necessario qualificarsi `super` con il nome dell'interfaccia: `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
    }
}
```

Perché utilizzare i metodi predefiniti?

La semplice risposta è che ti consente di evolvere un'interfaccia esistente senza rompere le implementazioni esistenti.

Ad esempio, hai un'interfaccia `Swim` che hai pubblicato 20 anni fa.

```
public interface Swim {
    void backStroke();
```

```
}
```

Abbiamo fatto un ottimo lavoro, la nostra interfaccia è molto popolare, ci sono molte implementazioni in tutto il mondo e non hai il controllo sul loro codice sorgente.

```
public class FooSwimmer implements Swim {
    public void backStroke() {
         System.out.println("Do backstroke");
    }
}
```

Dopo 20 anni, hai deciso di aggiungere nuove funzionalità all'interfaccia, ma sembra che la nostra interfaccia sia bloccata perché interromperà le implementazioni esistenti.

Fortunatamente Java 8 introduce una nuova funzione chiamata metodo Default.

Ora possiamo aggiungere un nuovo metodo all'interfaccia `Swim` .

```
public interface Swim {
    void backStroke();
    default void sideStroke() {
        System.out.println("Default sidestroke implementation. Can be overridden");
    }
}
```

Ora tutte le implementazioni esistenti della nostra interfaccia possono ancora funzionare. Ma, soprattutto, possono implementare il metodo appena aggiunto nel loro tempo.

Uno dei principali motivi di questo cambiamento, e uno dei suoi maggiori usi, è nel framework delle collezioni Java. Oracle non ha potuto aggiungere un metodo `foreach` all'interfaccia Iterable esistente senza rompere tutto il codice esistente che ha implementato Iterable. Aggiungendo metodi predefiniti, l'implementazione Iterable esistente erediterà l'implementazione predefinita.

Classe, classe astratta e precedenza del metodo di interfaccia

Le implementazioni in classi, incluse le dichiarazioni astratte, hanno la precedenza su tutte le impostazioni predefinite dell'interfaccia.

- Il metodo astratto della classe ha la precedenza sul metodo di default dell'interfaccia .

```
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 seguente dichiarazione

```
new FooSwimmer().backStroke();
```

Produrrà

```
AbstractSwimmer.backStroke
```

- Il metodo Class ha la precedenza su Interface Default Method

```
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 seguente dichiarazione

```
new FooSwimmer().backStroke();
```

Produrrà

```
FooSwimmer.backStroke
```

Metodo predefinito collisione di eredità multipla

Considera il prossimo esempio:

```
public interface A {
    default void foo() { System.out.println("A.foo"); }
}

public interface B {
    default void foo() { System.out.println("B.foo"); }
}
```

Qui ci sono due interfacce che dichiarano `default` metodo di `default foo` con la stessa firma.

Se tenterai di `extend` queste due interfacce nella nuova interfaccia dovrai scegliere tra due, perché

Java ti obbliga a risolvere esplicitamente questa collisione.

Innanzitutto , è possibile dichiarare il metodo `foo` con la stessa firma `abstract` , che sovrascriverà il comportamento `A` e `B`

```
public interface ABExtendsAbstract extends A, B {
    @Override
    void foo();
}
```

E quando `implement ABExtendsAbstract` nella `class` dovrai fornire l'implementazione di `foo` :

```
public class ABExtendsAbstractImpl implements ABExtendsAbstract {
    @Override
    public void foo() { System.out.println("ABImpl.foo"); }
}
```

In **secondo luogo** , è possibile fornire un'implementazione `default` completamente nuova. È anche possibile riutilizzare il codice dei metodi `A` e `B` `foo` accedendo ai metodi predefiniti sovrascritti dalla classe di implementazione .

```
public interface ABExtends extends A, B {
    @Override
    default void foo() { System.out.println("ABExtends.foo"); }
}
```

E quando `implement ABExtends` nella `class` , `not` dovrai fornire un'implementazione `foo` :

```
public class ABExtendsImpl implements ABExtends {}
```

Leggi Metodi predefiniti online: https://riptutorial.com/it/java/topic/113/metodi-predefiniti

Capitolo 119: Modello di memoria Java

Osservazioni

Il modello di memoria Java è la sezione del JLS che specifica le condizioni in base alle quali un thread è garantito per vedere gli effetti delle scritture di memoria effettuate da un altro thread. La sezione pertinente nelle edizioni recenti è "JLS 17.4 Memory Model" (in Java 8 , Java 7 , Java 6)

C'è stata un'importante revisione del Java Memory Model in Java 5 che (tra le altre cose) ha cambiato il modo in cui ha funzionato la `volatile` . Da allora, il modello di memoria è rimasto sostanzialmente invariato.

Examples

Motivazione per il modello di memoria

Considera il seguente esempio:

```
public class Example {
    public int a, b, c, d;

    public void doIt() {
       a = b + 1;
       c = d + 1;
    }
}
```

Se questa classe viene utilizzata è un'applicazione a thread singolo, il comportamento osservabile sarà esattamente come ci si aspetterebbe. Per esempio:

```
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);
    }
}
```

produrrà:

```
0, 0
1, 1
```

Per quanto riguarda il thread "principale" , le istruzioni nel metodo `main()` e nel metodo `doIt()` verranno eseguite nell'ordine in cui sono scritte nel codice sorgente. Questo è un chiaro requisito della Java Language Specification (JLS).

Consideriamo ora la stessa classe utilizzata in un'applicazione multi-thread.

```
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);
        }
    }
}
```

Cosa stamperà?

Infatti, secondo JLS non è possibile prevedere che questo verrà stampato:

- Probabilmente vedrai alcune righe di `0, 0` per iniziare.
- Quindi probabilmente vedrai linee come `N, N` o `N, N + 1` .
- Potresti vedere linee come `N + 1, N`
- In teoria, potresti persino vedere che le righe `0, 0` continuano per sempre [1] .

1 - In pratica, la presenza delle istruzioni `println` può provocare una sincronizzazione fortuita e uno svuotamento della cache della memoria. È probabile che si nascondano alcuni degli effetti che causerebbero il comportamento di cui sopra.

Quindi, come possiamo spiegarli?

Riordino dei compiti

Una possibile spiegazione per risultati imprevisti è che il compilatore JIT ha cambiato l'ordine dei compiti nel metodo `doIt()` . Il JLS richiede che le istruzioni *vengano* eseguite in ordine *dalla prospettiva del* thread *corrente* . In questo caso, nulla nel codice del metodo `doIt()` può osservare l'effetto di un (ipotetico) riordino di queste due affermazioni. Ciò significa che il compilatore JIT sarebbe autorizzato a farlo.

Perché dovrebbe farlo?

In un tipico hardware moderno, le istruzioni della macchina vengono eseguite utilizzando una pipeline di istruzioni che consente a una sequenza di istruzioni di essere in diverse fasi. Alcune fasi di esecuzione delle istruzioni richiedono più tempo di altre e le operazioni di memoria richiedono tempi più lunghi. Un compilatore intelligente può ottimizzare il throughput delle istruzioni della pipeline ordinando le istruzioni per massimizzare la quantità di sovrapposizione. Ciò potrebbe comportare l'esecuzione di parti di istruzioni fuori servizio. Il JLS lo consente a condizione che non influenzi il risultato del calcolo *dal punto di vista del thread corrente* .

Effetti delle cache di memoria

Una seconda spiegazione possibile è l'effetto della memorizzazione nella memoria cache. In un'architettura di computer classica, ogni processore ha un piccolo set di registri e una maggiore quantità di memoria. L'accesso ai registri è molto più rapido dell'accesso alla memoria principale. Nelle architetture moderne, ci sono cache di memoria che sono più lente dei registri, ma più veloci della memoria principale.

Un compilatore sfrutterà questo cercando di mantenere copie di variabili nei registri o nelle cache di memoria. Se una variabile non ha *bisogno* di essere scaricata nella memoria principale, o non ha *bisogno* di essere letta dalla memoria, ci sono significativi vantaggi in termini di prestazioni nel non farlo. Nei casi in cui il JLS non richiede che le operazioni di memoria siano visibili a un altro thread, è probabile che il compilatore JIT Java non aggiunga le istruzioni "read barrier" e "write barrier" che imporranno le letture e le scritture della memoria principale. Ancora una volta, i vantaggi in termini di prestazioni di questo sono significativi.

Sincronizzazione corretta

Finora, abbiamo visto che il JLS consente al compilatore JIT di generare codice che rende più veloce il codice a thread singolo riordinando o evitando le operazioni di memoria. Ma cosa succede quando altri thread possono osservare lo stato delle variabili (condivise) nella memoria principale?

La risposta è che gli altri thread sono soggetti ad osservare stati variabili che sembrano impossibili ... basati sull'ordine di codice delle istruzioni Java. La soluzione a questo è utilizzare la sincronizzazione appropriata. I tre approcci principali sono:

- Uso dei mutex primitivi e dei costrutti `synchronized` .
- Utilizzo di variabili `volatile` .
- Utilizzo del supporto di concorrenza di livello superiore; ad es. classi nei pacchetti `java.util.concurrent` .

Ma anche con questo, è importante capire dove è necessaria la sincronizzazione e quali effetti su cui puoi fare affidamento. È qui che entra in gioco il modello di memoria Java.

Il modello di memoria

Il modello di memoria Java è la sezione del JLS che specifica le condizioni in base alle quali un thread è garantito per vedere gli effetti delle scritture di memoria effettuate da un altro thread. Il modello di memoria è specificato con un discreto grado di *rigore formale* e (come risultato) richiede una lettura dettagliata e attenta da comprendere. Ma il principio di base è che certi costrutti creano una relazione "succede prima" tra la scrittura di una variabile di un thread e una successiva lettura della stessa variabile di un altro thread. Se esiste la relazione "accade prima", il compilatore JIT è *obbligato* a generare codice che assicurerà che l'operazione di lettura veda il valore scritto dalla scrittura.

Armato di questo, è possibile ragionare sulla coerenza della memoria in un programma Java e decidere se questo sarà prevedibile e coerente per *tutte le* piattaforme di esecuzione.

Le relazioni avvenute prima

(Quanto segue è una versione semplificata di ciò che dice la specifica del linguaggio Java. Per una comprensione più approfondita, è necessario leggere la specifica stessa.)

Le relazioni Happens-before sono la parte del modello di memoria che ci consente di comprendere e ragionare sulla visibilità della memoria. Come dice JLS (JLS 17.4.5):

> "Due *azioni* possono essere ordinate da una relazione di *successo prima* : se si *verifica* un'azione *, prima di* un'altra, la prima è visibile e ordinata prima della seconda".

Cosa significa questo?

Azioni

Le azioni a cui si riferisce la citazione sopra sono specificate in JLS 17.4.2 . Esistono 5 tipi di azioni elencate dalle specifiche:

- Leggi: lettura di una variabile non volatile.
- Scrivi: scrittura di una variabile non volatile.
- Azioni di sincronizzazione:
    - Lettura volatile: lettura di una variabile volatile.
    - Scrittura volatile: scrittura di una variabile volatile.
    - Serratura. Blocco di un monitor
    - Sbloccare. Sbloccare un monitor.
    - La (sintetica) prima e ultima azione di una discussione.
    - Azioni che avviano un thread o rilevano che un thread è terminato.
- Azioni esterne. Un'azione che ha un risultato che dipende dall'ambiente in cui il programma.
- Discussione delle azioni di divergenza. Questi modellano il comportamento di certi tipi di loop infinito.

Ordine di programmazione e ordine di sincronizzazione

Questi due ordini (JLS 17.4.3 e JLS 17.4.4) regolano l'esecuzione delle istruzioni in un Java

L'ordine del programma descrive l'ordine di esecuzione dell'istruzione all'interno di un singolo

thread.

L'ordine di sincronizzazione descrive l'ordine dell'esecuzione dell'istruzione per due istruzioni collegate da una sincronizzazione:

- Un'azione di sblocco sul monitor si *sincronizza con* tutte le azioni di blocco successive su quel monitor.
- Una scrittura su una variabile volatile si *sincronizza con* tutte le letture successive della stessa variabile da qualsiasi thread.
- Un'azione che avvia un thread (cioè la chiamata a `Thread.start()`) si *sincronizza con* la prima azione nel thread che inizia (ovvero la chiamata al metodo `run()` del thread).
- L'inizializzazione predefinita dei campi si *sincronizza con* la prima azione in ogni thread. (Vedi la JLS per una spiegazione di questo.)
- L'azione finale in un thread si *sincronizza con* qualsiasi azione in un altro thread che rileva la terminazione; ad esempio il ritorno di una chiamata `join()` o `isTerminated()` che restituisce `true` .
- Se un thread interrompe un altro thread, la chiamata di interrupt nel primo thread si *sincronizza, con* il punto in cui un altro thread rileva che il thread è stato interrotto.

Happens-before Order

Questo ordinamento (JLS 17.4.5) è ciò che determina se una scrittura di memoria è garantita per essere visibile a una lettura di memoria successiva.

Più specificatamente, una lettura di una variabile `v` è garantita per osservare una scrittura su `v` se e solo se *avviene la* `write(v)` - *prima di* `read(v)` E non vi è alcun intervento scritto su `v` . Se ci sono scritture intervenienti, allora la `read(v)` può vedere i risultati di esse piuttosto che la prima.

Le regole che definiscono l' *accaduto, prima di* ordinare, sono le seguenti:

- **Regola Happens-Before # 1** - Se xey sono azioni dello stesso thread e x viene prima di y in *ordine di programma* , allora x *accade-prima di* y.
- **Happens-Before Rule # 2** - C'è un fronte di happen-before dalla fine di un costruttore di un oggetto all'inizio di un finalizzatore per quell'oggetto.
- **Happens-Before Rule # 3** - Se un'azione x si *sincronizza con* un'azione successiva y, allora x *accade prima di* y.
- **Happens-Before Rule # 4** - Se x *accade-prima che* y e y *succedano-prima di* z allora x *accade-prima di* z.

Inoltre, varie classi nelle librerie standard Java vengono specificate come definizione delle relazioni *precedenti* . Puoi interpretare ciò nel senso che accade in *qualche modo* , senza bisogno

di sapere esattamente come verrà soddisfatta la garanzia.

Succede prima che il ragionamento si applicasse ad alcuni esempi

Presenteremo alcuni esempi per mostrare come applicare *prima - il* ragionamento per verificare che le scritture siano visibili alle letture successive.

Codice a thread singolo

Come ci si aspetterebbe, le scritture sono sempre visibili alle letture successive in un programma a thread singolo.

```
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)
    }
}
```

Prima regola: numero 1:

1. L'azione `write(a)` *avviene prima* dell'azione `write(b)` .
2. L'azione `write(b)` *avviene prima* dell'azione `read(a)` .
3. L'azione di `read(a)` *verifica prima* dell'azione *di* `read(a)` .

Dalla regola 4 di Happens-Before:

4. `write(a)` *accade-prima di* `write(b)` E `write(b)` *accade-prima di* `read(a)` IMPLICA `write(a)` *accade-prima di* `read(a)` .
5. `write(b)` *succede-prima* `read(a)` E `read(a)` *succede-prima* `read(b)` IMPLICA `write(b)` *succede-prima* `read(b)` .

Riassumendo:

6. La relazione `write(a)` *happens-before* `read(a)` significa che l'istruzione `a + b` è garantita per vedere il valore corretto di `a` .
7. La relazione `write(b)` *happens-before* `read(b)` significa che l'istruzione `a + b` è garantita per vedere il valore corretto di `b` .

Comportamento di 'volatile' in un esempio con 2 thread

Useremo il seguente codice di esempio per esplorare alcune implicazioni del modello di memoria per "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)
    }
}
```

Innanzitutto, considera la seguente sequenza di istruzioni che coinvolgono 2 thread:

1. Viene creata una singola istanza di `VolatileExample` ; chiamalo `ve` ,
2. `ve.update(1, 2)` è chiamato in un thread, e
3. `ve.observe()` è chiamato in un altro thread.

Prima regola: numero 1:

1. L'azione `write(a)` *avviene prima* dell'azione `volatile-write(a)` .
2. L'azione `volatile-read(a)` *avviene prima* dell'azione `read(b)` .

Dalla regola Happens-Before # 2:

3. L'azione `volatile-write(a)` nel primo thread *avviene prima* dell'azione `volatile-read(a)` nel secondo thread.

Dalla regola 4 di Happens-Before:

4. L'azione `write(b)` nel primo thread *avviene prima* dell'azione `read(b)` nel secondo thread.

In altre parole, per questa particolare sequenza, è garantito che il secondo thread vedrà l'aggiornamento della variabile non volatile `b` creata dal primo thread. Tuttavia, dovrebbe anche essere chiaro che se le assegnazioni nel metodo di `update` erano il contrario, o il metodo `observe()` leggeva la variabile `b` prima di `a` , allora la catena degli *eventi precedenti* sarebbe stata interrotta. La catena sarebbe anche rotta se `volatile-read(a)` nel secondo thread non era successivo alla `volatile-write(a)` nel primo thread.

Quando la catena è rotta, non vi è alcuna *garanzia* che `observe()` vedrà il valore corretto di `b` .

Volatile con tre fili

Supponiamo di aggiungere un terzo thread nell'esempio precedente:

1. Viene creata una singola istanza di `VolatileExample` ; chiamalo `ve` ,
2. `update` chiamate a due thread:
   - `ve.update(1, 2)` è chiamato in un thread,
   - `ve.update(3, 4)` è chiamato nel secondo thread,
3. `ve.observe()` viene successivamente chiamato in una terza discussione.

Per analizzare completamente questo, dobbiamo considerare tutti i possibili intrecci delle affermazioni nel primo thread e nel secondo. Invece, considereremo solo due di loro.

Scenario n. 1 - Supponiamo che l' `update(1, 2)` preceda l' `update(3,4)` otteniamo questa sequenza:

```
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
```

In questo caso, è facile vedere che c'è un ininterrotto *successo - prima della* catena da `write(b, 3)` a `read(b)` . Inoltre non vi è alcun intervento scritto a `b` . Quindi, per questo scenario, il terzo thread è garantito per vedere `b` come valore `3` .

Scenario 2 - Supponiamo che l' `update(1, 2)` e l' `update(3,4)` sovrappongano e le azioni siano intercalate come segue:

```
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
```

Ora, mentre c'è una catena di successi *prima* di `write(b, 3)` per `read(b)` , c'è un'azione di `write(b, 1)` interposta eseguita dall'altro thread. Ciò significa che non possiamo essere certi quale valore `read(b)` vedrà.

(A parte questo: ciò dimostra che non possiamo fare affidamento sulla `volatile` per garantire la visibilità delle variabili non volatili, tranne in situazioni molto limitate).

Come evitare di aver bisogno di capire il modello di memoria

Il modello di memoria è difficile da capire e difficile da applicare. È utile se hai bisogno di ragionare sulla correttezza del codice multi-thread, ma non vuoi dover fare questo ragionamento per ogni applicazione multi-thread che scrivi.

Se si adottano i principi seguenti quando si scrive codice concorrente in Java, è possibile evitare in *gran parte* la necessità di ricorrere al ragionamento degli eventi *precedenti* .

- Utilizzare strutture di dati immutabili ove possibile. Una classe immutabile correttamente implementata sarà thread-safe e non introdurrà problemi di sicurezza del thread quando lo si utilizza con altre classi.
- Comprendere ed evitare "pubblicazione non sicura".
- Usa i mutex primitivi o `Lock` objects per sincronizzare l'accesso allo stato in oggetti mutabili che devono essere thread-safe [1] .
- Utilizzare `Executor` / `ExecutorService` o il framework fork join piuttosto che tentare di creare direttamente i thread di gestione.

- Utilizza le classi `java.util.concurrent che forniscono blocchi avanzati, semafori, latch e barriere, invece di usare wait / notify / notifyAll direttamente.
- Utilizzare le versioni `java.util.concurrent` di mappe, insiemi, elenchi, code e deques piuttosto che la sinconizzazione esterna di raccolte non concorrenti.

Il principio generale è di provare ad utilizzare le librerie di concomitanza integrate di Java piuttosto che la "rotazione della propria" concorrenza. Puoi fare affidamento su di loro lavorando, se li usi correttamente.

---

1 - Non tutti gli oggetti devono essere thread-safe. Ad esempio, se un oggetto o un oggetto è *limitato* da un thread (ovvero è accessibile solo a un thread), la sua sicurezza del thread non è rilevante.

Leggi Modello di memoria Java online: https://riptutorial.com/it/java/topic/6829/modello-di-memoria-java

Capitolo 120: Modifica del codice byte

Examples

Cos'è il Bytecode?

Bytecode è l'insieme di istruzioni utilizzate da JVM. Per illustrare questo, prendiamo questo programma Hello World.

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

Questo è ciò che trasforma quando viene compilato in 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
```

Qual è la logica dietro a questo?

getstatic - Riconosce il valore di un campo statico di una classe. In questo caso, *PrintStream "Out"* del *sistema* .

ldc - Spingi una costante nello stack. In questo caso, la stringa "Hello World"

invokevirtual - Invoca un metodo su un riferimento caricato sullo stack e mette il risultato in pila. I parametri del metodo sono anche presi dalla pila.

Bene, ci deve essere più giusto?

Ci sono 255 opcode, ma non tutti sono ancora implementati. Una tabella con tutti gli opcode attuali può essere trovata qui: Elenchi di istruzioni bytecode Java .

Come posso scrivere / modificare bytecode?

Esistono diversi modi per scrivere e modificare bytecode. È possibile utilizzare un compilatore, utilizzare una libreria o utilizzare un programma.

Per scrivere:

- Gelsomino

- Krakatau

Per la modifica:

- biblioteche
  - ASM
  - Javassist
  - BCEL - *Non supporta Java 8+*
- Utensili
  - Bytecode-Viewer
  - JBytedit
  - reJ - *Non supporta Java 8+*
  - JBE - *Non supporta Java 8+*

Mi piacerebbe saperne di più sul bytecode!

Probabilmente esiste una pagina di documentazione specifica per bytecode. Questa pagina si concentra sulla modifica di bytecode utilizzando diverse librerie e strumenti.

Come modificare i file jar con ASM

Innanzitutto le classi del vaso devono essere caricate. Useremo tre metodi per questo processo:

- loadClasses (File)
- readJar (JarFile, JarEntry, Mappa)
- 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 questi metodi, caricare e modificare un file jar diventa una semplice questione di modifica dei ClassNode in una mappa. In questo esempio sostituiremo tutte le stringhe nel jar con quelle in maiuscolo utilizzando l'API Tree.

```
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();
                }
            }
        }
    }
}
```

Ora che tutte le stringhe di ClassNode sono state modificate, è necessario salvare le modifiche. Per salvare le modifiche e avere un output funzionante, alcune cose devono essere fatte:

- Esporta ClassNodes in byte
- Caricare voci jar non di classe *(ad esempio Manifest.mf / altre risorse binarie in jar)* come byte
- Salva tutti i byte in un nuovo barattolo

Dall'ultima parte sopra, creeremo tre metodi.

- processNodes (Map <String, ClassNode> nodi)

- loadNonClasses (File jarFile)
- saveAsJar (Mappa <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");
```

I metodi usati:

```
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();
    }
}
```

Questo è tutto. Tutte le modifiche verranno salvate in "sample-edit.jar".

Come caricare un ClassNode come classe

```
/**
 * 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;
    }
}
```

Come rinominare le classi in un file 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 è una classe esistente nella libreria ASM. Tuttavia, consente solo di modificare i nomi delle classi. Se si desidera rinominare campi e metodi, è necessario creare la propria implementazione della classe Remapper.

Javassist Basic

Javassist è una libreria di strumentazione bytecode che consente di modificare bytecode iniettando codice Java che verrà convertito in bytecode da Javassist e aggiunto alla classe / metodo instrument in fase di esecuzione.

Consente di scrivere il primo trasformatore che effettivamente prende una classe ipotetica "com.my.to.be.instrumented.MyClass" e aggiunge alle istruzioni di ciascun metodo una chiamata di 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;
    }
}
```

Ora per utilizzare questo trasformatore (in modo che la nostra JVM chiamerà il metodo transform su ogni classe al momento del caricamento) dobbiamo aggiungere questo strumento o questo 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());
    }
}
```

L'ultimo passo per iniziare il nostro primo esperimento con lo strumento è registrare effettivamente questa classe di agenti sull'esecuzione della macchina JVM. Il modo più semplice per farlo è registrarlo con un'opzione nel comando shell:

```
java -javaagent:myAgent.jar MyJavaApplication
```

Come possiamo vedere, il progetto agent / transformer viene aggiunto come jar all'esecuzione di qualsiasi applicazione denominata MyJavaApplication che deve contenere una classe denominata "com.my.to.be.instrumented.MyClass" per eseguire effettivamente il nostro codice inserito.

Leggi Modifica del codice byte online: https://riptutorial.com/it/java/topic/3747/modifica-del-codice-byte

Capitolo 121: Modificatori di non accesso

introduzione

I modificatori di ***non*** accesso ***non modificano l'accessibilità di variabili*** e metodi, ma forniscono loro ***proprietà speciali*** .

Examples

finale

`final` in Java può riferirsi a variabili, metodi e classi. Ci sono tre semplici regole:

- la variabile finale non può essere riassegnata
- il metodo finale non può essere annullato
- la lezione finale non può essere estesa

usi

Buona pratica di programmazione

Alcuni sviluppatori considerano buona pratica contrassegnare una variabile finale quando è possibile. Se hai una variabile che non dovrebbe essere cambiata, dovresti contrassegnarla come definitiva.

Un uso importante della parola chiave `final` se per i parametri del metodo. Se si desidera sottolineare che un metodo non modifica i parametri di input, contrassegnare le proprietà come finali.

```
public int sumup(final List<Integer> ints);
```

Questo sottolinea che il metodo di `sumup` non cambierà gli `ints` .

Accesso alla classe interna

Se la tua classe interiore anonima vuole accedere a una variabile, la variabile dovrebbe essere contrassegnata come `final`

```
  public IPrintName printName(){
    String name;
    return new IPrintName(){
        @Override
        public void printName(){
            System.out.println(name);
        }
    };
}
```

Questa classe non si compila, come il `name` della variabile, non è definitiva.

Java SE 8

Le variabili finali in modo efficace sono un'eccezione. Queste sono variabili locali che vengono scritte solo una volta e potrebbero quindi essere rese definitive. È possibile accedere alle variabili finali in modo efficace anche dalle classi di anonymus.

variabile `final static`

Anche se il codice sottostante è completamente legale quando la variabile `final foo` non è `static` , in caso di `static` non verrà compilata:

```
class TestFinal {
    private final static List foo;

    public Test() {
        foo = new ArrayList();
    }
}
```

Il motivo è, ripetiamolo, la *variabile finale non può essere riassegnata* . Poiché `foo` è statico, è condiviso tra tutte le istanze della classe `TestFinal` . Quando viene creata una nuova istanza di una classe `TestFinal` , viene richiamato il suo costruttore e quindi viene riassegnato che il compilatore non consente. Un modo corretto per inizializzare la variabile `foo` in questo caso è:

```
class TestFinal {
    private static final List foo = new ArrayList();
    //..
}
```

o usando un inizializzatore statico:

```
class TestFinal {
    private static final List foo;
    static {
        foo = new ArrayList();
    }
    //..
}
```

`final` metodi `final` sono utili quando la classe base implementa alcune funzionalità importanti che la classe derivata non dovrebbe modificare. Sono anche più veloci dei metodi non definitivi, perché non vi è alcun concetto di tavolo virtuale coinvolto.

Tutte le classi wrapper in Java sono definitive, come `Integer` , `Long` etc. I creatori di queste classi non volevano che chiunque potesse estendere Integer nella propria classe e modificare il comportamento di base della classe Integer. Uno dei requisiti per rendere immutabile una classe è che le sottoclassi non possano sovrascrivere i metodi. Il modo più semplice per farlo è dichiarare la classe come `final` .

volatile

Il modificatore `volatile` viene utilizzato nella programmazione multi-thread. Se dichiari un campo come `volatile` , è un segnale ai thread che devono leggere il valore più recente, non uno memorizzato localmente. Inoltre, `volatile` letture e le scritture `volatile` sono garantite per essere atomiche (l'accesso a un non `volatile long` o `double` non è atomico), evitando così determinati errori di lettura / scrittura tra più thread.

```
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;
    }
}
```

statico

La parola chiave `static` viene utilizzata su una classe, un metodo o un campo per farli funzionare indipendentemente da qualsiasi istanza della classe.

- I campi statici sono comuni a tutte le istanze di una classe. Non hanno bisogno di un'istanza per accedervi.
- I metodi statici possono essere eseguiti senza un'istanza della classe in cui si trovano. Tuttavia, possono accedere solo ai campi statici di quella classe.
- Le classi statiche possono essere dichiarate all'interno di altre classi. Non hanno bisogno di un'istanza della classe in cui sono istanziati.

```
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
```

astratto

L'astrazione è un processo che nasconde i dettagli dell'implementazione e mostra solo funzionalità all'utente. Una classe astratta non può mai essere istanziata. Se una classe viene dichiarata come astratta, l'unico scopo è quello di estendere la classe.

```
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");
    }
}
```

sincronizzato

Il modificatore sincronizzato viene utilizzato per controllare l'accesso di un particolare metodo o di un blocco da più thread. Solo un thread può entrare in un metodo o un blocco dichiarato come sincronizzato. la parola chiave sincronizzata funziona sul blocco intrinseco di un oggetto, nel caso di un metodo sincronizzato il blocco degli oggetti correnti e il metodo statico utilizza l'oggetto classe. Qualsiasi thread che tenta di eseguire un blocco sincronizzato deve prima acquisire il blocco oggetto.

```
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 variabile dichiarata come transitoria non verrà serializzata durante la serializzazione dell'oggetto.

```
public transient int limit = 55;   // will not persist
public int b; // will persist
```

strictf

Java SE 1.2

Il modificatore strictfp viene utilizzato per i calcoli in virgola mobile. Questo modificatore rende la variabile in virgola mobile più coerente su più piattaforme e garantisce che tutti i calcoli in virgola mobile siano eseguiti secondo gli standard IEEE 754 per evitare errori di calcolo (errori di arrotondamento), overflow e underflow su architettura a 32 e 64 bit. Questo non può essere applicato su metodi astratti, variabili o costruttori.

```
// strictfp keyword can be applied on methods, classes and interfaces.

strictfp class A{}

strictfp interface M{}

class A{
    strictfp void m(){}
}
```

Leggi Modificatori di non accesso online: https://riptutorial.com/it/java/topic/4401/modificatori-di-non-accesso

Capitolo 122: moduli

Sintassi

- richiede java.xml;
- richiede java.xml pubblico; # espone il modulo ai dipendenti per l'uso
- esporta com.example.foo; # dipendenti possono utilizzare tipi pubblici in questo pacchetto
- esporta com.example.foo.impl in com.example.bar; # limita l'utilizzo a un modulo

Osservazioni

L'uso di moduli è incoraggiato ma non richiesto, ciò consente al codice esistente di continuare a lavorare in Java 9. Consente inoltre una transizione graduale al codice modulare.

Ogni codice non modulare viene inserito in un *modulo senza nome* quando viene compilato. Questo è un modulo speciale che è in grado di utilizzare i tipi da tutti gli altri moduli ma **solo dai pacchetti che hanno una dichiarazione di `exports` .**

Tutti i pacchetti nel *modulo senza nome* vengono esportati automaticamente.

Le parole chiave, ad esempio il `module` ecc., Sono limitate in uso all'interno della dichiarazione del modulo, ma possono essere utilizzate come identificatori altrove.

Examples

Definire un modulo base

I moduli sono definiti in un file denominato `module-info.java` , denominato descrittore di modulo. Dovrebbe essere inserito nella root del codice sorgente:

```
|-- module-info.java
|-- com
    |-- example
        |-- foo
            |-- Foo.java
        |-- bar
            |-- Bar.java
```

Ecco un semplice descrittore di modulo:

```
module com.example {
    requires java.httpclient;
    exports com.example.foo;
}
```

Il nome del modulo deve essere univoco e si consiglia di utilizzare la stessa notazione di denominazione DNS inversa utilizzata dai pacchetti per garantire ciò.

Il modulo `java.base` , che contiene le classi base di Java, è implicitamente visibile a qualsiasi modulo e non ha bisogno di essere incluso.

La dichiarazione dei `requires` ci consente di utilizzare altri moduli, nell'esempio il modulo `java.httpclient` viene importato.

Un modulo può anche specificare quali pacchetti `exports` e quindi renderli visibili ad altri moduli.

Il pacchetto `com.example.foo` dichiarato nella clausola `exports` sarà visibile ad altri moduli. Eventuali sotto-pacchetti di `com.example.foo` non verranno esportati, avranno bisogno delle proprie dichiarazioni di `export` .

Al contrario, `com.example.bar` che non è elencato nelle clausole di `exports` non sarà visibile ad altri moduli.

Leggi moduli online: https://riptutorial.com/it/java/topic/5286/moduli

Capitolo 123: Motore JavaScript Nashorn

introduzione

Nashorn è un motore JavaScript sviluppato in Java da Oracle ed è stato rilasciato con Java 8. Nashorn consente l'integrazione di Javascript in applicazioni Java tramite JSR-223 e consente di sviluppare applicazioni JavaScript standalone e offre migliori prestazioni di runtime e una migliore conformità con ECMA specifica Javascript normalizzata.

Sintassi

- ScriptEngineManager // Fornisce un meccanismo di individuazione e installazione per le classi ScriptEngine; utilizza un SPI (Service Provider Interface)
- ScriptEngineManager.ScriptEngineManager () // Costruttore consigliato
- ScriptEngine // Fornisce l'interfaccia per il linguaggio di scripting
- ScriptEngine ScriptEngineManager.getEngineByName (String shortName) // Metodo factory per la specifica implementazione
- Object ScriptEngine.eval (String script) // Esegue lo script specificato
- Object ScriptEngine.eval (Reader reader) // Carica e quindi esegue uno script dall'origine specificata
- ScriptContext ScriptEngine.getContext () // Restituisce il provider predefinito di binding, lettori e scrittori
- void ScriptContext.setWriter (writer writer) // Imposta la destinazione in cui inviare l'output dello script a

Osservazioni

Nashorn è un motore JavaScript scritto in Java e incluso in Java 8. Tutto ciò che serve è raggruppato nel pacchetto `javax.script` .

Si noti che `ScriptEngineManager` fornisce un'API generica che consente di ottenere motori di script per vari linguaggi di scripting (ovvero non solo Nashorn, non solo JavaScript).

Examples

Imposta variabili globali

```
// 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
```

Ciao 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
```

Esegui il file 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!');
```

Intercettare l'output dello script

```
// 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!'
```

Valuta le stringhe aritmetiche

```
// 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.
```

Utilizzo di oggetti Java in JavaScript in Nashorn

È possibile passare oggetti Java al motore Nashorn per essere elaborati nel codice Java. Allo stesso tempo, ci sono alcune costruzioni specifiche di JavaScript (e Nashorn), e non è sempre chiaro come funzionano con gli oggetti java.

Di seguito c'è una tabella che descrive il comportamento degli oggetti Java nativi all'interno delle costruzioni JavaScript.

Costruzioni testate:

1. Espressione in se clausola. Nell'espressione JS la clausola if non deve essere di tipo booleano a differenza di Java. Viene valutato come falso per i cosiddetti valori falsy (null, indefinito, 0, stringhe vuote ecc.)
2. per ogni istruzione Nashorn ha un tipo speciale di loop - per ciascuno - che può scorrere su diversi oggetti JS e Java.
3. Ottenere dimensioni dell'oggetto. Negli oggetti JS è presente una lunghezza della proprietà, che restituisce le dimensioni di una matrice o di una stringa.

risultati:

genere	Se	per ciascuno	.lunghezza
Java null	falso	Nessuna iterazione	**Eccezione**
Stringa vuota Java	falso	Nessuna iterazione	0
Stringa Java	vero	Iterare su caratteri stringa	Lunghezza della corda
Java Integer / Long	valore! = 0	Nessuna iterazione	non definito
Java ArrayList	vero	Itera sugli elementi	Lunghezza della lista
Java HashMap	vero	Itera sopra i valori	nullo
Java HashSet	vero	Fa scorrere gli oggetti	non definito

Recommendatons:

- È consigliabile utilizzare `if (some_string)` per verificare se una stringa non è nullo e non vuota
- `for each` può essere tranquillamente utilizzato per iterare su qualsiasi raccolta, e non genera eccezioni se la raccolta non è iterable, null o undefined
- Prima di ottenere la lunghezza di un oggetto, è necessario controllarlo per null o undefined (lo stesso vale per qualsiasi tentativo di chiamare un metodo o ottenere una proprietà dell'oggetto Java)

Implementazione di un'interfaccia dallo 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
    }
}
```

Imposta e ottieni variabili globali

```
// 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
```

Leggi Motore JavaScript Nashorn online: https://riptutorial.com/it/java/topic/166/motore-javascript-nashorn

Capitolo 124: Networking

Sintassi

- nuovo Socket ("localhost", 1234); // Si connette a un server all'indirizzo "localhost" e alla porta 1234
- nuovo SocketServer ("localhost", 1234); // Crea un server socket in grado di ascoltare nuovi socket all'indirizzo localhost e sulla porta 1234
- socketServer.accept (); // Accetta un nuovo oggetto Socket che può essere utilizzato per comunicare con il client

Examples

Comunicazione client e server di base tramite socket

Server: avviare e attendere le connessioni in entrata

```
//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.
}
```

Server: gestione dei client

Gestiremo ciascun client in un thread separato in modo che più client possano interagire con il server contemporaneamente. Questa tecnica funziona bene fino a quando il numero di client è basso (<< 1000 client, a seconda dell'architettura del sistema operativo e del carico previsto di ogni thread).

```
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();
```

Client: connettersi al server e inviare un messaggio

```
// 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();
```

Socket di chiusura e eccezioni di gestione

Gli esempi di cui sopra hanno lasciato alcune cose per renderli più facili da leggere.

1. Proprio come i file e altre risorse esterne, è importante che comunichiamo al sistema operativo quando abbiamo finito con loro. Quando abbiamo finito con un socket, chiama `socket.close()` per chiuderlo correttamente.

2. Gli zoccoli gestiscono le operazioni di I / O (ingresso / uscita) che dipendono da una varietà di fattori esterni. Ad esempio, cosa succede se l'altro lato si disconnette improvvisamente? Cosa succede se ci sono errori di rete? Queste cose sono fuori dal nostro controllo. Questo è il motivo per cui molte operazioni socket possono generare eccezioni, in particolare `IOException` .

Un codice più completo per il cliente sarebbe quindi qualcosa del genere:

```
// "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
}
```

Server e client di base: esempi completi

Server:

```
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();
        }
    }
}
```

Caricamento di TrustStore e KeyStore da 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();
            }

    }
}
```

L'intestazione di un KeyStore funziona allo stesso modo, tranne sostituire la parola `Trust` in un nome oggetto con `Key` . Inoltre, l'array `KeyManager[]` deve essere passato al primo argomento di `SSLContext.init` . Questo è `SSLContext.init(keyMangers, trustMangers, null)`

Esempio di socket: lettura di una pagina Web utilizzando un socket semplice

```
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);
        }
    }
}
```

Dovresti ricevere una risposta che inizi con `HTTP/1.1 200 OK` , che indica una normale risposta HTTP, seguita dal resto dell'intestazione HTTP, seguita dalla pagina Web non elaborata in formato HTML.

Nota che il metodo `readFully()` è importante per prevenire un'eccezione EOF prematura. L'ultima riga della pagina web può mancare un ritorno, per segnalare la fine della riga, quindi `readLine()` si lamenterà, quindi è necessario leggerlo a mano o utilizzare i metodi di utilità da Apache commons-io IOUtils

Questo esempio è inteso come una semplice dimostrazione di connessione a una risorsa esistente utilizzando un socket, non è un modo pratico per accedere alle pagine Web. Se è necessario accedere a una pagina Web utilizzando Java, è preferibile utilizzare una libreria client

HTTP esistente come il client HTTP di Apache o il client HTTP di Google

Comunicazione client / server di base tramite UDP (Datagram)

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);
    }
}
```

In questo caso, passiamo all'indirizzo del server, tramite un argomento (`args[0]`). La porta che stiamo utilizzando è 4160.

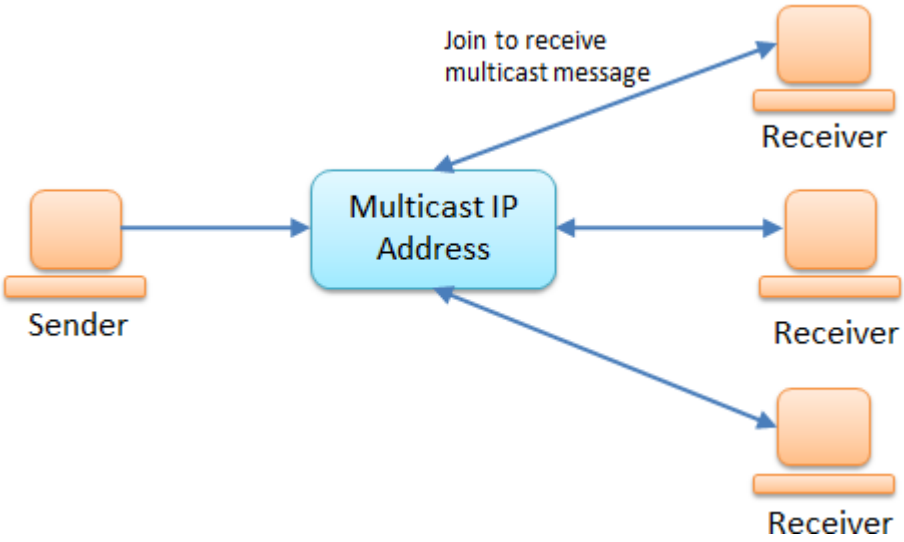
Server.java

```
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);
    }
}
```

Sul lato server, dichiarare un DatagramSocket sulla stessa porta che abbiamo inviato il nostro messaggio a (4160) e attendere una risposta.

multicasting

Il multicasting è un tipo di socket Datagram. A differenza dei datagrammi regolari, Multicasting non gestisce singolarmente ciascun client ma lo invia a un indirizzo IP e tutti i client sottoscritti riceveranno il messaggio.



Codice di esempio per un lato server:

```
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();
    }
}
```

Codice di esempio per un lato client:

```
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();
    }
}
```

Codice per l'esecuzione del server:

```
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();
    }
}
```

Codice per l'esecuzione di 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();
    }
}
```

Esegui prima il cliente: il cliente deve iscriversi all'IP prima che possa iniziare a ricevere qualsiasi pacchetto. Se si avvia il server e si chiama il metodo `send()` , quindi si crea un client (e si chiama `printMessage()`). Non succederà nulla perché il client si è connesso dopo che il messaggio è stato inviato.

Disattiva temporaneamente la verifica SSL (a scopo di test)

A volte in un ambiente di sviluppo o di test, la catena di certificati SSL potrebbe non essere stata

ancora completamente stabilita (ancora).

Per continuare a sviluppare e testare, puoi disattivare la verifica SSL a livello di codice installando un gestore di fiducia "all-trust":

```
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();
}
```

Download di un file utilizzando il canale

Se il file esiste già, verrà sovrascritto!

```
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 */ }
```

Gli appunti

- Non lasciare i blocchi di cattura vuoti!
- In caso di errore, controllare se il file remoto esiste
- Questa è un'operazione di blocco, può richiedere molto tempo con file di grandi dimensioni

Leggi Networking online: https://riptutorial.com/it/java/topic/149/networking

Capitolo 125: NIO - Networking

Osservazioni

`SelectionKey` definisce le diverse operazioni e informazioni selezionabili tra il selettore e il canale . In particolare, l' allegato può essere utilizzato per memorizzare informazioni relative alla connessione.

La gestione di `OP_READ` è piuttosto semplice. Tuttavia, è necessario prestare attenzione quando si ha a che fare con `OP_WRITE` : la maggior parte delle volte, i dati possono essere scritti in socket, quindi l'evento continuerà a sparare. Assicurati di registrare `OP_WRITE` solo prima di voler scrivere i dati (vedi la risposta).

Inoltre, `OP_CONNECT` dovrebbe essere cancellato una volta che il canale è stato connesso (perché, beh, *è* connesso. Vedi questo e quella risposte su SO). Quindi la rimozione di `OP_CONNECT` dopo `finishConnect()` riuscita.

Examples

Utilizzo del selettore per attendere gli eventi (ad esempio con OP_CONNECT)

NIO è apparso in Java 1.4 e ha introdotto il concetto di "Canali", che dovrebbero essere più veloci del normale I / O. Dal punto di vista della rete, `SelectableChannel` è il più interessante in quanto consente di monitorare diversi stati del Canale. Funziona in modo simile alla chiamata di sistema C `select()` : veniamo riattivati quando si verificano determinati tipi di eventi:

- connessione ricevuta (`OP_ACCEPT`)
- connessione realizzata (`OP_CONNECT`)
- dati disponibili in FIFO letto (`OP_READ`)
- i dati possono essere spinti per scrivere FIFO (`OP_WRITE`)

Permette la separazione tra la *rilevazione di* I / O socket (qualcosa può essere letto / scritto / ...) e l' *esecuzione* dell'I / O (lettura / scrittura / ...). Soprattutto, tutto il rilevamento I / O può essere eseguito in un singolo thread per più socket (client), mentre l'esecuzione dell'I / O può essere gestita in un pool di thread o in qualsiasi altro luogo. Ciò consente a un'applicazione di scalare facilmente il numero di client connessi.

Il seguente esempio mostra le basi:

1. Crea un `Selector`
2. Crea un `SocketChannel`
3. Registrare lo `SocketChannel` nel `Selector`
4. Loop con il `Selector` per rilevare gli eventi

```
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");
```

Darebbe il seguente risultato:

```
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
```

Leggi NIO - Networking online: https://riptutorial.com/it/java/topic/5513/nio---networking

Capitolo 126: NumberFormat

Examples

NumberFormat

Paesi diversi hanno formati di numeri diversi e considerando questo possiamo avere diversi formati usando Locale di java. L'uso della locale può aiutare nella formattazione

```
Locale locale = new Locale("en", "IN");
NumberFormat numberFormat = NumberFormat.getInstance(locale);
```

utilizzando il formato sopra è possibile eseguire varie attività

1. Numero di formato

   ```
   numberFormat.format(10000000.99);
   ```

2. Valuta il formato

   ```
   NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
   currencyFormat.format(10340.999);
   ```

3. Formato percentuale

   ```
   NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
   percentageFormat.format(10929.999);
   ```

4. Controllo numero di cifre

```
numberFormat.setMinimumIntegerDigits(int digits)
numberFormat.setMaximumIntegerDigits(int digits)
numberFormat.setMinimumFractionDigits(int digits)
numberFormat.setMaximumFractionDigits(int digits)
```

Leggi NumberFormat online: https://riptutorial.com/it/java/topic/7399/numberformat

Capitolo 127: Nuovo I / O file

Sintassi

- Paths.get (String first, String ... more) // Crea un'istanza Path in base agli elementi String
- Paths.get (URI uri) // Crea un'istanza Path da un URI

Examples

Creazione di percorsi

La classe `Path` è usata per rappresentare programmaticamente un percorso nel file system (e può quindi puntare a file e directory, anche a quelli non esistenti)

Un percorso può essere ottenuto usando i `Paths` classe helper:

```
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");
```

Recupero di informazioni su un percorso

Le informazioni su un percorso possono essere ottenute utilizzando i metodi di un oggetto `Path` :

- `toString()` restituisce la rappresentazione della stringa del percorso

  ```
  Path p1 = Paths.get("/var/www"); // p1.toString() returns "/var/www"
  ```

- `getFileName()` restituisce il nome del file (o, più specificamente, l'ultimo elemento del percorso

  ```
  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()` restituisce il numero di elementi che formano il percorso

  ```
  Path p1 = Paths.get("/var/www"); // p1.getNameCount() returns 2
  ```

- `getName(int index)` restituisce l'elemento nell'indice specificato

  ```
  Path p1 = Paths.get("/var/www"); // p1.getName(0) returns "var", p1.getName(1) returns
  "www"
  ```

- `getParent()` restituisce il percorso della directory padre

```
Path p1 = Paths.get("/var/www"); // p1.getParent().toString() returns "/var"
```

- `getRoot()` restituisce la radice del percorso

```
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:\\"
```

Manipolare i percorsi

Unire due percorsi

I percorsi possono essere uniti usando il metodo `resolve()` . Il percorso passato deve essere un percorso parziale, che è un percorso che non include l'elemento radice.

```
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"
```

Normalizzare un percorso

I percorsi possono contenere gli elementi `.` (che punta alla directory in cui ti trovi attualmente) e `..` (che punta alla directory principale).

Quando viene utilizzato in un percorso, `.` può essere rimosso in qualsiasi momento senza modificare la destinazione del percorso e `..` può essere rimosso insieme all'elemento precedente.

Con l'API Paths, questo viene fatto usando il metodo `.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"
```

Recupero delle informazioni usando il filesystem

Per interagire con il filesystem usate i metodi della classe `Files` .

Controllo dell'esistenza

Per verificare l'esistenza del file o della directory a cui punta il percorso, utilizzare i seguenti metodi:

```
Files.exists(Path path)
```

e

```
Files.notExists(Path path)
```

`!Files.exists(path)` non deve essere necessariamente uguale a `Files.notExists(path)` , perché ci sono tre possibili scenari:

- L'esistenza di un file o di una directory è verificata (`exists` restituisce `true` e `notExists` restituisce `false` in questo caso)
- L'inesistenza di un file o di una directory viene verificata (`exists` restituisce `false` e `notExists` restituisce `true`)
- Né l'esistenza né l'inesistenza di un file o di una directory possono essere verificati (ad esempio a causa di restrizioni di accesso): Entrambi `exists` e i non `nonExists` restituiscono false.

Verifica se un percorso punta a un file o a una directory

Questo viene fatto usando `Files.isDirectory(Path path)` e `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
```

Ottenere proprietà

Questo può essere fatto usando i seguenti metodi:

```
Files.isReadable(Path path)
Files.isWritable(Path path)
Files.isExecutable(Path path)

Files.isHidden(Path path)
Files.isSymbolicLink(Path path)
```

Ottenere il tipo MIME

```
Files.probeContentType(Path path)
```

Questo cerca di ottenere il tipo MIME di un file. Restituisce una stringa di tipo MIME, come questa:

- `text/plain` per i file di testo
- `text/html` per pagine HTML
- `application/pdf` per i file PDF
- `image/png` per i file PNG

Lettura dei file

I file possono essere letti in byte e in linea usando la classe `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()` facoltativamente un set di caratteri come parametro (l'impostazione predefinita è `StandardCharsets.UTF_8`):

```
List<String> linesOfContent = Files.readAllLines(p2, StandardCharsets.ISO_8859_1);
```

Scrivere file

I file possono essere scritti come morsi e linee usando la classe `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)
```

I file esistenti verranno sovrascritti, i file non esistenti verranno creati.

Leggi Nuovo I / O file online: https://riptutorial.com/it/java/topic/5519/nuovo-i---o-file

Capitolo 128: Oggetti immutabili

Osservazioni

Gli oggetti immutabili hanno uno stato fisso (nessun setter), quindi tutti gli stati devono essere noti al momento della creazione dell'oggetto.

Sebbene non sia tecnicamente necessario, è buona pratica rendere `final` tutti i campi. Questo renderà la classe immutable thread-safe (vedi Java Concurrency in Practice, 3.4.1).

Gli esempi mostrano diversi modelli che possono aiutare a raggiungere questo obiettivo.

Examples

Creare una versione immutabile di un tipo usando la copia difensiva.

Alcuni tipi e classi di base in Java sono fondamentalmente mutevoli. Ad esempio, tutti i tipi di array sono modificabili, così come le classi come `java.util.Data` . Questo può essere imbarazzante in situazioni in cui è imposto un tipo immutabile.

Un modo per affrontarlo consiste nel creare un wrapper immutabile per il tipo mutable. Ecco un semplice wrapper per una matrice di numeri interi

```
public class ImmutableIntArray {
    private final int[] array;

    public ImmutableIntArray(int[] array) {
        this.array = array.clone();
    }

    public int[] getValue() {
        return this.clone();
    }
}
```

Questa classe funziona usando la *copia difensiva* per isolare lo stato mutabile (l' `int[]`) da qualsiasi codice che possa mutarlo:

- Il costruttore usa `clone()` per creare una copia distinta dell'array dei parametri. Se il chiamante del costruttore ha successivamente modificato l'array di parametri, non avrebbe influenzato lo stato di `ImmutableIntArray` .
- Il metodo `getValue()` usa anche `clone()` per creare l'array che viene restituito. Se il chiamante dovesse cambiare l'array dei risultati, non avrebbe influenzato lo stato di `ImmutableIntArray` .

Potremmo anche aggiungere metodi a `ImmutableIntArray` per eseguire operazioni di sola lettura sull'array avvolto; ad esempio ottenere la sua lunghezza, ottenere il valore in un indice particolare, e così via.

Si noti che un tipo di wrapper immutabile implementato in questo modo non è compatibile con il tipo originale. Non puoi semplicemente sostituire il primo per quest'ultimo.

La ricetta per una lezione immutabile

Un oggetto immutabile è un oggetto il cui stato non può essere modificato. Una classe immutabile è una classe le cui istanze sono immutabili per progettazione e implementazione. La classe Java che viene presentata più comunemente come esempio di immutabilità è java.lang.String .

Quello che segue è un esempio stereotipato:

```
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 variazione su questo è dichiarare il costruttore come `private` e fornire invece un metodo `public static` .

La *ricetta standard* per una classe immutabile è la seguente:

- Tutte le proprietà devono essere impostate nel costruttore o nei metodi di fabbrica.
- Non dovrebbero esserci setter.
- Se è necessario includere setter per ragioni di compatibilità dell'interfaccia, non devono fare nulla o lanciare un'eccezione.
- Tutte le proprietà dovrebbero essere dichiarate come `private` e `final` .
- Per tutte le proprietà che fanno riferimento a tipi mutabili:
    - la proprietà deve essere inizializzata con una copia profonda del valore passato tramite il costruttore e
    - il getter della proprietà dovrebbe restituire una copia profonda del valore della proprietà.
- La classe dovrebbe essere dichiarata come `final` per impedire a qualcuno di creare una sottoclasse mutevole di una classe immutabile.

Un paio di altre cose da notare:

- L'immutabilità non impedisce all'oggetto di essere annullabile; ad es. `null` può essere assegnato a una variabile `String` .

- Se le proprietà di una classe immutable sono dichiarate `final` , le istanze sono intrinsecamente thread-safe. Ciò rende le classi immutabili un buon esempio per l'implementazione di applicazioni multi-thread.

Difetti di progettazione tipici che impediscono a una classe di essere immutabile

Utilizzo di alcuni setter, senza impostare tutte le proprietà necessarie nel / i costruttore / i

```
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); }
}
```

È facile dimostrare che la classe `Person` non è immutabile:

```
Person person = new Person("Joe");
person.setSurname("Average"); // NOT OK, change surname field after creation
```

Per risolvere il problema, elimina `setSurname()` e `setSurname()` il costruttore come segue:

```
public Person(String name, String surname) {
    this.name = name;
    this.surname = surname;
  }
```

Non contrassegnare le variabili di istanza come private e final

Dai un'occhiata alla seguente classe:

```
public final class Person {
    public String name;
    public Person(String name) {
        this.name = name;
     }
    public String getName() {
        return name;
    }

}
```

Il seguente frammento mostra che la suddetta classe non è immutabile:

```
Person person = new Person("Average Joe");
person.name = "Magic Mike"; // not OK, new name for person after creation
```

Per risolvere il problema, contrassegna semplicemente la proprietà del nome come `private` e `final` .

Esporre un oggetto mutevole della classe in un getter

Dai un'occhiata alla seguente classe:

```
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` classe dei `Names` sembra immutabile a prima vista, ma non è come mostra il seguente codice:

```
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"
```

Ciò è accaduto perché una modifica all'Elenco di riferimento restituito da `getNames()` può modificare l'elenco effettivo di `Names` .

Per risolvere questo problema, è sufficiente evitare i riferimenti che fanno riferimento a oggetti mutabili della classe *sia* per l'esecuzione di copie difensive ritorno, come segue:

```
public List<String> getNames() {
   return new ArrayList<String>(this.names); // copies elements
}
```

o progettando getter in modo che vengano restituiti solo altri *oggetti* e *primitive immutabili* , come segue:

```
public String getName(int index) {
    return names.get(index);
}
public int size() {
    return names.size();
}
```

Iniezione del costruttore con oggetto (i) che può essere modificato al di fuori della classe immutabile

Questa è una variazione del difetto precedente. Dai un'occhiata alla seguente classe:

```
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();
    }
}
```

Come classe `Names` prima, anche la classe `NewNames` sembra immutabile a prima vista, ma non lo è, infatti il seguente frammento dimostra il 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"
```

Per risolvere questo problema, come nel difetto precedente, è sufficiente creare copie difensive dell'oggetto senza assegnarlo direttamente alla classe immutabile, ovvero il costruttore può essere modificato come segue:

```
    public Names(List<String> names) {
        this.names = new ArrayList<String>(names);
    }
```

Lasciando ignorare i metodi della classe

Dai un'occhiata alla seguente classe:

```
public class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
      }
    public String getName() { return name;}
}
```

`Person` classe della `Person` sembra immutabile a prima vista, ma supponiamo che una nuova sottoclasse di `Person` sia definita:

```
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;
    }
}
```

ora la mutabilità di `Person` (im) può essere sfruttata attraverso il polimorfismo usando la nuova sottoclasse:

```
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
```

Per risolvere questo problema, *sia* contrassegnare la classe come `final` in modo che non possa essere esteso *o* dichiarare tutto il suo costruttore (s) come `private` .

Leggi Oggetti immutabili online: https://riptutorial.com/it/java/topic/2807/oggetti-immutabili

Capitolo 129: Oggetti sicuri

Sintassi

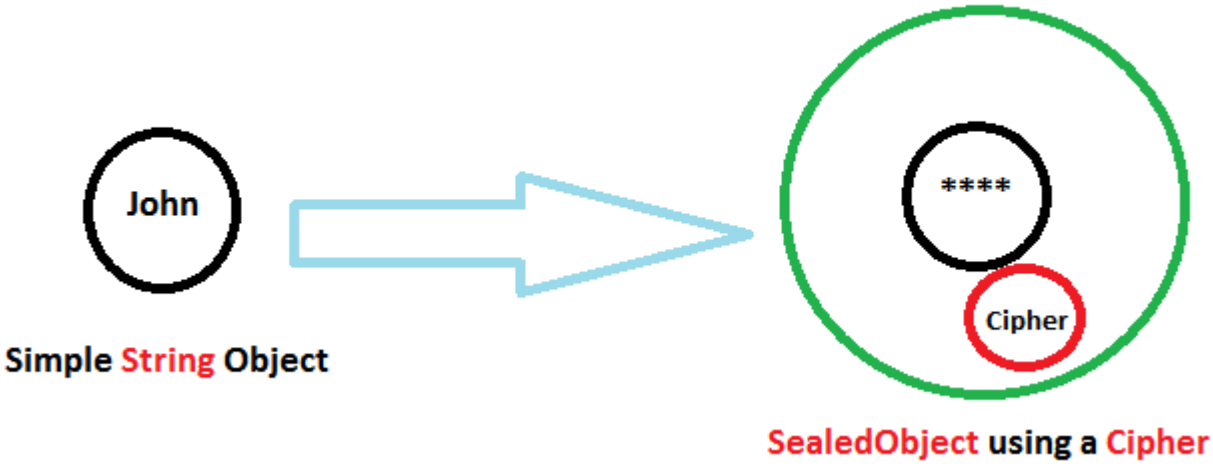
- SealedObject sealedObject = new SealedObject (obj, cipher);
- SignedObject signedObject = new SignedObject (obj, signingKey, signingEngine);

Examples

SealedObject (javax.crypto.SealedObject)

Questa classe consente a un programmatore di creare un oggetto e proteggere la sua riservatezza con un algoritmo crittografico.

Dato qualsiasi oggetto Serializable, si può creare un **SealedObject** che incapsula l'oggetto originale, in formato serializzato (cioè una "copia profonda"), e sigilla (crittografa) i suoi contenuti serializzati, usando un algoritmo crittografico come AES, DES, per proteggere la sua riservatezza Il contenuto crittografato può essere successivamente decrittografato (con l'algoritmo corrispondente utilizzando la chiave di decrittazione corretta) e de-serializzato, ottenendo l'oggetto originale.

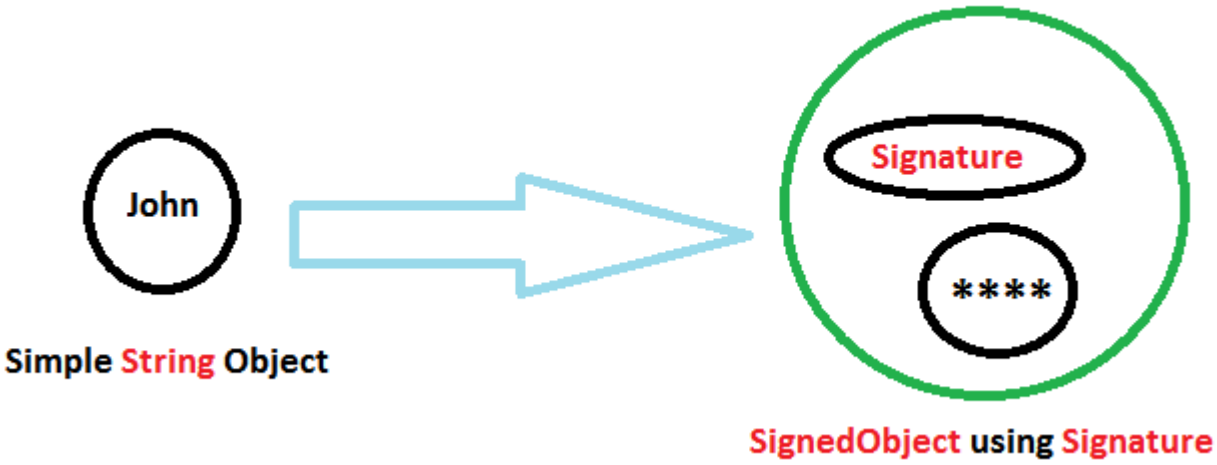



```
Serializable obj = new String("John");
// Generate key
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128);
SecretKey aesKey = kgen.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, aesKey);
SealedObject sealedObject = new SealedObject(obj, cipher);
System.out.println("sealedObject-" + sealedObject);
System.out.println("sealedObject Data-" + sealedObject.getObject(aesKey));
```

SignedObject (java.security.SignedObject)

SignedObject è una classe con lo scopo di creare oggetti runtime autentici la cui integrità non può essere compromessa senza essere rilevata.

Più in particolare, SignedObject contiene un altro oggetto Serializable, l'oggetto (to-be-) firmato e la sua 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());
```

Leggi Oggetti sicuri online: https://riptutorial.com/it/java/topic/5528/oggetti-sicuri

Capitolo 130: operatori

introduzione

Gli operatori nel linguaggio di programmazione Java sono simboli speciali che eseguono operazioni specifiche su uno, due o tre operandi e quindi restituiscono un risultato.

Osservazioni

Un *operatore* è un simbolo (o simboli) che indica a un programma Java di eseguire *un'operazione* su uno, due o tre *operandi* . Un operatore e i suoi operandi formano *un'espressione* (vedere l'argomento Espressioni). Gli operandi di un operatore sono essi stessi espressioni.

Questo argomento descrive i 40 operatori distinti definiti da Java. L'argomento delle espressioni separate spiega:

- in che modo operatori, operandi e altre cose sono combinati in espressioni,
- come vengono valutate le espressioni e
- come funzionano le espressioni di battitura, le conversioni e le espressioni.

Examples

The String Concatenation Operator (+)

Il simbolo `+` può significare tre operatori distinti in Java:

- Se non c'è un operando prima del `+` , allora è l'operatore unario Plus.
- Se ci sono due operandi e sono entrambi numerici. quindi è l'operatore di aggiunta binaria.
- Se ci sono due operandi e almeno uno di essi è una `String` , allora è l'operatore di concatenazione binaria.

Nel caso semplice, l'operatore Concatenazione unisce due stringhe per dare una terza stringa. Per esempio:

```
String s1 = "a String";
String s2 = "This is " + s1;    // s2 contains "This is a String"
```

Quando uno dei due operandi non è una stringa, viene convertito in una `String` come segue:

- Un operando il cui tipo è di tipo primitivo viene convertito *come se* chiamasse `toString()` sul valore box.
- Un operando il cui tipo è un tipo di riferimento viene convertito chiamando il metodo `toString()` dell'operando. Se l'operando è `null` o se il metodo `toString()` restituisce `null` , viene utilizzata la stringa letterale `"null"` .

Per esempio:

```
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 spiegazione per l'esempio `s5` è che il metodo `toString()` sui tipi di array è ereditato da `java.lang.Object` e il comportamento è quello di produrre una stringa che comprende il nome del tipo e l'hashcode dell'identità dell'oggetto.

L'operatore Concatenazione viene specificato per creare un nuovo oggetto `String` , tranne nel caso in cui l'espressione sia un'espressione costante. In quest'ultimo caso, l'espressione viene valutata al tipo di compilazione e il suo valore di runtime è equivalente a un valore letterale stringa. Ciò significa che non c'è nessun sovraccarico di runtime nel dividere un lungo letterale come questo:

```
String typing = "The quick brown fox " +
                "jumped over the " +
                "lazy dog";           // constant expression
```

Ottimizzazione ed efficienza

Come notato sopra, con l'eccezione delle espressioni costanti, ogni espressione di concatenazione di stringhe crea un nuovo oggetto `String` . Considera questo codice:

```
public String stars(int count) {
    String res = "";
    for (int i = 0; i < count; i++) {
        res = res + "*";
    }
    return res;
}
```

Nel metodo sopra, ogni iterazione del ciclo creerà una nuova `String` che è un carattere più lungo rispetto alla precedente iterazione. Ogni concatenazione copia tutti i caratteri nelle stringhe degli operandi per formare la nuova `String` . Quindi, le `stars(N)` :

- crea `N` nuovi oggetti `String` , e butta via tutto tranne l'ultimo,
- copia `N * (N + 1) / 2` caratteri e
- genera `O(N^2)` byte di spazzatura.

Questo è molto costoso per il grande `N` In effetti, qualsiasi codice che concatena stringhe in un ciclo è suscettibile di avere questo problema. Un modo migliore per scrivere questo sarebbe il seguente:

```
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, si dovrebbe impostare la capacità della `StringBuilder` , ma se questo non è pratico, la classe *crescerà* automaticamente la matrice supporto che il costruttore utilizza per tenere caratteri. (Nota: l'implementazione espande l'array di supporto in modo esponenziale. Questa strategia mantiene quella quantità di copia del personaggio su un `O(N)` piuttosto che su `O(N^2)` .)

Alcune persone applicano questo modello a tutte le concatenazioni di stringhe. Tuttavia, questo non è necessario perché il JLS *consente a* un compilatore Java di ottimizzare le concatenazioni di stringhe all'interno di una singola espressione. Per esempio:

```
String s1 = ...;
String s2 = ...;
String test = "Hello " + s1 + ". Welcome to " + s2 + "\n";
```

sarà *tipicamente* ottimizzato dal compilatore bytecode per qualcosa di simile;

```
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();
```

(Il compilatore JIT può ottimizzarlo ulteriormente se può dedurre che `s1` o `s2` non può essere `null` .) Ma si noti che questa ottimizzazione è consentita solo all'interno di una singola espressione.

In breve, se sei preoccupato dell'efficienza delle concatenazioni di stringhe:

- Ottimizza a mano se esegui una concatenazione ripetuta in un ciclo (o simile).
- Non ottimizzare a mano una singola espressione di concatenazione.

Gli operatori aritmetici (+, -, *, /,%)

Il linguaggio Java fornisce 7 operatori che eseguono l'aritmetica su valori interi e in virgola mobile.

- Ci sono due operatori `+` :
    - L'operatore di aggiunta binaria aggiunge un numero a un altro. (Esiste anche un operatore binario `+` che esegue la concatenazione di stringhe, come descritto in un esempio separato).
    - L'operatore unario plus non fa nulla oltre all'attivazione della promozione numerica (vedi sotto)
- Ci sono due `-` operatori:
    - L'operatore di sottrazione binaria sottrae un numero da un altro.
    - L'operatore unario meno equivale a sottrarre il suo operando da zero.

- L'operatore di moltiplicazione binario (*) moltiplica un numero per un altro.
- L'operatore di divisione binaria (/) divide un numero per un altro.
- L'operatore del resto [1] binario (%) calcola il resto quando un numero viene diviso per un altro.

1. Questo è spesso erroneamente definito come l'operatore "modulo". "Remainder" è il termine utilizzato da JLS. "Modulo" e "resto" non sono la stessa cosa.

Operand e tipi di risultato e promozione numerica

Gli operatori richiedono operandi numerici e producono risultati numerici. I tipi di operando possono essere qualsiasi tipo di carattere numerico primitivo (es. `byte` , `short` , `char` , `int` , `long` , `float` o `double`) o qualsiasi tipo di wrapper numerico definito in `java.lang` ; es. (`Byte` , `Character` , `Short` , `Integer` , `Long` , `Float` o `Double` .

Il tipo di risultato è determinato in base ai tipi dell'operando o degli operandi, come segue:

- Se uno degli operandi è `double` o `Double` , il tipo di risultato è `double` .
- Altrimenti, se uno degli operandi è `float` o `Float` , il tipo di risultato è `float` .
- Altrimenti, se uno degli operandi è `long` o `Long` , il tipo di risultato è `long` .
- Altrimenti, il tipo di risultato è `int` . Questo copre `byte` , `short` e `char` operandi così come `int.

Il tipo di risultato dell'operazione determina come viene eseguita l'operazione aritmetica e come vengono gestiti gli operandi

- Se il tipo di risultato è `double` , gli operandi vengono promossi a `double` e l'operazione viene eseguita utilizzando l'aritmetica in virgola mobile IEE 754 a 64 bit (binario a precisione doppia).
- Se il tipo di risultato è `float` , gli operandi vengono promossi in `float` e l'operazione viene eseguita utilizzando l'aritmetica in virgola mobile IEE 754 a 32 bit (binario a precisione singola).
- Se il tipo di risultato è `long` , gli operandi vengono promossi a `long` e l'operazione viene eseguita utilizzando l'aritmetica di numeri interi binari a due complementi con segno a 64 bit.
- Se il tipo di risultato è `int` , gli operandi vengono promossi a `int` e l'operazione viene eseguita utilizzando l'aritmetica di numeri interi binari a due complementi con segno a 32 bit.

La promozione viene eseguita in due fasi:

- Se il tipo di operando è un tipo di wrapper, il valore dell'operando è non *inserito* in un valore del tipo primitivo corrispondente.
- Se necessario, il tipo primitivo viene promosso al tipo richiesto:
    - La promozione di interi per `int` o `long` è senza perdite.
    - La promozione del `float` da `double` è senza perdite.
    - La promozione di un intero con un valore in virgola mobile può portare a una perdita di precisione. La conversione viene eseguita utilizzando la semantica IEE 768 "round-to-nearest".

Il significato della divisione

L'operatore / divide l'operando di sinistra `n` (il *dividendo*) e l'operando di destra `d` (il *divisore*) e produce il risultato `q` (il *quoziente*).

La divisione intera Java si arrotonda verso lo zero. La sezione 15.17.2 di JLS specifica il comportamento della divisione in intero Java come segue:

> Il quoziente prodotto per gli operatori `n` e `d` è un valore intero `q` cui grandezza è la più grande possibile mentre soddisfa `|d • q| ≤ |n|` . Inoltre, `q` è positivo quando `|n| ≥ |d|` e `n` e `d` hanno lo stesso segno, ma `q` è negativo quando `|n| ≥ |d|` e `n` e `d` hanno segni opposti.

Ci sono un paio di casi speciali:

- Se `n` è `MIN_VALUE` e il divisore è -1, si verifica un overflow di numero intero e il risultato è `MIN_VALUE` . Nessuna eccezione è lanciata in questo caso.
- Se `d` è 0, viene lanciata l'opzione ArithmeticException.

La divisione Java floating point ha altri casi da considerare. Tuttavia l'idea di base è che il risultato `q` è il valore più vicino alla soddisfazione `d . q = n` .

La divisione in virgola mobile non genererà mai un'eccezione. Invece, le operazioni che dividono per zero risultano in valori INF e NaN; vedi sotto.

Il significato di resto

A differenza di C e C ++, l'operatore rimanente in Java funziona con operazioni sia a interi che a virgola mobile.

Per i casi interi, il risultato di `a % b` è definito come il numero `r` tale che `(a / b) * b + r` è uguale a `a` , dove `/` , `*` e `+` sono gli operatori di numero intero Java appropriati. Questo vale in tutti i casi tranne quando `b` è zero. In quel caso, il resto risulta in `ArithmeticException` .

Dalla definizione sopra riportata risulta che `a % b` può essere negativa solo se `a` è negativa, ed è positiva solo se `a` è positiva. Inoltre, la grandezza di `a % b` è sempre inferiore alla grandezza di `b` .

L'operazione di resto del punto mobile è una generalizzazione del caso intero. Il risultato di `a % b` è il resto `r` è definito dalla relazione matematica `r = a - (b • q)` dove:

- `q` è un numero intero,
- è negativo solo se `a / b` è negativo e solo se `a / b` è positivo, e
- la sua grandezza è la più ampia possibile senza superare la grandezza del vero quoziente matematico di `a` e `b` .

Il resto in virgola mobile può produrre valori `INF` e `NaN` in casi limite come quando `b` è zero; vedi sotto. Non genererà un'eccezione.

Nota importante:

> Il risultato di un'operazione remainder a virgola mobile come calcolata da `%` **non è uguale** a quella prodotta dall'operazione resto definita da IEEE 754. Il resto IEEE 754 può essere calcolato utilizzando il metodo di libreria `Math.IEEEremainder` .

Overflow intero

I valori interi Java 32 e 64 bit sono firmati e utilizzano la rappresentazione binaria a due complementi. Ad esempio, l'intervallo di numeri rappresentabili come (32 bit) `int` -2^{31} a $+2^{31} - 1$.

Quando si aggiunge, si sottraggono o più due numeri interi a N bit (N == 32 o 64), il risultato dell'operazione potrebbe essere troppo grande per rappresentare un numero intero N bit. In questo caso, l'operazione porta a un *overflow di numeri interi* e il risultato può essere calcolato come segue:

- L'operazione matematica viene eseguita per fornire una rappresentazione intermedia del complemento a due dell'intero numero. Questa rappresentazione sarà più grande di N bit.
- Come risultato, vengono utilizzati i 32 o 64 bit inferiori della rappresentazione intermedia.

È necessario notare che l'overflow dei numeri interi non comporta eccezioni in nessuna circostanza.

Valori INF e NAN a virgola mobile

Java utilizza le rappresentazioni in virgola mobile IEE 754 per `float` e `double` . Queste rappresentazioni hanno alcuni valori speciali per rappresentare valori che non rientrano nel dominio dei numeri reali:

- I valori "infinito" o INF indicano numeri troppo grandi. Il valore `+INF` indica numeri troppo grandi e positivi. Il valore `-INF` denota numeri troppo grandi e negativi.
- Il "indefinito" / "non un numero" o NaN denotano valori derivanti da operazioni prive di significato.

I valori INF sono generati da operazioni mobili che causano un overflow, o dalla divisione per zero.

I valori NaN vengono prodotti dividendo zero per zero o calcolando zero resto zero.

Sorprendentemente, è possibile eseguire operazioni aritmetiche utilizzando gli operandi INF e NaN senza attivare eccezioni. Per esempio:

- Aggiungere + INF e un valore finito dà + INF.
- Aggiungere + INF e + INF dà + INF.
- Aggiungere + INF e -INF dà NaN.
- Dividere per INF fornisce +0.0 o -0.0.
- Tutte le operazioni con uno o più operandi NaN danno NaN.

Per i dettagli completi, fare riferimento alle sottosezioni rilevanti di JLS 15 . Si noti che questo è in

gran parte "accademico". Per i calcoli tipici, un `INF` o `NaN` significa che qualcosa è andato storto; ad esempio, hai dati di input incompleti o errati o il calcolo è stato programmato in modo errato.

The Equality Operators (==,! =)

Gli operatori `==` e `!=` Sono operatori binari che valutano `true` o `false` seconda che gli operandi siano uguali. L'operatore `==` restituisce `true` se gli operandi sono uguali e `false` altrimenti. L'operatore `!=` Dà `false` se gli operandi sono uguali e `true` altrimenti.

Questi operatori possono essere utilizzati operandi con tipi primitivi e di riferimento, ma il comportamento è significativamente diverso. Secondo JLS, ci sono in realtà tre serie distinte di questi operatori:

- Gli operatori booleani `==` e `!=` .
- Gli operatori numerici `==` e `!=` .
- Gli operatori di riferimento `==` e `!=` .

Tuttavia, in tutti i casi, il tipo di risultato degli operatori `==` e `!=` È `boolean` .

Gli operatori numerici `==` e `!=`

Quando uno (o entrambi) degli operandi di un operatore `==` o `!=` È un tipo numerico primitivo (`byte` , `short` , `char` , `int`, `long` , `float` o `double`), l'operatore è un confronto numerico. Il secondo operando deve essere un tipo numerico primitivo o un tipo numerico in scatola.

Il comportamento di altri operatori numerici è il seguente:

1. Se uno degli operandi è di tipo boxed, è unbox.
2. Se uno degli operandi ora è un `byte` , `short` o `char` , viene promosso a un `int` .
3. Se i tipi di operandi non sono gli stessi, l'operando con il tipo "più piccolo" viene promosso al tipo "più grande".
4. Il confronto viene quindi eseguito come segue:
    - Se gli operandi promossi sono `int` o `long` i valori vengono testati per vedere se sono identici.
    - Se gli operandi promossi sono `float` o `double` allora:
        - le due versioni di zero (`+0.0` e `-0.0`) sono considerate uguali
        - un valore `NaN` viene trattato come non uguale a nulla, e
        - altri valori sono uguali se le loro rappresentazioni IEEE 754 sono identiche.

Nota: è necessario fare attenzione quando si utilizza `==` e `!=` Per confrontare i valori in virgola mobile.

Gli operatori booleani `==` e `!=`

Se entrambi gli operandi sono `boolean` , o uno è `boolean` e l'altro è `Boolean` , questi operatori sono gli operatori booleani `==` e `!=` . Il comportamento è il seguente:

1. Se uno degli operandi è un `Boolean` , è unbox.
2. Gli operandi unbox vengono testati e il risultato booleano viene calcolato in base alla seguente tabella di verità

UN	B	A == B	A! = B
falso	falso	vero	falso
falso	vero	falso	vero
vero	falso	falso	vero
vero	vero	vero	falso

Ci sono due "trappole" che rendono consigliabile usare `==` e `!=` parsimonia con valori di verità:

- Se si utilizza `==` o `!=` Per confrontare due oggetti `Boolean` , vengono utilizzati gli operatori di riferimento. Questo potrebbe dare un risultato inaspettato; vedi Pitfall: usando == per confrontare oggetti di wrapper primitivi come Integer
- L'operatore `==` può essere erroneamente digitato come `=` . Per la maggior parte dei tipi di operandi, questo errore porta a un errore di compilazione. Tuttavia, per `Boolean` operandi `boolean` e `Boolean` l'errore porta a comportamenti di runtime non corretti; see Pitfall - Usando '==' per testare un booleano

Gli operatori di riferimento `==` e `!=`

Se entrambi gli operandi sono riferimenti a oggetti, gli operatori `==` e `!=` Testano se i due operandi si **riferiscono allo stesso oggetto** . Questo spesso non è quello che vuoi. Per verificare se due oggetti sono uguali *per valore* , dovrebbe essere usato il metodo `.equals()` .

```
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
```

Avvertenza: l'uso di `==` e `!=` Per confrontare i valori di `String` **non** è **corretto** nella maggior parte dei casi; vedere http://www.Scriptutorial.com/java/example/16290/pitfall--using- to- compare- strings . Un problema simile si applica ai tipi di wrapper primitivi; vedi http://www.Scriptutorial.com/java/example/8996/pitfall--using- a- compare- primitive-wrappers- objects- such- as- integer .

Informazioni sui casi limite di NaN

JLS 15.21.1 afferma quanto segue:

> Se uno degli operandi è `NaN` , il risultato di `==` è `false` ma il risultato di `!=` È `true` . In effetti, il test `x != x` è `true` se e solo se il valore di `x` è `NaN` .

Questo comportamento è (per la maggior parte dei programmatori) inaspettato. Se si verifica se un valore `NaN` è uguale a se stesso, la risposta è "No non lo è!". In altre parole, `==` non è *riflessivo* per i valori `NaN` .

Tuttavia, questa non è una "stranezza" di Java, questo comportamento è specificato negli standard IEEE 754 a virgola mobile, e scoprirete che è implementato dalla maggior parte dei moderni linguaggi di programmazione. (Per ulteriori informazioni, consultare http://stackoverflow.com/a/1573715/139985 ... notando che questo è scritto da qualcuno che era "nella stanza quando sono state prese le decisioni"!)

Gli operatori di incremento / decremento (++ / -)

Le variabili possono essere incrementate o decrementate di 1 usando rispettivamente gli operatori `++` e `--` .

Quando gli operatori `++` e `--` seguono le variabili, sono chiamati rispettivamente **post-incremento** e **post-decremento** .

```
int a = 10;
a++; // a now equals 11
a--; // a now equals 10 again
```

Quando gli operatori `++` e `--` precedono le variabili, le operazioni sono chiamate rispettivamente **pre-incremento** e **pre-decremento** .

```
int x = 10;
--x; // x now equals 9
++x; // x now equals 10
```

Se l'operatore precede la variabile, il valore dell'espressione è il valore della variabile dopo essere stato incrementato o decrementato. Se l'operatore segue la variabile, il valore dell'espressione è il valore della variabile prima di essere incrementato o decrementato.

```
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
```

Fare attenzione a non sovrascrivere post-incrementi o decrementi. Ciò accade se si utilizza un operatore post-in / decrement alla fine di un'espressione che viene riassegnato alla variabile in / decrementata stessa. L'in / decremento non avrà effetto. Anche se la variabile sul lato sinistro viene incrementata correttamente, il suo valore verrà immediatamente sovrascritto con il risultato valutato in precedenza dal lato destro dell'espressione:

```
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!)
```

Corretta:

```
int x = 0;
x = x++ + 1 + x;        // evaluates to x = 0 + 1 + 1
x++;                    // adds 1
System.out.println(x);  // prints 3
```

L'operatore condizionale (? :)

Sintassi

{condition-to-valutare} **?** *{statement-execution-on-true}* **:** *{istruzione-eseguito-su-falso}*

Come mostrato nella sintassi, l'Operatore Condizionale (noto anche come Operatore Ternario [1]) usa il `?` (punto interrogativo) e `:` (due punti) caratteri per abilitare un'espressione condizionale di due possibili risultati. Può essere usato per sostituire i blocchi `if-else` più lunghi per restituire uno dei due valori in base alla condizione.

```
result = testCondition ? value1 : value2
```

È equivalente a

```
if (testCondition) {
    result = value1;
} else {
    result = value2;
}
```

Può essere letto come ***"Se testCondition è vero, imposta il risultato su value1; altrimenti, imposta il risultato su value2 ".***

Per esempio:

```
// get absolute value using conditional operator
a = -10;
int absValue = a < 0 ? -a : a;
System.out.println("abs = " + absValue); // prints "abs = 10"
```

È 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 comune

È possibile utilizzare l'operatore condizionale per assegnazioni condizionali (come il controllo nullo).

```
String x = y != null ? y.toString() : ""; //where y is an object
```

Questo esempio è equivalente a:

```
String x = "";

if (y != null) {
    x = y.toString();
}
```

Poiché l'operatore condizionale ha la seconda precedenza più bassa, al di sopra degli operatori di assegnazione , raramente è necessario utilizzare la parentesi intorno alla *condizione* , ma è necessaria una parentesi attorno all'intero costrutto dell'operatore condizionale quando combinato con altri operatori:

```
// no parenthesis needed for expressions in the 3 parts
10 <= a && a < 19 ? b * 5 : b * 7

// parenthesis required
7 * (a > 0 ? 2 : 5)
```

La nidificazione degli operatori condizionali può essere eseguita anche nella terza parte, dove funziona più come concatenare o come un'istruzione 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 - Sia Java Language Specification che Java Tutorial chiamano l'operatore (`? :`) L'operatore *condizionale* . Il Tutorial dice che è "noto anche come Operatore Ternario" poiché è (attualmente) l'unico operatore ternario definito da Java. La terminologia "Operatore condizionale" è coerente con C e C ++ e altre lingue con un operatore equivalente.

Gli operatori bitwise e logici (~, &, |, ^)

Il linguaggio Java fornisce 4 operatori che eseguono operazioni bit a bit o logiche su operandi interi o booleani.

- L'operatore complementare (`~`) è un operatore unario che esegue un'inversione bit per bit o logica dei bit di un operando; vedi JLS 15.15.5. .
- L'operatore AND (`&`) è un operatore binario che esegue un "e" logico o bit a bit di due operandi; vedi JLS 15.22.2. .
- L'operatore OR (`|`) è un operatore binario che esegue un "inclusivo" o "logico" di bit di due operandi; vedi JLS 15.22.2. .
- L'operatore XOR (`^`) è un operatore binario che esegue un "esclusivo o" bitwise o logico di due operandi; vedi JLS 15.22.2. .

Le operazioni logiche eseguite da questi operatori quando gli operandi sono booleani possono essere riassunte come segue:

UN	B	~ A	A & B	A | B	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

Nota che per gli operandi interi, la tabella sopra descrive cosa succede per i singoli bit. Gli operatori operano effettivamente su tutti i 32 o 64 bit dell'operando o degli operandi in parallelo.

Tipi di operandi e tipi di risultato.

Le normali conversioni aritmetiche si applicano quando gli operandi sono numeri interi. Casi d'uso comuni per gli operatori bit a bit

---

L'operatore `~` viene utilizzato per invertire un valore booleano o modificare tutti i bit in un operando intero.

L'operatore `&` viene usato per "mascherare" alcuni dei bit in un operando intero. Per esempio:

```
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
```

Il `|` l'operatore è usato per combinare i valori di verità di due operandi. Per esempio:

```
int word2 = 0b01011111;
// Combine the bottom 2 bits of word1 with the top 30 bits of word2
int combined = (word & mask) | (word2 & ~mask);   // -> 0b01011110
```

L'operatore ^ viene utilizzato per alternare o "sfogliare" i bit:

```
int word3 = 0b00101010;
int word4 = word3 ^ mask;             // -> 0b00101001
```

Per ulteriori esempi sull'uso degli operatori bit a bit, vedere Manipolazione bit

L'istanza di operatore

Questo operatore controlla se l'oggetto è di un particolare tipo di classe / interfaccia. **L'** operatore **instanceof** è scritto come:

```
( Object reference variable ) instanceof  (class/interface type)
```

Esempio:

```
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 );
   }
}
```

Ciò produrrebbe il seguente risultato:

```
true
```

Questo operatore restituirà comunque true se l'oggetto da confrontare è l'assegnazione compatibile con il tipo sulla destra.

Esempio:

```
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 );
   }
}
```

Ciò produrrebbe il seguente risultato:

```
true
```

The Assignment Operators (=, + =, - =, * =, / =,% =, << =, >> =, >>> =, & =, | = e ^ =)

L'operando di sinistra per questi operatori deve essere una variabile non finale o un elemento di una matrice. L'operando destro deve essere *compatibile* con l'operando di sinistra. Ciò significa che i tipi devono essere uguali oppure che il tipo di operando destro deve essere convertibile nel tipo di operandi di sinistra mediante una combinazione di boxing, unboxing o widening. (Per i dettagli completi, fare riferimento a JLS 5.2 .)

Il significato preciso degli operatori "operazione e assegnazione" è specificato da JLS 15.26.2 come:

> Un'espressione di assegnazione composta della forma `E1 op= E2` è equivalente a `E1 = (T) ((E1) op (E2))` , dove `T` è il tipo di `E1` , tranne che `E1` viene valutato solo una volta.

Nota che c'è un cast implicito del tipo prima dell'assegnazione finale.

1. `=`

L'operatore di assegnazione semplice: assegna il valore dell'operando di destra all'operando di sinistra.

> Esempio: `c = a + b` aggiungerà il valore di `a + b` al valore di `c` e assegnarlo a `c`

2. `+=`

L'operatore "aggiungi e assegna": aggiunge il valore dell'operando di destra al valore dell'operando di sinistra e assegna il risultato all'operando di sinistra. Se l'operando di sinistra ha tipo `String` , allora questo è un operatore "concatena e assegna".

> Esempio: `c += a` è all'incirca uguale a `c = c + a`

3. `-=`

L'operatore "sottrazione e assegnazione": sottrae il valore dell'operando di destra dal valore dell'operando della mano sinistra e assegna il risultato all'operando della mano sinistra.

> Esempio: `c -= a` è all'incirca uguale a `c = c - a`

4. `*=`

L'operatore "moltiplica e assegna": moltiplica il valore dell'operando di destra per il valore dell'operando di sinistra e assegna il risultato all'operando di sinistra. .

> Esempio: `c *= a` è all'incirca uguale a `c = c * a`

5. `/=`

L'operatore "divide e assegna": divide il valore dell'operando di destra per il valore dell'operando di sinistra e assegna il risultato all'operando di sinistra.

Esempio: `c /*= a` è approssimativamente uguale a `c = c / a`

6. `%=`

L'operatore "modulus and assign": calcola il modulo del valore dell'operando di destra per il valore dell'operando di sinistra e assegna il risultato all'operando di sinistra.

Esempio: `c %*= a` è approssimativamente uguale a `c = c % a`

7. `<<=`

L'operatore "spostamento a sinistra e assegnazione".

Esempio: `c <<= 2` equivale all'incirca a `c = c << 2`

8. `>>=`

L'operatore "aritmetico destro cambia e assegna".

Esempio: `c >>= 2` equivale all'incirca a `c = c >> 2`

9. `>>>=`

L'operatore "spostamento logico destro e assegnazione".

Esempio: `c >>>= 2` equivale all'incirca a `c = c >>> 2`

10. `&=`

L'operatore "bit a bit e assegnazione".

Esempio: `c &= 2` è approssimativamente uguale a `c = c & 2`

11. `|=`

L'operatore "bit a bit o assign".

Esempio: `c |= 2` equivale all'incirca a `c = c | 2`

12. `^=`

L'operatore "bitwise exclusive or and assign".

Esempio: `c ^= 2` equivale all'incirca a `c = c ^ 2`

Gli operatori condizionali e condizionali o (&& e ||)

Java fornisce un operatore condizionale e / o condizionale, che entrambi accettano uno o due operandi di tipo `boolean` e producono un risultato `boolean` . Questi sono:

- `&&` - l'operatore AND condizionale,
- `||` - gli operatori OR condizionali. La valutazione di `<left-expr> && <right-expr>` è equivalente al seguente pseudo-codice:

```
{
   boolean L = evaluate(<left-expr>);
   if (L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return false;
   }
}
```

La valutazione di `<left-expr> || <right-expr>` è equivalente al seguente pseudo-codice:

```
{
   boolean L = evaluate(<left-expr>);
   if (!L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return true;
   }
}
```

Come illustrato dallo pseudo-codice sopra, il comportamento degli operatori di cortocircuito è equivalente all'utilizzo delle istruzioni `if` / `else` .

Esempio: uso di && come guardia in un'espressione

L'esempio seguente mostra il modello di utilizzo più comune per l'operatore `&&` . Confrontare queste due versioni di un metodo per verificare se un `Integer` fornito è zero.

```
public boolean isZero(Integer value) {
    return value == 0;
}

public boolean isZero(Integer value) {
    return value != null && value == 0;
}
```

La prima versione funziona nella maggior parte dei casi, ma se l'argomento `value` è `null` , verrà `NullPointerException` una `NullPointerException` .

Nella seconda versione abbiamo aggiunto un test di "guardia". Il `value != null && value == 0` espressione viene valutata eseguendo prima il `value != null` test. Se il test `null` esito positivo (vale a dire vale `true`), viene valutato il `value == 0` espressione. Se il test `null` fallisce, allora la valutazione del `value == 0` viene saltata (cortocircuitata) e *non si* ottiene `NullPointerException` .

Esempio: usare && per evitare un calcolo costoso

L'esempio seguente mostra come `&&` può essere usato per evitare un calcolo relativamente costoso:

```
public boolean verify(int value, boolean needPrime) {
    return !needPrime | isPrime(value);
}

public boolean verify(int value, boolean needPrime) {
    return !needPrime || isPrime(value);
}
```

Nella prima versione, entrambi gli operandi del `|` sarà sempre valutato, quindi il metodo (costoso) `isPrime` verrà chiamato inutilmente. La seconda versione evita la chiamata non necessaria utilizzando `||` invece di `|` .

The Shift Operators (<<, >> e >>>)

Il linguaggio Java fornisce tre operatori per eseguire lo spostamento bit per bit su valori interi a 32 e 64 bit. Questi sono tutti operatori binari con il primo operando come valore da spostare, e il secondo operando che dice quanto lontano spostare.

- L'operatore `<<` o *spostamento a sinistra* sposta il valore dato dal primo operando *verso sinistra* per il numero di posizioni di bit date dal secondo operando. Le posizioni vuote all'estremità destra sono piene di zeri.
- L'operatore '>>' o *spostamento aritmetico* sposta il valore dato dal primo operando a *destra* del numero di posizioni di bit date dal secondo operando. Le posizioni vuote all'estremità sinistra vengono riempite copiando il bit più a sinistra. Questo processo è noto come *estensione del segno* .
- L'operatore ">>>" o *spostamento logico destro* sposta il valore dato dal primo operando a *destra* del numero di posizioni di bit fornite dal secondo operando. Le posizioni vuote all'estremità sinistra sono piene di zeri.

Gli appunti:

1. Questi operatori richiedono un valore `int` o `long` come primo operando e producono un valore con lo stesso tipo del primo operando. (Sarà necessario utilizzare un cast di tipo esplicito quando si assegna il risultato di uno spostamento a una variabile `byte` , `short` o `char` .)
2. Se si utilizza un operatore di spostamento con un primo operando che è un `byte` , `char` o `short` , viene promosso a un `int` e l'operazione produce un `int` .)
3. Il secondo operando viene ridotto *modulo il numero di bit dell'operazione* per dare l'ammontare dello spostamento. Per ulteriori informazioni sul **concetto matematico mod** , vedere Esempi di moduli .

4. I bit che vengono spostati dalla fine sinistra o destra dall'operazione vengono scartati. (Java non fornisce un operatore di "rotazione" primitivo).

5. L'operatore di spostamento aritmetico equivale a dividere un numero (a complemento di due) per una potenza di 2.

6. L'operatore di spostamento a sinistra equivale a moltiplicare un numero (a complemento di due) per una potenza di 2.

La seguente tabella ti aiuterà a vedere gli effetti dei tre operatori di turno. (I numeri sono stati espressi in notazione binaria per aiutare la visualizzazione).

operand1	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	0b0010000000000100
0b1000000000001011	31	0b1000000000000000	0b1111111111111111	0b0000000000000001

Ci sono esempi di utenti di operatori di shift nella manipolazione di Bit

L'operatore Lambda (->)

Da Java 8 in poi, l'operatore Lambda (`->`) è l'operatore utilizzato per introdurre un'espressione Lambda. Esistono due sintassi comuni, come illustrato da questi esempi:

Java SE 8

```
a -> a + 1              // a lambda that adds one to its argument
a -> { return a + 1; }  // an equivalent lambda using a block.
```

Un'espressione lambda definisce una funzione anonima, o più correttamente un'istanza di una classe anonima che implementa un'interfaccia *funzionale* .

(Questo esempio è incluso qui per completezza. Fare riferimento all'argomento Lambda

Expressions per il trattamento completo.)

The Relational Operators (<, <=,>,> =)

Gli operatori `<` , `<=` , `>` e `>=` sono operatori binari per il confronto di tipi numerici. Il significato degli operatori è come ci si aspetterebbe. Ad esempio, se `a` e `b` sono dichiarati come `byte` , `short` , `char` , `int` , `long` , `float` , `double` o i corrispondenti tipi di box:

```
- `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`.
```

Il tipo di risultato per questi operatori è `boolean` in tutti i casi.

Gli operatori relazionali possono essere utilizzati per confrontare i numeri con tipi diversi. Per esempio:

```
int i = 1;
long l = 2;
if (i < l) {
    System.out.println("i is smaller");
}
```

Gli operatori relazionali possono essere utilizzati quando uno o entrambi i numeri sono esempi di tipi numerici in scatola. Per esempio:

```
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");
}
```

Il comportamento preciso è riassunto come segue:

1. Se uno degli operandi è di tipo boxed, è unbox.
2. Se uno degli operandi ora è un `byte` , `short` o `char` , viene promosso a un `int` .
3. Se i tipi di operandi non sono gli stessi, l'operando con il tipo "più piccolo" viene promosso al tipo "più grande".
4. Il confronto viene eseguito sui valori `int` , `long` , `float` o `double` risultanti.

Devi stare attento con i confronti relazionali che coinvolgono numeri in virgola mobile:

- Le espressioni che calcolano i numeri in virgola mobile spesso causano errori di arrotondamento dovuti al fatto che le rappresentazioni in virgola mobile del computer hanno una precisione limitata.
- Quando si confronta un tipo intero e un tipo a virgola mobile, anche la conversione del numero intero in virgola mobile può portare a errori di arrotondamento.

Infine, Java supporta bit l'uso di operatori relazionali con tipi diversi da quelli sopra elencati. Ad

esempio, non è *possibile* utilizzare questi operatori per confrontare stringhe, matrici di numeri e così via.

Leggi operatori online: https://riptutorial.com/it/java/topic/176/operatori

Capitolo 131: Operazioni Java Floating Point

introduzione

I numeri in virgola mobile sono numeri che hanno parti frazionarie (di solito espresse con un punto decimale). In Java, esistono due tipi primitivi per numeri a virgola mobile che sono `float` (utilizza 4 byte) e `double` (usa 8 byte). Questa pagina di documentazione è per i dettagli con operazioni sugli esempi che possono essere eseguite su punti mobili in Java.

Examples

Confronto tra valori in virgola mobile

È necessario prestare attenzione quando si confrontano valori a virgola mobile (`float` o `double`) utilizzando operatori relazionali: `==` `!=` , `<` E così via. Questi operatori danno risultati in base alle rappresentazioni binarie dei valori in virgola mobile. Per esempio:

```
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"
    }
}
```

Il calcolo del `oneThird` ha introdotto un piccolo errore di arrotondamento e quando moltiplichiamo il `oneThird` per `3` otteniamo un risultato leggermente diverso da `1.0` .

Questo problema di rappresentazioni inesatte è più marcato quando si tenta di combinare il `double` e il `float` nei calcoli. Per esempio:

```
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
    }
}
```

Le rappresentazioni in virgola mobile utilizzate in Java per i tipi `float` e `double` hanno un numero limitato di cifre di precisione. Per il tipo `float` , la precisione è di 23 cifre binarie o di circa 8 cifre decimali. Per il `double` tipo, è 52 bit o circa 15 cifre decimali. Inoltre, alcune operazioni aritmetiche

introdurranno errori di arrotondamento. Pertanto, quando un programma confronta i valori in virgola mobile, pratica standard per definire un **delta accettabile** per il confronto. Se la differenza tra i due numeri è inferiore al delta, vengono considerati uguali. Per esempio

```
if (Math.abs(v1 - v2) < delta)
```

Esempio di confronto 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();
        }
    }
}
```

Risultato:

```
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
```

Anche per il confronto dei tipi primitivi a `double` e `float` possibile utilizzare il metodo di `compare` statico del tipo di boxing corrispondente. Per esempio:

```
double a = 1.0;
double b = 1.0001;

System.out.println(Double.compare(a, b));//-1
System.out.println(Double.compare(b, a));//1
```

Infine, determinare quali delta sono più appropriati per un confronto può essere complicato. Un approccio comunemente usato è quello di selezionare valori delta che secondo la nostra intuizione sono giusti. Tuttavia, se si conoscono la scala e l'accuratezza (vera) dei valori di input e i calcoli eseguiti, potrebbe essere possibile ottenere limiti matematicamente validi sulla precisione dei risultati, e quindi per i delta. (Esiste un ramo formale della matematica noto come analisi numerica che veniva insegnato agli scienziati computazionali che coprivano questo tipo di analisi).

OverFlow e UnderFlow

Float data type

Il tipo di dati float è un punto mobile IEEE 754 a 32 bit a precisione singola.

`Float` overflow

Il valore massimo possibile è `3.4028235e+38` , quando supera questo valore produce `Infinity`

```
float f = 3.4e38f;
float result = f*2;
System.out.println(result); //Infinity
```

`Float` **UnderFlow**

Il valore minimo è 1.4e-45f, quando va sotto questo valore produce `0.0`

```
    float f = 1e-45f;
    float result = f/1000;
    System.out.println(result);
```

doppio tipo di dati

Il doppio tipo di dati è un punto mobile IEEE 754 a `64-bit` a doppia precisione.

`Double` **OverFlow**

Il valore massimo possibile è `1.7976931348623157e+308` , quando supera questo valore produce `Infinity`

```
double d = 1e308;
double result=d*2;
System.out.println(result); //Infinity
```

`Double` **UnderFlow**

Il valore minimo è 4.9e-324, quando va sotto questo valore produce `0.0`

```
    double d = 4.8e-323;
    double result = d/1000;
    System.out.println(result); //0.0
```

Formattare i valori in virgola mobile

Virgola mobile I numeri possono essere formattati come numeri decimali utilizzando `String.format` con `'f'` flag `'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"
```

Virgola mobile I numeri possono essere formattati come numeri decimali utilizzando `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
```

Aderenza rigorosa alle specifiche IEEE

Per impostazione predefinita, le operazioni in virgola mobile su `float` e `double` *non* rispettano rigorosamente le regole della specifica IEEE 754. Un'espressione può utilizzare estensioni specifiche dell'implementazione nell'intervallo di questi valori; essenzialmente permettendo loro di essere *più* precisi del necessario.

`strictfp` disabilita questo comportamento. Si applica a una classe, un'interfaccia o un metodo e si applica a tutto ciò che vi è contenuto, come classi, interfacce, metodi, costruttori, inizializzatori variabili, ecc. Con `strictfp` , i valori intermedi di un'espressione a virgola mobile *devono* essere compresi il valore float impostato o il doppio valore impostato. Ciò fa sì che i risultati di tali espressioni siano esattamente quelli previsti dalla specifica IEEE 754.

Tutte le espressioni costanti sono implicitamente rigide, anche se non sono all'interno di uno scope `strictfp` .

Di conseguenza, `strictfp` ha l'effetto netto di rendere alcuni calcoli caso-angolo *meno* precisi e può anche *rallentare le* operazioni in virgola mobile (poiché la CPU sta facendo più lavoro per garantire che qualsiasi precisione extra nativa non influenzi il risultato). Tuttavia, fa sì che i risultati siano esattamente uguali su tutte le piattaforme. È quindi utile in cose come programmi scientifici, in cui la riproducibilità è più importante della velocità.

```
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;
        }
    }
}
```

Leggi Operazioni Java Floating Point online: https://riptutorial.com/it/java/topic/6167/operazioni-java-floating-point

Capitolo 132: Opzionale

introduzione

`Optional` è un oggetto contenitore che può contenere o meno un valore non nullo. Se è presente un valore, `isPresent()` restituirà `true` e `get()` restituirà il valore.

Vengono forniti ulteriori metodi che dipendono dalla presenza del valore contenuto, ad esempio `orElse()` , che restituisce un valore predefinito se value non è presente e `ifPresent()` che esegue un blocco di codice se il valore è presente.

Sintassi

- Optional.empty () // Crea un'istanza opzionale vuota.
- Optional.of (valore) // Restituisce un Facoltativo con il valore non null specificato. Una NullPointerException verrà lanciata se il valore passato è nullo.
- Optional.ofNullable (valore) // Restituisce un Facoltativo con il valore specificato che può essere nullo.

Examples

Restituisce il valore predefinito se Opzionale è vuoto

Non utilizzare semplicemente `Optional.get()` poiché potrebbe generare `NoSuchElementException` . I metodi `Optional.orElse(T)` e `Optional.orElseGet(Supplier<? extends T>)` forniscono un modo per fornire un valore predefinito nel caso in cui l'opzione sia vuota.

```
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 differenza cruciale tra `orElse` e `orElseGet` è che quest'ultimo viene valutato solo quando l'opzione è vuota mentre l'argomento fornito a quello precedente viene valutato anche se l'opzione non è vuota. `orElse` , `orElse` dovrebbe essere utilizzato solo per le costanti e mai per fornire valore basato su qualsiasi tipo di calcolo.

Carta geografica

Utilizza il metodo `map()` di `Optional` per lavorare con valori che potrebbero essere `null` senza fare controlli `null` espliciti:

(Si noti che le operazioni `map()` e `filter()` vengono valutate immediatamente, a differenza delle loro controparti Stream che vengono valutate solo su un'operazione *terminale* .)

Sintassi:

```
public <U> Optional<U> map(Function<? super T,? extends U> mapper)
```

Esempi di codice:

```
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"
```

Poiché Optional.map () restituisce un opzionale vuoto quando la sua funzione di mappatura restituisce null, è possibile concatenare diverse operazioni map () come una forma di dereferenziazione null-safe. Questo è anche noto come **concatenamento sicuro** .

Considera il seguente esempio:

```
String value = foo.getBar().getBaz().toString();
```

Qualsiasi `getBar` , `getBaz` e `toString` possono potenzialmente lanciare una `NullPointerException` .

Ecco un modo alternativo per ottenere il valore da `toString()` usando `Optional` :

```
String value = Optional.ofNullable(foo)
                       .map(Foo::getBar)
                       .map(Bar::getBaz)
                       .map(Baz::toString)
                       .orElse("");
```

Ciò restituirà una stringa vuota se una qualsiasi delle funzioni di mapping ha restituito null.

Di seguito è riportato un altro esempio, ma leggermente diverso. Stampa il valore solo se nessuna delle funzioni di mappatura ha restituito nulla.

```
Optional.ofNullable(foo)
        .map(Foo::getBar)
        .map(Bar::getBaz)
```

```
        .map(Baz::toString)
        .ifPresent(System.out::println);
```

Getta un'eccezione, se non c'è valore

Utilizzare il metodo `orElseThrow()` di `Optional` per ottenere il valore contenuto o generare un'eccezione, se non è stata impostata. È simile alla chiamata `get()` , tranne per il fatto che consente tipi di eccezioni arbitrarie. Il metodo accetta un fornitore che deve restituire l'eccezione da lanciare.

Nel primo esempio, il metodo restituisce semplicemente il valore contenuto:

```
Optional optional = Optional.of("something");

return optional.orElseThrow(IllegalArgumentException::new);
// returns "something" string
```

Nel secondo esempio, il metodo genera un'eccezione perché non è stato impostato un valore:

```
Optional optional = Optional.empty();

return optional.orElseThrow(IllegalArgumentException::new);
// throws IllegalArgumentException
```

È inoltre possibile utilizzare la sintassi lambda se è necessario lanciare un'eccezione con il messaggio:

```
optional.orElseThrow(() -> new IllegalArgumentException("Illegal"));
```

Filtro

`filter()` viene utilizzato per indicare che si desidera il valore *solo se* corrisponde al proprio predicato.

Pensalo come `if (!somePredicate(x)) { x = null; }` .

Esempi di codice:

```
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
```

Utilizzo di contenitori opzionali per tipi di numeri primitivi

`OptionalDouble` , `OptionalInt` e `OptionalLong` funzionano come `Optional` , ma sono specificamente progettati per includere tipi primitivi:

```
OptionalInt presentInt = OptionalInt.of(value);
OptionalInt absentInt = OptionalInt.empty();
```

Poiché i tipi numerici hanno un valore, non esiste una gestione speciale per null. I contenitori vuoti possono essere controllati con:

```
presentInt.isPresent(); // Is true.
absentInt.isPresent(); // Is false.
```

Allo stesso modo, esistono stenografie per aiutare la gestione del valore:

```
// 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();
```

Esegui il codice solo se è presente un valore

```
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.
```

Fornisci un valore predefinito usando un fornitore

Il *normale* metodo `orElse` riceve un `Object` , quindi potresti chiedervi perché esiste un'opzione per fornire un `Supplier` qui (il metodo `orElseGet`).

Tenere conto:

```
String value = "something";
return Optional.ofNullable(value)
               .orElse(getValueThatIsHardToCalculate()); // returns "something"
```

Chiamerebbe comunque `getValueThatIsHardToCalculate()` anche se il risultato non è utilizzato in quanto l'opzionale non è vuoto.

Per evitare questa penalità fornisci un fornitore:

```
String value = "something";
return Optional.ofNullable(value)
               .orElseGet(() -> getValueThatIsHardToCalculate()); // returns "something"
```

In questo modo `getValueThatIsHardToCalculate()` verrà chiamato solo se l' `Optional` è vuota.

FlatMap

`flatMap` è simile alla `map` . La differenza è descritta da javadoc come segue:

> Questo metodo è simile alla `map(Function)` , ma il mapper fornito è uno il cui risultato è già un `Optional` e, se invocato, `flatMap` non lo avvolge con un `Optional` .

In altre parole, quando si concatena una chiamata al metodo che restituisce un `Optional` , usando `Optional.flatMap` evita di creare `Optionals` nidificati.

Ad esempio, date le seguenti classi:

```
public class Foo {
    Optional<Bar> getBar(){
        return Optional.of(new Bar());
    }
}

public class Bar {
}
```

Se usi `Optional.map` , otterrai un `Optional` nidificata; cioè `Optional<Optional<Bar>>` .

```
Optional<Optional<Bar>> nestedOptionalBar =
    Optional.of(new Foo())
        .map(Foo::getBar);
```

Tuttavia, se si utilizza `Optional.flatMap` , si otterrà un semplice `Optional` ; cioè `Optional<Bar>` .

```
Optional<Bar> optionalBar =
    Optional.of(new Foo())
        .flatMap(Foo::getBar);
```

Leggi Opzionale online: https://riptutorial.com/it/java/topic/152/opzionale

Capitolo 133: Ora locale

Sintassi

- LocalTime time = LocalTime.now (); // Inizializza con l'orologio di sistema corrente
- LocalTime time = LocalTime.MIDNIGHT; // 00:00
- LocalTime time = LocalTime.NOON; // 12:00
- LocalTime time = LocalTime.of (12, 12, 45); // 12:12:45

Parametri

Metodo	Produzione
`LocalTime.of(13, 12, 11)`	13:12:11
`LocalTime.MIDNIGHT`	00:00
`LocalTime.NOON`	00:00
`LocalTime.now()`	Ora corrente dall'orologio di sistema
`LocalTime.MAX`	L'ora locale massima supportata 23: 59: 59.999999999
`LocalTime.MIN`	L'ora locale supportata minima 00:00
`LocalTime.ofSecondOfDay(84399)`	23:59:59, Ottiene il tempo dal valore della seconda giornata
`LocalTime.ofNanoOfDay(2000000000)`	00:00:02, ottiene il tempo dal valore di nanos-of-day

Osservazioni

Come indica il nome della classe, `LocalTime` rappresenta un'ora senza fuso orario. Non rappresenta una data. È un'etichetta semplice per un tempo determinato.

La classe è basata sul valore e il metodo `equals` deve essere usato quando si fanno i confronti.

Questa classe proviene dal pacchetto java.time.

Examples

Modifica del tempo

Puoi aggiungere ore, minuti, secondi e nanosecondi:

```
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)
```

Fusi orari e loro differenza di orario

```
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");
    }
}
```

Quantità di tempo tra due LocalTime

Ci sono due modi equivalenti per calcolare la quantità di unità di tempo tra due `LocalTime` : (1) attraverso il metodo `until(Temporal, TemporalUnit)` e attraverso (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);
    }
}
```

Intro

LocalTime è una classe immutabile e sicura per i thread, utilizzata per rappresentare il tempo, spesso vista come ora-min-sec. Il tempo è rappresentato con precisione al nanosecondo. Ad esempio, il valore "13: 45.30.123456789" può essere memorizzato in un LocalTime.

Questa classe non memorizza o rappresenta una data o un fuso orario. Invece, è una descrizione dell'ora locale vista su un orologio da parete. Non può rappresentare un istante sulla linea del tempo senza informazioni aggiuntive come un offset o un fuso orario. Questa è una classe basata sul valore, il metodo equals dovrebbe essere usato per i confronti.

campi

MAX: la durata massima supportata di LocalTime, '23: 59: 59.999999999 '. MEZZANOTTE, MIN, NOON

Metodi statici importanti

now (), now (Clock clock), now (ZoneId zone), parse (CharSequence text)

Metodi di istanza importanti

isAfter (LocalTime altro), isBefore (LocalTime altro), meno (TemporalAmount amountToSubtract), meno (long amountToSubtract, unità TemporalUnit), più (TemporalAmount amountToAdd), più (long amountToAdd, unità TemporalUnit)

```
ZoneId zone = ZoneId.of("Asia/Kolkata");
LocalTime now = LocalTime.now();
LocalTime now1 = LocalTime.now(zone);
LocalTime then = LocalTime.parse("04:16:40");
```

La differenza di tempo può essere calcolata in uno dei seguenti modi

```
long timeDiff = Duration.between(now, now1).toMinutes();
long timeDiff1 = java.time.temporal.ChronoUnit.MINUTES.between(now2, now1);
```

Puoi anche aggiungere / sottrarre ore, minuti o secondi da qualsiasi oggetto di LocalTime.

meno ore (lunghe oreToSubtract), menoMinute (lunghe oreToMinutes), minusNanos (lunghe nanosToSubtract), minusSeconds (lunghi secondiToSubtract), plusHours (lunghe oreToSubtract), plusMinutes (lunghe oreToMinutes), plusNanos (long nanosToSubtract), plusSeconds (long secondsToSubtract)

```
now.plusHours(1L);
now1.minusMinutes(20L);
```

Leggi Ora locale online: https://riptutorial.com/it/java/topic/3065/ora-locale

Capitolo 134: Oracle Official Code Standard

introduzione

La guida di stile ufficiale di Oracle per Java Programming Language è uno standard seguito dagli sviluppatori di Oracle e consigliato per essere seguito da qualsiasi altro sviluppatore Java. Copre nomi di file, organizzazione di file, indentazione, commenti, dichiarazioni, dichiarazioni, spazi bianchi, convenzioni di denominazione, pratiche di programmazione e include un esempio di codice.

Osservazioni

- Gli esempi sopra seguono rigorosamente la nuova guida di stile ufficiale di Oracle. In altre parole, *non sono* inventati in modo soggettivo dagli autori di questa pagina.
- La guida di stile ufficiale è stata scrupolosamente scritta per essere retrocompatibile con la guida di stile originale e la maggior parte del codice in circolazione.
- La guida ufficiale stile è stato pari rivisto , tra gli altri, Brian Goetz (Java Architect Language) e Mark Reinhold (Chief Architect del Java Platform).
- Gli esempi non sono normativi; Mentre intendono illustrare il modo corretto di formattazione del codice, potrebbero esserci altri modi per formattare correttamente il codice. Questo è un principio generale: ci possono essere diversi modi per formattare il codice, tutti aderendo alle linee guida ufficiali.

Examples

Convenzioni di denominazione

Nomi dei pacchetti

- I nomi dei pacchetti devono essere tutti in minuscolo senza caratteri di sottolineatura o altri caratteri speciali.
- I nomi dei pacchetti iniziano con la parte dell'autorità invertita dell'indirizzo Web della società dello sviluppatore. Questa parte può essere seguita da una sottostruttura del pacchetto dipendente dal progetto / struttura.
- Non usare la forma plurale. Seguire la convenzione dell'API standard che utilizza ad esempio `java.lang.annotation` e non `java.lang.annotations` .
- **Esempi:** `com.yourcompany.widget.button` , `com.yourcompany.core.api`

Nomi di classe, interfaccia e enum

- I nomi di classe e enum in genere dovrebbero essere nomi.
- I nomi delle interfacce dovrebbero in genere essere nomi o aggettivi che terminano con ... in grado.
- Usa maiuscole e minuscole con la prima lettera di ciascuna parola in maiuscolo (es. CamelCase).
- Abbina l'espressione regolare `^[AZ][a-zA-Z0-9]*$` .
- Usa parole intere ed evita di usare le abbreviazioni a meno che l'abbreviazione non sia più usata della forma lunga.
- Formatta un'abbreviazione come parola se fa parte di un nome di classe più lungo.
- **Esempi:** `ArrayList` , `BigInteger` , `ArrayIndexOutOfBoundsException` , `Iterable` .

Nomi dei metodi

I nomi dei metodi dovrebbero in genere essere verbi o altre descrizioni di azioni

- Dovrebbero corrispondere all'espressione regolare `^[az][a-zA-Z0-9]*$` .
- Usa maiuscole e minuscole con la prima lettera in minuscolo.
- **Esempi:** `toString` , `hashCode`

variabili

I nomi delle variabili dovrebbero essere in maiuscolo con la prima lettera in minuscolo

- Abbina l'espressione regolare `^[az][a-zA-Z0-9]*$`
- Ulteriore raccomandazione: variabili
- **Esempi:** `elements` , `currentIndex`

Digita le variabili

Per i casi semplici in cui ci sono poche variabili di tipo coinvolte usa una sola lettera maiuscola.

- Abbina l'espressione regolare `^[AZ][0-9]?$`
- Se una lettera è più descrittiva di un'altra (come `K` e `V` per le chiavi e i valori nelle mappe o `R` per un tipo di ritorno di funzione), usa quella, altrimenti usa `T`
- Per i casi complessi in cui le variabili di tipo a lettera singola diventano confuse, utilizzare nomi più lunghi scritti in lettere maiuscole e utilizzare il carattere di sottolineatura (`_`) per separare le parole.
- **Esempi:** `T` , `V` , `SRC_VERTEX`

costanti

Le costanti (campi `static final` cui contenuto è immutabile, per regole linguistiche o per convenzione) devono essere denominati con tutte le lettere maiuscole e il carattere di

sottolineatura (_) per separare le parole.

- Abbina l'espressione regolare `^[AZ][A-Z0-9]*(_[A-Z0-9]+)*$`
- **Esempi:** `BUFFER_SIZE` , `MAX_LEVEL`

Altre linee guida sulla denominazione

- Evitare metodi di occultamento / ombreggiamento, variabili e variabili di tipo negli ambiti esterni.
- Lascia che la verbosità del nome sia correlata alla dimensione dell'ambito. (Ad esempio, utilizzare nomi descrittivi per campi di classi grandi e nomi brevi per variabili locali di breve durata.)
- Quando si nominano membri statici pubblici, lasciare che l'identificatore sia auto-descrittivo se si ritiene che verranno importati staticamente.
- Ulteriori letture: Naming Section (nella guida ufficiale allo stile Java)

Fonte: linee guida stile Java da Oracle

File di origine Java

- Tutte le linee devono essere terminate con un carattere di avanzamento riga (LF, valore ASCII 10) e non per esempio CR o CR + LF.
- Potrebbe non esserci spazio bianco finale alla fine di una riga.
- Il nome di un file sorgente deve essere uguale al nome della classe che contiene seguito dall'estensione `.java` , anche per i file che contengono solo una classe privata del pacchetto. Questo non si applica ai file che non contengono dichiarazioni di classe, come `package-info.java` .

Personaggi speciali

- Oltre a LF, il solo spazio bianco consentito è Spazio (valore ASCII 32). Nota che ciò implica che altri caratteri di spazio bianco (in, ad esempio, caratteri letterali di stringa e caratteri) devono essere scritti in forma di escape.
- `\'` , `\"` , `\\` , `\t` , `\b` , `\r` , `\f` e `\n` dovrebbero essere preferiti rispetto ai corrispondenti caratteri ottali (ad esempio `\047`) o Unicode (ad esempio `\u0027`).
- In caso di necessità di andare contro le regole di cui sopra per motivi di test, il test dovrebbe *generare* l'input richiesto in modo programmatico.

Dichiarazione del pacchetto

```
package com.example.my.package;
```

La dichiarazione del pacchetto non deve essere incentrata sulla linea, indipendentemente dal fatto

che superi la lunghezza massima consigliata di una linea.

Importa le dichiarazioni

```
// 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;
```

- Le istruzioni di importazione dovrebbero essere ordinate ...
    - ... principalmente da non statici / statici con importazioni non statiche prima.
    - ... secondariamente dall'origine della confezione in base al seguente ordine
        - pacchetti java
        - pacchetti javax
        - pacchetti esterni (ad es. org.xml)
        - pacchetti interni (es. com.sun)
    - ... terziario per pacchetto e identificatore di classe in ordine lessicografico
- Le istruzioni di importazione non devono essere allineate alla linea, indipendentemente dal fatto che superi la lunghezza massima consigliata di una linea.
- Non dovrebbero essere presenti importazioni inutilizzate.

Importazioni con caratteri jolly

- Le importazioni con caratteri jolly non dovrebbero in generale essere utilizzate.
- Quando si importa un numero elevato di classi strettamente correlate (come l'implementazione di un visitatore su un albero con dozzine di classi distinte "nodo"), può essere utilizzata un'importazione con caratteri jolly.
- In ogni caso, non dovrebbe essere utilizzata più di una importazione di caratteri jolly per file.

Struttura di classe

Ordine dei membri della classe

I membri della classe dovrebbero essere ordinati come segue:

1. Campi (in ordine pubblico, protetto e privato)
2. Costruttori

3. Metodi di fabbrica
4. Altri metodi (in ordine pubblico, protetto e privato)

I campi e i metodi di ordinazione principalmente dai loro modificatori di accesso o identificatore non sono richiesti.

Ecco un esempio di questo ordine:

```
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 + "]";
    }

}
```

Raggruppamento di membri della classe

- I campi correlati dovrebbero essere raggruppati insieme.
- Un tipo annidato può essere dichiarato subito prima del suo primo utilizzo; altrimenti dovrebbe essere dichiarato prima dei campi.
- Costruttori e metodi sovraccaricati dovrebbero essere raggruppati per funzionalità e ordinati con crescente arbit. Ciò implica che la delega tra questi costrutti scorre verso il basso nel codice.
- I costruttori dovrebbero essere raggruppati insieme senza altri membri tra.
- Le varianti di overload di un metodo dovrebbero essere raggruppate insieme senza altri membri tra.

modificatori

```
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();
}
```

- I modificatori dovrebbero andare nel seguente ordine
    - Modificatore di accesso (`public` / `private` / `protected`)
    - `abstract`
    - `static`
    - `final`
    - `transient`
    - `volatile`
    - `default`
    - `synchronized`
    - `native`
    - `strictfp`
- I modificatori non dovrebbero essere scritti quando sono impliciti. Ad esempio, i metodi di interfaccia non devono essere dichiarati `public` né `abstract` e le enumerazioni e le interfacce nidificate non devono essere dichiarate statiche.
- I parametri del metodo e le variabili locali non devono essere dichiarati `final` meno che non migliorino la leggibilità o documentino una decisione di progettazione effettiva.
- I campi devono essere dichiarati `final` meno che non vi sia un valido motivo per renderli mutabili.

dentellatura

- Il livello di indentazione è di **quattro spazi** .
- Solo i caratteri di spazio possono essere usati per il rientro. **Nessun tab.**
- Le linee vuote non devono essere rientrate. (Ciò è implicito nella regola dello spazio bianco senza fine).
- `case` linee di `case` devono essere rientrate con quattro spazi e le istruzioni all'interno del caso dovrebbero essere rientrate con altri quattro spazi.

```
switch (var) {
    case TWO:
        setChoice("two");
        break;
    case THREE:
        setChoice("three");
        break;
    default:
        throw new IllegalArgumentException();
}
```

Fare riferimento alle istruzioni Wrapping per le linee guida su come indentare le linee di continuazione.

Dichiarazioni di avvolgimento

- Il codice sorgente e i commenti non dovrebbero in genere superare gli 80 caratteri per riga e raramente se mai superare i 100 caratteri per riga, incluso il rientro.

Il limite di caratteri deve essere valutato caso per caso. Ciò che conta davvero è la "densità" semantica e la leggibilità della linea. Rendere le linee gratuitamente lunghe le rende difficili da leggere; allo stesso modo, fare "tentativi eroici" per adattarli a 80 colonne può anche renderli difficili da leggere. La flessibilità qui delineata mira a consentire agli sviluppatori di evitare questi estremi, non massimizzare l'uso del monitor immobiliare.

- URL o comandi di esempio non devono essere incapsulati.

```
// 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(", "));
```

- Il wrapping ad un livello sintattico più alto è preferito rispetto al wrapping ad un livello sintattico più basso.

- Ci dovrebbe essere al massimo 1 affermazione per riga.

- Una linea di continuazione dovrebbe rientrare in uno dei quattro modi seguenti

    - **Variante 1** : con 8 spazi in più rispetto al rientro della riga precedente.
    - **Variante 2** : con 8 spazi aggiuntivi rispetto alla colonna iniziale dell'espressione avvolta.
    - **Variante 3** : allineata con la precedente espressione di pari livello (purché sia chiaro che è una linea di continuazione)
    - **Variante 4** : allineata alla precedente chiamata di metodo in un'espressione concatenata.

Dichiarazioni sul metodo di avvolgimento

```
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 {
    …
}
```

- Le dichiarazioni dei metodi possono essere formattate elencando gli argomenti verticalmente, oppure con una nuova riga e +8 spazi aggiuntivi
- Se una clausola throws deve essere avvolta, ponete l'interruzione di riga davanti alla clausola throws e assicuratevi che si distingua dall'elenco degli argomenti, facendo rientrare +8 rispetto alla dichiarazione della funzione, o +8 rispetto alla riga precedente.

Espressioni di avvolgimento

- Se una linea si avvicina al limite massimo di caratteri, considera sempre di scomporlo in più istruzioni / espressioni invece di avvolgere la linea.
- Pausa prima degli operatori.
- Pausa prima del. in chiamate di metodo concatenate.

```
popupMsg("Inbox notification: You have "
        + newMsgs + " new messages");

// Don't! Looks like two arguments
popupMsg("Inbox notification: You have " +
         newMsgs + " new messages");
```

Lo spazio bianco

Spazio bianco verticale

- Una singola riga vuota dovrebbe essere utilizzata per separare ...
    - Dichiarazione del pacchetto
    - Dichiarazioni di classe
    - Costruttori
    - metodi

    - Inizializzatori statici
    - Inizializzatori di istanze

- ... e può essere usato per separare i gruppi logici di

    - dichiarazioni di importazione
    - i campi
    - dichiarazioni

- È possibile utilizzare più righe vuote consecutive per separare i gruppi di membri correlati e non come interlinea standard tra i membri.

Spazio bianco orizzontale

- Un singolo spazio dovrebbe essere usato ...

    - Per separare le parole chiave dalle parentesi graffe o parentesi di apertura o chiusura vicine
    - Prima e dopo tutti gli operatori binari e gli operatori come simboli come le frecce nelle espressioni lambda e i due punti migliorati per cicli (ma non prima dei due punti di un'etichetta)
    - Dopo `//` inizia un commento.
    - Dopo le virgole che separano gli argomenti e il punto e virgola che separano le parti di un ciclo for.
    - Dopo la parentesi di chiusura di un cast.

- Nelle dichiarazioni variabili non è consigliabile allineare tipi e variabili.

Dichiarazioni variabili

- Una variabile per dichiarazione (e al massimo una dichiarazione per riga)
- Le parentesi quadre degli array dovrebbero essere al tipo (`String[] args`) e non alla variabile (`String args[]`).
- Dichiarare una variabile locale subito prima che venga utilizzata per la prima volta e inizializzarla il più vicino possibile alla dichiarazione.

annotazioni

Le annotazioni della dichiarazione devono essere riportate su una riga separata dalla dichiarazione che viene annotata.

```
@SuppressWarnings("unchecked")
public T[] toArray(T[] typeHolder) {
    ...
}
```

Tuttavia, poche o brevi annotazioni che annotano un metodo a linea singola possono essere

messe sulla stessa linea del metodo se migliora la leggibilità. Ad esempio, si può scrivere:

```
@Nullable String getName() { return name; }
```

Per una questione di coerenza e leggibilità, tutte le annotazioni dovrebbero essere messe sulla stessa riga o ogni annotazione dovrebbe essere messa su una riga separata.

```
// 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) {
    ...
}
```

Lambda Expressions

```
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));
```

- I lambda di espressione sono preferiti rispetto ai lambdas a blocco su una riga.
- I riferimenti al metodo dovrebbero generalmente essere preferiti rispetto alle espressioni lambda.
- Per i riferimenti al metodo dell'istanza associata, oi metodi con l'arità maggiore di uno, un'espressione lambda può essere più semplice da comprendere e quindi preferibile. Soprattutto se il comportamento del metodo non è chiaro dal contesto.
- I tipi di parametro dovrebbero essere omessi a meno che non migliorino la leggibilità.

- Se un'espressione lambda si estende su più di poche righe, prendere in considerazione la creazione di un metodo.

Parentesi ridondanti

```
return flag ? "yes" : "no";

String cmp = (flag1 != flag2) ? "not equal" : "equal";

// Don't do this
return (flag ? "yes" : "no");
```

- Le parentesi di raggruppamento ridondanti (ovvero le parentesi che non influiscono sulla valutazione) possono essere utilizzate se migliorano la leggibilità.
- Le parentesi di raggruppamento ridondanti dovrebbero in genere essere lasciate in espressioni più brevi che coinvolgono operatori comuni ma incluse in espressioni più lunghe o espressioni che coinvolgono operatori la cui precedenza e associatività non è chiara senza parentesi. Le espressioni ternarie con condizioni non banali appartengono a quest'ultima.
- L'intera espressione che segue una parola chiave `return` non deve essere racchiusa tra parentesi.

letterali

```
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` letterali `long` dovrebbero usare il suffisso `L` lettera maiuscola.
- I letterali esadecimali devono usare lettere maiuscole `A` - `F`
- Tutti gli altri prefissi, infissi e suffissi numerici devono utilizzare lettere minuscole.

Bretelle

```
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");
        }
    }
}
```

- Le parentesi graffe di apertura devono essere posizionate alla fine della riga corrente anziché su una linea di sua proprietà.

- Dovrebbe esserci una nuova riga davanti a una parentesi graffa di chiusura a meno che il blocco non sia vuoto (vedere i moduli brevi di seguito)
- Le parentesi graffe sono raccomandate anche laddove la lingua le rende facoltative, come ad esempio i corpi a singola fila e quelli a cappio.
    - Se un blocco si estende su più di una riga (inclusi i commenti), deve contenere parentesi.
    - Se uno dei blocchi in una istruzione `if` / `else` ha parentesi graffe, anche l'altro blocco deve esserlo.
    - Se il blocco arriva per ultimo in un blocco che lo racchiude, deve avere le parentesi graffe.
- La parola chiave `else` , `catch` e the `while` in `do…while` loop si posiziona sulla stessa riga della parentesi di chiusura del blocco precedente.

Forme brevi

```
enum Response { YES, NO, MAYBE }
public boolean isReference() { return true; }
```

Le raccomandazioni di cui sopra hanno lo scopo di migliorare l'uniformità (e quindi aumentare la familiarità / leggibilità). In alcuni casi, le "forme brevi" che si discostano dalle linee guida di cui sopra sono leggibili e possono essere utilizzate al loro posto. Questi casi includono per esempio dichiarazioni enum semplici e metodi banali e espressioni lambda.

Leggi Oracle Official Code Standard online: https://riptutorial.com/it/java/topic/2697/oracle-official-code-standard

Capitolo 135: Pacchi

introduzione

il pacchetto in java è usato per raggruppare classi e interfacce. Questo aiuta lo sviluppatore ad evitare conflitti quando ci sono un numero enorme di classi. Se usiamo questo pacchetto le classi possiamo creare una classe / interfaccia con lo stesso nome in diversi pacchetti. Usando i pacchetti possiamo importare il pezzo di nuovo in un'altra classe. Esistono molti *pacchetti integrati* in java come> 1.java.util> 2.java.lang> 3.java.io Possiamo definire i nostri *pacchetti definiti dall'utente* .

Osservazioni

I pacchetti forniscono protezione di accesso.

> la dichiarazione del pacchetto deve essere la prima riga del codice sorgente. Ci può essere solo un pacchetto in un file sorgente.

Con l'aiuto dei pacchetti è possibile evitare conflitti tra diversi moduli.

Examples

Utilizzo dei pacchetti per creare classi con lo stesso nome

Primo test.class:

```
package foo.bar

public class Test {

}
```

Anche Test.class in un altro pacchetto

```
package foo.bar.baz

public class Test {

}
```

Quanto sopra va bene perché le due classi esistono in diversi pacchetti.

Utilizzo di Scope protetto da pacchetto

In Java se non si fornisce un modificatore di accesso, l'ambito predefinito per le variabili è il livello protetto dal pacchetto. Ciò significa che le classi possono accedere alle variabili di altre classi

all'interno dello stesso pacchetto come se tali variabili fossero pubblicamente disponibili.

```
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
}
```

Questo metodo non funzionerà per una classe in un altro pacchetto:

```
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
}
```

Leggi Pacchi online: https://riptutorial.com/it/java/topic/8273/pacchi

Capitolo 136: Polimorfismo

introduzione

Il polimorfismo è uno dei principali concetti OOP (programmazione orientata agli oggetti). La parola Polymorphism deriva dalle parole greche "poly" e "morph". Poly significa "molti" e metamorfosi significa "forme" (molte forme).

Esistono due modi per eseguire il polimorfismo. **Sovraccarico del metodo** e **sovrascrittura del metodo** .

Osservazioni

`Interfaces` sono un altro modo per ottenere il polimorfismo in Java, a parte l'ereditarietà basata sulla classe. Le interfacce definiscono un elenco di metodi che costituiscono l'API del programma. Le classi devono `implement interface` ignorando tutti i suoi metodi.

Examples

Sovraccarico del metodo

L'overloading del metodo , noto anche come **overloading delle funzioni** , è la capacità di una classe di avere più metodi con lo stesso nome, a condizione che differiscano per numero o tipo di argomenti.

Il compilatore verifica la ***firma*** del metodo per l'overloading del metodo.

La firma del metodo consiste di tre cose:

1. Nome del metodo
2. Numero di parametri
3. Tipi di parametri

Se questi tre sono gli stessi per due metodi in una classe, il compilatore genera un **errore di metodo duplicato** .

Questo tipo di polimorfismo viene chiamato polimorfismo *statico* o *tempo di compilazione* perché il metodo appropriato da chiamare viene deciso dal compilatore durante il tempo di compilazione in base all'elenco degli argomenti.

```
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));
    }

}
```

Ciò comporterà:

```
2
6
4.000000
```

I metodi di overload possono essere statici o non statici. Questo inoltre non influisce sul sovraccarico del metodo.

```
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
    }
}
```

Inoltre, se si modifica il tipo di metodo restituito, non è possibile ottenerlo come overload del metodo.

```
public class Polymorph {
```

```
void methodOverloaded(){
    //No argument and No return type
}

int methodOverloaded(){
    //No argument and int return type
    return 0;
}
```

Metodo Overriding

Il metodo che sovrascrive è la capacità dei sottotipi di ridefinire (scavalcare) il comportamento dei loro supertipi.

In Java, questo si traduce in sottoclassi che sovrascrivono i metodi definiti nella super classe. In Java, tutte le variabili non primitive sono in realtà `references` , che sono simili ai puntatori alla posizione dell'oggetto reale in memoria. I `references` hanno solo un tipo, che è il tipo con cui sono stati dichiarati. Tuttavia, possono puntare a un oggetto del tipo dichiarato o di uno qualsiasi dei suoi sottotipi.

Quando un metodo viene chiamato su un `reference` , viene richiamato il **metodo** corrispondente **dell'oggetto reale che viene puntato** .

```
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
    }
}
```

Regole da tenere a mente

Per sovrascrivere un metodo nella sottoclasse, il metodo di sovrascrittura (cioè quello nella sottoclasse) **DEVE AVERE** :

- stesso nome
- stesso tipo di ritorno in caso di primitive (una sottoclasse è consentita per le classi, questo è anche noto come tipi di ritorno covarianti).
- stesso tipo e ordine dei parametri
- può lanciare solo quelle eccezioni dichiarate nella clausola di tiro del metodo della superclasse o delle eccezioni che sono sottoclassi delle eccezioni dichiarate. Può anche scegliere di non lanciare alcuna eccezione. I nomi dei tipi di parametri non contano. Ad esempio, void methodX (int i) è uguale a void methodX (int k)
- Non siamo in grado di sovrascrivere i metodi definitivi o statici. L'unica cosa che possiamo fare cambia solo il metodo body.

Aggiunta di comportamenti aggiungendo classi senza toccare il codice esistente

```
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");
    }
}
```

Spiegazione

a) La classe `MakeThingsFly` può funzionare con tutto ciò che è di tipo `FlyingMachine` .

b) Il metodo `letTheMachinesFly` funziona anche senza alcuna modifica (!) quando si aggiunge una nuova classe, ad esempio `PropellerPlane` :

```
public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
        for (FlyingMachine flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
    }
}
```

Questo è il potere del polimorfismo. Puoi implementare il principio open-closed con esso.

Funzioni virtuali

I metodi virtuali sono metodi in Java non statici e senza la parola chiave Final in primo piano. Tutti i metodi di default sono virtuali in Java. I Metodi Virtuali svolgono ruoli importanti nel Polymorphism perché le classi di bambini in Java possono sovrascrivere i metodi delle loro classi genitore se la funzione che viene sovrascritta non è statica e ha la stessa firma di metodo.

Esistono, tuttavia, alcuni metodi che non sono virtuali. Ad esempio, se il metodo è dichiarato privato o con la parola chiave final, il metodo non è Virtual.

Considera il seguente esempio modificato di ereditarietà con metodi virtuali da questo post StackOverflow Come funzionano le funzioni virtuali in C # e 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");

    }
}
```

Se invochiamo la classe B e chiamiamo hello () e boo (), otterremmo "No" e "Say haha" come output risultante perché B sovrascrive gli stessi metodi da A. Anche se l'esempio sopra è quasi identico a sovrascrittura del metodo, è importante capire che i metodi in classe A sono tutti, per impostazione predefinita, virtuali.

Inoltre, possiamo implementare metodi virtuali usando la parola chiave abstract. I metodi dichiarati con la parola chiave "abstract" non hanno una definizione di metodo, il che significa che il corpo del metodo non è ancora stato implementato. Considera di nuovo l'esempio sopra, eccetto che il metodo boo () è dichiarato astratto:

```
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");

    }
}
```

Se invochiamo boo () da B, l'output sarà ancora "Say haha" poiché B eredita il metodo astratto boo () e rende boo () output "Say haha".

Fonti utilizzate e ulteriori letture:

Come funzionano le funzioni virtuali in C # e Java?

Dai un'occhiata a questa ottima risposta che fornisce informazioni molto più complete sulle funzioni virtuali:

Puoi scrivere funzioni / metodi virtuali in Java?

Polimorfismo e diversi tipi di override

Dal tutorial di java

> La definizione di polimorfismo del dizionario fa riferimento a un principio in biologia in cui un organismo o una specie possono avere molte forme o stadi diversi. Questo principio può essere applicato anche alla programmazione orientata agli oggetti e ai linguaggi come il linguaggio Java. ***Sottoclassi di una classe possono definire i propri comportamenti univoci e tuttavia condividere alcune delle stesse funzionalità della classe genitore.***

Dai un'occhiata a questo esempio per capire i diversi tipi di sovrascrittura.

1. La classe base non fornisce implementazione e la sottoclasse deve sovrascrivere il metodo completo - (abstract)
2. La classe base fornisce l'implementazione predefinita e la sottoclasse può modificare il comportamento
3. La `super.methodName()` aggiunge l'estensione all'implementazione della classe base chiamando `super.methodName()` come prima istruzione
4. La classe base definisce la struttura dell'algoritmo (metodo Template) e la sottoclasse sovrascrive una parte dell'algoritmo

snippet di codice:

```
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();
        }
    }
}
```

produzione:

```
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:
```

Leggi Polimorfismo online: https://riptutorial.com/it/java/topic/980/polimorfismo

Capitolo 137: Pool Executor, ExecutorService e Thread

introduzione

L'interfaccia di Executor in Java fornisce un modo per disaccoppiare l'invio di attività dai meccanismi di esecuzione di ciascuna attività, inclusi i dettagli sull'utilizzo dei thread, la pianificazione, ecc. Normalmente viene utilizzato un Executor invece di creare esplicitamente i thread. Con gli Executor, gli sviluppatori non dovranno riscrivere in modo significativo il loro codice per essere in grado di regolare facilmente i criteri di esecuzione delle attività del loro programma.

Osservazioni

insidie

- Quando si pianifica un'attività per l'esecuzione ripetuta, a seconda di ScheduledExecutorService utilizzato, l'attività potrebbe essere sospesa da qualsiasi ulteriore esecuzione, se un'esecuzione dell'attività determina un'eccezione non gestita. Vedi Mother F ** k ScheduledExecutorService!

Examples

Fire and Forget - Runnable Tasks

Gli esecutori accettano un `java.lang.Runnable` che contiene codice (potenzialmente computazionalmente o altrimenti di lunga esecuzione o pesante) da eseguire in un'altra discussione.

L'utilizzo sarebbe:

```
Executor exec = anExecutor;
exec.execute(new Runnable() {
    @Override public void run() {
        //offloaded work, no need to get result back
    }
});
```

Si noti che con questo esecutore non si ha alcun modo per ottenere alcun valore calcolato.
Con Java 8, è possibile utilizzare lambda per abbreviare l'esempio di codice.

Java SE 8

```
Executor exec = anExecutor;
exec.execute(() -> {
    //offloaded work, no need to get result back
});
```

ThreadPoolExecutor

Un Executor comune utilizzato è il `ThreadPoolExecutor` , che si occupa della gestione dei thread. È possibile configurare la quantità minima di thread che l'executor deve sempre mantenere quando non c'è molto da fare (si chiama dimensione del core) e una dimensione massima del thread a cui il Pool può crescere, se c'è più lavoro da fare. Una volta che il carico di lavoro diminuisce, il Pool riduce lentamente il numero di thread fino a quando non raggiunge la dimensione minima.

```
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 Se si configura `ThreadPoolExecutor` con una coda *illimitata* , il conteggio dei thread non supererà `corePoolSize` poiché i nuovi thread vengono creati solo se la coda è piena:

ThreadPoolExecutor con tutti i parametri:

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

da JavaDoc

> Se ci sono più di corePoolSize ma meno di maximumPoolSize thread in esecuzione, verrà creato un nuovo thread solo se la coda è piena.

vantaggi:

1. La dimensione di BlockingQueue può essere controllata e gli scenari di esaurimento della memoria possono essere evitati. Le prestazioni dell'applicazione non saranno ridotte con le dimensioni limitate della coda limitata.
2. È possibile utilizzare esistenti o creare nuove politiche del gestore di rifiuto.
    1. Nel file ThreadPoolExecutor.AbortPolicy predefinito, il gestore rilascia una RejectedExecutionException di runtime in caso di rifiuto.
    2. In `ThreadPoolExecutor.CallerRunsPolicy` , il thread che richiama execute esegue l'attività. Ciò fornisce un semplice meccanismo di controllo di feedback che rallenta la velocità con cui vengono inviate nuove attività.

3. In `ThreadPoolExecutor.DiscardPolicy` , un'attività che non può essere eseguita viene semplicemente eliminata.

4. In `ThreadPoolExecutor.DiscardOldestPolicy` , se l'executor non viene arrestato, l'attività in testa alla coda di lavoro viene interrotta e quindi l'esecuzione viene ritentata (che può fallire nuovamente, provocando la ripetizione).

3. `ThreadFactory` possibile configurare Custom `ThreadFactory` , che è utile:

    1. Per impostare un nome di thread più descrittivo
    2. Per impostare lo stato del daemon del thread
    3. Per impostare la priorità del thread

Ecco un esempio di come utilizzare ThreadPoolExecutor

Recupero del valore dal calcolo - Callable

Se il tuo calcolo produce un valore di ritorno che in seguito è richiesto, una semplice attività Runnable non è sufficiente. Per tali casi è possibile utilizzare `ExecutorService.submit( Callable <T>)` che restituisce un valore al termine dell'esecuzione.

Il servizio restituirà un `Future` che è possibile utilizzare per recuperare il risultato dell'esecuzione dell'attività.

```
// 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
```

Quando hai bisogno di ottenere il risultato del futuro, chiama `future.get()`

- Aspetta indefinitamente che il futuro finisca con un risultato.

```
try {
    // Blocks current thread until future is completed
    Integer result = future.get();
catch (InterruptedException || ExecutionException e) {
    // handle appropriately
}
```

- Aspetta che il futuro finisca, ma non più del tempo specificato.

```
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
}
```

Se il risultato di un'attività pianificata o in esecuzione non è più necessario, puoi chiamare `Future.cancel(boolean)` per cancellarlo.

- Calling `cancel(false)` rimuove l'attività dalla coda delle attività da eseguire.
- La chiamata `cancel(true)` interromperà *anche* l'attività se è attualmente in esecuzione.

Pianificazione delle attività da eseguire a un'ora fissa, dopo un ritardo o ripetutamente

La classe `ScheduledExecutorService` fornisce metodi per pianificare attività singole o ripetute in diversi modi. Il seguente esempio di codice presuppone che il `pool` sia stato dichiarato e inizializzato come segue:

```
ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);
```

Oltre ai normali metodi `ExecutorService` , l'API `ScheduledExecutorService` aggiunge 4 metodi che pianificano le attività e restituiscono oggetti `ScheduledFuture` . Quest'ultimo può essere utilizzato per recuperare i risultati (in alcuni casi) e annullare le attività.

Avvio di un'attività dopo un ritardo fisso

L'esempio seguente pianifica l'avvio di un'attività dopo dieci minuti.

```
ScheduledFuture<Integer> future = pool.schedule(new Callable<>() {
        @Override public Integer call() {
            // do something
            return 42;
        }
    },
    10, TimeUnit.MINUTES);
```

Avvio di attività a una velocità fissa

L'esempio seguente pianifica l'avvio di un'attività dopo dieci minuti, quindi ripetutamente a una velocità di una volta ogni minuto.

```
ScheduledFuture<?> future = pool.scheduleAtFixedRate(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

L'esecuzione delle attività continuerà in base alla pianificazione fino a quando il `pool` verrà

arrestato, il `future` verrà annullato o una delle attività riscontra un'eccezione.

È garantito che le attività pianificate da una determinata chiamata `scheduledAtFixedRate` non si sovrappongano nel tempo. Se un'attività richiede più tempo del periodo prescritto, la successiva e successiva esecuzione delle attività potrebbe iniziare in ritardo.

Avvio di attività con un ritardo fisso

L'esempio seguente pianifica l'avvio di un'attività dopo dieci minuti, quindi ripetutamente con un ritardo di un minuto tra la fine di un'attività e la successiva.

```
ScheduledFuture<?> future = pool.scheduleWithFixedDelay(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

L'esecuzione delle attività continuerà in base alla pianificazione fino a quando il `pool` verrà arrestato, il `future` verrà annullato o una delle attività riscontra un'eccezione.

Gestire l'esecuzione rifiutata

Se

1. si tenta di inviare attività a un Executor di arresto o
2. la coda è saturata (possibile solo con quelli delimitati) e il numero massimo di thread è stato raggiunto,

Verrà richiamato `RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor)` .

Il comportamento predefinito prevede che venga generata una eccezione RejectedExecutionException al chiamante. Ma ci sono più comportamenti predefiniti disponibili:

- **ThreadPoolExecutor.AbortPolicy** (predefinito, genererà REE)
- **ThreadPoolExecutor.CallerRunsPolicy** (esegue l'attività sul thread del chiamante - *bloccandolo*)
- **ThreadPoolExecutor.DiscardPolicy** (attività scartare silenziosamente)
- **ThreadPoolExecutor.DiscardOldestPolicy** (**elimina in modo** silenzioso l'attività meno **recente** in coda e riprova l'esecuzione della nuova attività)

È possibile impostarli utilizzando uno dei costruttori 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) // <--
```

Puoi anche implementare il tuo comportamento estendendo l'interfaccia di RejectedExecutionHandler :

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
```

submit () vs execute () differenze nella gestione delle eccezioni

Generalmente il comando execute () viene usato per il fuoco e dimentica le chiamate (senza necessità di analizzare il risultato) e il comando submit () viene utilizzato per analizzare il risultato dell'oggetto Future.

Dovremmo essere consapevoli della differenza fondamentale dei meccanismi di gestione delle eccezioni tra questi due comandi.

Le eccezioni da submit () sono inghiottite dal framework se non le hai catturate.

Esempio di codice per capire la differenza:

Caso 1: invia il comando Runnable with execute (), che segnala l'eccezione.

```
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);
  }
}
```

produzione:

```
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: Sostituisci execute () con submit (): `service.submit(new Runnable(){` In questo caso, le eccezioni sono inghiottite dal framework poiché il metodo run () non le ha catturate esplicitamente.

produzione:

```
creating service
a and b=4:0
a and b=4:0
```

Caso 3: cambia il nuovo FixedThreadPool in ExtendedExecutor

```
//ExecutorService service = Executors.newFixedThreadPool(2);
 ExtendedExecutor service = new ExtendedExecutor();
```

produzione:

```
creating service
a and b=4:0
java.lang.ArithmeticException: / by zero
a and b=4:0
java.lang.ArithmeticException: / by zero
```

Ho dimostrato questo esempio per coprire due argomenti: Usa il tuo ThreadPoolExecutor personalizzato e gestisci Exectpion con ThreadPoolExecutor personalizzato.

Altra soluzione semplice al problema precedente: quando si utilizza il normale comando ExecutorService e invio, ottenere l'oggetto Future da submit () chiamata a comando get () API su Future. Cattura le tre eccezioni, che sono state citate nell'implementazione del metodo afterExecute. Vantaggio di ThreadPoolExecutor personalizzato rispetto a questo approccio: è necessario gestire il meccanismo di gestione delle eccezioni in un unico punto: ThreadPoolExecutor personalizzato.

Utilizzare i casi per diversi tipi di costrutti di concorrenza

1. ExecutorService

   ```
   ExecutorService executor = Executors.newFixedThreadPool(50);
   ```

   È semplice e facile da usare. Nasconde i dettagli di basso livello di `ThreadPoolExecutor` .

   Preferisco questo quando il numero di attività `Callable/Runnable` è di numero ridotto e l'accumulo di attività nella coda illimitata non aumenta la memoria e peggiora le prestazioni del sistema. Se si dispone di vincoli `CPU/Memory` , preferisco utilizzare `ThreadPoolExecutor` con limitazioni di capacità e `RejectedExecutionHandler` per gestire il rifiuto delle attività.

2. CountDownLatch

   `CountDownLatch` verrà inizializzato con un determinato conteggio. Questo conteggio viene decrementato dalle chiamate al metodo `countDown()` . I thread in attesa che questo conteggio raggiunga lo zero possono chiamare uno dei metodi `await()` . Calling `await()` blocca il thread finché il conteggio non raggiunge lo zero. *Questa classe abilita un thread java ad attendere fino a quando un altro set di thread completa le loro attività.*

   Casi d'uso:

   1. Raggiungere il parallelismo massimo: a volte vogliamo iniziare un numero di thread contemporaneamente per ottenere il massimo parallelismo
   2. Attendi il completamento di N thread prima di iniziare l'esecuzione
   3. Rilevamento deadlock.

3. ThreadPoolExecutor : fornisce più controllo. Se l'applicazione è vincolata dal numero di attività Runnable / Callable in sospeso, è possibile utilizzare la coda limitata impostando la capacità massima. Una volta che la coda raggiunge la capacità massima, è possibile definire RejectionHandler. Java fornisce quattro tipi di criteri `RejectedExecutionHandler` .

1. `ThreadPoolExecutor.AbortPolicy` , il gestore genera una runtime RejectedExecutionException in seguito a rifiuto.

2. ThreadPoolExecutor.CallerRunsPolicy`, il thread che richiama execute esegue l'attività. Ciò fornisce un semplice meccanismo di controllo di feedback che rallenta la velocità con cui vengono inviate nuove attività.

3. In `ThreadPoolExecutor.DiscardPolicy` , un'attività che non può essere eseguita viene semplicemente eliminata.

4. `ThreadPoolExecutor.DiscardOldestPolicy` , se l'executor non viene arrestato, l'attività in testa alla coda di lavoro viene interrotta e quindi l'esecuzione viene ritentata (che può fallire nuovamente, provocando la ripetizione).

Se si desidera simulare il comportamento di `CountDownLatch` , è possibile utilizzare il metodo `invokeAll()` .

4. Un altro meccanismo che non hai citato è ForkJoinPool

   `ForkJoinPool` stato aggiunto a Java in Java 7. `ForkJoinPool` è simile a Java `ExecutorService` ma con una differenza. `ForkJoinPool` rende facile per le attività suddividere il proprio lavoro in attività più piccole, che vengono poi inoltrate al `ForkJoinPool` . Il furto di attività avviene in `ForkJoinPool` quando i thread di lavoro liberi rubano le attività dalla coda di thread di lavoro occupata.

   Java 8 ha introdotto un'altra API in ExecutorService per creare pool di stealing del lavoro. Non è necessario creare `RecursiveTask` e `RecursiveAction` ma è comunque possibile utilizzare `ForkJoinPool` .

   ```
   public static ExecutorService newWorkStealingPool()
   ```

   Crea un pool di thread di lavoro-furto utilizzando tutti i processori disponibili come livello di parallelismo di destinazione.

   Per impostazione predefinita, richiederà un numero di core CPU come parametro.

Tutti questi quattro meccanismi sono complementari l'uno all'altro. A seconda del livello di granularità che si desidera controllare, è necessario scegliere quelli giusti.

Attendere il completamento di tutte le attività in ExecutorService

Diamo un'occhiata alle varie opzioni per attendere il completamento delle attività inviate a Executor

1. ExecutorService `invokeAll()`

   Esegue i compiti assegnati, restituendo un elenco di Futures che mantengono il loro stato e i risultati quando tutto è completato.

Esempio:

```
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

   Un aiuto di sincronizzazione che consente a uno o più thread di attendere fino al completamento di un insieme di operazioni eseguite in altri thread.

   Un **CountDownLatch** viene inizializzato con un determinato conteggio. Il blocco `countDown()` i metodi fino a quando il conteggio corrente non raggiunge lo zero a causa di invocazioni del metodo `countDown()` , dopo il quale tutti i thread in attesa vengono rilasciati e qualsiasi successiva chiamata di attesa attende immediatamente. Questo è un fenomeno one-shot: il conteggio non può essere resettato. Se è necessaria una versione che azzeri il conteggio, prendere in considerazione l'uso di **CyclicBarrier** .

3. ForkJoinPool o `newWorkStealingPool()` in Executor

4. Scorrere tutti gli oggetti `Future` creati dopo l'invio a `ExecutorService`

5. Modo consigliato di spegnimento dalla pagina di documentazione di Oracle di 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()`: avvia uno spegnimento ordinato in cui vengono eseguite le attività precedentemente inviate, ma non verranno accettate nuove attività.

`shutdownNow()`: tenta di interrompere tutte le attività che eseguono attivamente, interrompe l'elaborazione delle attività in attesa e restituisce un elenco delle attività che erano in attesa di esecuzione.

Nell'esempio precedente, se le attività richiedono più tempo per essere completate, è possibile modificare la condizione a condizione while

Sostituire

```
if (!pool.awaitTermination(60, TimeUnit.SECONDS))
```

con

```
while(!pool.awaitTermination(60, TimeUnit.SECONDS)) {
  Thread.sleep(60000);
```

}

Utilizzare i casi per diversi tipi di ExecutorService

Gli esecutori restituiscono diversi tipi di ThreadPools che soddisfano esigenze specifiche.

1. `public static ExecutorService newSingleThreadExecutor()`

   Crea un Executor che utilizza un singolo thread di lavoro che opera su una coda illimitata

   C'è una differenza tra `newFixedThreadPool(1)` e `newSingleThreadExecutor()` come dice il documento java per quest'ultimo:

A differenza del newFixedThreadPool (1) altrimenti equivalente, l'executor restituito non è riconfigurabile per utilizzare thread aggiuntivi.

Ciò significa che un nuovo `newFixedThreadPool` può essere riconfigurato successivamente nel programma da: `((ThreadPoolExecutor) fixedThreadPool).setMaximumPoolSize(10)` Questo non è possibile per `newSingleThreadExecutor`

Casi d'uso:

1. Si desidera eseguire le attività inoltrate in sequenza.
2. È necessaria solo una discussione per gestire tutte le richieste

Contro:

1. La coda illimitata è dannosa

2. `public static ExecutorService newFixedThreadPool(int nThreads)`

   Crea un pool di thread che riutilizza un numero fisso di thread che operano su una coda non associata condivisa. In qualsiasi momento, al massimo i thread nThreads saranno attività di elaborazione attive. Se vengono inoltrate attività aggiuntive quando tutti i thread sono attivi, attenderanno in coda fino a quando un thread non sarà disponibile

   Casi d'uso:

   1. Uso efficace dei nuclei disponibili. Configura `nThreads` come `Runtime.getRuntime().availableProcessors()`
   2. Quando si decide che il numero di thread non deve superare un numero nel pool di thread

   Contro:

   1. La coda illimitata è dannosa.

3. `public static ExecutorService newCachedThreadPool()`

   Crea un pool di thread che crea nuovi thread in base alle necessità, ma riutilizzerà i thread creati in precedenza quando sono disponibili

   Casi d'uso:

   1. Per attività asincrone di breve durata

   Contro:

   1. La coda illimitata è dannosa.
   2. Ogni nuova attività creerà un nuovo thread se tutti i thread esistenti sono occupati. Se l'attività richiede una lunga durata, verrà creato più numero di thread, il che peggiorerà le prestazioni del sistema. Alternativa in questo caso: `newFixedThreadPool`

4. `public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`

Crea un pool di thread in grado di pianificare i comandi da eseguire dopo un determinato ritardo o da eseguire periodicamente.

Casi d'uso:

1. Gestione di eventi ricorrenti con ritardi, che avverranno in futuro in determinati intervalli di tempo

Contro:

1. La coda illimitata è dannosa.

5. `public static ExecutorService newWorkStealingPool()`

Crea un pool di thread di lavoro-furto utilizzando tutti i processori disponibili come livello di parallelismo di destinazione

Casi d'uso:

1. Per dividere e conquistare il tipo di problemi.
2. Uso efficace dei thread inattivi. I thread inattivi rubano le attività dai thread occupati.

Contro:

1. La dimensione della coda illimitata è dannosa.

È possibile visualizzare uno svantaggio comune in tutti questi ExecutorService: coda illimitata. Questo sarà risolto con ThreadPoolExecutor

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

Con `ThreadPoolExecutor` , è possibile

1. Controlla la dimensione del pool di thread in modo dinamico
2. Imposta la capacità per `BlockingQueue`
3. Definisci `RejectionExecutionHander` quando la coda è piena
4. `CustomThreadFactory` per aggiungere alcune funzionalità aggiuntive durante la creazione del thread `(public Thread newThread(Runnable r)`

Utilizzo di pool di thread

I pool di thread vengono utilizzati principalmente i metodi di chiamata in `ExecutorService` .

I seguenti metodi possono essere utilizzati per inviare lavori per l'esecuzione:

Metodo	Descrizione
`submit`	Esegue un lavoro inviato e restituisce un futuro che può essere utilizzato per

Metodo	Descrizione
	ottenere il risultato
`execute`	Esegui l'attività in un momento futuro senza ottenere alcun valore di ritorno
`invokeAll`	Esegui un elenco di attività e restituisci un elenco di Futures
`invokeAny`	Esegue tutto ma restituisce solo il risultato di uno che ha avuto successo (senza eccezioni)

Una volta che hai finito il Thread Pool puoi chiamare `shutdown()` per chiudere il Thread Pool. Questo esegue tutte le attività in sospeso. Per attendere l'esecuzione di tutte le attività, è possibile eseguire il loop su `awaitTermination` o `isShutdown()` .

Leggi Pool Executor, ExecutorService e Thread online: https://riptutorial.com/it/java/topic/143/pool-executor--executorservice-e-thread

Capitolo 138: Preferenze

Examples

Aggiunta di listener di eventi

Ci sono due tipi di eventi emessi da una `Preferences` oggetto: `PreferenceChangeEvent` e `NodeChangeEvent` .

PreferenceChangeEvent

Un `PreferenceChangeEvent` viene emesso da un oggetto `Properties` ogni volta che cambia una coppia chiave-valore del nodo. `PreferenceChangeEvent` può essere ascoltato con `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();
    }
});
```

Questo listener non ascolterà le coppie di valori-chiave modificati dei nodi figli.

NodeChangeEvent

Questo evento verrà generato ogni volta che viene aggiunto o rimosso un nodo figlio di 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();
    }
});
```

Ottenere sottonodi delle preferenze

`Preferences` oggetti `Preferences` rappresentano sempre un nodo specifico in un intero albero delle `Preferences` , un po 'come questo:

```
/userRoot
├── com
│   └── mycompany
│       └── myapp
│           ├── darkApplicationMode=true
│           ├── showExitConfirmation=false
│           └── windowMaximized=true
└── org
    └── myorganization
        └── anotherapp
            ├── defaultFont=Helvetica
            ├── defaultSavePath=/home/matt/Documents
            └── exporting
                ├── defaultFormat=pdf
                └── openInBrowserAfterExport=false
```

Per selezionare il nodo /com/mycompany/myapp :

1. Per convenzione, in base al pacchetto di una classe:

   ```
   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. Per percorso relativo:

   ```
   Preferences myApp = Preferences.userRoot().node("com/mycompany/myapp");
   ```

   L'utilizzo di un percorso relativo (un percorso che non inizia con a /) causerà la risoluzione del percorso rispetto al nodo genitore su cui è stato risolto. Ad esempio, il seguente esempio restituirà il nodo del percorso /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. Per via assoluta:

   ```
   Preferences myApp = Preferences.userRoot().node("/com/mycompany/myapp");
   ```

   L'utilizzo di un percorso assoluto sul nodo radice non sarà diverso dall'utilizzo di un percorso relativo. La differenza è che, se chiamato su un sub-nodo, il percorso sarà

risolto rispetto al nodo radice.

```
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
```

Accesso alle preferenze di coordinamento tra più istanze di applicazioni

Tutte le istanze di Preferences sono sempre thread-safe attraverso i thread di una singola Java Virtual Machine (JVM). Poiché le Preferences possono essere condivise tra più JVM, esistono metodi speciali che gestiscono la sincronizzazione delle modifiche tra macchine virtuali.

Se si dispone di un'applicazione che deve essere eseguita solo in una **singola istanza** , **non** è richiesta **alcuna sincronizzazione esterna** .

Se si dispone di un'applicazione che viene eseguita in **più istanze** su un singolo sistema e quindi l'accesso alle Preferences deve essere coordinato tra le JVM sul sistema, allora il **metodo sync()** di qualsiasi nodo Preferences può essere usato per assicurare che le modifiche al nodo Preferences siano visibile ad altre JVM sul 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();
}
```

Esportare le preferenze

Preferences nodi di Preferences possono essere esportati in un documento XML che rappresenta quel nodo. L'albero XML risultante può essere nuovamente importato. Il documento XML risultante ricorderà se è stato esportato dall'utente o dalle Preferences sistema.

Per esportare un singolo nodo, ma **non i suoi nodi figli** :

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) {}
    }
}
```

Per esportare un singolo nodo **con i relativi nodi figlio :**

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) {}
    }
}
```

Importazione delle preferenze

Preferences nodi delle Preferences possono essere importati da un documento XML. L'importazione è pensata per essere utilizzata insieme alla funzionalità di esportazione delle Preferences , poiché crea i documenti XML corrispondenti corretti.

I documenti XML ricorderanno se sono stati esportati dall'utente o dalle Preferences sistema. Pertanto, possono essere importati nuovamente nei loro rispettivi alberi delle Preferences , senza che tu debba capire o sapere da dove provengono. La funzione statica rileva automaticamente se il documento XML è stato esportato dall'utente o dalle Preferences sistema e le importerà automaticamente nella struttura da cui sono state esportate.

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) {}
    }
}
```

Rimozione dei listener di eventi

I listener di eventi possono essere rimossi nuovamente da qualsiasi nodo Properties , ma l'istanza del listener deve essere mantenuta per questo.

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 stesso vale per NodeChangeListener .

Ottenere i valori delle preferenze

Un valore di un nodo Preferences può essere di tipo String , boolean , byte[] , double , float , int o long . Tutte le invocazioni devono fornire un valore predefinito, nel caso in cui il valore specificato non sia presente nel 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);
```

Impostazione dei valori delle preferenze

Per memorizzare un valore nel nodo Preferences , viene utilizzato uno dei metodi putXXX() . Un valore di un nodo Preferences può essere di 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 le preferenze

Preferences possono essere utilizzate per memorizzare le impostazioni dell'utente che riflettono le impostazioni dell'applicazione personale dell'utente, ad esempio il carattere del loro editor, se preferiscono che l'applicazione sia avviata in modalità a schermo intero, se hanno spuntato la casella di controllo "non mostrare più" e come quello.

```
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);
        }
    }
}
```

Leggi Preferenze online: https://riptutorial.com/it/java/topic/582/preferenze

Capitolo 139: Processi

Osservazioni

Si noti che l'API raccomanda che, a partire dalla versione 1.5, il modo preferito per creare un processo sia l'utilizzo di ProcessBuilder.start() .

Un'altra osservazione importante è che il valore di uscita prodotto da waitFor dipende dal programma / script in esecuzione. Ad esempio, i codici di uscita prodotti da **calc.exe** sono diversi da **notepad.exe** .

Examples

Esempio semplice (versione Java <1.5)

Questo esempio chiamerà il calcolatore di Windows. È importante notare che il codice di uscita varierà in base al programma / script che viene chiamato.

```
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();
        }
    }
}
```

Utilizzando la classe ProcessBuilder

La classe ProcessBuilder semplifica l'invio di un comando tramite la riga di comando. Tutto ciò che serve è un elenco di stringhe che compongono i comandi da inserire. Basta chiamare il metodo start () sull'istanza ProcessBuilder per eseguire il comando.

Se hai un programma chiamato Add.exe che prende due argomenti e li aggiunge, il codice sarebbe simile a questo:

```
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();
}
```

Alcune cose da tenere a mente:

- L'array di comandi deve essere tutto un array di stringhe
- I comandi devono essere nell'ordine (nell'array) che sarebbero se avessi fatto la chiamata al programma nella linea di comando stessa (cioè il nome dell'exe non può andare dopo il primo argomento
- Quando si imposta la directory di lavoro è necessario passare in un oggetto File e non solo il nome del file come una stringa

Blocco contro chiamate non bloccanti

In generale quando si effettua una chiamata alla riga di comando, il programma invierà il comando e quindi continuerà la sua esecuzione.

Tuttavia potresti voler aspettare che il programma chiamato finisca prima di continuare la tua esecuzione (ad esempio, il programma chiamato scriverà i dati su un file e il tuo programma avrà bisogno di accedere a quei dati).

Questo può essere fatto facilmente chiamando il metodo waitFor() Process restituita.

Esempio di utilizzo:

```
//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

L'avvio di processi esterni da Java utilizzando direttamente l'API java.lang.ProcessBuilder non elaborata può risultare un po 'macchinoso. La libreria Apache Commons Exec lo rende un po 'più semplice. La libreria ch.vorburger.exec si estende ulteriormente su Commons Exec per renderlo veramente 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 e ProcessBuilder non capiscono la sintassi della shell

I metodi Runtime.exec(String ...) e Runtime.exec(String) consentono di eseguire un comando come processo esterno [1] . Nella prima versione, si fornisce il nome del comando e gli argomenti del comando come elementi separati dell'array di stringhe e il runtime Java richiede al sistema di runtime del sistema operativo di avviare il comando esterno. La seconda versione è ingannevolmente facile da usare, ma presenta alcune insidie.

Prima di tutto, ecco un esempio di utilizzo di exec(String) usato in modo sicuro:

```
Process p = Runtime.exec("mkdir /tmp/testDir");
p.waitFor();
if (p.exitValue() == 0) {
    System.out.println("created the directory");
}
```

Spazi nei nomi dei percorsi

Supponiamo di generalizzare l'esempio sopra in modo da poter creare una directory arbitraria:

```
Process p = Runtime.exec("mkdir " + dirPath);
// ...
```

Questo in genere funziona, ma fallirà se dirPath è (ad esempio) "/ home / utente / Documenti". Il problema è che exec(String) divide la stringa in un comando e argomenti semplicemente cercando spazi bianchi. La stringa di comando:

```
"mkdir /home/user/My Documents"
```

sarà diviso in:

```
"mkdir", "/home/user/My", "Documents"
```

e questo causerà il fallimento del comando "mkdir" perché prevede un argomento, non due.

Di fronte a questo, alcuni programmatori cercano di aggiungere citazioni attorno al percorso. Questo non funziona neanche:

```
"mkdir \"/home/user/My Documents\""
```

sarà diviso in:

```
"mkdir", "\"/home/user/My", "Documents\""
```

I caratteri di doppia virgoletta aggiuntivi che sono stati aggiunti nel tentativo di "quotare" gli spazi sono trattati come qualsiasi altro carattere non di uno spazio bianco. In effetti, tutto ciò che citiamo o sfuggiamo agli spazi fallirà.

Il modo di affrontare questo particolare problema è usare il sovraccarico exec(String ...) .

```
Process p = Runtime.exec("mkdir", dirPath);
// ...
```

Questo funzionerà se dirpath include caratteri di spaziatura perché questo overload di exec non tenta di dividere gli argomenti. Le stringhe vengono passate alla exec sistema exec sistema operativo così com'è.

Reindirizzamento, pipeline e altra sintassi della shell

Supponiamo di voler reindirizzare l'input o l'output di un comando esterno o eseguire una pipeline. Per esempio:

```
Process p = Runtime.exec("find / -name *.java -print 2>/dev/null");
```

o

```
Process p = Runtime.exec("find source -name *.java | xargs grep package");
```

(Il primo esempio elenca i nomi di tutti i file Java nel file system e il secondo stampa le istruzioni del package [2] nei file Java nell'albero "origine".)

Questi non funzioneranno come previsto. Nel primo caso, il comando "trova" verrà eseguito con "2> / dev / null" come argomento del comando. Non sarà interpretato come un reindirizzamento. Nel secondo esempio, il carattere pipe ("|") e le opere successive verranno dati al comando "find".

Il problema qui è che i metodi exec e ProcessBuilder non capiscono alcuna sintassi della shell. Ciò include reindirizzamenti, pipeline, espansione variabile, globbing e così via.

In alcuni casi (ad esempio, il reindirizzamento semplice) è possibile ottenere facilmente l'effetto desiderato utilizzando ProcessBuilder . Tuttavia, questo non è vero in generale. Un approccio alternativo è quello di eseguire la riga di comando in una shell; per esempio:

```
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");
```

Notate che nel secondo esempio, dovevamo sfuggire al carattere jolly ("*") perché vogliamo che il carattere jolly venga interpretato da "find" piuttosto che dalla shell.

I comandi incorporati della shell non funzionano

Supponiamo che i seguenti esempi non funzionino su un sistema con una shell simile 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
```

Ci sono un paio di motivi per cui questo non funzionerà:

1. I comandi "cd" e "export" sono comandi incorporati nella shell. Non esistono come

eseguibili distinti.

2. Perché i builtin della shell facciano ciò che dovrebbero fare (ad esempio cambiano la directory di lavoro, aggiornano l'ambiente), hanno bisogno di cambiare il luogo in cui si trova lo stato. Per un'applicazione normale (inclusa un'applicazione Java) lo stato è associato al processo dell'applicazione. Ad esempio, il processo figlio che eseguiva il comando "cd" non poteva cambiare la directory di lavoro del suo processo "java" genitore. Allo stesso modo, un processo di exec 'd non può cambiare la directory di lavoro per un processo che lo segue.

Questo ragionamento si applica a tutti i comandi incorporati della shell.

---

1 - È possibile utilizzare ProcessBuilder , ma ciò non è pertinente al punto di questo esempio.

2 - Questo è un po 'approssimativo e pronto ... ma ancora una volta, i difetti di questo approccio non sono rilevanti per l'esempio.

Leggi Processi online: https://riptutorial.com/it/java/topic/4682/processi

Capitolo 140: Programmazione parallela con il framework Fork / Join

Examples

Fork / Join Tasks in Java

Il framework fork / join in Java è ideale per un problema che può essere suddiviso in parti più piccole e risolto in parallelo. I passaggi fondamentali di un problema di fork / join sono:

- Dividi il problema in più parti
- Risolvi ognuno dei pezzi in parallelo tra loro
- Combina ciascuna delle sub-soluzioni in un'unica soluzione globale

ForkJoinTask è l'interfaccia che definisce tale problema. In genere è previsto che sottoclassi una delle sue implementazioni astratte (di solito il RecursiveTask) piuttosto che implementare direttamente l'interfaccia.

In questo esempio, andremo a sommare una raccolta di numeri interi, dividendo fino a raggiungere dimensioni di batch non superiori a dieci.

```
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;
        }
    }
}
```

Un'istanza di questa attività può ora essere passata a un'istanza di 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();
```

Leggi Programmazione parallela con il framework Fork / Join online:
https://riptutorial.com/it/java/topic/4245/programmazione-parallela-con-il-framework-fork---join

Capitolo 141: Programmazione simultanea (thread)

introduzione

Il calcolo simultaneo è una forma di calcolo in cui diversi calcoli vengono eseguiti contemporaneamente anziché in modo sequenziale. Il linguaggio Java è progettato per supportare la programmazione concorrente attraverso l'uso di thread. Gli oggetti e le risorse sono accessibili da più thread; ogni thread può potenzialmente accedere a qualsiasi oggetto nel programma e il programmatore deve garantire che l'accesso in lettura e scrittura agli oggetti sia sincronizzato correttamente tra i thread.

Osservazioni

Argomenti correlati su StackOverflow:

- Tipi atomici
- Pool Executor, ExecutorService e Thread
- Estensione del Thread rispetto all'implementazione Runnable

Examples

Multithreading di base

Se hai molte attività da eseguire e tutte queste attività non dipendono dal risultato di quelle precedenti, puoi utilizzare il **Multithreading** per il tuo computer per eseguire tutte queste attività contemporaneamente utilizzando più processori, se il tuo computer lo può fare. Questo può rendere **più veloce l'** esecuzione del tuo programma se hai delle grosse attività indipendenti.

```
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);
        }
    }
}
```

Il codice del metodo di esecuzione delle varie istanze CountAndPrint verrà eseguito in un ordine non prevedibile. Un frammento di un'esecuzione di esempio potrebbe essere simile a questo:

```
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
...
```

Producer-Consumer

Un semplice esempio di soluzione di problemi produttore-consumatore. Si noti che le classi JDK (AtomicBoolean e BlockingQueue) vengono utilizzate per la sincronizzazione, riducendo la possibilità di creare una soluzione non valida. Consultare Javadoc per vari tipi di BlockingQueue ; la scelta di un'implementazione differente può modificare drasticamente il comportamento di questo esempio (come 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 needeed 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

Uno strumento utile in Java Concurrency è ThreadLocal - questo ti permette di avere una variabile che sarà unica per un dato thread. Pertanto, se lo stesso codice viene eseguito in thread diversi, queste esecuzioni non condivideranno il valore, ma ogni thread ha una propria variabile che è *locale al thread* .

Ad esempio, questo è frequentemente usato per stabilire il contesto (come le informazioni di autorizzazione) della gestione di una richiesta in un servlet. Potresti fare qualcosa del genere:

```
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
    }
}
```

Ora, invece di passare MyUserContext in ogni singolo metodo, puoi invece utilizzare MyServlet.getContext() dove ti serve. Ora, naturalmente, questo introduce una variabile che deve essere documentata, ma è thread-safe, che elimina molti aspetti negativi nell'utilizzo di una variabile così ad alto scope.

Il vantaggio chiave qui è che ogni thread ha la propria variabile locale del thread nel contenitore dei contexts . Finché lo si utilizza da un punto di ingresso definito (come richiedere che ogni servlet mantenga il suo contesto, o magari aggiungendo un filtro servlet), si può fare affidamento sul fatto che questo contesto sia lì quando ne hai bisogno.

CountDownLatch

CountDownLatch

> Un aiuto di sincronizzazione che consente a uno o più thread di attendere fino al completamento di un insieme di operazioni eseguite in altri thread.

1. Un CountDownLatch viene inizializzato con un determinato conteggio.
2. Il blocco countDown() i metodi fino a quando il conteggio corrente non raggiunge lo zero a causa di invocazioni del metodo countDown() , dopo il quale tutti i thread in attesa vengono rilasciati e qualsiasi successiva chiamata di attesa attende immediatamente.
3. Questo è un fenomeno one-shot: il conteggio non può essere resettato. Se è necessaria una versione che azzeri il conteggio, prendere in considerazione l'uso di CyclicBarrier .

Metodi chiave:

```
public void await() throws InterruptedException
```

> Fa in modo che il thread corrente attenda fino a quando il latch non viene contato fino a zero, a meno che il thread non venga interrotto.

```
public void countDown()
```

> Riduce il conteggio del latch, rilasciando tutti i thread in attesa se il conteggio raggiunge lo zero.

Esempio:

```
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();
        }
    }
}
```

produzione:

```
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
```

Spiegazione:

1. CountDownLatch è inizializzato con un contatore di 5 in thread principale
2. Il thread principale è in attesa usando il metodo await() .
3. Sono state create cinque istanze di DoSomethingInAThread . Ogni istanza decrementava il contatore con il metodo countDown() .
4. Quando il contatore diventa zero, il thread principale riprenderà

Sincronizzazione

In Java, esiste un meccanismo di blocco a livello di lingua incorporato: il blocco synchronized , che può utilizzare qualsiasi oggetto Java come blocco intrinseco (ad esempio, ogni oggetto Java può avere un monitor ad esso associato).

I blocchi intrinseci forniscono l'atomicità a gruppi di dichiarazioni. Per capire cosa questo significhi per noi, diamo un'occhiata a un esempio in cui la synchronized è utile:

```
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();
}
```

In questo caso, se non fosse stato per il blocco synchronized , sarebbero stati coinvolti più problemi di concorrenza. Il primo sarebbe con l'operatore post-incremento (non è atomico in sé), e il secondo sarebbe che osserveremmo il valore di t dopo che una quantità arbitraria di altri thread ha avuto la possibilità di modificarla. Tuttavia, poiché abbiamo acquisito un blocco intrinseco, qui non ci saranno condizioni di gara e l'uscita conterrà numeri da 1 a 100 nel loro ordine normale.

I blocchi intrinseci in Java sono *mutex* (cioè blocchi di esecuzione reciproca). L'esecuzione reciproca significa che se un thread ha acquisito il blocco, il secondo sarà costretto ad aspettare che il primo lo rilasci prima che possa acquisire il blocco per se stesso. Nota: un'operazione che può mettere il thread nello stato di attesa (sospensione) viene chiamata un'operazione di *blocco* . Pertanto, l'acquisizione di un blocco è un'operazione di blocco.

I blocchi intrinseci in Java sono *rientranti* . Ciò significa che se un thread tenta di acquisire un lock che già possiede, non lo bloccherà e lo acquisterà con successo. Ad esempio, il seguente codice *non si* bloccherà se chiamato:

```
public void bar(){
    synchronized(this){
        ...
    }
}
public void foo(){
    synchronized(this){
        bar();
    }
}
```

Accanto ai blocchi synchronized , ci sono anche metodi synchronized .

I seguenti blocchi di codice sono praticamente equivalenti (anche se il bytecode sembra essere diverso):

1. blocco synchronized su this :

```
public void foo() {
    synchronized(this) {
        doStuff();
    }
}
```

2. metodo synchronized :

```
 public synchronized void foo() {
     doStuff();
 }
```

Allo stesso modo per static metodi static , questo:

```
class MyClass {
    ...
    public static void bar() {
        synchronized(MyClass.class) {
            doSomeOtherStuff();
        }
    }
}
```

ha lo stesso effetto di questo:

```
class MyClass {
    ...
    public static synchronized void bar() {
        doSomeOtherStuff();
    }
}
```

Operazioni atomiche

Un'operazione atomica è un'operazione che viene eseguita "tutto in una volta", senza alcuna possibilità che altri thread osservino o modificino lo stato durante l'esecuzione dell'operazione atomica.

Considera un **BAD EXAMPLE** .

```
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();
}
```

In questo caso, ci sono due problemi. Il primo problema è che l'operatore di post-incremento *non* è atomico. Comprende più operazioni: ottieni il valore, aggiungi 1 al valore, imposta il valore. Ecco perché se eseguiamo l'esempio, è probabile che non vedremo t: 100 nell'output: due thread potrebbero contemporaneamente ottenere il valore, incrementarlo e impostarlo: diciamo che il valore di t è 10 e due i thread stanno aumentando t. Entrambi i thread imposteranno il valore da t a 11, poiché il secondo thread osserva il valore di t prima che il primo thread abbia finito di incrementarlo.

Il secondo problema riguarda il modo in cui stiamo osservando t. Quando stampiamo il valore di t, il valore potrebbe essere già stato modificato da un thread diverso dopo l'operazione di incremento di questo thread.

Per risolvere questi problemi, utilizzeremo java.util.concurrent.atomic.AtomicInteger , che ha molte operazioni atomiche da utilizzare.

```
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();
}
```

Il metodo incrementAndGet di AtomicInteger incrementa atomicamente e restituisce il nuovo valore, eliminando così la condizione di gara precedente. Si noti che in questo esempio le linee saranno ancora fuori servizio perché non facciamo nessuno sforzo per sequenziare le chiamate println e questo non rientra nell'ambito di questo esempio, poiché richiederebbe la sincronizzazione e l'obiettivo di questo esempio è mostrare come utilizzare AtomicInteger per eliminare le condizioni di gara relative allo stato.

Creazione di un sistema deadlocked di base

Un deadlock si verifica quando due azioni in competizione aspettano che l'altro finisca, e quindi non lo fa mai. In Java è presente un blocco associato a ciascun oggetto. Per evitare modifiche simultanee eseguite da più thread su un singolo oggetto, possiamo utilizzare synchronized parola chiave synchronized , ma tutto ha un costo. L'uso errato di parole chiave synchronized può portare a sistemi bloccati chiamati sistemi con deadlock.

Considera che ci sono 2 thread che lavorano su 1 istanza, Lets chiama i thread come First e Second, e diciamo che abbiamo 2 risorse R1 e R2. Prima acquisisce R1 e ha bisogno anche di R2 per il suo completamento mentre Second acquisisce R2 e necessita di R1 per il completamento.

quindi dì al tempo t = 0,

Prima ha R1 e Second ha R2. ora First sta aspettando R2 mentre Second è in attesa di R1. questa attesa è indefinita e questo porta a un punto morto.

```
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());
        }
    }
}
```

Uscita di questo programma:

```
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
```

Mettere in pausa l'esecuzione

Thread.sleep fa in modo che il thread corrente sospenda l'esecuzione per un periodo specificato. Questo è un mezzo efficace per rendere il tempo del processore disponibile per gli altri thread di un'applicazione o altre applicazioni che potrebbero essere in esecuzione su un sistema informatico. Ci sono due metodi di sleep sovraccarico nella classe Thread.

Uno che specifica il tempo di sonno al millisecondo

```
public static void sleep(long millis) throws InterruptedException
```

Uno che specifica il tempo di sonno per il nanosecondo

```
public static void sleep(long millis, int nanos)
```

Metti in pausa l'esecuzione per 1 secondo

```
Thread.sleep(1000);
```

È importante notare che questo è un suggerimento per lo scheduler del kernel del sistema operativo. Questo potrebbe non essere necessariamente preciso, e alcune implementazioni non considerano nemmeno il parametro nanosecondo (possibilmente arrotondando al millisecondo più vicino).

Si consiglia di includere una chiamata a Thread.sleep in try / catch e catch InterruptedException .

Visualizzazione di barriere di lettura / scrittura durante l'utilizzo sincronizzato / volatile

Come sappiamo, dovremmo usare synchronized parola chiave synchronized per rendere l'esecuzione di un metodo o blocco esclusivo. Ma pochi di noi potrebbero non essere a conoscenza di un aspetto più importante dell'utilizzo di parole chiave synchronized e volatile : *oltre a rendere atomica un'unità di codice, fornisce anche una barriera di lettura / scrittura* . Cos'è questa barriera di lettura / scrittura? Discutiamo di questo usando un esempio:

```
class Counter {

  private Integer count = 10;

  public synchronized void incrementCount() {
    count++;
  }

  public Integer getCount() {
    return count;
  }
}
```

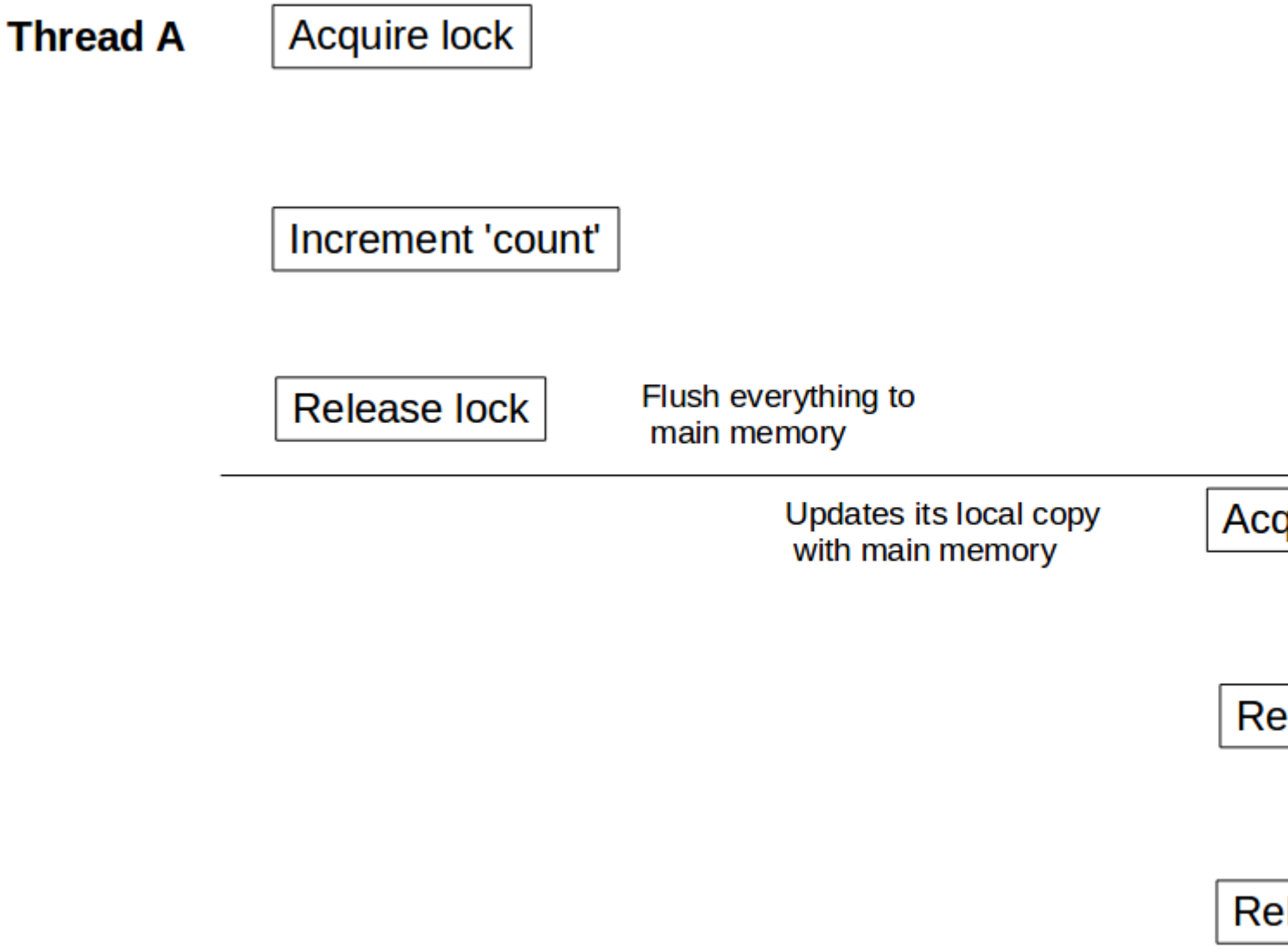
Supponiamo che un thread *A* richiami incrementCount() prima di un altro thread *B* chiama getCount() . In questo scenario non è garantito che B vedrà il valore aggiornato del count . Può ancora vedere count come 10 , anche è anche possibile che non veda mai il valore aggiornato di count mai.

Per comprendere questo comportamento, è necessario capire come il modello di memoria Java si integra con l'architettura hardware. In Java, ogni thread ha il proprio stack di thread. Questo stack contiene: stack di chiamata del metodo e variabile locale creata in quel thread. In un sistema multi core, è del tutto possibile che due thread vengano eseguiti contemporaneamente in core separati. In tale scenario è possibile che parte della pila di un thread si trovi all'interno del registro / cache di un core. Se all'interno di un thread, si accede a un oggetto usando synchronized parola chiave synchronized (o volatile), dopo synchronized blocco synchronized quel thread sincronizza la sua copia locale di quella variabile con la memoria principale. Questo crea una barriera di lettura / scrittura e si assicura che il thread veda l'ultimo valore di quell'oggetto.

Ma nel nostro caso, poiché il thread B non ha usato l'accesso sincronizzato per count , potrebbe essere il valore di riferimento del count memorizzato nel registro e potrebbe non vedere mai gli aggiornamenti dal thread A. Per assicurarsi che B veda l'ultimo valore di conteggio dobbiamo fare getCount() sincronizzato pure.

```
public synchronized Integer getCount() {
  return count;
}
```

Ora, quando il thread A viene eseguito con il count aggiornamento, sblocca l'istanza di Counter , allo stesso tempo crea una barriera di scrittura e svuota tutte le modifiche apportate all'interno di quel blocco alla memoria principale. Allo stesso modo quando il thread B acquisisce il lock sulla stessa istanza di Counter , entra in una barriera di lettura e legge il valore del count dalla memoria principale e vede tutti gli aggiornamenti.



Lo stesso effetto di visibilità vale anche per volatile letture / scritture volatile . Tutte le variabili aggiornate prima della scrittura in volatile verranno scaricate nella memoria principale e tutte le letture dopo la lettura della variabile volatile saranno dalla memoria principale.

Creazione di un'istanza java.lang.Thread

Esistono due approcci principali per la creazione di un thread in Java. In sostanza, creare un thread è facile come scrivere il codice che verrà eseguito in esso. I due approcci differiscono nel punto in cui definisci quel codice.

In Java, un thread è rappresentato da un oggetto: un'istanza di java.lang.Thread o la sua sottoclasse. Quindi il primo approccio è creare quella sottoclasse e sovrascrivere il metodo *run()* .

Nota : userò *Thread* per fare riferimento alla classe e al *thread* java.lang.Thread per fare

riferimento al concetto logico dei thread.

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread running!");
        }
    }
}
```

Da quando abbiamo già definito il codice da eseguire, il thread può essere creato semplicemente come:

```
MyThread t = new MyThread();
```

La classe Thread contiene anche un costruttore che accetta una stringa, che verrà utilizzata come nome del thread. Questo può essere particolarmente utile durante il debug di un programma multi-thread.

```
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");
```

Il secondo approccio è definire il codice usando java.lang.Runnable e il suo unico metodo *run ()* . La classe Thread consente quindi di eseguire tale metodo in un thread separato. Per ottenere ciò, creare il thread utilizzando un costruttore che accetta un'istanza dell'interfaccia Runnable .

```
Thread t = new Thread(aRunnable);
```

Questo può essere molto potente se combinato con riferimenti a lambda o metodi (solo Java 8):

```
Thread t = new Thread(operator::hardWork);
```

Puoi anche specificare il nome del thread.

```
Thread t = new Thread(operator::hardWork, "Pi operator");
```

Praticamente parlando, puoi usare entrambi gli approcci senza preoccupazioni. Tuttavia la saggezza generale dice di usare quest'ultimo.

Per ognuno dei quattro costruttori menzionati, esiste anche un'alternativa che accetta un'istanza di java.lang.ThreadGroup come primo parametro.

```
ThreadGroup tg = new ThreadGroup("Operators");
```

```
Thread t = new Thread(tg, operator::hardWork, "PI operator");
```

Il ThreadGroup rappresenta un set di thread. È possibile solo aggiungere una discussione a un ThreadGroup utilizzando il costruttore di Thread . Il ThreadGroup può quindi essere utilizzato per gestire tutti i suoi Thread insieme, così come il Thread può ottenere informazioni dal suo ThreadGroup .

Quindi per sumarize, il Thread può essere creato con uno di questi costruttori pubblici:

```
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)
```

L'ultimo ci consente di definire le dimensioni dello stack desiderate per il nuovo thread.

Spesso la leggibilità del codice soffre quando si creano e si configurano molti thread con le stesse proprietà o con lo stesso modello. È qui che può essere utilizzato java.util.concurrent.ThreadFactory . Questa interfaccia consente di incapsulare la procedura di creazione del thread attraverso il modello factory e il suo unico metodo *newThread (Runnable)* .

```
class WorkerFactory implements ThreadFactory {
    private int id = 0;

    @Override
    public Thread newThread(Runnable r) {
        return new Thread(r, "Worker " + id++);
    }
}
```

Discussione Interruzione / interruzione di thread

Ogni thread Java ha un flag di interrupt, che inizialmente è falso. L'interruzione di un thread, in sostanza, non è altro che impostare quel flag su true. Il codice in esecuzione su quel thread può controllare la bandiera occasionalmente e agire su di esso. Il codice può anche ignorarlo completamente. Ma perché ogni discussione ha una tale bandiera? Dopotutto, avere una bandiera booleana su un thread è qualcosa che possiamo semplicemente organizzarci, se e quando ne abbiamo bisogno. Bene, ci sono metodi che si comportano in un modo speciale quando il thread su cui stanno girando viene interrotto. Questi metodi sono chiamati metodi di blocco. Questi sono metodi che inseriscono il thread nello stato WAITING o TIMED_WAITING. Quando un thread si trova in questo stato, interrompendolo, si genera un InterruptedException sul thread interrotto, anziché il flag di interrupt impostato su true e il thread diventa nuovamente RUNNABLE. Il codice che richiama un metodo di blocco è costretto ad occuparsi di InterruptedException, poiché si tratta di un'eccezione controllata. Quindi, e quindi il suo nome, un interrupt può avere l'effetto di interrompere un WAIT, chiudendolo efficacemente. Si noti che non tutti i metodi che sono in qualche modo in attesa (ad es. Bloccando IO) rispondono all'interruzione in questo modo, poiché non mettono il thread in uno stato di attesa. Infine, un thread con il proprio flag di interrupt, che immette un metodo di blocco (cioè tenta di entrare in uno stato di attesa), genererà immediatamente un InterruptedException e il flag di interrupt verrà cancellato.

Oltre a questi meccanismi, Java non assegna alcun significato semantico speciale all'interruzione. Il codice è libero di interpretare un'interruzione come preferisce. Ma il più delle volte l'interruzione viene utilizzata per segnalare a un thread che dovrebbe smettere di funzionare al più presto. Ma, come dovrebbe essere chiaro da quanto sopra, è il codice su quel thread a reagire a tale interruzione in modo appropriato al fine di interrompere la corsa. Fermare un thread è una collaborazione. Quando un thread viene interrotto, il codice in esecuzione può essere a diversi livelli in profondità nello stacktrace. La maggior parte del

codice non chiama un metodo di blocco e termina abbastanza tempestivamente per non ritardare indebitamente l'interruzione del thread. Il codice che dovrebbe preoccuparsi principalmente di essere reattivo all'interruzione, è il codice che è in un ciclo che gestisce le attività finché non ne rimane nessuna, o finché non viene impostato un flag che lo segnala per interrompere quel ciclo. I loop che gestiscono attività potenzialmente infinite (ovvero continuano a funzionare in linea di principio) dovrebbero controllare il flag di interrupt per uscire dal ciclo. Per i cicli finiti, la semantica può stabilire che tutte le attività devono essere completate prima di terminare, oppure potrebbe essere opportuno lasciare alcune attività non gestite. Il codice che richiama i metodi di blocco sarà costretto ad occuparsi di InterruptedException. Se tutto è semanticamente possibile, può semplicemente propagare l'InterruptedException e dichiarare di lanciarlo. In quanto tale, diventa esso stesso un metodo di blocco per quanto riguarda i suoi chiamanti. Se non può propagare l'eccezione, dovrebbe almeno impostare il flag interrotto, in modo che i chiamanti più in alto nello stack sappiano anche che il thread è stato interrotto. In alcuni casi, il metodo deve continuare ad attendere indipendentemente da InterruptedException, nel qual caso deve ritardare l'impostazione del flag interrotto fino a quando non viene eseguito in attesa, ciò può comportare l'impostazione di una variabile locale, che deve essere controllata prima di uscire dal metodo quindi interrompi il suo thread.

Esempi:

Esempio di codice che interrompe la gestione delle attività in caso di interruzione

```
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
    }
}
```

Esempio di codice che ritarda l'impostazione del flag di interrupt fino al completamento completo:

```
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
    }
}
```

Esempio di codice che ha un elenco fisso di attività ma potrebbe chiudersi presto quando viene interrotto

```
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
    }
}
```

Esempio di produttore / consumatore multiplo con coda globale condivisa

Sotto il codice vengono presentati più programmi Producer / Consumer. Entrambi i thread Producer e Consumer condividono la stessa coda globale.

```
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();
            }
        }
    }
}
```

produzione:

```
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
```

e così via .....

Spiegazione:

1. sharedQueue , che è LinkedBlockingQueue è condiviso tra tutti i thread Producer e Consumer.
2. I thread di produzione producono continuamente un intero per ogni 200 milli secondi e lo sharedQueue a sharedQueue
3. Consumer thread Consumer consuma continuamente integer da sharedQueue .
4. Questo programma è implementato senza costrutti espliciti synchronized o Lock . BlockingQueue è la chiave per raggiungerlo.

   Le implementazioni BlockingQueue sono progettate per essere utilizzate principalmente per code produttore-consumatore.

   Le implementazioni di BlockingQueue sono thread-safe. Tutti i metodi di accodamento ottengono i loro effetti atomicamente utilizzando blocchi interni o altre forme di controllo della concorrenza.

Scrittura esclusiva / accesso di lettura simultaneo

Talvolta è necessario che un processo scriva e legga contemporaneamente gli stessi "dati".

L'interfaccia ReadWriteLock e l'implementazione ReentrantReadWriteLock consentono un pattern di accesso che può essere descritto come segue:

1. Ci può essere un numero qualsiasi di lettori concorrenti dei dati. Se è concesso almeno un accesso al lettore, non è possibile l'accesso allo scrittore.
2. Ci può essere al massimo un singolo writer per i dati. Se è concesso l'accesso allo scrittore, nessun lettore può accedere ai dati.

Un'implementazione potrebbe essere simile a:

```
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 : Questo caso d'uso preciso ha una soluzione più pulita usando AtomicInteger , ma ciò che è descritto qui è un modello di accesso, che funziona indipendentemente dal fatto che i dati qui siano un numero intero come una variante Atomica.

NOTA 2 : Il blocco sulla parte di lettura è davvero necessario, anche se potrebbe non sembrare così per il lettore casuale. Infatti, se non si blocca sul lato del lettore, qualsiasi numero di cose può andare storto, tra cui:

1. Le scritture dei valori primitivi non sono garantite per essere atomiche su tutte le JVM, quindi il lettore potrebbe vedere ad esempio solo 32 bit di una scrittura a 64 bit se i data erano di tipo a 64 bit
2. La visibilità della scrittura da un thread che non l'ha eseguita è garantita dalla JVM solo se stabiliamo la *relazione Happen Before* tra le scritture e le letture. Questa relazione viene stabilita quando sia i lettori che gli scrittori utilizzano i rispettivi blocchi, ma non altrimenti

Java SE 8

Nel caso in cui siano richieste prestazioni più elevate, sotto certi tipi di utilizzo, è disponibile un tipo di blocco più rapido, denominato StampedLock , che, tra le altre cose, implementa una modalità di blocco ottimistica. Questo blocco funziona in modo molto diverso da ReadWriteLock e questo esempio non è trasportabile.

Oggetto eseguibile

L'interfaccia Runnable definisce un singolo metodo, run() , pensato per contenere il codice eseguito nel thread.

L'oggetto Runnable viene passato al costruttore Thread . E viene chiamato il metodo start() di Thread.

Esempio

```
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();
    }
}
```

Esempio in Java8:

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

```
    Runnable r = () -> System.out.println("Hello world");
    new Thread(r).start();
}
```

Sottoclasse Runnable vs Thread

Un impiego di oggetti Runnable è più generale, perché l'oggetto Runnable può creare una sottoclasse di una classe diversa da Thread .

Thread sottoclasse di Thread è più semplice da utilizzare nelle applicazioni semplici, ma è limitata dal fatto che la classe di attività deve essere un discendente di Thread .

Un oggetto Runnable è applicabile alle API di gestione del thread di alto livello.

Semaforo

Un semaforo è un sincronizzatore di alto livello che mantiene un insieme di *permessi* che possono essere acquisiti e rilasciati dai thread. Un semaforo può essere immaginato come un contatore di *permessi* che verrà decrementato quando un thread acquisisce e incrementato quando un thread viene rilasciato. Se la quantità di *permessi* è 0 quando un thread tenta di acquisire, il thread si bloccherà fino a quando non verrà reso disponibile un permesso (o finché il thread non viene interrotto).

Un semaforo è inizializzato come:

```
Semaphore semaphore = new Semaphore(1); // The int value being the number of permits
```

Il costruttore del semaforo accetta un parametro booleano aggiuntivo per l'equità. Se impostato su false, questa classe non fornisce garanzie sull'ordine in cui i thread acquisiscono i permessi. Quando l'equità è impostata su true, il semaforo garantisce che i thread che invocano uno dei metodi di acquisizione siano selezionati per ottenere i permessi nell'ordine in cui è stata elaborata la loro chiamata di tali metodi. È dichiarato nel modo seguente:

```
Semaphore semaphore = new Semaphore(1, true);
```

Ora diamo un'occhiata a un esempio di javadocs, in cui il semaforo viene utilizzato per controllare l'accesso a un gruppo di elementi. In questo esempio viene utilizzato un semaforo per fornire funzionalità di blocco al fine di garantire che ci siano sempre elementi da ottenere quando viene chiamato 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
    }
}
```

Aggiungi due array `int` usando un Threadpool

Un Threadpool ha una coda di compiti, di cui ognuno verrà eseguito su uno di questi Thread.

L'esempio seguente mostra come aggiungere due array int usando 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));
```

Gli appunti:

1. Questo esempio è puramente illustrativo. In pratica, non ci sarà alcuna accelerazione utilizzando i thread per un'attività così piccola. È probabile un rallentamento, dal momento che i costi generali di creazione e programmazione dell'attività acquisteranno il tempo necessario per eseguire un'attività.
2. Se stavi usando Java 7 e versioni precedenti, dovresti utilizzare le classi anonime invece di lambda per implementare le attività.

Ottieni lo stato di tutti i thread avviati dal tuo programma escludendo i thread di sistema

Snippet di codice:

```
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();
        }
    }
}
```

Produzione:

```
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
```

Spiegazione:

Thread.getAllStackTraces().keySet() restituisce tutti i Thread , inclusi i thread di applicazione e i thread di sistema. Se sei interessato solo allo stato di Thread, avviato dalla tua applicazione, iterare il Thread set controllando il Thread Group di un particolare thread con il thread del tuo programma principale.

In assenza di una condizione ThreadGroup superiore, il programma restituisce lo stato di sotto Thread di sistema:

```
Reference Handler
Signal Dispatcher
Attach Listener
Finalizer
```

Callable e Future

Mentre Runnable fornisce un mezzo per avvolgere il codice da eseguire in un thread diverso, ha una limitazione nel senso che non può restituire un risultato dell'esecuzione. L'unico modo per ottenere un valore di ritorno dall'esecuzione di un Runnable è assegnare il risultato a una variabile accessibile in un ambito esterno a Runnable .

Callable stato introdotto in Java 5 come peer to Runnable . Callable è essenzialmente lo stesso eccetto che ha un metodo di call invece di run . Il metodo di call ha la capacità aggiuntiva di restituire un risultato ed è inoltre autorizzato a generare eccezioni controllate.

Il risultato di una *richiesta di attività Callable* è disponibile per essere sfruttato tramite un futuro

Future può essere considerato un contenitore di sorta che ospita il risultato del calcolo Callable . Il calcolo del callable può continuare in un altro thread, e qualsiasi tentativo di toccare il risultato di un Future bloccherà e restituirà il risultato solo quando sarà disponibile.

Interfaccia richiamabile

```
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 l'esempio Callable e Future:

```
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());
}
```

Il nostro Callable che esegue un lungo calcolo

```
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
    }
}
```

Produzione

```
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
```

Altre operazioni permesse su Future

Mentre get() è il metodo per estrarre il risultato reale, il futuro ha la possibilità

- get(long timeout, TimeUnit unit) definisce il periodo di tempo massimo durante il thread corrente aspetterà un risultato;
- Per annullare l'operazione, annullare la chiamata cancel(mayInterruptIfRunning) . Il flag mayInterrupt indica che l'attività deve essere interrotta se è stata avviata ed è in esecuzione in questo momento;
- Per verificare se l'attività è completata / terminata chiamando isDone() ;
- Per verificare se la lunga attività è stata annullata isCancelled() .

Blocchi come aiuti di sincronizzazione

Prima dell'introduzione simultanea del pacchetto di Java 5 il threading era di livello più basso. L'introduzione di questo pacchetto forniva numerosi ausili / costrutti di programmazione simultanei di livello superiore.

I lock sono meccanismi di sincronizzazione dei thread che essenzialmente hanno lo stesso scopo dei blocchi sincronizzati o delle parole chiave.

Blocco intrinseco

```
int count = 0; // shared among multiple threads

public void doSomething() {
    synchronized(this) {
        ++count; // a non-atomic operation
    }
}
```

Sincronizzazione tramite blocchi

```
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
    }
}
```

I blocchi hanno anche funzionalità disponibili che il blocco intrinseco non offre, come il blocco ma che rimangono reattivi all'interruzione o che cercano di bloccare e non bloccano quando non sono in grado di farlo.

Chiusura, sensibile all'interruzione

```
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
            }
        }
    }
}
```

Fai qualcosa solo quando puoi bloccare

```
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
            }
        }
    }
}
```

Sono disponibili diverse varianti di blocco. Per ulteriori dettagli, consultare i documenti API qui

Leggi Programmazione simultanea (thread) online:
https://riptutorial.com/it/java/topic/121/programmazione-simultanea--thread-

Capitolo 142: Proprietà Classe

introduzione

L'oggetto properties contiene coppie di chiavi e valori sia come stringa. La classe java.util.Properties è la sottoclasse di Hashtable.

Può essere utilizzato per ottenere il valore della proprietà in base alla chiave della proprietà. La classe Proprietà fornisce metodi per ottenere dati dal file delle proprietà e memorizzare i dati nel file delle proprietà. Inoltre, può essere utilizzato per ottenere le proprietà del sistema.

Vantaggio del file delle proprietà

La ricompilazione non è richiesta, se le informazioni vengono modificate dal file delle proprietà: Se vengono modificate le informazioni da

Sintassi

- In un file di proprietà:
- chiave = valore
- #commento

Osservazioni

Un oggetto Properties è una mappa le cui chiavi e valori sono stringhe per convenzione. Sebbene i metodi di Map possano essere utilizzati per accedere ai dati, i metodi type-safe getProperty , setProperty e stringPropertyNames vengono generalmente utilizzati.

Le proprietà sono spesso archiviate in file di proprietà Java, che sono semplici file di testo. Il loro formato è documentato accuratamente nel metodo Properties.load . In sintesi:

- Ogni coppia chiave / valore è una riga di testo con uno spazio, uguale (=), o due punti (:) tra la chiave e il valore. Gli uguali o i due punti possono avere qualsiasi quantità di spazi bianchi prima e dopo di essa, che viene ignorata.
- Lo spazio bianco principale viene sempre ignorato, lo spazio bianco finale è sempre incluso.
- Una barra rovesciata può essere utilizzata per sfuggire a qualsiasi carattere (eccetto in minuscolo u).
- Una barra rovesciata alla fine della riga indica che la riga successiva è una continuazione della linea corrente. Tuttavia, come per tutte le linee, gli spazi bianchi iniziali nella linea di continuazione vengono ignorati.
- Proprio come nel codice sorgente di Java, \u seguita da quattro cifre esadecimali rappresenta un carattere UTF-16.

La maggior parte dei framework, incluse le strutture di Java SE come java.util.ResourceBundle, carica i file delle proprietà come InputStreams. Quando si carica un file di proprietà da un InputStream, quel file può contenere solo caratteri ISO 8859-1 (ovvero caratteri nell'intervallo 0-255). Qualsiasi altro personaggio deve essere rappresentato come \u escape. Tuttavia, puoi scrivere un file di testo in qualsiasi codifica e utilizzare lo strumento native2ascii (che viene fornito con ogni JDK) per farlo scappare per te.

Se si sta caricando un file di proprietà con il proprio codice, può essere in qualsiasi codifica, purché si crei un Reader (come un InputStreamReader) basato sul set di caratteri corrispondente. È quindi possibile caricare il file utilizzando load (Reader) anziché il metodo di caricamento legacy (InputStream).

È anche possibile memorizzare le proprietà in un semplice file XML, che consente al file stesso di definire la codifica. Tale file può essere caricato con il metodo loadFromXML . La DTD che descrive la struttura di tali file XML si trova all'indirizzo http://java.sun.com/dtd/properties.dtd .

Examples

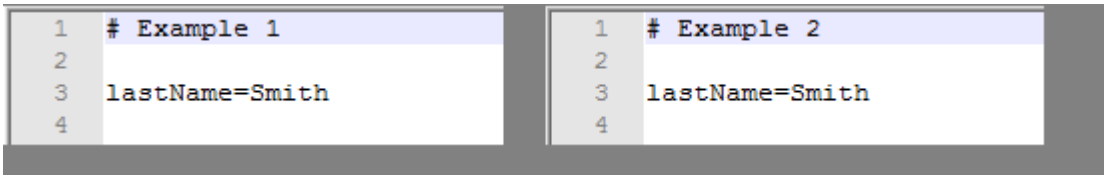
Caricamento delle proprietà

Per caricare un file di proprietà in bundle con la tua applicazione:

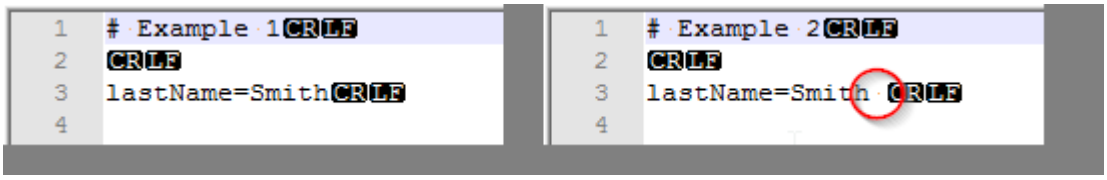
```
public class Defaults {

    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);
        }
    }

}
```

Avvertenza sui file di proprietà: spazi bianchi finali

Dai un'occhiata da vicino a questi due file di proprietà che sono apparentemente completamente identici:



tranne che non sono davvero identici:



(gli screenshot provengono da Notepad ++)

Poiché lo spazio bianco finale è conservato, il valore di lastName sarebbe "Smith" nel primo caso e "Smith " nel secondo caso.

Molto raramente questo è ciò che gli utenti si aspettano e uno e può solo speculare perché questo è il comportamento predefinito della classe Properties . È tuttavia facile creare una versione avanzata di Properties che corregge questo problema. La seguente classe, **TrimmedProperties** , fa proprio questo. È una sostituzione drop-in per la classe di proprietà standard.

```
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;
    }
}
```

Salvataggio delle proprietà come XML

Memorizzazione delle proprietà in un file XML

Il modo in cui si archiviano i file delle proprietà come file XML è molto simile al modo in cui li si archivia come file .properties . Semplicemente invece di usare lo 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");
}
```

Quando apri il file, sarà simile a questo.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>testing properties with xml</comment>
<entry key="age">23</entry>
<entry key="color">green</entry>
<entry key="name">Steve</entry>
</properties>
```

Caricamento delle proprietà da un file XML

Ora per caricare questo file come properties è necessario chiamare il loadFromXML() posto di load() che si utilizzerà con i file regolari .propeties .

```
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);
    }
}
```

Quando si esegue questo codice, nella console verrà visualizzato quanto segue:

```
age=23
color=green
name=Steve
```

Leggi Proprietà Classe online: https://riptutorial.com/it/java/topic/576/proprieta-classe

Capitolo 143: Registrazione (java.util.logging)

Examples

Utilizzo del logger predefinito

Questo esempio mostra come utilizzare l'API di registrazione predefinita.

```
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()});
    }

}
```

Livelli di registrazione

Java Logging Api ha 7 livelli . I livelli in ordine decrescente sono:

- SEVERE (valore più alto)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (valore più basso)

Il livello predefinito è INFO (ma dipende dal sistema e utilizza una macchina virtuale).

Nota : ci sono anche dei livelli OFF (può essere usato per disattivare la registrazione) e ALL (la prospettiva di OFF).

Esempio di codice per questo:

```
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");
    }
}
```

Per impostazione predefinita, in esecuzione questa classe verranno visualizzati solo i messaggi con livello superiore a 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
```

Registrazione di messaggi complessi (in modo efficiente)

Diamo un'occhiata ad un campione di registrazione che puoi vedere in molti programmi:

```
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
}
```

L'esempio sopra sembra perfettamente a posto, ma molti programmatori dimenticano che Java VM è uno stack machine. Ciò significa che tutti i parametri del metodo vengono calcolati **prima** dell'esecuzione del metodo.

Questo fatto è cruciale per la registrazione in Java, specialmente per la registrazione di

qualcosa in livelli bassi come FINE , FINER , FINEST che sono disabilitati di default. Diamo un'occhiata al bytecode Java per il metodo takeOrder() .

Il risultato per javap -c LoggingComplex.class è qualcosa del genere:

```
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 riga 39 esegue la registrazione effettiva. Tutto il lavoro precedente (caricamento di variabili, creazione di nuovi oggetti, concatenazione di stringhe nel metodo di format) può essere inutile se il livello di registrazione è impostato su un valore superiore a FINE (e di default è). Tale registrazione può essere molto inefficiente, consumando risorse di memoria e processore non necessarie.

Ecco perché dovresti chiedere se il livello che vuoi utilizzare è abilitato.

La strada giusta dovrebbe essere:

```
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
}
```

Da quando Java 8:

La classe Logger ha metodi aggiuntivi che prendono come parametro un parametro Supplier<String> , che può essere semplicemente fornito da 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
}
```

Il metodo get() fornitori - in questo caso il lambda - viene chiamato solo quando il livello corrispondente è abilitato e quindi la costruzione if non è più necessaria.

Leggi Registrazione (java.util.logging) online:
https://riptutorial.com/it/java/topic/2010/registrazione--java-util-logging-

Capitolo 144: Remote Method Invocation (RMI)

Osservazioni

RMI richiede 3 componenti: client, server e un'interfaccia remota condivisa. L'interfaccia remota condivisa definisce il contratto client-server specificando i metodi che un server deve implementare. L'interfaccia deve essere visibile al server in modo che possa implementare i metodi; l'interfaccia deve essere visibile al client in modo che conosca i metodi ("servizi") forniti dal server.
Qualsiasi oggetto che implementa un'interfaccia remota è destinato a svolgere il ruolo di server. Come tale, una relazione client-server in cui il server può anche invocare metodi nel client è in realtà una relazione server-server. Questo è chiamato *callback* poiché il server può richiamare il "client". Con questo in mente, è accettabile utilizzare il *client di* designazione per i server che funzionano come tali.

L'interfaccia remota condivisa è un'interfaccia che estende Remote . Un oggetto che funziona come server subisce quanto segue:

1. Implementa l'interfaccia remota condivisa, esplicitamente o implicitamente, estendendo UnicastRemoteObject che implementa Remote .
2. Esportato, implicitamente se estende UnicastRemoteObject o esplicitamente passando a UnicastRemoteObject#exportObject .
3. Associato in un registro, direttamente tramite il Registry o indirettamente tramite Naming . Questo è necessario solo per stabilire una comunicazione iniziale poiché ulteriori stub possono essere passati direttamente tramite RMI.

Nell'impostazione del progetto, i progetti client e server non sono completamente correlati, ma ognuno specifica un progetto condiviso nel suo percorso di generazione. Il progetto condiviso contiene le interfacce remote.

Examples

Client-Server: richiamo di metodi in una JVM da un'altra

L'interfaccia remota condivisa:

```
package remote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteServer extends Remote {

    int stringToInt(String string) throws RemoteException;
}
```

Il server che implementa l'interfaccia remota condivisa:

```
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();
        }
    }
}
```

Il client che richiama un metodo sul server (in remoto):

```
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();
        }
    }
}
```

Produzione:

```
 Server ricevuto: "120"
```

```
Cliente ricevuto: 120
```

Callback: invocazione di metodi su un "client"

Panoramica

In questo esempio 2 client inviano informazioni reciprocamente attraverso un server. Un client invia al server un numero che viene inoltrato al secondo client. Il secondo client dimezza il numero e lo invia al primo client attraverso il server. Il primo cliente fa lo stesso. Il server interrompe la comunicazione quando il numero restituito da uno dei client è inferiore a 10. Il valore restituito dal server ai client (il numero convertito in rappresentazione stringa) quindi retrocede il processo.

1. Un server di login si lega a un registro.
2. Un client cerca il server di login e chiama il metodo di login con le sue informazioni. Poi:
    - Il server di login memorizza le informazioni del cliente. Include lo stub del cliente con i metodi di callback.
    - Il server di login crea e restituisce un server stub ("connessione" o "sessione") al client da memorizzare. Include lo stub del server con i suoi metodi incluso un metodo di logout (non utilizzato in questo esempio).
3. Un client chiama il passInt del server con il nome del client del destinatario e un int .
4. Il server chiama la half sul client destinatario con quella int . Ciò avvia una comunicazione avanti e indietro (chiamate e callback) finché non viene arrestata dal server.

Le interfacce remote condivise

Il server di login:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteLogin extends Remote {

    RemoteConnection login(String name, RemoteClient client) throws RemoteException;
}
```

Il server:

```
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;
}
```

Il cliente:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;
```

```
public interface RemoteClient extends Remote {

    void half(int i) throws RemoteException;
}
```

Le implementazioni

Il server di login:

```
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();
        }
    }
}
```

Il server:

```
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);
    }
}
```

Il 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 + "\"");
    }
}
```

Esecuzione dell'esempio:

1. Esegui il server di login.
2. Esegui un client con gli argomenti Client2 Client1 1097 .
3. Esegui un client con gli argomenti Client1 Client2 1098 .

Le uscite appariranno su 3 console poiché ci sono 3 JVM. eccoli raggruppati insieme:

```
Client2 registrato
Client1 loggato
Server ricevuto da Client1: 120
Server ricevuto da Client2: 60
Server ricevuto da Client1: 30
Server ricevuto da Client2: 15
Server ricevuto da Client1: 7
```

```
Client1 ricevuto: "7"
Client2 ricevuto: "15"
Client1 ricevuto: "30"
Client2 ricevuto: "60"
```

Esempio RMI semplice con implementazione client e server

Questo è un semplice esempio RMI con cinque classi Java e due pacchetti, *server* e *client* .

Pacchetto server

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;
    }
}
```

Server.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();
    }
}
```

Pacchetto del 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);
        }
    }
}
```

Metti alla prova la tua domanda

- Avvia il metodo principale di Server.java. Produzione:

```
Server : Registry created.
Server : PersonList registered
```

- Avvia il metodo principale di PersonTest.java. Produzione:

```
Peter Pan
Pippi Langstrumpf
```

Leggi Remote Method Invocation (RMI) online: https://riptutorial.com/it/java/topic/171/remote-method-invocation--rmi-

Capitolo 145: ricorsione

introduzione

La ricorsione si verifica quando un metodo chiama se stesso. Un tale metodo è chiamato **ricorsivo** . Un metodo ricorsivo può essere più conciso di un approccio non ricorsivo equivalente. Tuttavia, per una ricorsione profonda, a volte una soluzione iterativa può consumare meno spazio di stack finito di un thread.

Questo argomento include esempi di ricorsione in Java.

Osservazioni

Progettare un metodo ricorsivo

Quando si progetta un metodo ricorsivo, tenere presente che è necessario:

- **Caso base.** Questo definirà quando la tua ricorsione si fermerà e produrrà il risultato. Il caso base nell'esempio fattoriale è:

```
if (n <= 1) {
    return 1;
}
```

- **Chiamata ricorsiva.** In questa affermazione si richiama il metodo con un parametro modificato. La chiamata ricorsiva nell'esempio fattoriale sopra è:

```
else {
    return n * factorial(n - 1);
}
```

Produzione

In questo esempio si calcola il numero fattoriale n-esimo. I primi fattoriali sono:

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

...

Eliminazione di Java e Tail-call

I compilatori Java attuali (fino a Java 9 incluso) non eseguono l'eliminazione di tail-call. Questo può influire sulle prestazioni degli algoritmi ricorsivi e, se la ricorsione è abbastanza profonda, può portare a crash di StackOverflowError ; vedere La ricorsione profonda è problematica in Java

Examples

L'idea di base della ricorsione

Cos'è la ricorsione:

In generale, la ricorsione è quando una funzione invoca se stessa, direttamente o indirettamente. Per esempio:

```
// This method calls itself "infinitely"
public void useless() {
    useless();  // method calls itself (directly)
}
```

Condizioni per l'applicazione della ricorsione a un problema:

Esistono due presupposti per l'utilizzo di funzioni ricorsive per risolvere un problema specifico:

1. Ci deve essere una condizione di base per il problema, che sarà l'endpoint per la ricorsione. Quando una funzione ricorsiva raggiunge la condizione di base, non effettua ulteriori chiamate (più profonde) ricorsive.
2. Ogni livello di ricorsione dovrebbe tentare un problema più piccolo. La funzione ricorsiva divide così il problema in parti sempre più piccole. Supponendo che il problema sia finito, ciò garantirà la terminazione della ricorsione.

In Java c'è una terza precondizione: non dovrebbe essere necessario recidare troppo profondamente per risolvere il problema; vedere La ricorsione profonda è problematica in Java

Esempio

La seguente funzione calcola i fattoriali usando la ricorsione. Si noti come il metodo factorial chiama all'interno della funzione. Ogni volta che chiama se stesso, riduce il parametro n di 1. Quando n raggiunge 1 (condizione di base), la funzione non verrà eseguita più in profondità.

```
public int factorial(int n) {
    if (n <= 1) { // the base condition
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Questo non è un modo pratico di calcolare i fattoriali in Java, dal momento che non prende in considerazione l'overflow dei numeri interi o l'overflow dello stack delle chiamate (ad esempio le eccezioni StackOverflowError) per i grandi valori di n .

Calcolo del numero dell'N ° Fibonacci

Il seguente metodo calcola il numero Nth Fibonacci usando la ricorsione.

```
public int fib(final int n) {
    if (n > 2) {
        return fib(n - 2) + fib(n - 1);
    }
    return 1;
}
```

Il metodo implementa un caso base (n <= 2) e un caso ricorsivo (n> 2). Questo illustra l'uso della ricorsione per calcolare una relazione ricorsiva.

Tuttavia, sebbene questo esempio sia illustrativo, è anche inefficiente: ogni singola istanza del metodo chiamerà la funzione stessa due volte, portando ad una crescita esponenziale nel numero di volte in cui la funzione viene chiamata con l'aumentare di N. La funzione precedente è O (2^N), ma una soluzione iterativa equivalente ha complessità O (N). Inoltre, esiste un'espressione "forma chiusa" che può essere valutata nelle moltiplicazioni in virgola mobile O (N).

Calcolo della somma di numeri interi da 1 a N

Il seguente metodo calcola la somma di interi da 0 a N usando la ricorsione.

```
public int sum(final int n) {
    if (n > 0) {
        return n + sum(n - 1);
    } else {
        return n;
    }
}
```

Questo metodo è O (N) e può essere ridotto a un ciclo semplice mediante l'ottimizzazione della coda di chiamata. Infatti esiste un'espressione di *forma chiusa* che calcola la somma in operazioni O(1) .

Calcolare l'ennesima potenza di un numero

Il seguente metodo calcola il valore di num elevato alla potenza di exp utilizzando la ricorsione:

```
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);
}
```

Questo illustra i principi sopra menzionati: il metodo ricorsivo implementa un caso base (due casi, n = 0 e n = 1) che termina la ricorsione e un caso ricorsivo che chiama di nuovo il metodo. Questo metodo è O (N) e può essere ridotto a un ciclo semplice mediante l'ottimizzazione della coda di chiamata.

Invertire una stringa usando Ricorsione

Di seguito è riportato un codice ricorsivo per invertire una stringa

```
/**
 * 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);
    }
}
```

Attraversare una struttura dati Albero con ricorsione

Considera la classe Node con 3 dati membri, il puntatore figlio sinistro e il puntatore figlio destro come sotto.

```
public class Node {
    public int data;
    public Node left;
    public Node right;

    public Node(int data){
        this.data = data;
    }
}
```

Possiamo attraversare l'albero costruito collegando più oggetti della classe Node come di seguito, l'attraversamento è chiamato attraversamento in ordine dell'albero.

```
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
        }
    }
```

Come dimostrato sopra, usando la **ricorsione** possiamo attraversare la **struttura dei dati dell'albero** senza utilizzare altre strutture dati che non sono possibili con l'approccio **iterativo** .

Tipi di ricorsione

La ricorsione può essere classificata come **ricorsione della testa** o **ricorsione della coda** , a seconda di dove viene collocata la chiamata al metodo ricorsivo.

Nella **ricorsione della testa** , la chiamata ricorsiva, quando accade, viene prima di un'altra elaborazione nella funzione (pensate che avvenga in alto, o in testa, della funzione).

Nella **ricorsione di coda** , è l'opposto: l'elaborazione avviene prima della chiamata ricorsiva. Scegliere tra i due stili ricorsivi può sembrare arbitrario, ma la scelta può fare la differenza.

Una funzione con un percorso con una singola chiamata ricorsiva all'inizio del percorso utilizza quella che viene chiamata ricorsione della testa. La funzione fattoriale di una mostra precedente utilizza la ricorsione della testa. La prima cosa che fa una volta che determina che la ricorsione è necessaria è chiamarsi con il parametro decrementato. Una funzione con una singola chiamata ricorsiva alla fine di un percorso utilizza la ricorsione della coda.

```
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);
    }
```

Se la chiamata ricorsiva si verifica alla fine di un metodo, viene chiamata tail recursion . La ricorsione della coda è similar to a loop . Il method executes all the statements before jumping

into the next recursive call .

Se la chiamata ricorsiva si verifica beginning of a method, it is called a head recursion . Il method saves the state before jumping into the next recursive call .

Riferimento: la differenza tra ricorsione testa e coda

StackOverflowError e ricorsione in loop

Se una chiamata ricorsiva diventa "troppo profonda", ciò provoca un StackOverflowError . Java assegna un nuovo frame per ogni chiamata di metodo sullo stack del suo thread. Tuttavia, lo spazio della pila di ciascun thread è limitato. Troppi frame sullo stack portano allo Stack Overflow (SO).

Esempio

```
public static void recursion(int depth) {
    if (depth > 0) {
        recursion(depth-1);
    }
}
```

Chiamare questo metodo con parametri di grandi dimensioni (es. recursion(50000) probabilmente si tradurrà in un overflow dello stack. Il valore esatto dipende dalla dimensione dello stack di thread, che a sua volta dipende dalla costruzione del thread, dai parametri della riga di comando come -Xss o dal dimensione predefinita per JVM.

Soluzione

Una ricorsione può essere convertita in un ciclo memorizzando i dati per ogni chiamata ricorsiva in una struttura dati. Questa struttura di dati può essere archiviata nell'heap anziché nella pila di thread.

In generale, i dati necessari per ripristinare lo stato di un'invocazione di metodo possono essere memorizzati in uno stack e un ciclo while può essere utilizzato per "simulare" le chiamate ricorsive. I dati che possono essere richiesti includono:

- l'oggetto per cui è stato chiamato il metodo (solo i metodi di istanza)
- i parametri del metodo
- variabili locali
- la posizione corrente nell'esecuzione o il metodo

Esempio

La seguente classe consente la ricorsività di una struttura ad albero che stampa fino ad una profondità specificata.

```
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(")");
        }
    }

}
```

per esempio

```
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();
```

stampe

```
(((...)20(...))10((...)30))
```

Questo potrebbe essere convertito nel seguente ciclo:

```
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: questo è solo un esempio dell'approccio generale. Spesso puoi trovare un modo molto migliore per rappresentare un frame e / o memorizzare i dati del frame.

La ricorsione profonda è problematica in Java

Considera il seguente metodo ingenuo per aggiungere due numeri positivi usando la ricorsione:

```
public static int add(int a, int b) {
    if (a == 0) {
        return b;
    } else {
        return add(a - 1, b + 1);  // TAIL CALL
    }
}
```

Questo è algoritmicamente corretto, ma ha un grosso problema. Se si chiama add a large a , si bloccherà con StackOverflowError , su qualsiasi versione di Java fino a (almeno) Java 9.

In un tipico linguaggio di programmazione funzionale (e in molti altri linguaggi) il compilatore ottimizza la ricorsione della coda . Il compilatore noterebbe che la chiamata da add (sulla linea taggata) è una chiamata di coda e riscriverebbe efficacemente la ricorsione come un ciclo. Questa trasformazione è chiamata eliminazione di coda.

Tuttavia, i compilatori Java di generazione corrente non eseguono l'eliminazione delle chiamate tail. (Questa non è una semplice svista, ci sono sostanziali motivi tecnici per questo, vedi sotto). Invece, ogni chiamata ricorsiva di add fa sì che un nuovo frame venga allocato nello stack del thread. Ad esempio, se si chiama add(1000, 1) , occorreranno 1000 chiamate ricorsive per arrivare alla risposta 1001 .

Il problema è che la dimensione dello stack di thread Java è fissa quando viene creato il thread. (Ciò include il thread "principale" in un programma a thread singolo.) Se vengono allocati troppi frame di stack, lo stack andrà in overflow. La JVM lo rileva e lancia un StackOverflowError .

Un approccio per affrontare questo è semplicemente utilizzare uno stack più grande. Esistono opzioni JVM che controllano la dimensione predefinita di uno stack e puoi anche specificare la dimensione dello stack come parametro del costruttore di Thread . Sfortunatamente, questo "mette fuori" solo l'overflow dello stack. Se è necessario eseguire un calcolo che richiede uno stack ancora più grande, viene restituito StackOverflowError .

La vera soluzione consiste nell'identificare algoritmi ricorsivi in cui è probabile una ricorsione profonda e eseguire *manualmente* l'ottimizzazione della coda di chiamata a livello di codice sorgente. Ad esempio, il nostro metodo add può essere riscritto come segue:

```
public static int add(int a, int b) {
    while (a != 0) {
       a = a - 1;
       b = b + 1;
    }
    return b;
}
```

(Ovviamente, ci sono modi migliori per aggiungere due numeri interi. Quanto sopra è semplicemente per illustrare l'effetto dell'eliminazione manuale della coda di chiamata.)

Perché l'eliminazione tail-call non è implementata in Java (ancora)

Ci sono una serie di motivi per cui l'aggiunta dell'eliminazione delle chiamate tail a Java non è facile. Per esempio:

- Alcuni codici potrebbero fare affidamento su StackOverflowError per (ad esempio) posizionare un limite sulla dimensione di un problema computazionale.
- I responsabili della sicurezza di Sandbox si affidano spesso all'analisi dello stack di chiamate quando decidono se consentire al codice non privilegiato di eseguire un'azione privilegiata.

Come spiega John Rose in "Tail calls in the VM" :

> *"Gli effetti della rimozione del frame dello stack del chiamante sono visibili per alcune API, in particolare per i controlli di controllo degli accessi e per la traccia dello stack, come se il chiamante avesse chiamato direttamente il callee.Tutti i privilegi posseduti dal chiamante vengono eliminati dopo che il controllo è stato trasferito al tuttavia, il collegamento e l'accessibilità del metodo del callee sono calcolati prima del trasferimento del controllo e tengono conto del chiamante di coda. "*

In altre parole, l'eliminazione di chiamata di coda potrebbe far sì che un metodo di controllo degli accessi ritenga erroneamente che un'API sensibile alla sicurezza sia stata chiamata da codice attendibile.

Leggi ricorsione online: https://riptutorial.com/it/java/topic/914/ricorsione

Capitolo 146: Riferimenti dell'oggetto

Osservazioni

Questo dovrebbe aiutare a capire una "Eccezione puntatore nullo" - si ottiene uno di questi perché un riferimento all'oggetto è nullo, ma il codice del programma si aspetta che il programma utilizzi qualcosa in quel riferimento oggetto. Tuttavia, questo merita il proprio argomento ...

Examples

Riferimenti dell'oggetto come parametri del metodo

Questo argomento spiega il concetto di *riferimento a* un *oggetto* ; è rivolto a chi è nuovo alla programmazione in Java. Dovresti già avere familiarità con alcuni termini e significati: definizione della classe, metodo principale, istanza dell'oggetto e il richiamo dei metodi "su" un oggetto e passaggio dei parametri ai metodi.

```
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;
  }
}
```

Per essere completamente competente nella programmazione Java, dovresti essere in grado di spiegare questo esempio a qualcun altro in cima alla tua testa. I suoi concetti sono fondamentali per capire come funziona Java.

Come puoi vedere, abbiamo un main che istanzia un oggetto alla person variabile e chiama un metodo per impostare il campo del name in quell'oggetto su "Bob" . Quindi chiama un altro metodo e passa la person come uno dei due parametri; l'altro parametro è una variabile intera, impostata su 5.

Il metodo chiamato imposta il valore del name sull'oggetto passato su "Linda" e imposta la variabile intera passata a 99, quindi restituisce.

Quindi cosa verrebbe stampato?

```
Linda 5
```

Quindi, perché la modifica apportata alla person effetto main , ma la modifica apportata all'intero non lo è?

Quando viene effettuata la chiamata, il metodo principale passa un *riferimento oggetto* per person al metodo setPersonName ; qualsiasi modifica che setAnotherName apporta a quell'oggetto è parte di quell'oggetto e quindi tali modifiche fanno ancora parte dell'oggetto quando il metodo restituisce.

Un altro modo di dire la stessa cosa: la person punta a un oggetto (memorizzato sull'heap, se sei interessato). Qualsiasi modifica apportata dal metodo a quell'oggetto viene effettuata "su quell'oggetto" e non viene influenzata dal fatto che il metodo che esegue la modifica sia ancora attivo o sia ritornato. Quando il metodo ritorna, tutte le modifiche apportate all'oggetto sono ancora memorizzate su quell'oggetto.

Confrontalo con il numero intero che viene passato. Poiché questo è un *primitivo* int (e non un'istanza dell'oggetto Integer), viene passato "per valore", indicando che il suo valore è fornito al metodo, non un puntatore al numero intero originale passato. Il metodo può cambiarlo per il metodo propri scopi, ma ciò non influisce sulla variabile utilizzata quando viene effettuata la chiamata al metodo.

In Java, tutti i primitivi vengono passati per valore. Gli oggetti vengono passati per riferimento, il che significa che un puntatore all'oggetto viene passato come parametro a qualsiasi metodo che li preleva.

Una cosa meno ovvia questo significa: non è possibile per un metodo chiamato creare un *nuovo* oggetto e restituirlo come uno dei parametri. L'unico modo per un metodo per restituire un oggetto creato, direttamente o indirettamente, dalla chiamata al metodo, è come valore di ritorno dal metodo. Prima vediamo come non funzionerebbe, e poi come funzionerebbe.

Aggiungiamo un altro metodo al nostro piccolo esempio qui:

```
private static void getAnotherObjectNot(Person person) {
  person = new Person();
  person.setName("George");
}
```

E, di nuovo nella parte main , sotto la chiamata a setAnotherName , chiamiamo questo metodo e un'altra chiamata println:

```
getAnotherObjectNot(person);
System.out.println(person.getName());
```

Ora il programma dovrebbe stampare:

```
Linda 5
Linda
```

Cosa è successo all'oggetto che aveva George? Bene, il parametro passato era un puntatore a Linda; quando il metodo getAnotherObjectNot creato un nuovo oggetto, ha sostituito il riferimento all'oggetto Linda con un riferimento all'oggetto George. L'oggetto Linda esiste ancora (nell'heap), il metodo main può ancora accedervi, ma il metodo getAnotherObjectNot non sarebbe in grado di fare nulla con esso dopo, perché non ha alcun riferimento ad esso. Sembrerebbe che lo scrittore del codice intendesse per il metodo creare un nuovo oggetto e passarlo indietro, ma se così fosse, non ha funzionato.

Se questo è ciò che lo scrittore voleva fare, avrebbe bisogno di restituire l'oggetto appena creato dal metodo, qualcosa del genere:

```
private static Person getAnotherObject() {
  Person person = new Person();
  person.setName("Mary");
  return person;
}
```

Quindi chiamalo così:

```
Person mary;
mary = getAnotherObject();
System.out.println(mary.getName());
```

E l'intero output del programma ora sarebbe:

```
Linda 5
Linda
Mary
```

Ecco l'intero programma, con entrambe le aggiunte:

```
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;
  }
}
```

Leggi Riferimenti dell'oggetto online: https://riptutorial.com/it/java/topic/5454/riferimenti-dell-oggetto

Capitolo 147: Risorse (sul classpath)

introduzione

Java consente il recupero di risorse basate su file memorizzate all'interno di un JAR insieme a classi compilate. Questo argomento si concentra sul caricamento di tali risorse e sulla loro disponibilità per il tuo codice.

Osservazioni

Una *risorsa* è data da file con un nome simile a un percorso, che risiede nel classpath. L'uso più comune delle risorse è il raggruppamento di immagini di applicazioni, suoni e dati di sola lettura (come la configurazione di default).

È possibile accedere alle risorse con i metodi ClassLoader.getResource e ClassLoader.getResourceAsStream . Il caso d'uso più comune è disporre di risorse inserite nello stesso pacchetto della classe che le legge; i metodi Class.getResource e Class.getResourceAsStream servono questo caso di utilizzo comune.

L'unica differenza tra un metodo getResource e il metodo getResourceAsStream è che il primo restituisce un URL, mentre quest'ultimo apre quell'URL e restituisce un InputStream.

I metodi di ClassLoader accettano un nome di risorsa simile a un percorso come argomento e ricercano ogni posizione nel classpath del ClassLoader per una voce corrispondente a tale nome.

- Se una posizione del percorso di classe è un file .jar, una voce di jar con il nome specificato è considerata una corrispondenza.
- Se una posizione del percorso di classe è una directory, un file relativo in tale directory con il nome specificato è considerato una corrispondenza.

Il nome della risorsa è simile alla porzione di percorso di un URL relativo. Su *tutte le piattaforme,* utilizza le barre (/) come separatori di directory. Non deve iniziare con una barra.

I metodi corrispondenti di Class sono simili, ad eccezione di:

- Il nome della risorsa può iniziare con una barra, nel qual caso la barra iniziale viene rimossa e il resto del nome viene passato al metodo corrispondente di ClassLoader.
- Se il nome della risorsa non inizia con una barra, viene considerato come relativo alla classe il cui metodo getResource o getResourceAsStream viene chiamato. Il nome effettivo della risorsa diventa *package* / *name* , dove *package* è il nome del pacchetto a cui appartiene la classe, con ogni periodo sostituito da una barra, e *name* è l'argomento originale dato al metodo.

Per esempio:

```
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);
    }
}
```

Le risorse dovrebbero essere collocate in pacchetti con nome, piuttosto che nella radice di un file .jar, per lo stesso motivo le classi sono collocate in pacchetti: per evitare collisioni tra più fornitori. Ad esempio, se più file .jar si trovano nel classpath e più di uno di essi contiene una voce config.properties nella sua radice, le chiamate ai metodi getResource o getResourceAsStream restituiranno i file config.properties da qualsiasi .jar è elencato per primo in il classpath. Questo comportamento non è prevedibile in ambienti in cui l'ordine del classpath non è sotto il controllo diretto dell'applicazione, come Java EE.

Tutti i metodi getResource e getResourceAsStream restituiscono null se la risorsa specificata non esiste. Poiché le risorse devono essere aggiunte all'applicazione al momento della compilazione, le loro posizioni dovrebbero essere note al momento della scrittura del codice; un errore nel trovare una risorsa in fase di esecuzione è in genere il risultato di un errore del programmatore.

Le risorse sono di sola lettura. Non c'è modo di scrivere su una risorsa. Gli sviluppatori principianti spesso commettono l'errore di assumere che dal momento che la risorsa è un file fisico separato durante lo sviluppo in un IDE (come Eclipse), sarà sicuro trattarlo come un file fisico separato nel caso generale. Tuttavia, questo non è corretto; le applicazioni sono quasi sempre distribuite come archivi come file .jar o .war e, in questi casi, una risorsa non sarà un file separato e non sarà scrivibile. (Il metodo getFile della classe URL *non* è una soluzione per questo, nonostante il suo nome, restituisce semplicemente la porzione di percorso di un URL, che non è in alcun modo garantito che sia un nome file valido).

Non esiste un modo sicuro per elencare le risorse in fase di runtime. Ancora una volta, poiché gli sviluppatori sono responsabili dell'aggiunta di file di risorse all'applicazione al momento della compilazione, gli sviluppatori dovrebbero già conoscere i loro percorsi. Mentre ci sono soluzioni alternative, non sono affidabili e alla fine falliranno.

Examples

Caricamento di un'immagine da una risorsa

Per caricare un'immagine in bundle:

```
package com.example;

public class ExampleApplication {
    private Image getIcon() throws IOException {
        URL imageURL = ExampleApplication.class.getResource("icon.png");
        return ImageIO.read(imageURL);
    }
}
```

Caricamento della configurazione predefinita

Per leggere le proprietà di configurazione predefinite:

```
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;
```

```
    }
}
```

Caricamento della risorsa con lo stesso nome da più JAR

Risorsa con lo stesso percorso e nome possono esistere in più di un file JAR sul classpath. I casi comuni sono risorse che seguono una convenzione o che fanno parte di una specifica di imballaggio. Esempi per tali risorse sono

- META-INF / MANIFEST.MF
- META-INF / beans.xml (Specifiche CDI)
- Proprietà ServiceLoader contenenti provider di implementazione

Per accedere a *tutte* queste risorse in diversi jar, è necessario utilizzare ClassLoader, che ha un metodo per questo. L' Enumeration restituita può essere convenientemente convertita in una List utilizzando una funzione Collections.

```
Enumeration<URL> resEnum = MyClass.class.getClassLoader().getResources("META-
INF/MANIFEST.MF");
ArrayList<URL> resources = Collections.list(resEnum);
```

Trovare e leggere le risorse usando un classloader

Il caricamento delle risorse in Java comprende i seguenti passaggi:

1. Trovare la Class o ClassLoader che troverà la risorsa.
2. Trovare la risorsa
3. Ottenere il flusso di byte per la risorsa.
4. Lettura ed elaborazione del flusso di byte.
5. Chiusura del flusso di byte.

Gli ultimi tre passaggi vengono in genere eseguiti passando l'URL a un metodo o costruttore di libreria per caricare la risorsa. In questo caso, in genere utilizzi un metodo getResource . È anche possibile leggere i dati della risorsa nel codice dell'applicazione. In questo caso, in genere si utilizza getResourceAsStream .

Percorsi di risorse assoluti e relativi

Le risorse che possono essere caricate dal classpath sono contraddistinte da un *percorso* . La sintassi del percorso è simile a un percorso file UNIX / Linux. Consiste di nomi semplici separati da caratteri di barra (/). Un *percorso relativo* inizia con un nome e un *percorso assoluto* inizia con un separatore.

Come descrivono gli esempi di Classpath, il classpath di una JVM definisce uno spazio dei nomi sovrapponendo gli spazi dei nomi delle directory e dei file JAR o ZIP nel classpath. Quando viene risolto un percorso assoluto, i classloader interpretano l'iniziale / come significato della radice dello spazio dei nomi. Al contrario, un percorso relativo *può essere* risolto rispetto a qualsiasi "cartella" nello spazio dei nomi. La cartella utilizzata dipenderà dall'oggetto che si utilizza per risolvere il percorso.

Ottenere una classe o un classloader

Una risorsa può essere localizzata usando un oggetto Class o un oggetto ClassLoader . Un oggetto di Class può risolvere percorsi relativi, quindi in genere si utilizzerà uno di questi se si dispone di una risorsa relativa (di classe). Esistono diversi modi per ottenere un oggetto Class . Per esempio:

- Un *letterale di classe* ti darà l'oggetto Class per qualsiasi classe che puoi nominare nel

codice sorgente Java; ad es. String.class fornisce l'oggetto Class per il tipo String .

- Object.getClass() ti darà l'oggetto Class per il tipo od qualsiasi oggetto; ad esempio "hello".getClass() è un altro modo per ottenere la Class del tipo String .

- Il Class.forName(String) (se necessario) carica dinamicamente una classe e restituisce il suo oggetto Class ; ad esempio Class.forName("java.lang.String") .

Un oggetto ClassLoader viene in genere ottenuto chiamando getClassLoader() su un oggetto Class . È anche possibile ottenere il classloader predefinito della JVM utilizzando il metodo ClassLoader.getSystemClassLoader() statico.

I metodi get

Una volta che hai un'istanza Class o ClassLoader , puoi trovare una risorsa, usando uno dei seguenti metodi:

metodi	Descrizione
`ClassLoader.getResource(path)` `ClassLoader.getResources(path)`	Restituisce un URL che rappresenta la posizione della risorsa con il percorso specificato.
`ClassLoader.getResources(path)` `Class.getResources(path)`	Restituisce `Enumeration<URL>` fornisce gli URL che possono essere utilizzati per individuare la risorsa `foo.bar` ; vedi sotto.
`ClassLoader.getResourceAsStream(path)` `Class.getResourceStream(path)`	Restituisce un `InputStream` dal quale è possibile leggere il contenuto della risorsa `foo.bar` come una sequenza di byte.

Gli appunti:

- La differenza principale tra le versioni ClassLoader e Class dei metodi è nel modo in cui i percorsi relativi vengono interpretati.
    - I metodi Class risolvono un percorso relativo nella "cartella" che corrisponde al pacchetto classes.
    - I metodi ClassLoader trattano i percorsi relativi come se fossero assoluti; cioè risolvili nella "cartella radice" dello spazio dei nomi del classpath.

- Se non è possibile trovare la risorsa richiesta (o le risorse), i methods return getResource e getResourceAsStream methods return null , and the methods return an empty getResources methods return an empty un'enumerazione vuota`.

- Gli URL restituiti saranno risolvibili utilizzando URL.toStream() . Potrebbero essere file: URL o altri URL convenzionali, ma se la risorsa risiede in un file JAR, saranno jar: URL che identificano il file JAR e una risorsa specifica al suo interno.

- Se il codice utilizza un metodo getResourceAsStream (o URL.toStream()) per ottenere un InputStream , è responsabile della chiusura dell'oggetto flusso. La mancata chiusura del flusso potrebbe causare una perdita di risorse.

Leggi Risorse (sul classpath) online: https://riptutorial.com/it/java/topic/2433/risorse--sul-classpath-

Capitolo 148: Scanner

Sintassi

- Scanner scanner = nuovo scanner (fonte di origine);
- Scanner scanner = nuovo scanner (System.in);

Parametri

Parametro	Dettagli
fonte	L'origine potrebbe essere uno di String, File o qualsiasi tipo di InputStream

Osservazioni

La classe Scanner è stata introdotta in Java 5. Il metodo reset() è stato aggiunto in Java 6 e un paio di nuovi costruttori sono stati aggiunti in Java 7 per l'interoperabilità con la (allora) nuova interfaccia Path .

Examples

Lettura dell'input del sistema tramite Scanner

```
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);
```

L'oggetto scanner è inizializzato per leggere l'input dalla tastiera. Quindi per l'input qui sotto da keyboar, produrrà l'output come Reading from keyboard

```
Reading
from
keyboard
  //space
```

Lettura dell'input di file tramite Scanner

```
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();
```

```
}
```

Qui viene creato un oggetto Scanner passando un oggetto File contenente il nome di un file di testo come input. Questo file di testo verrà aperto dall'oggetto File e letto dall'oggetto scanner nelle seguenti righe. scanner.hasNext() controllerà per vedere se c'è una riga successiva di dati nel file di testo. La combinazione di questo con un ciclo while ti consentirà di scorrere ogni riga di dati nel file Names.txt . Per recuperare i dati stessi, possiamo usare metodi come nextLine() , nextInt() , nextBoolean() , ecc. Nell'esempio sopra, scanner.nextLine() viene utilizzato. nextLine() riferisce alla seguente riga in un file di testo e combinandola con un oggetto scanner consente di stampare il contenuto della linea. Per chiudere un oggetto scanner, dovresti usare .close() .

Usando try with resources (da Java 7 in poi), il codice sopra citato può essere scritto elegantemente come sotto.

```
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!");
}
```

Leggi l'intero input come una stringa usando Scanner

È possibile utilizzare Scanner per leggere tutto il testo nell'input come String, utilizzando \Z (intero input) come delimitatore. Ad esempio, questo può essere usato per leggere tutto il testo in un file di testo in una riga:

```
String content = new Scanner(new File("filename")).useDelimiter("\\Z").next();
System.out.println(content);
```

Ricorda che dovrai chiudere lo Scanner, così come catturare l' IoException questo può lanciare, come descritto nell'esempio di lettura dell'ingresso del file usando Scanner .

Utilizzando delimitatori personalizzati

È possibile utilizzare delimitatori personalizzati (espressioni regolari) con Scanner, con .useDelimiter(",") , per determinare come viene letto l'input. Funziona in modo simile a String.split(...) . Ad esempio, è possibile utilizzare Scanner per leggere da un elenco di valori separati da virgola in una stringa:

```
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();
}
```

Questo ti permetterà di leggere ogni elemento nell'input individualmente. Si noti che **non** si dovrebbe usare questo per analizzare i dati CSV, invece, utilizzare un adeguato libreria parser CSV, vedere CSV parser per Java per altre possibilità.

Schema generale che viene generalmente richiesto per le attività

Di seguito è riportato come utilizzare correttamente la classe java.util.Scanner per leggere in modo interattivo l'input dell'utente da System.in correttamente (a volte indicato come stdin , in particolare in C, C ++ e altri linguaggi nonché in Unix e Linux). Dimostra in modo idiomatico le cose più comuni che devono essere fatte.

```
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);
    }
```

```
}
```

Leggi un int dalla riga di comando

```
import java.util.Scanner;

Scanner s = new Scanner(System.in);
int number = s.nextInt();
```

Se vuoi leggere un int dalla riga di comando, usa questo snippet. Prima di tutto, devi creare un oggetto Scanner, che ascolti System.in, che è di default la riga di comando, quando avvii il programma dalla riga di comando. Successivamente, con l'aiuto dell'oggetto Scanner, si legge la prima int che l'utente passa alla riga di comando e la memorizza nel numero variabile. Ora puoi fare tutto ciò che vuoi con quello memorizzato int.

Chiusura accurata di uno scanner

può succedere che si usi uno scanner con System.in come parametro per il costruttore, quindi è necessario essere consapevoli che chiudendo lo scanner si chiuderà anche InputStream dando come prossimo ogni tentativo di leggere l'input su quello (o qualsiasi altro oggetto scanner) genererà java.util.NoSuchElementException o java.lang.IllegalStateException

esempio:

```
    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();
```

Leggi Scanner online: https://riptutorial.com/it/java/topic/551/scanner

Capitolo 149: Scegliere le collezioni

introduzione

Java offre un'ampia varietà di collezioni. Scegliere quale Collezione usare può essere complicata. Vedi la sezione Esempi per un diagramma di flusso facile da seguire per scegliere la giusta Collezione per il lavoro.

Examples

Diagramma di flusso delle raccolte Java

Utilizzare il seguente diagramma di flusso per scegliere la raccolta corretta per il lavoro.

Questo diagramma di flusso era basato su [http://i.stack.imgur.com/aSDsG.png) .

Leggi Scegliere le collezioni online: https://riptutorial.com/it/java/topic/10846/scegliere-le-collezioni

Capitolo 150: serializzazione

introduzione

Java fornisce un meccanismo, chiamato serializzazione di oggetti in cui un oggetto può essere rappresentato come una sequenza di byte che include i dati dell'oggetto, nonché informazioni sul tipo dell'oggetto e i tipi di dati memorizzati nell'oggetto.

Dopo che un oggetto serializzato è stato scritto in un file, può essere letto dal file e deserializzato cioè, le informazioni sul tipo e i byte che rappresentano l'oggetto ei suoi dati possono essere utilizzati per ricreare l'oggetto in memoria.

Examples

Serializzazione di base in Java

Cos'è la serializzazione

La serializzazione è il processo di conversione dello stato di un oggetto (inclusi i suoi riferimenti) in una sequenza di byte, nonché il processo di ricostruzione di quei byte in un oggetto live in un momento futuro. La serializzazione viene utilizzata quando si desidera mantenere l'oggetto. Viene anche utilizzato da Java RMI per passare oggetti tra JVM, come argomenti in un richiamo di metodo da un client a un server o come valori di ritorno da una chiamata di metodo o come eccezioni generate da metodi remoti. In generale, la serializzazione viene utilizzata quando vogliamo che l'oggetto esista oltre la durata della JVM.

java.io.Serializable è un'interfaccia marker (non ha corpo). È appena usato per "contrassegnare" le classi Java come serializzabili.

Il runtime di serializzazione associa a ciascuna classe serializzabile un numero di versione, chiamato serialVersionUID , utilizzato durante la *de-* serializzazione per verificare che il mittente e il destinatario di un oggetto serializzato abbiano caricato classi per quell'oggetto che sono compatibili con la serializzazione. Se il destinatario ha caricato una classe per l'oggetto che ha un serialVersionUID diverso da quello della classe del mittente corrispondente, la deserializzazione produrrà una InvalidClassException . Una classe serializzabile può dichiarare il proprio serialVersionUID esplicitamente dichiarando un campo denominato serialVersionUID che deve essere static, final, e di tipo long :

```
ANY-ACCESS-MODIFIER static final long serialVersionUID = 1L;
```

Come rendere una classe idonea per la serializzazione

Per mantenere un oggetto, la rispettiva classe deve implementare l'interfaccia 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;
    }
}
```

Come scrivere un oggetto in un file

Ora dobbiamo scrivere questo oggetto su un file system. Utilizziamo java.io.ObjectOutputStream per questo scopo.

```
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();
        }
     }
 }
```

Come ricreare un oggetto dal suo stato serializzato

L'oggetto memorizzato può essere letto dal file system in un secondo momento utilizzando java.io.ObjectInputStream come mostrato di seguito:

```
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 classe serializzata è in forma binaria. La deserializzazione può essere problematica se la definizione della classe cambia: consultare il capitolo sul controllo delle versioni degli oggetti serializzati della specifica di serializzazione Java per i dettagli.

La serializzazione di un oggetto serializza l'intero grafico dell'oggetto di cui è la radice e opera correttamente in presenza di grafici ciclici. Viene fornito un metodo reset() per forzare ObjectOutputStream a dimenticare gli oggetti che sono già stati serializzati.

Campi transitori - Serializzazione

Serializzazione con Gson

La serializzazione con Gson è semplice e genera un JSON corretto.

```
public class Employe {

    private String firstName;
    private String lastName;
    private int age;
    private BigDecimal salary;
    private List<String> skills;

    //getters and setters
}
```

(Serializzazione)

```
//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"]}
```

Si noti che non è possibile serializzare oggetti con riferimenti circolari poiché ciò comporterà una ricorsione infinita.

(Deserializzazione)

```
//it's very simple...
//Assuming that json is the previous String object....

Employe obj2 = gson.fromJson(json, Employe.class); // obj2 is just like obj
```

Serializzazione con Jackson 2

Di seguito è una implementazione che dimostra come un oggetto può essere serializzato nella sua stringa JSON corrispondente.

```
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;
    }
}
```

serializzazione:

```
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
}
```

Produzione:

```
{
  "idx" : 1,
  "name" : "abc"
}
```

Puoi omettere la stampante Pretty Default se non ne hai bisogno.

La dipendenza usata qui è la seguente:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.6.3</version>
</dependency>
```

Serializzazione personalizzata

In questo esempio vogliamo creare una classe che genererà e produrrà in console, un numero casuale tra un intervallo di due numeri interi che vengono passati come argomenti durante l'inizializzazione.

```
    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();
            }
        }
    }
    }
```

Ora se vogliamo rendere questa classe serializzabile ci saranno alcuni problemi. La discussione è una delle determinate classi a livello di sistema che non sono serializzabili. Quindi dobbiamo dichiarare il thread come **transitorio** . In questo modo saremo in grado di serializzare gli oggetti di questa classe, ma avremo ancora un problema. Come puoi vedere nel costruttore impostiamo i valori minimo e massimo del nostro randomizzatore e dopo questo iniziamo il thread che è responsabile della generazione e della stampa del valore casuale. Pertanto, quando si ripristina l'oggetto persistente chiamando il ***readObject ()*** il costruttore non verrà eseguito di nuovo poiché non vi è creazione di un nuovo oggetto. In tal caso, dobbiamo sviluppare una **serializzazione personalizzata** fornendo due metodi all'interno della classe. Questi metodi sono:

```
private void writeObject(ObjectOutputStream out) throws IOException;
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

Quindi aggiungendo la nostra implementazione in ***readObject ()*** possiamo iniziare e iniziare la nostra discussione:

```
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();
}
}
```

Ecco il principale per il nostro esempio:

```
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();
        }

}
}
```

Se esegui il main vedrai che ci sono due thread in esecuzione per ogni istanza RangeRandom e questo perché il metodo ***Thread.start ()*** è ora sia nel costruttore che in ***readObject ()*** .

Versioning e serialVersionUID

Quando si implementa l'interfaccia java.io.Serializable per rendere serializzabile una classe, il compilatore cerca un campo static final denominato serialVersionUID di tipo long . Se la classe non ha dichiarato esplicitamente questo campo, il compilatore crea uno di questi campi e lo assegna con un valore che deriva da un calcolo dipendente serialVersionUID di serialVersionUID . Questo calcolo dipende da vari aspetti della classe e segue le specifiche di serializzazione dell'oggetto fornite da Sun. Ma non è garantito che il valore sia lo stesso in tutte le implementazioni del compilatore.

Questo valore viene utilizzato per verificare la compatibilità delle classi rispetto alla serializzazione e questo viene fatto durante la de-serializzazione di un oggetto salvato. Serialization Runtime verifica che serialVersionUID letto dai dati de-serializzati e che serialVersionUID dichiarato nella classe sia esattamente lo stesso. In caso contrario, genera una InvalidClassException .

Si consiglia vivamente di dichiarare e inizializzare in modo esplicito il campo finale statico di tipo long e denominato 'serialVersionUID' in tutte le classi che si desidera rendere serializzabili invece di fare affidamento sul calcolo predefinito del valore per questo campo anche se non si intende usa il controllo delle versioni. **Il calcolo del 'serialVersionUID' è estremamente sensibile e può variare da un'implementazione del compilatore a un altro e quindi è possibile che si InvalidClassException l' InvalidClassException anche per la stessa classe solo perché sono state utilizzate diverse implementazioni del compilatore sul mittente e sul destinatario del processo di serializzazione.**

```
public class Example implements Serializable {
    static final long serialVersionUID = 1L /*or some other value*/;
    //...
}
```

Finché serialVersionUID è lo stesso, la serializzazione Java può gestire diverse versioni di una classe. I cambiamenti compatibili e incompatibili sono;

Modifiche compatibili

- **Aggiunta di campi:** quando la classe da ricostituire ha un campo che non si verifica nel flusso, quel campo nell'oggetto verrà inizializzato sul valore predefinito per il suo tipo. Se è necessaria l'inizializzazione specifica della classe, la classe può fornire un metodo readObject che può inizializzare il campo con valori non predefiniti.
- **Aggiunta di classi:** lo stream conterrà la gerarchia di tipi di ciascun oggetto nello stream. Confrontando questa gerarchia nello stream con la classe corrente è possibile rilevare classi aggiuntive. Poiché non vi sono informazioni nel flusso da cui inizializzare l'oggetto, i campi della classe verranno inizializzati sui valori predefiniti.
- **Rimozione classi: il** confronto tra la gerarchia di classi nello stream con quella della classe corrente può rilevare che una classe è stata cancellata. In questo caso, i campi e gli oggetti corrispondenti a quella classe vengono letti dallo stream. I campi primitivi vengono scartati, ma vengono creati gli oggetti a cui fa riferimento la classe eliminata, poiché possono essere indirizzati in seguito nello stream. Verranno raccolti automaticamente quando lo stream viene raccolto o ripristinato.
- **Aggiunta di metodi writeObject / readObject:** se la versione che legge il flusso ha questi metodi, readObject è previsto, come al solito, di leggere i dati richiesti scritti nello stream dalla serializzazione predefinita. Dovrebbe chiamare defaultReadObject prima di

leggere qualsiasi dato facoltativo. Come al solito, il metodo writeObject richiama defaultWriteObject per scrivere i dati richiesti e quindi può scrivere dati opzionali.
- **Aggiunta di java.io.Serializable:** equivale ad aggiungere tipi. Non ci saranno valori nello stream per questa classe, quindi i suoi campi verranno inizializzati su valori predefiniti. Il supporto per sottoclassi di classi non serializzabili richiede che il supertipo della classe abbia un costruttore no-arg e la classe stessa venga inizializzata su valori predefiniti. Se il costruttore no-arg non è disponibile, viene generata l'InvalidClassException.
- **Modifica dell'accesso a un campo:** i modificatori di accesso public, package, protected e private non hanno alcun effetto sulla possibilità della serializzazione di assegnare valori ai campi.
- **Modifica di un campo da statico a nonstatic o transiente a non transitorio:** quando si utilizza la serializzazione predefinita per calcolare i campi serializzabili, questa modifica equivale all'aggiunta di un campo alla classe. Il nuovo campo verrà scritto nello stream, ma le classi precedenti ignoreranno il valore poiché la serializzazione non assegnerà valori a campi statici o transitori.

Cambiamenti incompatibili

- **Eliminazione di campi:** se un campo viene eliminato in una classe, lo stream scritto non conterrà il suo valore. Quando il flusso viene letto da una classe precedente, il valore del campo verrà impostato sul valore predefinito perché non è disponibile alcun valore nel flusso. Tuttavia, questo valore predefinito può compromettere negativamente la capacità della versione precedente di adempiere al proprio contratto.
- **Spostamento delle classi in alto o in basso nella gerarchia:** questo non può essere consentito poiché i dati nel flusso vengono visualizzati nella sequenza errata.
- **Modifica di un campo non statico in statico o un campo non transitorio in transitorio:** quando si fa affidamento sulla serializzazione predefinita, questa modifica equivale all'eliminazione di un campo dalla classe. Questa versione della classe non scriverà quei dati nello stream, quindi non sarà disponibile per essere letto dalle precedenti versioni della classe. Come quando si elimina un campo, il campo della versione precedente verrà inizializzato sul valore predefinito, il che può causare il fallimento della classe in modi imprevisti.
- **Modifica del tipo dichiarato di un campo primitivo:** ogni versione della classe scrive i dati con il suo tipo dichiarato. Le versioni precedenti della classe che tentano di leggere il campo non riusciranno perché il tipo di dati nel flusso non corrisponde al tipo del campo.
- Modifica del metodo writeObject o readObject in modo che non scriva più né legga i dati di campo predefiniti o non li modifichi in modo che tentino di scriverli o leggerli quando la versione precedente no. I dati di campo predefiniti devono essere costantemente visualizzati o non visualizzati nello stream.
- La modifica di una classe da Serializable a Externalizable o viceversa è una modifica incompatibile poiché il flusso conterrà dati incompatibili con l'implementazione della classe disponibile.
- Modifica di una classe da un tipo non enum a un tipo enum o viceversa poiché lo stream conterrà dati che sono incompatibili con l'implementazione della classe disponibile.
- La rimozione di Serializable o Externalizable è una modifica incompatibile poiché, una volta scritta, non fornirà più i campi necessari per le versioni precedenti della classe.
- L'aggiunta del metodo writeReplace o readResolve a una classe non è compatibile se il comportamento produce un oggetto incompatibile con qualsiasi versione precedente della classe.

Deserializzazione JSON personalizzata con Jackson

Consumiamo API di riposo come formato JSON e quindi unmarshal su un POJO.
L'org.codehaus.jackson.map.ObjectMapper di Jackson "funziona" immediatamente e in realtà non facciamo nulla nella maggior parte dei casi. Ma a volte abbiamo bisogno di un deserializzatore personalizzato per soddisfare le nostre esigenze personalizzate e questo tutorial ti guiderà attraverso il processo di creazione del tuo deserializzatore personalizzato.

Diciamo che abbiamo le seguenti entità.

```
public class User {
    private Long id;
    private String name;
    private String email;

    //getter setter are omitted for clarity
}
```

E

```
public class Program {
    private Long id;
    private String name;
    private User createdBy;
    private String contents;

    //getter setter are omitted for clarity
}
```

Iniziamo con il serializzare / marescializzare un oggetto.

```
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();
```

final String json = objectMapper.writeValueAsString (programma); System.out.println (JSON);

Il codice sopra produrrà seguente JSON-

```
{
    "id": 1,
    "name": "Program @# 1",
    "createdBy": {
        "id": 1,
        "name": "Bazlur Rahman",
        "email": "example@example.com"
    },
    "contents": "Some contents"
}
```

Ora può fare il contrario molto facilmente. Se disponiamo di questo JSON, possiamo eseguire unmarshal su un oggetto programma usando ObjectMapper come segue:

Ora diciamo, questo non è il caso reale, avremo un JSON diverso da un'API che non corrisponde alla nostra classe Program .

```
{
"id": 1,
"name": "Program @# 1",
"ownerId": 1
"contents": "Some contents"
```

```
}
```

Guarda la stringa JSON, puoi vedere, ha un campo diverso che è owenerId.

Ora se vuoi serializzare questo JSON come abbiamo fatto prima, avrai delle eccezioni.

Ci sono due modi per evitare le eccezioni e avere questo serializzato -

Ignora i campi sconosciuti

Ignora l' onwerId . Aggiungi la seguente annotazione nella classe Program

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Program {}
```

Scrivi un deserializzatore personalizzato

Ma ci sono casi in cui hai effettivamente bisogno di questo campo owerId . Diciamo che vuoi relazionarlo come un id della classe User .

In tal caso, è necessario scrivere un deserializzatore personalizzato-

Come puoi vedere, devi prima accedere a JsonNode da JonsParser . E quindi puoi facilmente estrarre informazioni da un JsonNode usando il metodo get() . e devi assicurarti del nome del campo. Dovrebbe essere il nome esatto, l'errore di ortografia causerà eccezioni.

E infine, devi registrare ProgramDeserializer su 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);
```

In alternativa, è possibile utilizzare l'annotazione per registrare direttamente il deserializzatore:

```
@JsonDeserialize(using = ProgramDeserializer.class)
public class Program {
}
```

Leggi serializzazione online: https://riptutorial.com/it/java/topic/767/serializzazione

Capitolo 151: ServiceLoader

Osservazioni

ServiceLoader può essere utilizzato per ottenere istanze di classi che estendono un determinato tipo (= servizio) specificato in un file contenuto in un file .jar . Il servizio che viene esteso / implementato è spesso un'interfaccia, ma non è necessario.

Le classi di estensione / implementazione devono fornire un costruttore di argomenti zero per ServiceLoader per istanziarle.

Per essere scoperto dal ServiceLoader un file di testo con il nome del nome di tipo completo del servizio implementato deve essere memorizzato all'interno della META-INF/services nel file jar. Questo file contiene un nome completo di una classe che implementa il servizio per riga.

Examples

Servizio logger

L'esempio seguente mostra come istanziare una classe per la registrazione tramite ServiceLoader .

Servizio

```
package servicetest;

import java.io.IOException;

public interface Logger extends AutoCloseable {

    void log(String message) throws IOException;
}
```

Implementazioni del servizio

La seguente implementazione scrive semplicemente il messaggio su 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 seguente implementazione scrive i messaggi in un file di testo:

```
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

Il file META-INF/services/servicetest.Logger elenca i nomi delle implementazioni di Logger .

```
servicetest.logger.ConsoleLogger
servicetest.logger.FileLogger
```

uso

Il seguente metodo main scrive un messaggio a tutti i logger disponibili. I logger vengono istanziati utilizzando 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);
        }
    }
}
```

Semplice esempio ServiceLoader

ServiceLoader è un meccanismo integrato semplice e facile da usare per il caricamento dinamico delle implementazioni dell'interfaccia. Con il caricatore di servizio, che fornisce mezzi per l'istanziazione (ma non il cablaggio), è possibile creare un semplice meccanismo di iniezione delle dipendenze in Java SE. Con l'interfaccia ServiceLoader e la separazione

dell'implementazione diventa naturale e i programmi possono essere estesi in modo conveniente. In realtà molte API Java sono implementate in base al ServiceLoader

I concetti di base sono

- Operativo su *interfacce* di servizi
- Ottenere l'implementazione (s) del servizio tramite ServiceLoader
- Fornire l'implementazione di servizi

Iniziamo dall'interfaccia e la accounting-api.jar in un vaso, chiamato ad esempio accounting-api.jar

```
package example;

public interface AccountingService {

  long getBalance();
}
```

Ora forniamo un'implementazione di tale servizio in un jar denominato accounting-impl.jar , contenente un'implementazione del servizio

```
package example.impl;
import example.AccountingService;

public interface DefaultAccountingService implements AccouningService {

  public long getBalance() {
    return balanceFromDB();
  }

  private long balanceFromDB(){
    ...
  }
}
```

inoltre, accounting-impl.jar contiene un file che dichiara che questo jar fornisce un'implementazione di AccountingService . Il file deve avere un percorso che inizia con META-INF/services/ e deve avere lo stesso nome del nome *completo* dell'interfaccia:

- META-INF/services/example.AccountingService

Il contenuto del file è il nome *completo* dell'implementazione:

```
example.impl.DefaultAccountingService
```

Dato che entrambi i jar si trovano nel classpath del programma, che utilizza AccountingService , è possibile ottenere un'istanza del servizio utilizzando ServiceLauncher

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
AccountingService service = loader.next();
long balance = service.getBalance();
```

Come il ServiceLoader è un Iterable , supporta diversi operatori di attuazione, in cui il programma può scegliere tra:

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
for(AccountingService service : loader) {
   //...
}
```

Si noti che quando si invoca next() verrà sempre creata una nuova istanza. Se si desidera riutilizzare un'istanza, è necessario utilizzare il metodo iterator() del ServiceLoader o del ciclo for-each, come mostrato sopra.

Leggi ServiceLoader online: https://riptutorial.com/it/java/topic/5433/serviceloader

Capitolo 152: Servizio di stampa Java

introduzione

L' API del servizio di stampa Java offre funzionalità per scoprire servizi di stampa e inviare richieste di stampa per loro.

Include attributi di stampa estensibili basati sugli attributi standard specificati in Internet Printing Protocol (IPP) 1.1 dalla specifica IETF, RFC 2911 .

Examples

Alla scoperta dei servizi di stampa disponibili

Per scoprire tutti i servizi di stampa disponibili, possiamo usare la classe PrintServiceLookup . Vediamo come:

```
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());
        }
    }

}
```

Questo programma, se eseguito su un ambiente Windows, stamperà qualcosa come questo:

```
Print service name: Fax
Print service name: Microsoft Print to PDF
Print service name: Microsoft XPS Document Viewer
```

Scoperta del servizio di stampa predefinito

Per scoprire il servizio di stampa predefinito, possiamo usare la classe PrintServiceLookup . Vediamo come:

```
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());
    }

}
```

Creazione di un lavoro di stampa da un servizio di stampa

Un lavoro di stampa è una richiesta di stampa di qualcosa in un servizio di stampa specifico. Consiste fondamentalmente in:

- i dati che verranno stampati (vedi Creazione del documento che verrà stampato)
- un insieme di attributi

Dopo aver raccolto l'istanza di servizio di stampa corretta, possiamo richiedere la creazione di un lavoro di stampa:

```
DocPrintJob printJob = printService.createPrintJob();
```

L'interfaccia di DocPrintJob ci fornisce il metodo di print :

```
printJob.print(doc, pras);
```

L'argomento doc è un Doc : i dati che verranno stampati.

E l'argomento pras è un'interfaccia PrintRequestAttributeSet : un insieme di PrintRequestAttribute . Sono esempi di attributi di richiesta di stampa:

- quantità di copie (1, 2 ecc.),
- orientamento (verticale o orizzontale)
- cromaticità (monocromatico, a colori)
- qualità (bozza, normale, alta)
- lati (unilaterali, bilaterali ecc.)
- e così via...

Il metodo di stampa può generare una PrintException .

Costruire il documento che verrà stampato

Doc è un'interfaccia e l'API del servizio di stampa Java fornisce un'implementazione semplice denominata SimpleDoc .

Ogni istanza di Doc è composta essenzialmente da due aspetti:

- il contenuto dei dati di stampa (una e-mail, un'immagine, un documento, ecc.)
- il formato dei dati di stampa, chiamato DocFlavor (tipo MIME + classe di rappresentazione).

Prima di creare l'oggetto Doc , dobbiamo caricare il nostro documento da qualche parte. Nell'esempio, verrà caricato un file specifico dal disco:

```
FileInputStream pdfFileInputStream = new FileInputStream("something.pdf");
```

Quindi ora dobbiamo scegliere un DocFlavor che corrisponda ai nostri contenuti. La classe DocFlavor ha una serie di costanti per rappresentare i tipi di dati più comuni. Prendiamo quello di INPUT_STREAM.PDF :

```
DocFlavor pdfDocFlavor = DocFlavor.INPUT_STREAM.PDF;
```

Ora possiamo creare una nuova istanza di SimpleDoc :

```
Doc doc = new SimpleDoc(pdfFileInputStream, pdfDocFlavor , null);
```

L'oggetto doc ora può essere inviato alla richiesta del lavoro di stampa (vedere Creazione di un lavoro di stampa da un servizio di stampa).

Definizione degli attributi della richiesta di stampa

A volte abbiamo bisogno di determinare alcuni aspetti della richiesta di stampa. Li chiameremo *attributo* .

Sono esempi di attributi di richiesta di stampa:

- quantità di copie (1, 2 ecc.),
- orientamento (verticale o orizzontale)
- cromaticità (monocromatico, a colori)
- qualità (bozza, normale, alta)
- lati (unilaterali, bilaterali ecc.)
- e così via...

Prima di scegliere uno di essi e quale valore ognuno avrà, per prima cosa dobbiamo creare un insieme di attributi:

```
PrintRequestAttributeSet pras = new HashPrintRequestAttributeSet();
```

Ora possiamo aggiungerli. Alcuni esempi sono:

```
pras.add(new Copies(5));
pras.add(MediaSize.ISO_A4);
pras.add(OrientationRequested.PORTRAIT);
pras.add(PrintQuality.NORMAL);
```

pras oggetto pras può essere inviato alla richiesta del lavoro di stampa (vedere Creazione di un lavoro di stampa da un servizio di stampa).

Modifica dello stato della richiesta di lavoro di stampa in ascolto

Per la maggior parte dei client di stampa, è estremamente utile sapere se un processo di stampa ha avuto esito positivo o negativo.

L'API del servizio di stampa Java fornisce alcune funzionalità per essere informati su questi scenari. Tutto quello che dobbiamo fare è:

- fornire un'implementazione per l'interfaccia PrintJobListener e
- registrare questa implementazione nel processo di stampa.

Quando lo stato del lavoro di stampa cambia, saremo avvisati. Possiamo fare tutto ciò che è necessario, ad esempio:

- aggiornare un'interfaccia utente,
- avviare un altro processo aziendale,
- registrare qualcosa nel database,
- o semplicemente registrarlo.

Nell'esempio seguente, registreremo ogni modifica dello stato dei lavori di stampa:

```
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 ");
    }
}
```

Infine, possiamo aggiungere la nostra implementazione del listener del lavoro di stampa sul lavoro di stampa prima della richiesta di stampa stessa, come segue:

```
DocPrintJob printJob = printService.createPrintJob();

printJob.addPrintJobListener(new LoggerPrintJobListener());

printJob.print(doc, pras);
```

L'argomento *pje PrintJobEvent*

Si noti che ogni metodo ha un argomento PrintJobEvent pje . Non lo usiamo in questo esempio per semplicità, ma puoi usarlo per esplorare lo stato. Per esempio:

```
pje.getPrintJob().getAttributes();
```

Restituirà un'istanza dell'oggetto PrintJobAttributeSet ed è possibile eseguirli in modo for-each.

Un altro modo per raggiungere lo stesso obiettivo

Un'altra opzione per raggiungere lo stesso obiettivo è l'estensione della classe PrintJobAdapter , come dice il nome, è un adattatore per PrintJobListener . Implementando l'interfaccia dobbiamo obbligatoriamente implementarli tutti. Il vantaggio di questo modo è che abbiamo bisogno di sovrascrivere solo i metodi che vogliamo. Vediamo come funziona:

```
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 =( ");
    }
}
```

Si noti che sostituiamo solo alcuni metodi specifici.

Allo stesso modo nell'esempio che implementa l'interfaccia PrintJobListener , aggiungiamo il listener al lavoro di stampa prima di inviarlo alla stampa:

```
printJob.addPrintJobListener(new LoggerPrintJobAdapter());

printJob.print(doc, pras);
```

Leggi Servizio di stampa Java online: https://riptutorial.com/it/java/topic/10178/servizio-di-stampa-java

Capitolo 153: Sicurezza e crittografia

Examples

Calcola hash crittografici

Per calcolare gli hash di blocchi relativamente piccoli di dati utilizzando diversi algoritmi:

```
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 questo risultato:

```
MD5    hash: E99E768582F6DD5A3BA2D9C849DF736E
SHA1   hash: 0135FAA6323685BA8A8FF8D3F955F0C36949D8FB
SHA256 hash: 8D35C97BCD902B96D1B551741BBE8A7F50BB5A690B4D0225482EAA63DBFB9DED
```

Potrebbero essere disponibili algoritmi aggiuntivi a seconda dell'implementazione della piattaforma Java.

Genera dati casuali crittograficamente

Per generare campioni di dati casuali crittograficamente:

```
final byte[] sample = new byte[16];

new SecureRandom().nextBytes(sample);

System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

Produce risultati simili a:

```
Sample: E4F14CEA2384F70B706B53A6DF8C5EFE
```

Si noti che la chiamata a nextBytes() potrebbe nextBytes() mentre l'entropia viene raccolta a seconda dell'algoritmo utilizzato.

Per specificare l'algoritmo e il provider:

```
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 risultati simili a:

```
Provider: SUN version 1.8
Algorithm: SHA1PRNG
Sample: C80C44BAEB352FD29FBBE20489E4C0B9
```

Genera coppie di chiavi pubbliche / private

Per generare coppie di chiavi utilizzando diversi algoritmi e dimensioni chiave:

```
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();
```

Ulteriori algoritmi e dimensioni delle chiavi potrebbero essere disponibili per l'implementazione della piattaforma Java.

Per specificare una fonte di casualità da utilizzare durante la generazione delle chiavi:

```
final KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");

generator.initialize(2048, SecureRandom.getInstance("SHA1PRNG", "SUN"));

final KeyPair pair = generator.generateKeyPair();
```

Calcola e verifica le firme digitali

Per calcolare 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();
```

Si noti che l'algoritmo della firma deve essere compatibile con l'algoritmo utilizzato per generare la coppia di chiavi.

Per verificare 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 questo risultato:

```
Signature: true
```

Criptare e decrittografare i dati con chiavi pubbliche / private

Per crittografare i dati con una chiave pubblica:

```
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 risultati simili a:

```
Message: Hello
Encrypted: 5641FBB9558ECFA9ED...
```

Si noti che durante la creazione dell'oggetto Cipher , è necessario specificare una trasformazione compatibile con il tipo di chiave utilizzata. (Vedi Nomi algoritmo standard JCA per un elenco di trasformazioni supportate.). Per i dati di crittografia RSA, la lunghezza message.getBytes() deve essere inferiore alla dimensione della chiave. Vedi questo SO Rispondi per i dettagli.

Per decrittografare i dati:

```
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 il seguente risultato:

```
Decrypted: Hello
```

Leggi Sicurezza e crittografia online: https://riptutorial.com/it/java/topic/7529/sicurezza-e-crittografia

Capitolo 154: Sicurezza e crittografia

introduzione

Le pratiche di sicurezza in Java possono essere separate in due categorie ampie e vagamente definite; Sicurezza della piattaforma Java e programmazione Java sicura.

Le pratiche di sicurezza della piattaforma Java si occupano della gestione della sicurezza e dell'integrità della JVM. Include argomenti come la gestione dei provider JCE e le politiche di sicurezza.

Le pratiche di programmazione Java sicure riguardano i modi migliori per scrivere programmi Java sicuri. Include argomenti come l'uso di numeri casuali e crittografia e la prevenzione delle vulnerabilità.

Osservazioni

Mentre gli esempi dovrebbero essere chiaramente fatti, alcuni argomenti che devono essere trattati sono:

1. Il concetto / struttura del fornitore JCE
2. Elemento dell'elenco

Examples

Il JCE

Java Cryptography Extension (JCE) è un framework integrato in JVM per consentire agli sviluppatori di utilizzare in modo semplice e sicuro la crittografia nei loro programmi. Lo fa fornendo un'interfaccia semplice e portatile ai programmatori, mentre utilizza un sistema di JCE Provider per implementare in modo sicuro le operazioni crittografiche sottostanti.

Chiavi e gestione delle chiavi

Mentre JCE protegge le operazioni di crittografia e la generazione delle chiavi, è compito dello sviluppatore gestire effettivamente le proprie chiavi. Maggiori informazioni devono essere fornite qui.

Una best practice comunemente accettata per la gestione delle chiavi in fase di esecuzione è quella di archiviarle solo come array di byte e mai come stringhe. Questo perché le stringhe Java sono immutabili e non possono essere "cancellate" o "azzerate" manualmente in memoria; mentre un riferimento a una stringa può essere rimosso, la stringa esatta rimarrà in memoria fino a quando il suo segmento di memoria non viene raccolto e riutilizzato. Un utente malintenzionato avrebbe una grande finestra in cui è possibile scaricare la memoria del programma e trovare facilmente la chiave. Al contrario, gli array di byte sono mutabili e possono avere il loro contenuto sovrascritto sul posto; è una buona idea azzerare le tue chiavi non appena non ne hai più bisogno.

Vulnerabilità comuni di Java

Ha bisogno di contenuti

Preoccupazioni di rete

Ha bisogno di contenuti

Casualità e tu

Ha bisogno di contenuti

Per la maggior parte delle applicazioni, la classe java.utils.Random è una fonte perfetta di dati "casuali". Se devi scegliere un elemento casuale da un array, o generare una stringa casuale, o creare un identificatore "univoco" temporaneo, dovresti probabilmente usare Random .

Tuttavia, molti sistemi crittografici si affidano alla casualità per la loro sicurezza e la casualità fornita da Random non è semplicemente di qualità sufficientemente alta. Per qualsiasi operazione di crittografia che richiede un input casuale, è consigliabile utilizzare SecureRandom .

Hashing e convalida

Più informazioni necessarie

Una funzione hash crittografica è un membro di una classe di funzioni con tre proprietà vitali; coerenza, unicità e irreversibilità.

Coerenza: dati gli stessi dati, una funzione di hash restituirà sempre lo stesso valore. Cioè, se X = Y, f (x) sarà sempre uguale a f (y) per la funzione hash f.

Unicità: non ci saranno mai due input per una funzione hash nello stesso output. Cioè, se X! = Y, f (x)! = F (y), per qualsiasi valore di X e Y.

Irreversibilità: è difficile, se non impossibile, "invertire" una funzione hash. Cioè, dato solo f (X), non dovrebbe esserci modo di trovare l'X originale in corto di mettere ogni possibile valore di X attraverso la funzione f (forza bruta). Non dovrebbe esserci alcuna funzione f1 tale che f1 (f (X)) = X.

Molte funzioni non hanno almeno uno di questi attributi. Ad esempio, MD5 e SHA1 sono noti per avere collisioni, cioè due ingressi che hanno lo stesso output, quindi mancano di unicità. Alcune funzioni che si ritiene attualmente siano sicure sono SHA-256 e SHA-512.

Leggi Sicurezza e crittografia online: https://riptutorial.com/it/java/topic/9371/sicurezza-e-crittografia

Capitolo 155: Singletons

introduzione

Un singleton è una classe che ha sempre una sola istanza. Per ulteriori informazioni sul *modello di progettazione* Singleton, fare riferimento all'argomento Singleton nel tag Design Patterns .

Examples

Enum Singleton

Java SE 5

```
public enum Singleton {
    INSTANCE;

    public void execute (String arg) {
        // Perform operation here
    }
}
```

Le enumerazioni hanno costruttori privati, sono definitive e forniscono un adeguato meccanismo di serializzazione. Sono anche molto concisi e pigramente inizializzati in modo sicuro.

La JVM garantisce che i valori di enum non verranno istanziati più di una volta ciascuno, il che conferisce al modello singolare enum una difesa molto forte contro gli attacchi di riflessione.

Ciò di cui il modello enumerabile *non* protegge è rappresentato dagli altri sviluppatori che aggiungono fisicamente più elementi al codice sorgente. Di conseguenza, se scegli questo stile di implementazione per i tuoi singleton, è imperativo che tu dichiari chiaramente che non dovrebbero essere aggiunti nuovi valori a tali enumerazioni.

Questo è il modo consigliato di implementare il modello singleton, come spiegato da Joshua Bloch in Effective Java.

Filetto sicuro Singleton con doppio bloccaggio controllato

Questo tipo di Singleton è thread-safe e impedisce il blocco non necessario dopo la creazione dell'istanza 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;
    }
}
```

Va sottolineato - nelle versioni precedenti a Java SE 5, l'implementazione di cui sopra non è corretta e dovrebbe essere evitata. Non è possibile implementare il blocco del doppio controllo correttamente in Java prima di Java 5.

Singleton senza uso di Enum (inizializzazione desiderosa)

```
public class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Si può sostenere che questo esempio è in *effetti* un'inizializzazione pigra. Sezione 12.4.1 degli stati della specifica del linguaggio Java :

> Una classe o un tipo di interfaccia T verrà inizializzato immediatamente prima della prima occorrenza di una delle seguenti operazioni:
>
> - T è una classe e viene creata un'istanza di T
> - T è una classe e viene invocato un metodo statico dichiarato da T
> - Viene assegnato un campo statico dichiarato da T
> - Viene utilizzato un campo statico dichiarato da T e il campo non è una variabile costante
> - T è una classe di livello superiore e viene eseguita un'istruzione di asserzione nidificata in modo lessico in T.

Pertanto, fintanto che non ci sono altri campi statici o metodi statici nella classe, l'istanza Singleton non verrà inizializzata finché il metodo getInstance() viene invocato la prima volta.

Inizializzazione pigra sicura del thread usando la classe del titolare | Implementazione di 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() {}
}
```

Questo inizializza la variabile INSTANCE alla prima chiamata a Singleton.getInstance() ,

sfruttando le garanzie di sicurezza del thread del linguaggio per l'inizializzazione statica senza richiedere ulteriore sincronizzazione.

Questa implementazione è nota anche come modello di Bill Pugh singleton. [Wiki]

Estensione di singleton (ereditarietà singleton)

In questo esempio, la classe base Singleton fornisce il metodo getMessage() che restituisce "Hello world!" Messaggio.

Le sottoclassi UppercaseSingleton e LowercaseSingleton sostituiscono il metodo getMessage () per fornire una rappresentazione appropriata del messaggio.

```
//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());
    }
}
```

Leggi Singletons online: https://riptutorial.com/it/java/topic/130/singletons

Capitolo 156: Socket Java

introduzione

I socket sono un'interfaccia di rete di basso livello che aiuta a creare una connessione tra due programmi principalmente client che possono o meno essere in esecuzione sulla stessa macchina.

Socket Programming è uno dei concetti di networking più diffusi.

Osservazioni

Esistono due tipi di traffico Internet Protocol -
1. TCP - Transmission Control Protocol 2. UDP - User Datagram Protocol

TCP è un protocollo orientato alla connessione.
UDP è un protocollo senza connessione.

TCP è adatto per applicazioni che richiedono un'elevata affidabilità e il tempo di trasmissione è relativamente meno critico.

UDP è adatto per applicazioni che richiedono una trasmissione rapida ed efficiente, come i giochi. La natura stateless di UDP è anche utile per i server che rispondono a piccole query da un numero enorme di client.

In parole più semplici -
Usa TCP quando non puoi permetterti di perdere dati e quando il tempo di inviare e ricevere dati non ha importanza. Usa UDP quando non puoi permetterti di perdere tempo e quando la perdita di dati non è importante.

È assolutamente garantito che i dati trasferiti rimangano intatti e arrivino nello stesso ordine in cui è stato inviato in caso di TCP.
mentre non vi è alcuna garanzia che i messaggi o i pacchetti inviati raggiungano affatto l'UDP.

Examples

Un semplice back server echo TCP

Il nostro server back echo TCP sarà un thread separato. È semplice come un inizio. Restituirà solo ciò che viene inviato, ma in forma maiuscola.

```
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);
        }
    }

}
```

Ora per accettare connessioni. Aggiorniamo il metodo di esecuzione.

```
@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);
    }
}
```

Ora se puoi aprire telnet e provare a connetterti vedrai un messaggio di benvenuto.

È necessario connettersi con la porta specificata e l'indirizzo IP.

Dovresti vedere un risultato simile a questo:

```
Welcome!

Connection to host lost.
```

Bene, la connessione è andata persa perché l'abbiamo interrotta. A volte dovremmo programmare il nostro client TCP. In questo caso, abbiamo bisogno di un client per richiedere l'input dell'utente e inviarlo attraverso la rete, ricevere l'input in maiuscolo.

Se il server invia prima i dati, il client deve prima leggere i dati.

```
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);
            }
        }
    }
}
```

Ora aggiorna il server

```
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();
```

E ora esegui il server e il client, dovresti avere un output simile a questo

```
Welcome!
>
```

Leggi Socket Java online: https://riptutorial.com/it/java/topic/9923/socket-java

Capitolo 157: Sockets

introduzione

Un socket è un punto finale di un collegamento di comunicazione bidirezionale tra due programmi in esecuzione sulla rete.

Examples

Leggi dalla presa

```
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
}
```

Leggi Sockets online: https://riptutorial.com/it/java/topic/9918/sockets

Capitolo 158: SortedMap

introduzione

Introduzione alla mappa ordinata.

Examples

Introduzione alla mappa ordinata.

Punto chiave :-

- L'interfaccia SortedMap estende la mappa.
- le voci sono mantenute in ordine crescente.

Metodi di mappa ordinata:

- Comparatore comparatore ().
- Oggetto firstKey ().
- SortedMap headMap (Object end).
- Oggetto lastKey ().
- SortedMap subMap (Inizio oggetto, Fine oggetto).
- SortedMap tailMap (Inizio dell'oggetto).

Esempio

```
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"));
   }
```

Leggi SortedMap online: https://riptutorial.com/it/java/topic/10748/sortedmap

Capitolo 159: StringBuffer

introduzione

Introduzione alla classe Java StringBuffer.

Examples

Classe Buffer di stringhe

Punti chiave :-

- usato per creare una stringa mutabile (modificabile).
- **Mutabile** : - Che può essere cambiato.
- è thread-safe, vale a dire che più thread non possono accedervi contemporaneamente.

Metodi: -

- append di StringBuffer sincronizzato pubblico (String s)
- inserto StringBuffer pubblico sincronizzato (int offset, String s)
- sostituisce StringBuffer sincronizzato pubblico (int startIndex, int endIndex, String str)
- delete StringBuffer sincronizzato pubblico (int startIndex, int endIndex)
- public synchronized StringBuffer reverse ()
- capacità pubblica int ()
- public void ensureCapacity (int minimumCapacity)
- char char pubblico (indice int)
- public int length ()
- sottostringa String pubblica (int beginIndex)
- sottostringa pubblica String (int beginIndex, int endIndex)

Esempio che mostra diffrence tra implementazione String e 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
 }
}
```

Leggi StringBuffer online: https://riptutorial.com/it/java/topic/10757/stringbuffer

Capitolo 160: StringBuilder

introduzione

La classe Java StringBuilder viene utilizzata per creare una stringa mutabile (modificabile). La classe Java StringBuilder è uguale alla classe StringBuffer, tranne per il fatto che non è sincronizzata. È disponibile da JDK 1.5.

Sintassi

- nuovo StringBuilder ()
- nuovo StringBuilder (capacità int)
- nuovo StringBuilder (CharSequence seq)
- nuovo StringBuilder (builder StringBuilder)
- new StringBuilder (stringa di stringhe)
- nuovo StringJoiner (delimitatore CharSequence)
- nuovo StringJoiner (delimitatore CharSequence, prefisso CharSequence, suffisso CharSequence)

Osservazioni

La creazione di un nuovo StringBuilder con tipo char come parametro comporterebbe la chiamata al costruttore con argomento int capacity e non a quello con argomento String string :

```
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

Ripeti una stringa n volte

Problema: creare una String contenente n ripetizioni di una String s .

L'approccio banale dovrebbe concatenare ripetutamente la String

```
final int n = ...
final String s = ...
String result = "";

for (int i = 0; i < n; i++) {
    result += s;
}
```

Questo crea n nuove istanze di stringa contenenti da 1 ad n ripetizioni di s risultante in un tempo di esecuzione di O(s.length() * n²) = O(s.length() * (1+2+...+(n-1)+n)) .

Per evitare questo, è necessario utilizzare StringBuilder , che consente invece di creare la String in O(s.length() * n) :

```
final int n = ...
final String s = ...

StringBuilder builder = new StringBuilder();

for (int i = 0; i < n; i++) {
    builder.append(s);
}

String result = builder.toString();
```

Confronto tra StringBuffer, StringBuilder, Formatter e StringJoiner

Le classi StringBuffer , StringBuilder , Formatter e StringJoiner sono classi di utilità Java SE principalmente utilizzate per assemblare stringhe da altre informazioni:

- La classe StringBuffer è presente da Java 1.0 e offre una varietà di metodi per la creazione e la modifica di un "buffer" contenente una sequenza di caratteri.
- La classe StringBuilder è stata aggiunta in Java 5 per risolvere i problemi di prestazioni con la classe StringBuffer originale. Le API per le due clases sono essenzialmente le stesse. La principale differenza tra StringBuffer e StringBuilder è che il primo è thread-safe e sincronizzato e il secondo no.

Questo esempio mostra come è possibile utilizzare 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 classe StringBuffer è usata allo stesso modo: basta cambiare StringBuilder in StringBuffer nel precedente)

Le classi StringBuffer e StringBuilder sono adatte sia per assemblare che per modificare stringhe; cioè forniscono metodi per sostituire e rimuovere caratteri e aggiungerli in vari modi. Le due classi remining sono specifiche per il compito di assemblare le stringhe.

- La classe Formatter è stata aggiunta in Java 5 ed è modellata liberamente sulla funzione sprintf nella libreria standard C. Prende una stringa di *formato* con *specificatori di formato* incorporato e una sequenza di altri argomenti e genera una stringa convertendo gli argomenti in testo e sostituendoli al posto degli specificatori di formato. I dettagli degli specificatori di formato indicano come gli argomenti vengono convertiti in testo.
- La classe StringJoiner è stata aggiunta in Java 8. È un programma di formattazione per scopi speciali che formatta succintamente una sequenza di stringhe con separatori tra loro. È progettato con un'API *fluente* e può essere utilizzato con stream Java 8.

Ecco alcuni esempi tipici di utilizzo 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 classe StringJoiner non è l'ideale per l'attività di cui sopra, quindi ecco un esempio di formattazione di una matrice di stringhe.

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
for (String s : new String[]{"A", "B", "C"}) {
    sj.add(s);
}
System.out.println(sj);
// Prints "[A, B, C]"
```

I casi d'uso per le 4 classi possono essere riassunti:

- StringBuilder adatto per qualsiasi assemblaggio di stringhe o attività di modifica delle stringhe.
- StringBuffer usa (solo) quando si richiede una versione thread-safe di StringBuilder .
- Formatter offre funzionalità di formattazione delle stringhe molto più ricche, ma non è efficiente come StringBuilder . Questo perché ogni chiamata a Formatter.format(...) comporta:
    - analizzando la stringa di format ,
    - creare e popolare un array *varargs* e
    - autoboxing di argomenti di tipo primitivo.
- StringJoiner fornisce una formattazione succinta ed efficiente di una sequenza di stringhe con separatori, ma non è adatta per altre attività di formattazione.

Leggi StringBuilder online: https://riptutorial.com/it/java/topic/1037/stringbuilder

Capitolo 161: stringhe

introduzione

Le stringhe (java.lang.String) sono pezzi di testo memorizzati nel tuo programma. Le stringhe **non** sono un tipo di dati primitivi in Java , tuttavia, sono molto comuni nei programmi Java.

In Java, le stringhe sono immutabili, il che significa che non possono essere modificate. (Fare clic qui per una spiegazione più approfondita dell'immutabilità).

Osservazioni

Poiché le stringhe Java sono immutabili , tutti i metodi che manipolano una String **restituiranno un nuovo oggetto String** . Non cambiano la String originale. Ciò include i metodi di sottostringa e di sostituzione che i programmatori C e C ++ si aspettano di modificare l'oggetto String destinazione.

Utilizzare un StringBuilder invece di String se si desidera concatenare più di due oggetti String cui valori non possono essere determinati in fase di compilazione. Questa tecnica è più performante rispetto alla creazione di nuovi oggetti String e concatenandoli perché StringBuilder è modificabile.

StringBuffer può anche essere utilizzato per concatenare oggetti String . Tuttavia, questa classe è meno performante perché è progettata per essere thread-safe e acquisisce un mutex prima di ogni operazione. Dal momento che non si ha quasi mai bisogno di thread-safe quando si concatenano le stringhe, è meglio usare StringBuilder .

Se puoi esprimere una concatenazione di stringhe come una singola espressione, allora è meglio usare l'operatore + . Il compilatore Java convertirà un'espressione contenente + concatenazioni in una sequenza efficiente di operazioni utilizzando String.concat(...) o StringBuilder . Il consiglio di usare StringBuilder esplicitamente si applica solo quando la concatenazione coinvolge più espressioni.

Non memorizzare informazioni sensibili nelle stringhe. Se qualcuno è in grado di ottenere un dump della memoria della tua applicazione in esecuzione, sarà in grado di trovare tutti gli oggetti String esistenti e leggerne il contenuto. Ciò include oggetti String non raggiungibili e in attesa di garbage collection. Se questo è un problema, è necessario cancellare i dati sensibili delle stringhe non appena ne hai finito. Non puoi farlo con gli oggetti String poiché sono immutabili. Pertanto, è consigliabile utilizzare un oggetto char[] per conservare dati di carattere sensibili e pulirli (ad es. Sovrascriverli con caratteri '\000') una volta terminato.

Tutte le istanze String vengono create nell'heap, anche le istanze corrispondenti ai valori letterali stringa. La particolarità dei valori letterali delle stringhe è che la JVM garantisce che tutti i letterali uguali (ovvero che siano costituiti dagli stessi caratteri) siano rappresentati da un singolo oggetto String (questo comportamento è specificato in JLS). Questo è implementato dai caricatori di classi JVM. Quando un programma di caricamento classe carica una classe, analizza i valori letterali stringa utilizzati nella definizione di classe, ogni volta che ne rileva uno, controlla se esiste già un record nel pool di stringhe per questo valore letterale (utilizzando il valore letterale come chiave) . Se esiste già una voce per il letterale, viene utilizzato il riferimento a un'istanza di String archiviata come coppia per quel letterale. Altrimenti, viene creata una nuova istanza String e un riferimento all'istanza viene archiviato per il letterale (utilizzato come chiave) nel pool di stringhe. (Vedi anche internamento stringa).

Il pool di stringhe viene tenuto nell'heap Java ed è soggetto alla normale garbage collection.

Java SE 7

Nelle versioni di Java precedenti a Java 7, il pool di stringhe era contenuto in una parte speciale dello heap nota come "PermGen". Questa parte è stata raccolta solo occasionalmente.

Java SE 7

In Java 7, il pool di stringhe è stato spostato da "PermGen".

Si noti che i valori letterali stringa sono implicitamente raggiungibili da qualsiasi metodo che li utilizza. Ciò significa che gli oggetti String corrispondenti possono essere raccolti solo se il codice stesso è garbage collector.

Fino a Java 8, gli oggetti String vengono implementati come un array di dati UTF-16 (2 byte per carattere). C'è una proposta in Java 9 per implementare String come array di byte con un campo flag di codifica per notare se la stringa è codificata come byte (LATIN-1) o chars (UTF-16).

Examples

Confronto di stringhe

Per confrontare le stringhe per l'uguaglianza, è necessario utilizzare i metodi equals o equalsIgnoreCase dell'oggetto String.

Ad esempio, il seguente snippet determinerà se le due istanze di String sono uguali su tutti i caratteri:

```
String firstString = "Test123";
String secondString = "Test" + 123;

if (firstString.equals(secondString)) {
   // Both Strings have the same content.
}
```

Dimostrazione dal vivo

Questo esempio li confronterà, indipendentemente dal loro caso:

```
String firstString = "Test123";
String secondString = "TEST123";

if (firstString.equalsIgnoreCase(secondString)) {
    // Both Strings are equal, ignoring the case of the individual characters.
}
```

Dimostrazione dal vivo

Si noti che equalsIgnoreCase non consente di specificare un Locale . Ad esempio, se si confrontano le due parole "Taki" e "TAKI" in inglese sono uguali; tuttavia, in turco sono diversi (in turco, il minuscolo I è ı). Per casi come questo, la conversione di entrambe le stringhe in minuscolo (o maiuscolo) con Locale e il confronto con gli equals è la soluzione.

```
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
```

Dimostrazione dal vivo

Non utilizzare l'operatore == per confrontare le stringhe

A meno che tu non possa garantire che tutte le stringhe siano state internate (vedi sotto), non **devi** usare gli operatori == o != Per confrontare le stringhe. Questi operatori verificano effettivamente i riferimenti e, poiché più oggetti String possono rappresentare la stessa stringa, è possibile fornire una risposta errata.

Utilizzare invece il String.equals(Object) , che confronterà gli oggetti String in base ai loro valori. Per una spiegazione dettagliata, fai riferimento a Pitfall: usando == per confrontare le stringhe .

Confronto tra stringhe in un'istruzione switch

Java SE 7

A partire da Java 1.7, è possibile confrontare una variabile String a valori letterali in un'istruzione switch . Assicurati che String non sia nullo, altrimenti genererà sempre una NullPointerException . I valori vengono confrontati usando String.equals , ovvero case sensitive.

```
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;
}
```

Dimostrazione dal vivo

Confronto di stringhe con valori costanti

Quando si confronta una String con un valore costante, è possibile inserire il valore costante sul lato sinistro degli equals per garantire che non si otterrà NullPointerException se l'altra stringa è null .

```
"baz".equals(foo)
```

Mentre foo.equals("baz") lancia una NullPointerException se foo è null , "baz".equals(foo) valuterà a false .

Java SE 7

Un'alternativa più leggibile consiste nell'utilizzare Objects.equals() , che esegue un controllo nullo su entrambi i parametri: Objects.equals(foo, "baz") .

(**Nota**: è discutibile se sia meglio evitare NullPointerExceptions in generale, o lasciarli accadere e quindi correggere la causa principale, vedere qui e qui . Certamente, chiamare la strategia di evitamento "best practice" non è giustificabile).

Ordinamenti di stringhe

La classe String implementa Comparable<String> con il metodo String.compareTo (come descritto all'inizio di questo esempio). Ciò rende l'ordinamento naturale degli oggetti String ordine con distinzione tra maiuscole e minuscole. La classe String fornisce una costante di Comparator<String> chiamata CASE_INSENSITIVE_ORDER adatta per l'ordinamento senza distinzione tra maiuscole e minuscole.

Confronto con stringhe internate

La specifica del linguaggio Java (JLS 3.10.6) afferma quanto segue:

> "Inoltre, una stringa letterale fa sempre riferimento alla stessa istanza della classe String poiché stringhe letterali o, più in generale, stringhe che rappresentano i valori delle espressioni costanti, vengono *internate in* modo da condividere istanze univoche, utilizzando il metodo String.intern . "

Ciò significa che è possibile confrontare i riferimenti a due *valori letterali* stringa usando == . Inoltre, lo stesso vale per i riferimenti agli oggetti String che sono stati prodotti utilizzando il metodo String.intern() .

Per esempio:

```
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");
}
```

Dietro le quinte, il meccanismo di interning mantiene una tabella hash che contiene tutte le stringhe internamente ancora *raggiungibili* . Quando si chiama intern() su una String , il metodo cerca l'oggetto nella tabella hash:

- Se la stringa viene trovata, tale valore viene restituito come stringa internata.
- Altrimenti, una copia della stringa viene aggiunta alla tabella hash e tale stringa viene restituita come stringa internata.

È possibile utilizzare l'interning per consentire il confronto delle stringhe utilizzando == . Tuttavia, ci sono problemi significativi con questo; vedi Pitfall - Le stringhe Internazionali in modo che tu possa usare == è una cattiva idea per i dettagli. Non è raccomandato nella maggior parte dei casi.

Modifica del caso di caratteri all'interno di una stringa

Il tipo String fornisce due metodi per la conversione di stringhe tra maiuscole e minuscole:

- toUpperCase per convertire tutti i caratteri in maiuscolo

- toLowerCase per convertire tutti i caratteri in lettere minuscole

Questi metodi restituiscono entrambe le stringhe convertite come nuove istanze String : gli oggetti String originali non vengono modificati perché String è immutabile in Java. Vedi questo per saperne di più sull'immutabilità: Immutabilità di stringhe in 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"
```

I caratteri non alfabetici, come cifre e segni di punteggiatura, non sono influenzati da questi metodi. Si noti che questi metodi possono anche trattare in modo errato determinati caratteri Unicode in determinate condizioni.

Nota : questi metodi sono *sensibili alle impostazioni internazionali* e potrebbero produrre risultati imprevisti se utilizzati su stringhe che devono essere interpretate indipendentemente dalle impostazioni internazionali. Gli esempi sono identificatori di linguaggio di programmazione, chiavi di protocollo e tag HTML .

Ad esempio, "TITLE".toLowerCase() in una locale turca restituisce " tıtle ", dove ı (\u0131) è il carattere LATIN SMALL LETTER DOTLESS I. Per ottenere risultati corretti per le stringhe insensibili della locale, passare Locale.ROOT come parametro al metodo di conversione caso corrispondente (ad es toLowerCase(Locale.ROOT) o toUpperCase(Locale.ROOT)).

Anche se l'uso di Locale.ENGLISH è corretto anche per la maggior parte dei casi, il modo in cui la **lingua è invariata** è Locale.ROOT .

Un elenco dettagliato dei caratteri Unicode che richiedono un involucro speciale può essere trovato sul sito Web del Consorzio Unicode .

Modifica del caso di un carattere specifico all'interno di una stringa ASCII:

Per cambiare il caso di un carattere specifico di una stringa ASCII, è possibile utilizzare l'algoritmo seguente:

passi:

1. Dichiara una stringa.
2. Inserisci la stringa.
3. Converti la stringa in un array di caratteri.
4. Inserisci il carattere che deve essere cercato.
5. Cerca il personaggio nell'array di caratteri.
6. Se trovato, controlla se il carattere è in minuscolo o maiuscolo.
    - Se è maiuscolo, aggiungi 32 al codice ASCII del carattere.
    - Se Minuscolo, sottrai 32 dal codice ASCII del carattere.
7. Cambia il carattere originale dall'array di caratteri.
8. Converti nuovamente l'array di caratteri nella stringa.

Voilà, il caso del personaggio è cambiato.

Un esempio del codice dell'algoritmo è:

```
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);
```

Trovare una stringa all'interno di un'altra stringa

Per verificare se una particolare stringa a è contenuta in una stringa b oppure no, possiamo usare il metodo String.contains() con la seguente sintassi:

```
b.contains(a); // Return true if a is contained in b, false otherwise
```

Il metodo String.contains() può essere utilizzato per verificare se è possibile trovare un CharSequence nella stringa. Il metodo cerca la stringa a nella stringa b in modo maiuscole e minuscole.

```
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 live su Ideone

Per trovare la posizione esatta in cui una stringa inizia all'interno di un'altra stringa, utilizzare 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 live su Ideone

Il metodo String.indexOf() restituisce il primo indice di un char o di una String in un'altra String . Il metodo restituisce -1 se non viene trovato.

Nota : il metodo String.indexOf() è sensibile al maiuscolo / minuscolo.

Esempio di ricerca che ignora il 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 live su Ideone

Ottenere la lunghezza di una stringa

Per ottenere la lunghezza di un oggetto String , chiamare il metodo length() su di esso. La lunghezza è uguale al numero di unità di codice UTF-16 (caratteri) nella stringa.

```
String str = "Hello, World!";
System.out.println(str.length()); // Prints out 13
```

Demo live su Ideone

Un char in una stringa è il valore UTF-16. I codepoint Unicode i cui valori sono ≥ 0x1000 (ad esempio, la maggior parte degli emoji) utilizzano due posizioni char. Per contare il numero di punti di codice Unicode in una stringa, indipendentemente dal fatto che ciascun punto di codice rientri in un valore di char UTF-16, è possibile utilizzare il metodo codePointCount :

```
int length = str.codePointCount(0, str.length());
```

È inoltre possibile utilizzare un flusso di codepoint, come di Java 8:

```
int length = str.codePoints().count();
```

sottostringhe

```
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")
```

Le sottostringhe possono anche essere applicate per tagliare e aggiungere / sostituire caratteri nella sua stringa originale. Ad esempio, hai affrontato una data cinese contenente caratteri cinesi, ma vuoi memorizzarla come una stringa di data in formato well.

```
String datestring = "2015□11□17□"
datestring = datestring.substring(0, 4) + "-" + datestring.substring(5,7) + "-" +
datestring.substring(8,10);
//Result will be 2015-11-17
```

Il metodo della sottostringa estrae un pezzo di una String . Quando viene fornito un parametro, il parametro è l'inizio e il pezzo si estende fino alla fine della String . Quando vengono dati due parametri, il primo parametro è il carattere di partenza e il secondo parametro è l'indice del carattere subito dopo la fine (il carattere dell'indice non è incluso). Un modo semplice per verificare è la sottrazione del primo parametro dal secondo dovrebbe fornire la lunghezza prevista della stringa.

Java SE 7

Nelle versioni JDK <7u6 il metodo della substring crea un'istanza di una String che condivide lo stesso char[] backup char[] della String originale e dispone dei campi di offset e count interni impostati per l'inizio e la lunghezza del risultato. Tale condivisione può causare perdite di memoria, che possono essere prevenute chiamando la new String(s.substring(...)) per forzare la creazione di una copia, dopo di che char[] può essere garbage collection.

Java SE 7

Da JDK 7u6 il metodo della substring copia sempre l'intero array char[] sottostante, rendendo la complessità lineare rispetto alla costante precedente ma garantendo l'assenza di perdite di memoria contemporaneamente.

Ottenere l'ennesimo carattere in una stringa

```
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"
```

Per ottenere l'ennesimo carattere in una stringa, chiama semplicemente charAt(n) su una String , dove n è l'indice del carattere che desideri recuperare

NOTA: l' indice n inizia a 0 , quindi il primo elemento è n = 0.

Separatore di nuove linee indipendente dalla piattaforma

Poiché il nuovo separatore di linee varia da piattaforma a piattaforma (ad es. \n su sistemi di tipo Unix o \r\n su Windows) è spesso necessario avere un modo indipendente dalla piattaforma di accedervi. In Java può essere recuperato da una proprietà di sistema:

```
System.getProperty("line.separator")
```

Java SE 7

Poiché il nuovo separatore di righe è così comunemente necessario, da Java 7 su un metodo di scelta rapida che restituisce esattamente lo stesso risultato del codice precedente è disponibile:

```
System.lineSeparator()
```

Nota : poiché è molto improbabile che il nuovo separatore di riga cambi durante l'esecuzione del programma, è consigliabile archiviarlo in una variabile finale statica anziché recuperarlo dalla proprietà di sistema ogni volta che è necessario.

Quando si utilizza String.format , utilizzare %n anziché \n o '\ r \ n' per generare un nuovo separatore di riga indipendente dalla piattaforma.

```
System.out.println(String.format('line 1: %s.%nline 2: %s%n', lines[0],lines[1]));
```

Aggiunta del metodo toString () per oggetti personalizzati

Supponiamo di aver definito la seguente classe Person :

```
public class Person {

    String name;
    int age;

    public Person (int age, String name) {
        this.age = age;
        this.name = name;
    }
}
```

Se istanziate un nuovo oggetto Person :

```
Person person = new Person(25, "John");
```

e più avanti nel tuo codice usi la seguente dichiarazione per stampare l'oggetto:

```
System.out.println(person.toString());
```

Demo live su Ideone

otterrai un risultato simile al seguente:

```
Person@7ab89d
```

Questo è il risultato dell'implementazione del metodo toString() definito nella classe Object , una superclasse di Person . La documentazione di Object.toString() afferma:

> Il metodo toString per l'oggetto classe restituisce una stringa composta dal nome della classe di cui l'oggetto è un'istanza, il carattere at-sign `@ 'e la rappresentazione esadecimale senza firma del codice hash dell'oggetto. In altre parole, questo metodo restituisce una stringa uguale al valore di:
>
> ```
> getClass().getName() + '@' + Integer.toHexString(hashCode())
> ```

Quindi, per un output significativo, dovrai **sovrascrivere** il metodo toString() :

```
@Override
public String toString() {
    return "My name is " + this.name + " and my age is " + this.age;
}
```

Ora l'output sarà:

```
My name is John and my age is 25
```

Puoi anche scrivere

```
System.out.println(person);
```

Demo live su Ideone

Infatti, println() richiama implicitamente il metodo toString sull'oggetto.

Spaccare le corde

È possibile dividere una String su un particolare carattere di delimitazione o un'espressione regolare , è possibile utilizzare il metodo String.split() con la seguente firma:

```
public String[] split(String regex)
```

Si noti che la delimitazione del carattere o dell'espressione regolare viene rimossa dalla matrice di stringhe risultante.

Esempio utilizzando il carattere di delimitazione:

```
String lineFromCsvFile = "Mickey;Bolton;12345;121216";
String[] dataCells = lineFromCsvFile.split(";");
// Result is dataCells = { "Mickey", "Bolton", "12345", "121216"};
```

Esempio usando l'espressione regolare:

```
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?"};
```

Puoi anche dividere direttamente un valore letterale String :

```
String[] firstNames = "Mickey, Frank, Alicia, Tom".split(", ");
// Result is firstNames = {"Mickey", "Frank", "Alicia", "Tom"};
```

Avvertenza : non dimenticare che il parametro viene sempre considerato come un'espressione regolare.

```
"aaa.bbb".split("."); // This returns an empty array
```

Nell'esempio precedente . viene considerato come il carattere jolly di espressione regolare che corrisponde a qualsiasi carattere e poiché ogni carattere è un delimitatore, il risultato è un array vuoto.

Divisione basata su un delimitatore che è un meta-carattere regex

I seguenti personaggi sono considerati speciali (detti anche meta-caratteri) in espressioni regolari

```
  < > - = ! ( ) [ ] { } \ ^ $ | ? * + .
```

Per dividere una stringa in base a uno dei delimitatori sopra indicati, è necessario Pattern.quote() l' *escape* usando \\ o utilizzare Pattern.quote() :

- Utilizzando Pattern.quote() :

  ```
  String s = "a|b|c";
  String regex = Pattern.quote("|");
  String[] arr = s.split(regex);
  ```

- Sfuggire ai caratteri speciali:

  ```
  String s = "a|b|c";
  String[] arr = s.split("\\|");
  ```

Split rimuove i valori vuoti

split(delimiter) per default rimuove le stringhe vuote finali dall'array dei risultati. Per disattivare questo meccanismo, è necessario utilizzare la versione di split(delimiter, limit) con limite impostato su valore negativo come

```
String[] split = data.split("\\|", -1);
```

split(regex) restituisce internamente il risultato di split(regex, 0) .

Il parametro limit controlla il numero di volte in cui il pattern è applicato e quindi influenza la lunghezza dell'array risultante.
Se il limite *n* è maggiore di zero, il pattern verrà applicato al massimo *n - 1* volte, la lunghezza dell'array non sarà maggiore di *n* e l'ultima voce dell'array conterrà tutti gli input oltre l'ultimo delimitatore corrispondente.
Se *n* è negativo, il pattern verrà applicato il maggior numero possibile di volte e l'array può avere una lunghezza qualsiasi.

Se *n* è zero, il pattern verrà applicato il maggior numero di volte possibile, l'array può avere una lunghezza qualsiasi e le stringhe vuote verranno eliminate.

Divisione con StringTokenizer

Oltre al metodo split() , le stringhe possono anche essere suddivise usando StringTokenizer .

StringTokenizer è ancora più restrittivo di String.split() , e anche un po 'più difficile da usare. È essenzialmente progettato per estrarre token delimitati da un set fisso di caratteri (dati come String). Ogni personaggio fungerà da separatore. A causa di questa restrizione, è circa il doppio della velocità di String.split() .

I set di caratteri predefiniti sono spazi vuoti (\t\n\r\f). L'esempio seguente stamperà ogni parola separatamente.

```
String str = "the lazy fox jumped over the brown fence";
StringTokenizer tokenizer = new StringTokenizer(str);
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

Questo verrà stampato:

```
the
lazy
fox
jumped
over
the
brown
fence
```

È possibile utilizzare diversi set di caratteri per la separazione.

```
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());
}
```

Questo verrà stampato:

```
j
mp
d ov
r
```

Unione di stringhe con un delimitatore

Java SE 8

Un array di stringhe può essere unito usando il metodo statico String.join() :

```
String[] elements = { "foo", "bar", "foobar" };
String singleString = String.join(" + ", elements);

System.out.println(singleString); // Prints "foo + bar + foobar"
```

Allo stesso modo, c'è un metodo String.join() sovraccarico per Iterable s.

Per avere un controllo dettagliato sull'unione , è possibile utilizzare la classe 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]"
```

Per unire un flusso di stringhe, puoi utilizzare il raccoglitore di join:

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", "));
System.out.println(joined); // Prints "foo, bar, foobar"
```

C'è anche un'opzione per definire prefisso e suffisso :

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", ", "{", "}"));
System.out.println(joined); // Prints "{foo, bar, foobar}"
```

Inversione di archi

Ci sono un paio di modi in cui puoi invertire una stringa per farla tornare indietro.

1. StringBuilder / StringBuffer:

```
 String code = "code";
 System.out.println(code);

 StringBuilder sb = new StringBuilder(code);
 code = sb.reverse().toString();

 System.out.println(code);
```

2. Array di caratteri:

```
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));
```

Conteggio delle occorrenze di una sottostringa o di un carattere in una stringa

countMatches metodo countMatches di org.apache.commons.lang3.StringUtils viene in genere

utilizzato per contare le occorrenze di una sottostringa o di un carattere in 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
```

In caso contrario, per le stesse API Java standard è possibile utilizzare le espressioni regolari:

```
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;
}
```

Concatenazione di stringhe e StringBuilder

La concatenazione di stringhe può essere eseguita usando l'operatore + . Per esempio:

```
String s1 = "a";
String s2 = "b";
String s3 = "c";
String s = s1 + s2 + s3; // abc
```

Normalmente un'implementazione del compilatore eseguirà la suddetta concatenazione usando metodi che coinvolgono un StringBuilder sotto il cofano. Una volta compilato, il codice sarà simile al seguente:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append("c").toString();
```

StringBuilder ha diversi metodi di overloading per aggiungere tipi diversi, ad esempio, per aggiungere un int posto di una String . Ad esempio, un'implementazione può convertire:

```
String s1 = "a";
String s2 = "b";
String s = s1 + s2 + 2; // ab2
```

al seguente:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append(2).toString();
```

Gli esempi sopra illustrano una semplice operazione di concatenazione che viene effettivamente eseguita in un singolo punto nel codice. La concatenazione implica una singola istanza di StringBuilder . In alcuni casi, una concatenazione viene eseguita in modo cumulativo come in un ciclo:

```
String result = "";
for(int i = 0; i < array.length; i++) {
    result += extractElement(array[i]);
}
return result;
```

In questi casi, l'ottimizzazione del compilatore di solito non viene applicata e ogni iterazione creerà un nuovo oggetto StringBuilder . Questo può essere ottimizzato trasformando esplicitamente il codice per utilizzare un singolo StringBuilder :

```
StringBuilder result = new StringBuilder();
for(int i = 0; i < array.length; i++) {
    result.append(extractElement(array[i]));
}
return result.toString();
```

Un StringBuilder verrà inizializzato con uno spazio vuoto di soli 16 caratteri. Se si sa in anticipo che si costruiscono stringhe più grandi, può essere utile inizializzarlo con dimensioni sufficienti in anticipo, in modo che il buffer interno non debba essere ridimensionato:

```
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
```

Se stai producendo molte stringhe, è consigliabile riutilizzare StringBuilder :

```
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());
}
```

Se (e solo se) più thread scrivono nello *stesso* buffer, utilizzare StringBuffer , che è una versione synchronized di StringBuilder . Ma poiché solitamente un solo thread scrive su un buffer, di solito è più veloce usare StringBuilder senza sincronizzazione.

Usando il metodo concat ():

```
String string1 = "Hello ";
String string2 = "world";
String string3 = string1.concat(string2);  // "Hello world"
```

Ciò restituisce una nuova stringa che è stringa1 con stringa2 aggiunta ad essa alla fine. Puoi anche usare il metodo concat () con stringhe letterali, come in:

```
"My name is ".concat("Buyya");
```

Sostituzione di parti di stringhe

Due modi per sostituire: regex o per corrispondenza esatta.

Nota: l'oggetto String originale sarà invariato, il valore restituito mantiene la stringa modificata.

Corrispondenza esatta

Sostituisci un singolo carattere con un altro singolo carattere:

```
String replace(char oldChar, char newChar)
```

> Restituisce una nuova stringa risultante dalla sostituzione di tutte le occorrenze di oldChar in questa stringa con newChar.

```
String s = "popcorn";
System.out.println(s.replace('p','W'));
```

Risultato:

```
WoWcorn
```

Sostituisci la sequenza di caratteri con un'altra sequenza di caratteri:

```
String replace(CharSequence target, CharSequence replacement)
```

> Sostituisce ogni sottostringa di questa stringa che corrisponde alla sequenza di destinazione letterale con la sequenza di sostituzione letterale specificata.

```
String s = "metal petal et al.";
System.out.println(s.replace("etal","etallica"));
```

Risultato:

```
metallica petallica et al.
```

regex

Nota : il raggruppamento usa il carattere $ per fare riferimento ai gruppi, ad esempio $1 .

Sostituisci tutte le partite:

```
String replaceAll(String regex, String replacement)
```

> Sostituisce ogni sottostringa di questa stringa che corrisponde all'espressione regolare data con la sostituzione fornita.

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Risultato:

```
spiral metallica petallica et al.
```

Sostituisci solo la prima partita:

```
String replaceFirst(String regex, String replacement)
```

Sostituisce la prima sottostringa di questa stringa che corrisponde all'espressione regolare data con la sostituzione specificata

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Risultato:

```
spiral metallica petal et al.
```

Rimuovi spazi bianchi dall'inizio e alla fine di una stringa

Il metodo trim() restituisce una nuova stringa con gli spazi bianchi iniziali e finali rimossi.

```
String s = new String("   Hello World!!  ");
String t = s.trim();  // t = "Hello World!!"
```

Se si trim una stringa che non ha spazi bianchi da rimuovere, verrà restituita la stessa istanza String.

Si noti che il metodo trim() ha la propria nozione di spazio bianco , che differisce dalla nozione utilizzata dal metodo Character.isWhitespace() :

- Tutti i caratteri di controllo ASCII con i codici da U+0000 a U+0020 sono considerati spazi bianchi e vengono rimossi da trim() . Questo include U+0020 'SPACE' , U+0009 'CHARACTER TABULATION' , U+000A 'LINE FEED' e U+000D 'CARRIAGE RETURN' , ma anche i caratteri come U+0007 'BELL' .
- Gli spazi bianchi Unicode come U+00A0 'NO-BREAK SPACE' o U+2003 'EM SPACE' *non* sono riconosciuti da trim() .

Pool di stringhe e archiviazione heap

Come molti oggetti Java, **tutte le** istanze String vengono create sullo heap, anche letterali. Quando la JVM trova un valore letterale String che non ha riferimenti equivalenti nell'heap, la JVM crea un'istanza String corrispondente nell'heap **e** memorizza anche un riferimento all'istanza String appena creata nel pool String. Qualsiasi altro riferimento allo stesso letterale String viene sostituito con l'istanza String precedentemente creata nell'heap.

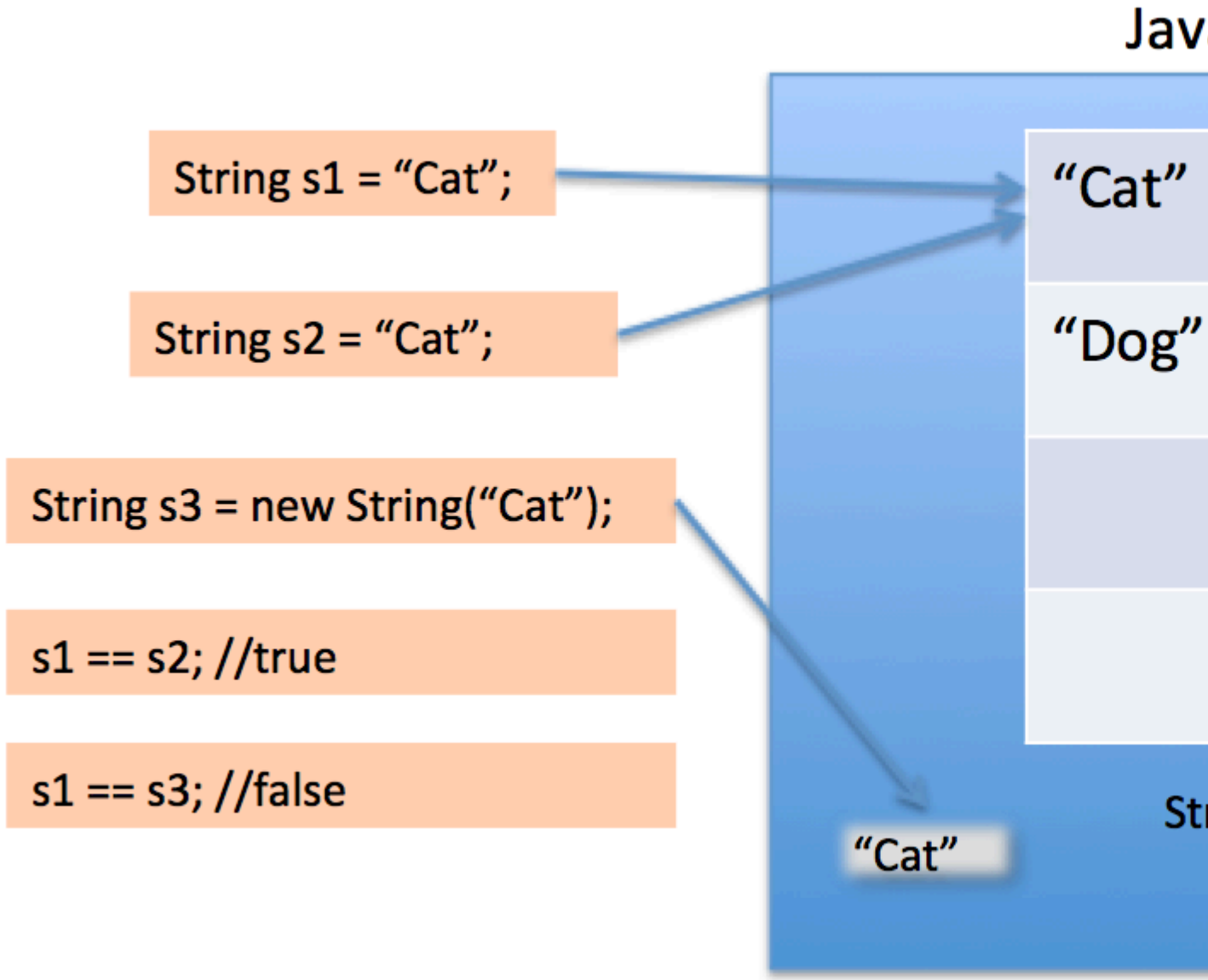
Diamo un'occhiata al seguente esempio:

```
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);
    }
}
```

L'output di quanto sopra è:

```
true
true
true
true
```



Quando si usano le doppie virgolette per creare una stringa, prima cerca String con lo stesso valore nel pool String, se trovato restituisce solo il riferimento altrimenti crea una nuova stringa nel pool e quindi restituisce il riferimento.

Tuttavia, utilizzando un nuovo operatore, imponiamo la classe String per creare un nuovo oggetto String nello spazio heap. Possiamo usare il metodo intern () per metterlo nel pool o fare riferimento ad un altro oggetto String dal pool di stringhe con lo stesso valore.

Anche il pool di stringhe stesso viene creato nell'heap.

Java SE 7

Prima di Java 7, i **valori letterali** String venivano memorizzati nel pool costante di runtime nell'area del metodo di PermGen , che aveva una dimensione fissa.

Il pool di stringhe risiedeva anche in PermGen .

Java SE 7

RFC: 6962931

> In JDK 7, le stringhe internate non sono più allocate nella generazione permanente dell'heap Java, ma sono allocate nella parte principale dell'heap Java (noto come generazioni giovani e meno recenti), insieme agli altri oggetti creati dall'applicazione . Questo cambiamento comporterà più dati residenti nell'heap principale di Java e meno dati nella generazione permanente e, pertanto, potrebbe richiedere la regolazione delle dimensioni dell'heap. La maggior parte delle applicazioni vedrà solo differenze relativamente piccole nell'utilizzo dell'heap a causa di questa modifica, ma applicazioni più grandi che caricano molte classi o fanno un uso massiccio del metodo String.intern() vedranno differenze più significative.

Interruttore non sensibile al maiuscolo / minuscolo

Java SE 7

switch stesso non può essere parametrizzato senza distinzione tra maiuscole e minuscole, ma se strettamente necessario, può comportarsi in modo insensibile alla stringa di input utilizzando toLowerCase() o toUpperCase :

```
switch (myString.toLowerCase()) {
     case "case1" :
        ...
     break;
     case "case2" :
        ...
     break;
}
```

diffidare

- Locale potrebbero influenzare il modo in cui i casi cambiano !
- Bisogna fare attenzione a non avere caratteri maiuscoli nelle etichette - quelli non verranno mai eseguiti!

Leggi stringhe online: https://riptutorial.com/it/java/topic/109/stringhe

Capitolo 162: Strutture di controllo di base

Osservazioni

Tutte le strutture di controllo, se non diversamente specificato, fanno uso di dichiarazioni di blocco . Questi sono indicati da parentesi graffe {} .

Ciò differisce dalle affermazioni normali , che *non* richiedono parentesi graffe, ma presentano anche un avvertimento rigido nel senso che solo la linea che *segue immediatamente* la frase precedente verrà presa in considerazione.

Pertanto, è perfettamente valido scrivere qualsiasi di queste strutture di controllo senza parentesi graffe, purché solo *una* istruzione segua l'inizio, ma è ***fortemente scoraggiata*** , in quanto può portare a implementazioni errate o codice non funzionante.

Esempio:

```
// 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

Se / Else If / 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");
}
```

Il blocco if verrà eseguito solo quando i è 1 o meno.

La condizione else if viene verificata solo se tutte le condizioni precedenti (nel precedente else if construct e parent if costrutti) sono state testate su false . In questo esempio, la condizione else if verrà verificata solo se i è maggiore o uguale a 2.

Se il risultato è true , viene eseguito il blocco e tutti i costrutti else if e else verranno saltati.

Se nessuna delle if e else if condizioni sono state testate su true , verrà eseguito il blocco else alla fine.

Per loop

```
for (int i = 0; i < 100; i++) {
    System.out.println(i);
```

```
}
```

I tre componenti del ciclo for (separati da ;) sono dichiarazione / inizializzazione variabile (qui int i = 0), la condizione (qui i < 100) e l'istruzione di incremento (qui i++). La dichiarazione delle variabili viene eseguita una volta come se fosse posizionata appena all'interno di { alla prima esecuzione. Quindi la condizione viene verificata, se è true il corpo del ciclo verrà eseguito, se è false il ciclo si fermerà. Supponendo che il ciclo continui, il corpo verrà eseguito e, infine, quando viene raggiunto il punto } l'istruzione di incremento verrà eseguita appena prima che la condizione venga nuovamente controllata.

Le parentesi graffe sono facoltative (è possibile una riga con un punto e virgola) se il ciclo contiene una sola istruzione. Tuttavia, è sempre consigliabile utilizzare le parentesi graffe per evitare equivoci e bug.

I componenti del ciclo for sono opzionali. Se la tua logica aziendale contiene una di queste parti, puoi omettere il componente corrispondente dal tuo ciclo 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 struttura for (;;) { function-body } è uguale ad un ciclo while (true) .

Nested For Loops

Qualsiasi istruzione di loop con un'altra istruzione loop all'interno chiamata loop annidato. Lo stesso modo per il loop che ha più loop interno è chiamato 'nested for loop'.

```
    for(;;){
        //Outer Loop Statements
        for(;;){
            //Inner Loop Statements
        }
        //Outer Loop Statements
    }
```

È possibile dimostrare che il ciclo annidato per stampare numeri a forma di triangolo.

```
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);
    }
 }
```

Mentre cicli

```
int i = 0;
while (i < 100) { // condition gets checked BEFORE the loop body executes
    System.out.println(i);
    i++;
}
```

Un ciclo while viene eseguito fintanto che la condizione tra parentesi è true . Questa viene anche chiamata la struttura del "ciclo di pre-test" poiché l'istruzione condizionale deve essere

soddisfatta prima che il corpo del ciclo principale venga eseguito ogni volta.

Le parentesi graffe sono facoltative se il ciclo contiene una sola istruzione, ma alcune convenzioni di stile di codifica preferiscono avere le parentesi a prescindere.

do ... while Loop

Il ciclo do...while differenzia da altri loop in quanto è garantito che venga eseguito **almeno una volta** . Viene anche chiamata la struttura "loop post-test" perché l'istruzione condizionale viene eseguita dopo il corpo del ciclo principale.

```
int i = 0;
do {
    i++;
    System.out.println(i);
} while (i < 100); // Condition gets checked AFTER the content of the loop executes.
```

In questo esempio, il ciclo verrà eseguito finché non verrà stampato il numero 100 (anche se la condizione è i < 100 e non i <= 100), poiché la condizione del ciclo viene valutata *dopo* l'esecuzione del ciclo.

Con la garanzia di almeno un'esecuzione, è possibile dichiarare le variabili all'esterno del ciclo e inizializzarle all'interno.

```
String theWord;
Scanner scan = new Scanner(System.in);
do {
    theWord = scan.nextLine();
} while (!theWord.equals("Bird"));

System.out.println(theWord);
```

In questo contesto, la theWord è definita al di fuori del ciclo, ma poiché è garantito avere un valore basato sul suo flusso naturale, la theWord verrà inizializzata.

Per ciascuno

Java SE 5

Con Java 5 e versioni successive, è possibile utilizzare per ogni ciclo, anche noto come for-loops avanzato:

```
List strings = new ArrayList();

strings.add("This");
strings.add("is");
strings.add("a for-each loop");

for (String string : strings) {
    System.out.println(string);
}
```

Per ogni loop può essere utilizzato per iterare su Array e implementazioni dell'interfaccia Iterable , il successivo include classi Collections , come List o Set .

La variabile di ciclo può essere di qualsiasi tipo che è assegnabile dal tipo di origine.

La variabile di ciclo per un ciclo Iterable<T> per Iterable<T> o T[] può essere di tipo S , se

- T extends S

- sia T che S sono tipi primitivi e assegnabili senza cast
- S è un tipo primitivo e T può essere convertito in un tipo assegnabile a S dopo la conversione di unboxing.
- T è un tipo primitivo e può essere convertito in S mediante la conversione in autoboxing.

Esempi:

```
T elements = ...
for (S s : elements) {
}
```

T	S	compilazioni
int []	lungo	sì
lungo[]	int	no
Iterable<Byte>	lungo	sì
Iterable<String>	CharSequence	sì
Iterable<CharSequence>	Stringa	no
int []	Lungo	no
int []	Numero intero	sì

Se altro

```
int i = 2;
if (i < 2) {
  System.out.println("i is less than 2");
} else {
  System.out.println("i is greater than 2");
}
```

Un'istruzione if esegue il codice condizionalmente a seconda del risultato della condizione tra parentesi. Quando la condizione tra parentesi è vera, entrerà nel blocco di if statement che è definito da parentesi graffe come { e } . la staffa di apertura fino alla staffa di chiusura è lo scopo dell'istruzione if.

Il blocco else è facoltativo e può essere omesso. Funziona se il if affermazione è false e non viene eseguito se il if affermazione è vera perché in quel caso if istruzione viene eseguita.

Vedi anche: Ternario Se

Passare la dichiarazione

L'istruzione switch è la dichiarazione di ramo a più vie di Java. E 'usato per prendere il posto di lunga if - else if - else catene, e renderli più leggibili. Tuttavia, a differenza delle dichiarazioni if , non si possono usare disuguaglianze; ogni valore deve essere definito concretamente.

Esistono tre componenti critici per l'istruzione switch :

- case : questo è il valore che viene valutato per l'equivalenza con l'argomento dell'istruzione switch .
- default : si tratta di un'espressione catch-all facoltativa, nel case cui nessuna delle espressioni case valuti come true .
- Completamento brusco della dichiarazione del case ; di solito break : ciò è necessario per

evitare la valutazione indesiderata di ulteriori dichiarazioni di un case .

Ad eccezione di continue , è possibile utilizzare qualsiasi istruzione che possa causare il completamento brusco di una dichiarazione . Ciò comprende:

- break
  - return
  - throw

Nell'esempio seguente, una tipica istruzione switch viene scritta con quattro possibili casi, incluso quello 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");
}
```

Omettendo la break o qualsiasi affermazione che avrebbe un completamento brusco, possiamo sfruttare i casi noti come "fall-through", che valutano rispetto a diversi valori. Questo può essere usato per creare intervalli per cui un valore ha successo, ma non è ancora flessibile come le disuguaglianze.

```
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");
}
```

In caso di foo == 1 l'output sarà:

```
I'm equal or greater than one
I'm one, two, or three
```

In caso di foo == 3 l'output sarà:

```
I'm one, two, or three
```

Java SE 5

L'istruzione switch può anche essere utilizzata 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

L'istruzione switch può anche essere utilizzata 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");
    }
}
```

Operatore ternario

A volte è necessario verificare una condizione e impostare il valore di una variabile.

Per es.

```
String name;

if (A > B) {
    name = "Billy";
} else {
    name = "Jimmy";
}
```

Questo può essere facilmente scritto in una riga come

```
String name = A > B ? "Billy" : "Jimmy";
```

Il valore della variabile è impostato sul valore immediatamente dopo la condizione, se la condizione è vera. Se la condizione è falsa, verrà assegnato il secondo valore alla variabile.

Rompere

L'istruzione break termina un ciclo (come for , while) o la valutazione di un'istruzione switch .

Ciclo continuo:

```
while(true) {
   if(someCondition == 5) {
       break;
   }
}
```

Il ciclo nell'esempio funzionerebbe per sempre. Ma quando una someCondition uguale a 5 in qualche punto dell'esecuzione, allora il ciclo termina.

Se i loop multipli sono in cascata, solo il loop più interno termina utilizzando break .

Prova ... Catch ... Finalmente

La struttura di controllo try { ... } catch (...) { ... } viene utilizzata per la gestione delle eccezioni .

```
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.");
}
```

Questo stamperebbe:

```
Inserimento non valido. 'abc' non è un numero intero valido.
```

Una finally clausola può essere aggiunto dopo la catch . La clausola finally sarebbe sempre stata eseguita, indipendentemente dal fatto che fosse stata lanciata un'eccezione.

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");
}
```

Questo stamperebbe:

```
Inserimento non valido. 'abc' non è un numero intero valido.
Questo codice verrà sempre eseguito, anche se viene generata un'eccezione
```

Interruzione / proseguimento annidato

È possibile break / continue con un ciclo esterno usando le istruzioni label:

```
outerloop:
for(...) {
    innerloop:
    for(...) {
        if(condition1)
            break outerloop;

        if(condition2)
            continue innerloop; // equivalent to: continue;
    }
}
```

Non c'è altro uso per le etichette in Java.

Continua la dichiarazione in Java

L'istruzione continue viene utilizzata per saltare i passaggi rimanenti nell'iterazione corrente e iniziare con l'iterazione del ciclo successivo. Il controllo passa dall'istruzione continue al valore di passo (incremento o decremento), se presente.

```
String[] programmers = {"Adrian", "Paul", "John", "Harry"};

    //john is not printed out
    for (String name : programmers) {
        if (name.equals("John"))
            continue;
        System.out.println(name);
    }
```

L'istruzione continue può anche fare in modo che il controllo del programma passi al valore di passo (se esiste) di un ciclo denominato:

```
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;
    }
}
```

Leggi Strutture di controllo di base online:
https://riptutorial.com/it/java/topic/118/strutture-di-controllo-di-base

Capitolo 163: sun.misc.Unsafe

Osservazioni

La classe Unsafe consente a un programma di eseguire operazioni che non sono consentite dal compilatore Java. I programmi normali dovrebbero evitare l'uso di Unsafe .

AVVERTENZE

1. Se commetti un errore utilizzando le API non Unsafe , le tue applicazioni potrebbero causare l'arresto anomalo della JVM e / o mostrare sintomi difficili da diagnosticare.
2. L'API Unsafe è soggetta a modifiche senza preavviso. Se lo si utilizza nel codice, potrebbe essere necessario riscrivere il codice quando si cambiano le versioni di Java.

Examples

Instantiating sun.misc.Unsafe tramite riflessione

```
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 ha un costruttore privato e il metodo statico getUnsafe() è protetto con un controllo del classloader per garantire che il codice sia stato caricato con il programma di caricamento di classe primario. Pertanto, un metodo per caricare l'istanza è utilizzare la riflessione per ottenere il campo statico.

Istanziazione di sun.misc.Unsafe tramite bootclasspath

```
public class UnsafeLoader {
    public static Unsafe loadUnsafe() {
        return Unsafe.getUnsafe();
    }
}
```

Mentre questo esempio verrà compilato, è probabile che fallisca in fase di esecuzione a meno che la classe Unsafe non sia stata caricata con il classloader primario. Per garantire che ciò accada, la JVM dovrebbe essere caricata con gli argomenti appropriati, come:

```
java -Xbootclasspath:$JAVA_HOME/jre/lib/rt.jar:./UnsafeLoader.jar foo.bar.MyApp
```

La classe foo.bar.MyApp può quindi utilizzare UnsafeLoader.loadUnsafe() .

Ottenere istanza di non sicuro

Unsafe è memorizzato come campo privato a cui non è possibile accedere direttamente. Il costruttore è privato e l'unico metodo per accedere al public static Unsafe getUnsafe() ha accesso privilegiato. Con l'uso della riflessione, c'è un work-around per rendere accessibili i campi privati:

```
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;
}
```

Usi di non sicuro

Alcuni usi non sicuri sono i seguenti:

Uso	API
Off heap / allocazione diretta della memoria, riallocazione e deallocazione	`allocateMemory(bytes)` , `reallocateMemory(address, bytes)` e `freeMemory(address)`
Recinti di memoria	`loadFence()` , `storeFence()` , `fullFence()`
Filo di corrente di parcheggio	`park(isAbsolute, time)` , `unpark(thread)`
Campo diretto e / o accesso alla memoria	`get*` e `put*` famiglia di metodi
Lancio di eccezioni non controllate	`throwException(e)`
CAS e operazioni atomiche	`compareAndSwap*` famiglia di metodi
Impostazione della memoria	`setMemory`
Operazioni volatili o concorrenti	`get*Volatile` , `put*Volatile` , `putOrdered*`

La famiglia di metodi get e put è relativa a un determinato oggetto. Se l'oggetto è nullo, viene trattato come un indirizzo assoluto.

```
// 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);
```

Alcuni metodi sono definiti solo per int e long. Puoi usare questi metodi su float e doubles usando floatToRawIntBits , intBitsToFloat, doubleToRawLongBits , longBitsToDouble`

Leggi sun.misc.Unsafe online: https://riptutorial.com/it/java/topic/6771/sun-misc-unsafe

Capitolo 164: super parola chiave

Examples

Uso di parole chiave super con esempi

la parola chiave super svolge un ruolo importante in tre punti

1. Livello costruttore
2. Livello del metodo
3. Livello variabile

Livello costruttore

super parola chiave super viene utilizzata per chiamare il costruttore della classe genitore. Questo costruttore può essere costruttore predefinito o costruttore parametrizzato.

- Costruttore predefinito: super();
- Costruttore parametrizzato: 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() deve essere la prima istruzione nel costruttore altrimenti otterremo il messaggio di errore di compilazione.

Livello del metodo

super parola chiave super può anche essere utilizzata in caso di sovrascrittura del metodo. super parola chiave super può essere utilizzata per invocare o chiamare il metodo della classe genitore.

```
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 : se non esiste una modalità di sovrascrittura, non è necessario utilizzare la parola chiave super per chiamare il metodo della classe genitore.

Livello variabile

super è usato per riferirsi alla variabile istanza della classe genitore immediata. In caso di ereditarietà, potrebbe esserci la possibilità che la classe base e la classe derivata possano avere membri di dati simili. Per differenziare tra il membro dati della classe base / genitore e classe derivata / figlia, nel contesto della classe derivata i dati della classe base i membri devono essere preceduti da una super parola chiave.

```
//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 : se non scriviamo la parola chiave super prima del nome del membro dei dati della classe di base, verrà riferito come membro dei dati della classe corrente e i membri dei dati della classe di base sono nascosti nel contesto della classe derivata.

Leggi super parola chiave online: https://riptutorial.com/it/java/topic/5764/super-parola-chiave

Capitolo 165: tabella hash

introduzione

Hashtable è una classe nelle raccolte Java che implementa l'interfaccia Mappa ed estende la classe Dizionario

Contiene solo elementi unici e la sua sincronizzazione

Examples

tabella hash

```
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);
   }
}
```

Leggi tabella hash online: https://riptutorial.com/it/java/topic/10709/tabella-hash

Capitolo 166: Test unitario

introduzione

Il test delle unità è parte integrante dello sviluppo basato sui test e una funzionalità importante per la creazione di qualsiasi applicazione affidabile. In Java, il testing delle unità viene eseguito quasi esclusivamente utilizzando librerie e framework esterni, la maggior parte dei quali ha il proprio tag di documentazione. Questo mozzicone serve come mezzo per introdurre il lettore agli strumenti disponibili e alla relativa documentazione.

Osservazioni

Quadri di test unitari

Esistono numerosi framework disponibili per il test delle unità all'interno di Java. L'opzione più popolare di gran lunga è JUnit. È documentato sotto il seguente:

JUnit

JUnit4 - *Tag proposto per le funzionalità di JUnit4; non ancora implementato* .

Esistono altri framework di test unitari e sono disponibili documentazione:

TestNG

Strumenti di test unitario

Ci sono molti altri strumenti usati per i test unitari:

Mockito - Mocking quadro; consente agli oggetti di essere imitati. Utile per simulare il comportamento **previsto** di un'unità esterna nel test di una determinata unità, in modo da non collegare il comportamento dell'unità esterna ai test dell'unità dati.

JBehave - BDD Framework. Consente di collegare i test ai comportamenti degli utenti (consentendo la convalida dei requisiti / degli scenari). *Nessun tag dei documenti disponibile al momento della scrittura; ecco un link esterno* .

Examples

Che cos'è il test unitario?

Questo è un po 'un primer. È per lo più messo perché la documentazione è costretta ad avere un esempio, anche se è inteso come un articolo stub. Se conosci già le nozioni di base sui test unitari, sentiti libero di andare avanti alle osservazioni, laddove vengono citati specifici framework.

Il test unitario garantisce che un determinato modulo si comporti come previsto. Nelle applicazioni su larga scala, garantire l'esecuzione appropriata dei moduli nel vuoto è parte integrante della garanzia della fedeltà dell'applicazione.

Considera il seguente (banale) pseudo-esempio:

```
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;
    }
  }
}
```

Quindi questo esempio è banale; DataReader i dati da un file, passa a Capitalizer , che converte tutti i caratteri in maiuscolo, che viene quindi passato al Consumer . Ma DataReader è fortemente collegato al nostro ambiente applicativo, quindi rimandiamo il collaudo di questa catena fino a quando non saremo pronti a implementare una versione di prova.

Ora, supponiamo, da qualche parte lungo il percorso in una versione, per ragioni sconosciute, il metodo getVal() in Capitalizer cambiato dalla restituzione di una stringa toUpperCase() a una stringa toLowerCase() :

```
  // Another team's module.
  class Capitalizer {
    ...

    public String getVal() {
      return dr.readVal().toLowerCase();
    }
  }
```

Chiaramente, questo rompe il comportamento previsto. Ma, a causa degli ardui processi coinvolti

con l'esecuzione di DataReader , non lo noteremo fino alla prossima implementazione del test. Così trascorrono giorni / settimane / mesi con questo bug che si trova nel nostro sistema, e quindi il product manager lo vede e si rivolge immediatamente a te, il leader del team associato al Consumer . "Perché sta succedendo questo? Che cosa avete cambiato voi ragazzi?" Ovviamente, sei senza tracce. Non hai idea di cosa sta succedendo. Non hai cambiato alcun codice che dovrebbe essere toccato da questo; perché è improvvisamente rotto?

Alla fine, dopo una discussione tra le squadre e la collaborazione, il problema viene tracciato e il problema risolto. Ma, si pone la domanda; come è stato possibile prevenirlo?

Ci sono due cose ovvie:

I test devono essere automatizzati

La nostra fiducia nel test manuale fa passare questo inosservato per troppo tempo. Abbiamo bisogno di un modo per automatizzare il processo in base al quale i bug vengono introdotti **all'istante** . Non 5 settimane da ora. Non più di 5 giorni. Non 5 minuti da ora. Proprio adesso.

Devi apprezzare che, in questo esempio, ho espresso un bug **molto banale** che è stato introdotto e inosservato. In un'applicazione industriale, con decine di moduli costantemente aggiornati, questi possono insinuarsi dappertutto. Correggere qualcosa con un modulo, solo per rendersi conto che lo stesso comportamento "aggiustato" era invocato in qualche modo altrove (internamente o esternamente).

Senza una convalida rigorosa, le cose si insinueranno nel sistema. E 'possibile che, se trascurata abbastanza lontano, questo si tradurrà in tanto lavoro in più cercando di risolvere i cambiamenti (e quindi fissare le correzioni, ecc), che un prodotto sarà effettivamente **aumentare** nel lavoro rimanente come lo sforzo è messo in esso. Non vuoi essere in questa situazione.

I test devono essere a grana fine

Il secondo problema notato nel nostro esempio precedente è la quantità di tempo impiegato per tracciare il bug. Il product manager ti ha inviato un ping quando i tester l'hanno notato, hai investigato e hai scoperto che il Capitalizer restituiva dati apparentemente cattivi, hai fatto il ping al team di Capitalizer con i tuoi risultati, hanno investigato, ecc. Ecc. Ecc.

Lo stesso punto che ho fatto sopra sulla quantità e la difficoltà di questo banale esempio qui. Ovviamente chiunque sia ragionevolmente esperto di Java potrebbe trovare rapidamente il problema introdotto. Ma spesso è molto, molto più difficile da rintracciare e comunicare problemi. Forse il team di Capitalizer ti ha fornito un JAR senza fonte. Forse si trovano dall'altra parte del mondo e le ore di comunicazione sono molto limitate (forse alle e-mail che vengono inviate una volta al giorno). Può portare a bug che impiegano settimane o più a tracciare (e, di nuovo, potrebbero esserci molti di questi per una data release).

Al fine di mitigare questo, vogliamo test rigorosi il più **fine** possibile (si desidera anche test a grana grossa per garantire che i moduli interagiscano correttamente, ma non è questo il nostro punto focale qui). Vogliamo specificare rigorosamente come funziona tutta la funzionalità rivolta verso l'esterno (almeno) e testare tale funzionalità.

Inserisci il test unitario

Immagina se avessimo un test, in particolare assicurando che il metodo getVal() di Capitalizer restituito una stringa in maiuscolo per una determinata stringa di input. Inoltre, immagina che il test sia stato eseguito prima ancora che avessimo commesso un codice. Il bug introdotto nel sistema (ovvero toUpperCase() viene sostituito con toLowerCase()) non causerebbe problemi perché il bug non sarebbe mai **stato** introdotto nel sistema. Lo prenderemmo in un test, lo sviluppatore avrebbe (si spera) realizzato il proprio errore e sarebbe stata raggiunta una soluzione alternativa su come introdurre l'effetto desiderato.

Ci sono alcune omissioni qui riportate su **come** implementare questi test, ma quelli sono coperti nella documentazione specifica del framework (collegata nelle osservazioni). Si spera che questo sia un esempio del **perché il** test unitario è importante.

Leggi Test unitario online: https://riptutorial.com/it/java/topic/8155/test-unitario

Capitolo 167: ThreadLocal

Osservazioni

Ideale per oggetti che dipendono da internals durante il richiamo di una chiamata, ma altrimenti sono stateless, come SimpleDateFormat , Marshaller

Per l'utilizzo di ThreadLocal Random , prendere in considerazione l'utilizzo di ThreadLocalRandom

Examples

Inizializzazione funzionale di Java 8 ThreadLocal

```
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);
    }
}
```

Utilizzo thread di base

Java ThreadLocal è usato per creare variabili locali del thread. È noto che i thread di un oggetto condividono le sue variabili, quindi la variabile non è thread-safe. Possiamo usare la sincronizzazione per la sicurezza dei thread, ma se vogliamo evitare la sincronizzazione, ThreadLocal ci permette di creare variabili che sono locali al thread, cioè solo che quel thread può leggere o scrivere su quelle variabili, quindi gli altri thread eseguono lo stesso pezzo di codice non sarà in grado di accedere alle altre variabili ThreadLocal.

Questo può essere usato possiamo usare variabili ThreadLocal . in situazioni in cui si dispone di un pool di thread, ad esempio in un servizio Web. Ad esempio, la creazione di un oggetto SimpleDateFormat ogni volta per ogni richiesta richiede molto tempo e non è possibile creare uno statico in quanto SimpleDateFormat non è thread-safe, quindi possiamo creare un ThreadLocal in modo da poter eseguire operazioni thread-safe senza l'overhead di creare SimpleDateFormat ogni tempo.

Il seguente codice mostra come può essere usato:

Ogni thread ha la propria variabile ThreadLocal e possono usare i suoi metodi get() e set() per ottenere il valore predefinito o cambiarne il valore local in Thread.

ThreadLocal istanze ThreadLocal sono in genere campi statici privati in classi che desiderano associare lo stato a un thread.

Ecco un piccolo esempio che mostra l'uso di ThreadLocal nel programma java e dimostra che ogni thread ha la propria copia della variabile 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());
    }

}
```

Produzione:

```
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
```

Come possiamo vedere dall'output, Thread-0 ha cambiato il valore di formattatore, ma il formattatore di default di thread-2 è lo stesso del valore inizializzato.

Più thread con un oggetto condiviso

In questo esempio abbiamo un solo oggetto ma è condiviso tra / eseguito su thread differenti. L'uso normale dei campi per salvare lo stato non sarebbe possibile perché anche l'altro thread lo vedrebbe (o probabilmente non vedrebbe).

```
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();
    }
}
```

In Foo contiamo partendo da zero. Invece di salvare lo stato in un campo, memorizziamo il nostro numero corrente nell'oggetto ThreadLocal che è staticamente accessibile. Si noti che la sincronizzazione in questo esempio non è correlata all'utilizzo di ThreadLocal ma assicura un output della console migliore.

```
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) {
```

```
                }
            }
        }
    }
}
```

Dall'output possiamo vedere che ogni thread conta per se stesso e non usa il valore dell'altro:

```
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
```

Leggi ThreadLocal online: https://riptutorial.com/it/java/topic/2001/threadlocal

Capitolo 168: Tipi atomici

introduzione

I tipi atomici di Java sono semplici tipi mutabili che forniscono operazioni di base che sono thread-safe e atomiche senza ricorrere al locking. Sono concepiti per l'uso nei casi in cui il blocco sarebbe un collo di bottiglia di concorrenza, o dove vi è il rischio di deadlock o livelock.

Parametri

Parametro	Descrizione
impostato	Set volatile del campo
ottenere	Lettura volatile del campo
lazySet	Questa è un'operazione ordinata dal punto vendita del campo
compareAndSet	Se il valore è il valore di expansion, quindi inviato al nuovo valore
getAndSet	ottenere il valore corrente e aggiornarlo

Osservazioni

Molti su essenzialmente combinazioni di letture volatili o scritture e operazioni CAS . Il modo migliore per capire questo è guardare direttamente il codice sorgente. Ad esempio AtomicInteger , Unsafe.getAndSet

Examples

Creazione di tipi atomici

Per il codice multi-threaded semplice, l'utilizzo della sincronizzazione è accettabile. Tuttavia, l'utilizzo della sincronizzazione ha un impatto di vivacità e, man mano che la base di codice diventa più complessa, aumenta la probabilità che ti ritroverai con Deadlock , Starvation o Livelock .

In caso di concorrenza più complessa, l'utilizzo di variabili atomiche è spesso un'alternativa migliore, poiché consente di accedere a una singola variabile in modo thread-safe senza il sovraccarico dell'utilizzo di metodi o blocchi di codice sincronizzati.

Creazione di un tipo AtomicInteger :

```
AtomicInteger aInt = new AtomicInteger() // Create with default value 0

AtomicInteger aInt = new AtomicInteger(1) // Create with initial value 1
```

Allo stesso modo per altri tipi di istanze.

```
AtomicIntegerArray aIntArray = new AtomicIntegerArray(10) // Create array of specific length
AtomicIntegerArray aIntArray = new AtomicIntegerArray(new int[] {1, 2, 3}) // Initialize array
with another array
```

Allo stesso modo per altri tipi atomici.

C'è un'eccezione notevole che non ci sono float e double types. Questi possono essere simulati usando Float.floatToIntBits(float) e Float.intBitsToFloat(int) per float , nonché Double.doubleToLongBits(double) e Double.longBitsToDouble(long) per i doppi.

Se si desidera utilizzare sun.misc.Unsafe è possibile utilizzare qualsiasi variabile primitiva come atomica utilizzando l'operazione atomica in sun.misc.Unsafe . Tutti i tipi primitivi devono essere convertiti o codificati in int o long per utilizzarli in questo modo. Per maggiori informazioni su questo: sun.misc.Unsafe .

Motivazione per i tipi atomici

Il modo più semplice per implementare applicazioni multi-thread è utilizzare la sincronizzazione integrata e le primitive di blocco di Java; ad esempio la parola chiave synchronized . L'esempio seguente mostra come possiamo usare synchronized per accumulare conteggi.

```
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;
    }
}
```

Questa implementazione funzionerà correttamente. Tuttavia, se si dispone di un numero elevato di thread che effettuano molte chiamate simultanee sullo stesso oggetto Counters , è possibile che la sincronizzazione sia un collo di bottiglia. In particolare:

1. Ogni chiamata al metodo synchronized inizierà con il thread corrente acquisendo il blocco per l'istanza Counters .
2. Il thread manterrà il blocco mentre controlla il valore number e aggiorna il contatore.
3. Infine, rilascerà il blocco, consentendo l'accesso ad altri thread.

Se un thread tenta di acquisire il blocco mentre un altro lo trattiene, il thread tentante verrà bloccato (arrestato) al passaggio 1 finché il blocco non verrà rilasciato. Se più thread sono in attesa, uno di loro lo otterrà e gli altri continueranno a essere bloccati.

Questo può portare a un paio di problemi:

- Se c'è un sacco di *contesa* per il lock (cioè un sacco di thread tenta di acquisirlo), quindi alcuni thread possono essere bloccati per un lungo periodo di tempo.
- Quando un thread è bloccato in attesa del blocco, il sistema operativo in genere prova a passare l'esecuzione a un thread diverso. Questo *cambio di contesto* comporta un impatto relativamente grande sulle prestazioni del processore.
- Quando ci sono più thread bloccati sullo stesso lock, non ci sono garanzie che ognuno di

essi verrà trattato "abbastanza" (cioè ogni thread è garantito che sia programmato per essere eseguito). Questo può portare alla *fame di thread* .

Come si implementa i tipi atomici?

Cominciamo riscrivendo l'esempio sopra usando i contatori 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;
    }
}
```

Abbiamo sostituito int[] con un AtomicInteger[] e inizializzato con un'istanza in ogni elemento. Abbiamo anche aggiunto le chiamate a incrementAndGet() e get() al posto delle operazioni sui valori int .

Ma la cosa più importante è che possiamo rimuovere la parola chiave synchronized perché il blocco non è più necessario. Questo funziona perché le operazioni incrementAndGet() e get() sono *atomiche* e *thread-safe* . In questo contesto, significa che:

- Ogni contatore dell'array sarà solo *osservabile* nello stato "prima" di un'operazione (come un "incremento") o nello stato "dopo".
- Supponendo che l'operazione avvenga al tempo T , nessun thread sarà in grado di vedere lo stato "prima" dopo il tempo T

Inoltre, mentre due thread potrebbero effettivamente tentare di aggiornare la stessa istanza di AtomicInteger nello stesso momento, le implementazioni delle operazioni assicurano che solo un incremento si verifica alla volta nell'istanza specificata. Questo viene fatto senza blocco, spesso con prestazioni migliori.

Come funzionano i tipi atomici?

Tipicamente i tipi atomici si basano su istruzioni hardware specializzate nel set di istruzioni della macchina target. Ad esempio, i set di istruzioni basati su Intel forniscono un'istruzione CAS (Compare and Swap) che eseguirà una sequenza specifica di operazioni di memoria atomicamente.

Queste istruzioni di basso livello vengono utilizzate per implementare operazioni di livello

superiore nelle API delle rispettive classi AtomicXxx . Ad esempio, (di nuovo, in pseudocodice C-like):

```
private volatile num;

int increment() {
  while (TRUE) {
    int old = num;
    int new = old + 1;
    if (old == compare_and_swap(&num, old, new)) {
      return new;
    }
  }
}
```

Se non c'è contesa su AtomicXxxx , il test if avrà esito positivo e il ciclo terminerà immediatamente. Se c'è contesa, il if fallirà per tutti tranne uno dei thread, e "ruoterà" nel ciclo per un piccolo numero di cicli del ciclo. In pratica, la rotazione è più veloce di ordini di grandezza (eccetto che per livelli *non realistici* di contesa, dove le prestazioni sincronizzate sono migliori rispetto alle classi atomiche perché quando l'operazione CAS fallisce, il tentativo di aggiungere altro contesa) che sospendere il thread e passare a un altro uno.

Per inciso, le istruzioni CAS vengono in genere utilizzate da JVM per implementare il *blocco non mirato* . Se la JVM può vedere che un blocco non è attualmente bloccato, tenterà di utilizzare un CAS per acquisire il blocco. Se il CAS ha esito positivo, non è necessario eseguire la costosa programmazione dei thread, il cambio di contesto e così via. Per ulteriori informazioni sulle tecniche utilizzate, vedere Bloccaggio di parte in HotSpot .

Leggi Tipi atomici online: https://riptutorial.com/it/java/topic/5963/tipi-atomici

Capitolo 169: Tipi di dati di riferimento

Examples

Istanziare un tipo di riferimento

```
Object obj = new Object(); // Note the 'new' keyword
```

Dove:

- Object è un tipo di riferimento.
- obj è la variabile in cui memorizzare il nuovo riferimento.
- Object() è la chiamata a un costruttore di Object .

Che succede:

- Lo spazio in memoria è allocato per l'oggetto.
- Il costruttore Object() viene chiamato per inizializzare quello spazio di memoria.
- L'indirizzo di memoria è memorizzato in obj , in modo che faccia *riferimento* all'oggetto appena creato.

Questo è diverso dai primitivi:

```
int i = 10;
```

Dove il valore effettivo 10 è memorizzato in i .

dereferenziazione

Il dereferenziamento avviene con . operatore:

```
Object obj = new Object();
String text = obj.toString(); // 'obj' is dereferenced.
```

Il dereferenziamento *segue* l'indirizzo di memoria memorizzato in un riferimento, nel luogo in memoria in cui risiede l'oggetto reale. Quando un oggetto è stato trovato, viene chiamato il metodo richiesto (toString in questo caso).

Quando un riferimento ha il valore null , la dereferenziazione risulta in una NullPointerException :

```
Object obj = null;
obj.toString(); // Throws a NullpointerException when this statement is executed.
```

null indica l'assenza di un valore, ovvero il *seguente* indirizzo di memoria non conduce da nessuna parte. Quindi non esiste alcun oggetto su cui possa essere chiamato il metodo richiesto.

Leggi Tipi di dati di riferimento online: https://riptutorial.com/it/java/topic/1046/tipi-di-dati-di-riferimento

Capitolo 170: Tipi di dati primitivi

introduzione

Gli 8 tipi di dati primitivi byte , short , int , long , char , boolean , float e double sono i tipi che memorizzano la maggior parte dei dati numerici grezzi nei programmi Java.

Sintassi

- int aInt = 8; // La parte di definizione (numero) di questa dichiarazione int è detta letterale.
- int hexInt = 0x1a; // = 26; È possibile definire valori letterali con valori esadecimali preceduti da *0x* .
- int binInt = 0b11010; // = 26; È inoltre possibile definire valori letterali binari; prefisso con *0b* .
- long good Long = 10000000000L; // Per impostazione predefinita, i valori letterali interi sono di tipo int. Aggiungendo la L alla fine del letterale stai dicendo al compilatore che il letterale è lungo. Senza questo il compilatore genererebbe un errore "Numero intero troppo grande".
- double aDouble = 3,14; // I letterali a virgola mobile sono di tipo double per impostazione predefinita.
- float aFloat = 3.14F; // Per impostazione predefinita questo letterale sarebbe stato un double (e causato un errore "Tipi incompatibili"), ma aggiungendo una F diciamo al compilatore che è un float.

Osservazioni

Java ha 8 *tipi di dati primitivi* , vale a dire boolean , byte , short , char , int , long , float e double . (Tutti gli altri tipi sono tipi di *riferimento,* inclusi tutti i tipi di array e tipi / classi di oggetti incorporati che hanno un significato speciale nel linguaggio Java, ad esempio String , Class e Throwable e le relative sottoclassi.)

Il risultato di tutte le operazioni (addizione, sottrazione, moltiplicazione, ecc.) Su un tipo primitivo è almeno un int , quindi aggiungendo un short a un short produce un int , come aggiungere un byte a un byte o un char a un char . Se vuoi assegnare il risultato di quel ritorno ad un valore dello stesso tipo, devi lanciarlo. per esempio

```
byte a = 1;
byte b = 2;
byte c = (byte) (a + b);
```

La mancata esecuzione dell'operazione comporterà un errore di compilazione.

Ciò è dovuto alla seguente parte della Java Language Spec, §2.11.1 :

> Un compilatore codifica carichi di valori letterali di tipi byte e short utilizzando le istruzioni Java Virtual Machine che firmano-estendono tali valori ai valori di tipo int al momento della compilazione o in fase di esecuzione. Carichi di valori letterali di tipi boolean e char sono codificati usando istruzioni che estendono a zero il valore letterale a un valore di tipo int al momento della compilazione o in fase di esecuzione. [..]. Quindi, la maggior parte delle operazioni sui valori di tipi reali boolean , byte , char e short vengono eseguite correttamente mediante istruzioni che operano su valori di tipo computazionale int .

La ragione di ciò è specificata anche in quella sezione:

> Data la **dimensione dell'opcode a byte singolo** di Java Virtual Machine, i tipi di codifica in opcode mettono pressione sul design del set di istruzioni. Se ogni istruzione digitata supportava tutti i tipi di dati di runtime della Java Virtual Machine, ci sarebbero più istruzioni di quante potrebbero essere rappresentate in un byte . [...] È possibile utilizzare istruzioni separate per convertire tra tipi di dati non supportati e supportati, se necessario.

Examples

Il primitivo int

Un tipo di dati primitivo come int detiene i valori direttamente nella variabile che lo utilizza, nel frattempo una variabile che è stata dichiarata utilizzando Integer contiene un riferimento al valore.

In base all'API java : "La classe Integer racchiude un valore del tipo primitivo int in un oggetto. Un oggetto di tipo Integer contiene un singolo campo il cui tipo è int."

Per impostazione predefinita, int è un numero intero con int a 32 bit. Può memorizzare un valore minimo di -2^{31} e un valore massimo di $2^{31} - 1$.

```
int example = -42;
int myInt = 284;
int anotherInt = 73;

int addedInts = myInt + anotherInt; // 284 + 73 = 357
int subtractedInts = myInt - anotherInt; // 284 - 73 = 211
```

Se è necessario memorizzare un numero al di fuori di questo intervallo, dovrebbe essere usato a long . Il superamento dell'intervallo di valori di int causa un overflow di un intero, causando il valore che supera l'intervallo da aggiungere al sito opposto dell'intervallo (positivo diventa negativo e viceversa). Il valore è ((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
```

I valori massimi e minimi di int possono essere trovati a:

```
int high = Integer.MAX_VALUE;    // high == 2147483647
int low = Integer.MIN_VALUE;     // low == -2147483648
```

Il valore predefinito di un int è *0*

```
int defaultInt;    // defaultInt == 0
```

Il breve primitivo

Un short è un intero con segno a 16 bit. Ha un valore minimo di -2^{15} (-32.768) e un valore massimo di $2^{15} -1$ (32.767)

```
short example = -48;
short myShort = 987;
short anotherShort = 17;

short addedShorts = (short) (myShort + anotherShort); // 1,004
```

```
short subtractedShorts = (short) (myShort - anotherShort); // 970
```

I valori massimi e minimi del short possono essere trovati a:

```
short high = Short.MAX_VALUE;        // high == 32767
short low = Short.MIN_VALUE;         // low == -32768
```

Il valore predefinito di un short è *0*

```
short defaultShort;    // defaultShort == 0
```

Il lungo primitivo

Per impostazione predefinita, long è un intero con segno a 64 bit (in Java 8, può essere firmato o non firmato). Firmato, può memorizzare un valore minimo di -2^{63} e un valore massimo di $2^{63} - 1$, e senza segno può memorizzare un valore minimo di 0 e un valore massimo di $2^{64} - 1$

```
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
```

I valori massimo e minimo di long possono essere trovati a:

```
long high = Long.MAX_VALUE;    // high == 9223372036854775807L
long low = Long.MIN_VALUE;     // low == -9223372036854775808L
```

Il valore predefinito di long è *0L*

```
long defaultLong;    // defaultLong == 0L
```

Nota: la lettera "L" aggiunta alla fine del letterale long fa distinzione tra maiuscole e minuscole, tuttavia è buona norma utilizzare il capitale in quanto è più semplice distinguere dal primo:

```
2L == 2l;            // true
```

Avviso: cache Java istanze di oggetti interi nell'intervallo da -128 a 127. Il ragionamento è spiegato qui: https://blogs.oracle.com/darcy/entry/boxing_and_caches_integer_valueof

I seguenti risultati possono essere trovati:

```
Long val1 = 127L;
Long val2 = 127L;

System.out.println(val1 == val2); // true

Long val3 = 128L;
Long val4 = 128L;
```

```
System.out.println(val3 == val4); // false
```

Per confrontare correttamente 2 valori Long Object, utilizzare il codice seguente (da Java 1.7 in poi):

```
Long val3 = 128L;
Long val4 = 128L;

System.out.println(Objects.equal(val3, val4)); // true
```

Confrontare un primitivo lungo ad un oggetto lungo non darà come risultato un falso negativo come confrontare 2 oggetti con ==.

Il primitivo booleano

Un boolean può memorizzare uno dei due valori, 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
```

Il valore predefinito di un boolean è *falso*

```
boolean defaultBoolean;    // defaultBoolean == false
```

Il byte primitivo

Un byte è un intero con byte a 8 bit. Può memorizzare un valore minimo di -2^7 (-128) e un valore massimo di $2^7 - 1$ (127)

```
byte example = -36;
byte myByte = 96;
byte anotherByte = 7;

byte addedBytes = (byte) (myByte + anotherByte); // 103
byte subtractedBytes = (byte) (myBytes - anotherByte); // 89
```

I valori massimi e minimi di byte possono essere trovati su:

```
byte high = Byte.MAX_VALUE;        // high == 127
byte low = Byte.MIN_VALUE;         // low == -128
```

Il valore predefinito di un byte è *0*

```
byte defaultByte;    // defaultByte == 0
```

Il galleggiante primitivo

Un float è un numero a virgola mobile IEEE 754 a 32 bit a precisione singola. Per impostazione predefinita, i decimali vengono interpretati come doppi. Per creare un float , è sufficiente aggiungere un f al letterale decimale.

```
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
```

I float gestiscono le cinque operazioni aritmetiche comuni: addizione, sottrazione, moltiplicazione, divisione e modulo.

Nota: quanto segue può variare leggermente a causa di errori in virgola mobile. Alcuni risultati sono stati arrotondati per chiarezza e leggibilità (il risultato stampato dell'esempio di addizione era in realtà 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
```

A causa del modo in cui i numeri in virgola mobile vengono memorizzati (ad esempio in formato binario), molti numeri non hanno una rappresentazione esatta.

```
float notExact = 3.1415926f;
System.out.println(notExact); // 3.1415925
```

Sebbene l'uso di float bene per la maggior parte delle applicazioni, non è necessario utilizzare né float né double per memorizzare rappresentazioni esatte di numeri decimali (come importi monetari) o numeri in cui è richiesta una maggiore precisione. Invece, dovrebbe essere usata la classe BigDecimal .

Il valore predefinito di un float è *0.0f* .

```
float defaultFloat;    // defaultFloat == 0.0f
```

Un float è preciso a circa un errore di 1 su 10 milioni.

Nota: Float.POSITIVE_INFINITY , Float.NEGATIVE_INFINITY , Float.NaN sono float valori. NaN sta per risultati di operazioni che non possono essere determinate, come la divisione di 2 valori infiniti. Inoltre 0f e -0f sono diversi, ma == si -0f vero:

```
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
```

Il doppio primitivo

Un double è un numero a virgola mobile IEEE 754 a 64 bit a doppia precisione.

```
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
```

A causa del modo in cui vengono memorizzati i numeri in virgola mobile, molti numeri non hanno una rappresentazione esatta.

```
double notExact = 1.32 - 0.42; // result should be 0.9
System.out.println(notExact); // 0.9000000000000001
```

Mentre l'uso del double va bene per la maggior parte delle applicazioni, non è necessario utilizzare né float né double per memorizzare numeri precisi come la valuta. Invece, dovrebbe essere usata la classe BigDecimal

Il valore predefinito di un double è *0.0d*

```
public double defaultDouble;    // defaultDouble == 0.0
```

Nota: Double.POSITIVE_INFINITY , Double.NEGATIVE_INFINITY , Double.NaN sono valori double . NaN sta per risultati di operazioni che non possono essere determinate, come la divisione di 2 valori infiniti. Inoltre 0d e -0d sono diversi, ma == si -0d vero:

```
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
```

Il primitivo char

Un char può memorizzare un singolo carattere Unicode a 16 bit. Un letterale di carattere è racchiuso tra virgolette singole

```
char myChar = 'u';
char myChar2 = '5';
char myChar3 = 65; // myChar3 == 'A'
```

Ha un valore minimo di \u0000 (0 nella rappresentazione decimale, chiamata anche *carattere null*) e un valore massimo di \uffff (65.535).

Il valore predefinito di un char è \u0000 .

```
char defaultChar;    // defaultChar == \u0000
```

Per definire un char di ' valore, è necessario utilizzare una sequenza di escape (carattere preceduto da una barra rovesciata):

```
char singleQuote = '\'';
```

Ci sono anche altre sequenze di 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
```

È possibile dichiarare un char di qualsiasi carattere Unicode.

```
char heart = '\u2764';
System.out.println(Character.toString(heart)); // Prints a line containing "❤".
```

È anche possibile aggiungere a un char . ad esempio per scorrere tutte le lettere minuscole, è possibile fare quanto segue:

```
for (int i = 0; i <= 26; i++) {
    char letter = (char) ('a' + i);
    System.out.println(letter);
}
```

Rappresentazione del valore negativo

Java e molti altri linguaggi memorizzano numeri interi negativi in una rappresentazione chiamata notazione *complemento a 2* .

Per una rappresentazione binaria univoca di un tipo di dati utilizzando n bit, i valori sono codificati in questo modo:

I bit meno significativi di n-1 memorizzano un numero integrale positivo x in rappresentazione integrale. Il valore più significativo memorizza un valore di bit vith s . Il valore ripreso da quei bit è

$x - s * 2^{n-1}$

cioè se il bit più significativo è 1, allora un valore che è solo di 1 più grande del numero che potresti rappresentare con gli altri bit ($2^{n-2} + 2^{n-3} + ... + 2^{1} + 2^{0} = 2^{n-1} - 1$) viene sottratto consentendo una rappresentazione binaria univoca per ogni valore da -2^{n-1} (s = 1; x = 0) a $2^{n-1} - 1$ (s = 0; x = $2^{n-1} - 1$).

Questo ha anche il bell'effetto collaterale, che è possibile aggiungere le rappresentazioni binarie come se fossero numeri binari positivi:

$$v1 = x1 - s1 * 2^{n-1}$$
$$v2 = x2 - s2 * 2^{n-1}$$

s1	s2	x1 + x2 overflow	risultato aggiuntivo
0	0	No	x1 + x2 = v1 + v2
0	0	sì	troppo grande per essere rappresentata con il tipo di dati (overflow)
0	1	No	$x1 + x2 - 2^{n-1} = x1 + x2 - s2 * 2^{n-1} = v1 + v2$
0	1	sì	$(x1 + x2) \bmod 2^{n-1} = x1 + x2 - 2^{n-1} = v1 + v2$
1	0	*	vedi sopra (scambia i contatti)
1	1	No	troppo piccolo per essere rappresentato con il tipo di dati ($x1 + x2 - 2^{n} < -2^{n-1}$; underflow)
1	1	sì	$(x1 + x2) \bmod 2^{n-1} - 2^{n-1} = (x1 + x2 - 2^{n-1}) - 2^{n-1} = (x1 - s1 * 2^{n-1}) + (x2 - s2 * 2^{n-1}) = v1 + v2$

Si noti che questo fatto rende facile trovare la rappresentazione binaria dell'inverso additivo (cioè il valore negativo):

Osservare che aggiungendo il complemento bit per bit al numero risulta che tutti i bit siano 1. Ora aggiungi 1 per rendere il valore overflow e ottieni l'elemento neutro 0 (tutti i bit 0).

Quindi il valore negativo di un numero i può essere calcolata utilizzando (ignorando possibile promozione a int qui)

```
(~i) + 1
```

Esempio: prendendo il valore negativo di 0 (byte):

Il risultato della negazione di 0 è 11111111 . L'aggiunta di 1 fornisce un valore di 100000000 (9 bit). Poiché un byte può solo memorizzare 8 bit, il valore più a sinistra viene troncato e il risultato è 00000000

Originale	Processi	Risultato
0 (00000000)	Negare	-0 (11111111)
11111111	Aggiungi 1 al binario	100000000
100000000	Tronca a 8 bit	00000000 (-0 è uguale a 0)

Consumo di memoria dei primitivi rispetto ai primitivi inscatolati

Primitivo	Tipo in scatola	Dimensione della memoria di primitivo / in scatola
booleano	booleano	1 byte / 16 byte
byte	Byte	1 byte / 16 byte
corto	Corto	2 byte / 16 byte
carbonizzare	carbonizzare	2 byte / 16 byte
int	Numero intero	4 byte / 16 byte
lungo	Lungo	8 byte / 16 byte
galleggiante	Galleggiante	4 byte / 16 byte
Doppio	Doppio	8 byte / 16 byte

Gli oggetti in scatola richiedono sempre 8 byte per tipo e gestione della memoria, e poiché la dimensione degli oggetti è sempre un multiplo di 8, *tutti i* tipi di box *richiedono 16 byte totali* . *Inoltre* , ogni utilizzo di un oggetto scatolato comporta la memorizzazione di un riferimento che tiene conto di altri 4 o 8 byte, a seconda delle opzioni JVM e JVM.

Nelle operazioni a uso intensivo di dati, il consumo di memoria può avere un notevole impatto sulle prestazioni. Il consumo di memoria aumenta ulteriormente quando si utilizzano gli array: un array float[5] richiede solo 32 byte; mentre un Float[5] memorizza 5 valori distinti non nulli richiederà un totale di 112 byte (su 64 bit senza puntatori compressi, questo aumenta a 152 byte).

Cache di valore in scatola

Gli overheads spaziali dei tipi di box possono essere mitigati in misura maggiore dalle cache del valore in box. Alcuni tipi di box implementano una cache di istanze. Ad esempio, per impostazione predefinita, la classe Integer memorizzerà nella cache le istanze per rappresentare numeri nell'intervallo da -128 a +127 . Ciò, tuttavia, non riduce il costo aggiuntivo derivante dall'indirizzamento aggiuntivo della memoria.

Se si crea un'istanza di un tipo in box mediante autoboxing o chiamando il valueOf(primitive) statico valueOf(primitive) , il sistema runtime tenterà di utilizzare un valore memorizzato nella cache. Se l'applicazione utilizza molti valori nell'intervallo che viene memorizzato nella cache, questo può ridurre in modo sostanziale la penalità della memoria dell'utilizzo di tipi di box. Certamente, se crei istanze di valore in box "a mano", è meglio usare valueOf piuttosto che new . (La new operazione crea sempre una nuova istanza.) Se, tuttavia, la maggior parte dei valori *non* si *trova* nell'intervallo memorizzato nella cache, può essere più veloce chiamare la new e salvare la ricerca della cache.

Conversione di primitivi

In Java, possiamo convertire tra valori interi e valori a virgola mobile. Inoltre, poiché ogni carattere corrisponde a un numero nella codifica Unicode, i tipi di char possono essere convertiti in e da tipi interi e in virgola mobile. boolean è l'unico tipo di dati primitivo che non può essere convertito in o da qualsiasi altro tipo di dati primitivo.

Esistono due tipi di conversioni: *conversione allargata* e *conversione restringimento* .

Una *conversione allargata* è quando un valore di un tipo di dati viene convertito in un valore di un altro tipo di dati che occupa più bit rispetto al primo. In questo caso non vi è alcun problema di perdita di dati.

Corrispondentemente, una *conversione restringimento* è quando un valore di un tipo di dati viene convertito in un valore di un altro tipo di dati che occupa meno bit del primo. La perdita di dati può verificarsi in questo caso.

Java esegue automaticamente l' *ampliamento delle conversioni* . Ma se si desidera eseguire una *conversione di restringimento* (se si è certi che non si verificherà alcuna perdita di dati), è possibile forzare Java ad eseguire la conversione utilizzando un costrutto linguistico noto come cast .

Conversione allargata:

```
int a = 1;
double d = a;    // valid conversion to double, no cast needed (widening)
```

Riduzione della conversione:

```
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
```

Cheatsheet Tipi primitivi

Tabella che mostra le dimensioni e l'intervallo di valori di tutti i tipi primitivi:

tipo di dati	rappresentazione numerica	intervallo di valori	valore predefinito
booleano	n / A	falso e vero	falso
byte	Firmato a 8 bit	-2^7 a $2^7 - 1$	0
		Da -128 a +127	
corto	Firmato a 16 bit	-2^{15} a $2^{15} - 1$	0
		Da -32.768 a +32.767	
int	Firmato a 32 bit	-2^{31} a $2^{31} - 1$	0
		-2,147,483,648 a +2,147,483,647	
lungo	Firmato a 64 bit	-2^{63} a $2^{63} - 1$	0L
		-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	
galleggiante	32 virgola mobile	1.401298464e-45 a 3.402823466e + 38 (positivo o negativo)	0.0f
Doppio	Virgola mobile a 64 bit	4.94065645841246544e-324d a 1.79769313486231570e + 308d	0.0D

tipo di dati	rappresentazione numerica	intervallo di valori	valore predefinito
		(positivo o negativo)	
carbonizzare	16 bit senza segno	Da 0 a $2^{16} - 1$	0
		Da 0 a 65.535	

Gli appunti:

1. La specifica del linguaggio Java richiede che i tipi integrali firmati (da byte a long) utilizzino la rappresentazione binaria del complemento a due e che i tipi a virgola mobile utilizzino rappresentazioni in virgola mobile binarie IEE 754 standard.
2. Java 8 e versioni successive forniscono metodi per eseguire operazioni aritmetiche non firmate su int e long . Sebbene questi metodi consentano a un programma di *trattare i* valori dei rispettivi tipi come non firmati, i tipi restano tipi firmati.
3. Il più piccolo punto fluttuante mostrato sopra è *subnormale* ; cioè hanno meno precisione di un valore *normale* . I numeri normali più piccoli sono 1.175494351e-38 e 2.2250738585072014e-308
4. Un char rappresenta convenzionalmente *un'unità di codice* Unicode / UTF-16.
5. Sebbene un boolean contenga solo un bit di informazione, la sua dimensione in memoria varia a seconda dell'implementazione di Java Virtual Machine (si veda il tipo booleano).

Leggi Tipi di dati primitivi online: https://riptutorial.com/it/java/topic/148/tipi-di-dati-primitivi

Capitolo 171: Tipi di riferimento

Examples

Diversi tipi di riferimento

java.lang.ref pacchetto java.lang.ref fornisce classi di oggetti di riferimento, che supportano un grado limitato di interazione con il garbage collector.

Java ha quattro tipi di riferimento principali. Loro sono:

- Forte riferimento
- Riferimento debole
- Riferimento morbido
- Riferimento Phantom

1. Forte riferimento

Questa è la solita forma di creazione di oggetti.

```
MyObject myObject = new MyObject();
```

Il titolare variabile ha un forte riferimento all'oggetto creato. Finché questa variabile è MyObject e contiene questo valore, l' MyObject non verrà raccolta dal garbage collector.

2. Riferimento debole

Quando non si desidera mantenere un oggetto più lungo e si deve cancellare / liberare la memoria allocata per un oggetto il prima possibile, questo è il modo per farlo.

```
WeakReference myObjectRef = new WeakReference(MyObject);
```

Semplicemente, un riferimento debole è un riferimento che non è abbastanza forte da costringere un oggetto a rimanere in memoria. I riferimenti deboli ti consentono di sfruttare l'abilità del garbage collector di determinare la raggiungibilità per te, quindi non devi farlo tu stesso.

Quando hai bisogno dell'oggetto che hai creato, usa semplicemente il metodo .get() :

```
myObjectRef.get();
```

Il seguente codice esemplificherà questo:

```
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. Riferimento software

I riferimenti soft sono leggermente più forti dei riferimenti deboli. È possibile creare un oggetto di riferimento morbido come segue:

```
SoftReference myObjectRef = new SoftReference(MyObject);
```

Possono trattenere la memoria più fortemente del riferimento debole. Se disponi di risorse / risorse di memoria sufficienti, il garbage collector non pulirà i riferimenti software in modo altrettanto entusiasta come riferimenti deboli.

I riferimenti software sono pratici da utilizzare nella memorizzazione nella cache. È possibile creare oggetti di riferimento morbidi come cache, dove sono conservati fino a esaurimento della memoria. Quando la tua memoria non può fornire abbastanza risorse, il garbage collector rimuoverà i riferimenti software.

```
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. Riferimento Phantom

Questo è il tipo di referenziazione più debole. Se hai creato un riferimento a un oggetto usando Phantom Reference, il metodo get() restituirà sempre null!

L'uso di questo riferimento è che "Gli oggetti di riferimento Phantom, che vengono messi in coda dopo che il collector determina che i loro referenti potrebbero essere recuperati, vengono spesso utilizzati per pianificare le azioni di pulizia pre-mortem in un modo più flessibile di quanto sia possibile con Meccanismo di finalizzazione Java. " - Da Phantom Reference Javadoc di Oracle.

Puoi creare un oggetto di Phantom Reference come segue:

```
PhantomReference myObjectRef = new PhantomReference(MyObject);
```

Leggi Tipi di riferimento online: https://riptutorial.com/it/java/topic/4017/tipi-di-riferimento

Capitolo 172: Tipo di conversione

Sintassi

- TargetType target = (SourceType) source;

Examples

Colata primitiva non numerica

Il tipo boolean non può essere lanciato su / da nessun altro tipo primitivo.

È possibile eseguire il cast di un char su / da qualsiasi tipo numerico utilizzando i mapping di punti di codice specificati da Unicode. Un char viene rappresentato in memoria come un valore intero a 16 bit senza segno (2 byte), quindi il casting a byte (1 byte) farà cadere 8 di questi bit (questo è sicuro per i caratteri ASCII). I metodi di utilità della classe Character utilizzano int (4 byte) per il trasferimento da / verso i valori del punto di codice, ma un short (2 byte) sarebbe anche sufficiente per la memorizzazione di un punto di codice 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
```

Colata primitiva numerica

I primitivi numerici possono essere espressi in due modi. *Il* casting *implicito si* verifica quando il tipo di origine ha un intervallo più piccolo rispetto al tipo di destinazione.

```
//Implicit casting
byte byteVar = 42;
short shortVar = byteVar;
int intVar = shortVar;
long longVar = intvar;
float floatVar = longVar;
double doubleVar = floatVar;
```

Il casting *esplicito* deve essere eseguito quando il tipo di sorgente ha un intervallo più ampio del tipo di destinazione.

```
//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;
```

Quando si esegue il casting di primitive in virgola mobile (float , double) su numeri interi primitivi, il numero viene **arrotondato per difetto** .

Fusione dell'oggetto

Come con i primitivi, gli oggetti possono essere espressi in modo esplicito e implicito.

Il casting implicito si verifica quando il tipo di origine estende o implementa il tipo di destinazione (casting su una superclasse o un'interfaccia).

Il casting esplicito deve essere eseguito quando il tipo di sorgente viene esteso o implementato dal tipo di destinazione (casting in un sottotipo). Questo può produrre un'eccezione di runtime (ClassCastException) quando l'oggetto in fase di cast non è del tipo di destinazione (o del sottotipo di destinazione).

```
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)
```

Promozione numerica di base

```
 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.
}
```

Verifica se un oggetto può essere lanciato usando instanceof

Java fornisce l'operatore instanceof per verificare se un oggetto è di un certo tipo o una sottoclasse di quel tipo. Il programma può quindi scegliere di lanciare o non lanciare quell'oggetto di conseguenza.

```
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.
}
```

Leggi Tipo di conversione online: https://riptutorial.com/it/java/topic/1392/tipo-di-conversione

Capitolo 173: Tokenizer di stringa

introduzione

La classe java.util.StringTokenizer ti consente di dividere una stringa in token. È un modo semplice per rompere la corda.

L'insieme di delimitatori (i caratteri che separano i token) può essere specificato al momento della creazione o in base al token.

Examples

StringTokenizer Dividi per spazio

```
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());
     }
   }
}
```

Produzione:

Mela

palla

gatto

cane

StringTokenizer Dividi con la virgola ','

```
public static void main(String args[]) {
        StringTokenizer st = new StringTokenizer("apple,ball cat,dog", ",");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
```

Produzione:

Mela

palla gatto

cane

Leggi Tokenizer di stringa online: https://riptutorial.com/it/java/topic/10563/tokenizer-di-stringa

Capitolo 174: Trappole comuni di Java

introduzione

Questo argomento delinea alcuni degli errori più comuni fatti dai principianti in Java.

Ciò include qualsiasi errore comune nell'uso del linguaggio Java o comprensione dell'ambiente di runtime.

Gli errori associati a specifiche API possono essere descritti in argomenti specifici di tali API. Le stringhe sono un caso speciale; sono coperti nella specifica del linguaggio Java. Dettagli diversi dagli errori comuni possono essere descritti in questo argomento su Stringhe .

Examples

Pitfall: usare == per confrontare oggetti di wrapper primitivi come Integer

(Questo errore si applica ugualmente a tutti i tipi di wrapper primitivi, ma lo illustreremo per Integer e int .)

Quando si lavora con oggetti Integer , si è tentati di usare == per confrontare i valori, perché è ciò che si farebbe con i valori int . E in alcuni casi questo sembrerà funzionare:

```
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
```

Qui abbiamo creato due oggetti Integer con il valore 1 e li abbiamo confrontati (in questo caso ne abbiamo creati uno da una String e uno da un letterale int . Ci sono altre alternative). Inoltre, osserviamo che i due metodi di confronto (== e equals) rendono entrambi true .

Questo comportamento cambia quando scegliamo valori diversi:

```
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
```

In questo caso, solo il confronto tra equals produce il risultato corretto.

La ragione di questa differenza di comportamento è che la JVM conserva una cache di oggetti Integer per l'intervallo da -128 a 127. (Il valore superiore può essere sostituito con la proprietà di sistema "java.lang.Integer.IntegerCache.high" o Argomento JVM "-XX: AutoBoxCacheMax = size"). Per i valori in questo intervallo, Integer.valueOf() restituirà il valore memorizzato nella cache anziché crearne uno nuovo.

Pertanto, nel primo esempio le chiamate Integer.valueOf(1) e Integer.valueOf("1") restituito la stessa istanza di Integer memorizzata nella cache. Al contrario, nel secondo esempio Integer.valueOf(1000) e Integer.valueOf("1000") entrambi creato e restituito nuovi oggetti Integer .

L'operatore == per i tipi di riferimento verifica l'uguaglianza di riferimento (ovvero lo stesso oggetto). Pertanto, nel primo esempio int1_1 == int1_2 è true perché i riferimenti sono gli stessi. Nel secondo esempio int2_1 == int2_2 è falso perché i riferimenti sono diversi.

Trappola: dimenticando di liberare risorse

Ogni volta che un programma apre una risorsa, come un file o una connessione di rete, è importante liberare la risorsa una volta che hai finito di usarla. Un'analoga cautela dovrebbe essere presa se dovessero essere lanciate delle eccezioni durante le operazioni su tali risorse. Si potrebbe obiettare che FileInputStream ha un finalizzatore che richiama il metodo close() su un evento di garbage collection; tuttavia, poiché non possiamo essere sicuri quando verrà avviato un ciclo di garbage collection, lo stream di input può consumare risorse del computer per un periodo di tempo indefinito. La risorsa deve essere chiuso in un finally sezione di un blocco 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();
        }
    }
}
```

A partire da Java 7 c'è una dichiarazione veramente utile e chiara introdotta in Java 7, in particolare per questo caso, chiamata 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();
        }
    }
}
```

L'istruzione *try-with-resources* può essere utilizzata con qualsiasi oggetto che implementa l'interfaccia Closeable o AutoCloseable . Assicura che ogni risorsa sia chiusa entro la fine dell'istruzione. La differenza tra le due interfacce è che il metodo close() di Closeable genera una IOException che deve essere gestita in qualche modo.

Nei casi in cui la risorsa è già stata aperta ma dovrebbe essere chiusa in sicurezza dopo l'uso, è possibile assegnarla a una variabile locale all'interno delle risorse try-with

Java SE 7

```
private static void printFileJava7(InputStream extResource) throws IOException {
    try (InputStream input = extResource) {
        ... //access resource
    }
}
```

La variabile di risorsa locale creata nel costruttore try-with-resources è effettivamente definitiva.

Trappola: perdite di memoria

Java gestisce automaticamente la memoria. Non è necessario liberare memoria manualmente. La memoria di un oggetto sull'heap può essere liberata da un garbage collector quando l'oggetto non è più *raggiungibile* da un thread attivo.

Tuttavia, è possibile impedire la liberazione della memoria, consentendo agli oggetti di essere raggiungibili e non più necessari. Indipendentemente dal fatto che si tratti di una perdita di memoria o di un packratting della memoria, il risultato è lo stesso: un aumento non necessario della memoria allocata.

Le perdite di memoria in Java possono accadere in vari modi, ma la ragione più comune sono i riferimenti agli oggetti eterni, perché il garbage collector non può rimuovere oggetti dall'heap mentre ci sono ancora riferimenti a essi.

Campi statici

È possibile creare tale riferimento definendo la classe con un campo static contenente una certa raccolta di oggetti e dimenticando di impostare il campo static su null dopo che la raccolta non è più necessaria. static campi static sono considerati root GC e non vengono mai raccolti. Un altro problema sono le perdite nella memoria non heap quando viene utilizzato JNI .

Perdita di Classloader

Di gran lunga, tuttavia, il tipo più insidioso di perdita di memoria è la perdita di classloader . Un classloader contiene un riferimento a ogni classe che ha caricato e ogni classe contiene un riferimento al suo programma di caricamento classi. Ogni oggetto contiene anche un riferimento alla sua classe. Pertanto, se anche un *singolo* oggetto di una classe caricato da un classloader non è garbage, non è possibile raccogliere una *singola* classe caricata da quel classloader. Poiché ogni classe fa riferimento anche ai suoi campi statici, non possono essere raccolti neanche.

Perdita di accumulo L'esempio di perdita di accumulo potrebbe essere simile al seguente:

```
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();
}
```

Questo esempio crea due attività pianificate. La prima attività prende l'ultimo numero da un deque chiamato numbers e, se il numero è divisibile per 51, stampa il numero e la dimensione della deque. Il secondo compito mette i numeri nella coda. Entrambe le attività sono pianificate a una velocità fissa e vengono eseguite ogni 10 ms.

Se il codice viene eseguito, vedrai che la dimensione della deque aumenta in modo permanente.

Questo alla fine farà riempire la deque di oggetti che consumano tutta la memoria heap disponibile.

Per evitare ciò preservando la semantica di questo programma, possiamo usare un metodo diverso per prendere i numeri dal deque: pollLast . Contrariamente al metodo peekLast , pollLast restituisce l'elemento e lo rimuove dalla deque mentre peekLast restituisce solo l'ultimo elemento.

Pitfall: usare == per confrontare le stringhe

Un errore comune per i principianti Java è quello di utilizzare l'operatore == per verificare se due stringhe sono uguali. Per esempio:

```
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?");
            }
        }
    }
}
```

Il programma di cui sopra dovrebbe testare il primo argomento della riga di comando e stampare diversi messaggi quando non è la parola "ciao". Ma il problema è che non funzionerà. Il programma produrrà "Ti senti irritato oggi?" non importa quale sia il primo argomento della riga di comando.

In questo caso particolare, la String "ciao" viene inserita nel pool di stringhe mentre gli argomenti di String [0] risiedono nell'heap. Ciò significa che ci sono due oggetti che rappresentano lo stesso letterale, ciascuno con il suo riferimento. Poiché == verifica i riferimenti, non l'uguaglianza effettiva, il confronto produrrà un falso il più delle volte. Questo non significa che lo farà sempre.

Quando si usa == per testare le stringhe, ciò che si sta effettivamente testando è se due oggetti String sono lo stesso oggetto Java. Sfortunatamente, non è quello che significa l'uguaglianza delle stringhe in Java. In effetti, il modo corretto per testare le stringhe è utilizzare il metodo equals(Object) . Per un paio di stringhe, di solito vogliamo testare se consistono degli stessi caratteri nello stesso ordine.

```
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?");
            }
        }
    }
}
```

Ma in realtà peggiora. Il problema è che == *darà* la risposta attesa in alcune circostanze. Per esempio

```
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");
        }
    }
}
```

È interessante notare che questo verrà stampato "uguale", anche se stiamo testando le stringhe nel modo sbagliato. Perché? Poiché la specifica del linguaggio Java (Sezione 3.10.5: String Literals) stabilisce che qualsiasi stringa di due >> valori letterali << costituiti dagli stessi caratteri sarà effettivamente rappresentata dallo stesso oggetto Java. Quindi, il test == darà true per letterali uguali. (I valori letterali delle stringhe sono "internati" e aggiunti a un "pool di stringhe" condiviso quando viene caricato il codice, ma questo è in realtà un dettaglio di implementazione.)

Per aggiungere confusione, la specifica del linguaggio Java stabilisce anche che quando si dispone di un'espressione costante in fase di compilazione che concatena due valori letterali stringa, è equivalente a un singolo valore letterale. Così:

```
    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");
        }
    }
}
```

Questo produrrà "1. stesso" e "2. diverso". Nel primo caso, l'espressione + viene valutata al momento della compilazione e confrontiamo un oggetto String con se stesso. Nel secondo caso, viene valutato in fase di esecuzione e confrontiamo due diversi oggetti String

In sintesi, usare == per testare le stringhe in Java è quasi sempre errato, ma non è garantito dare la risposta sbagliata.

Pitfall: testare un file prima di tentare di aprirlo.

Alcune persone consigliano di applicare vari test a un file prima di tentare di aprirlo per fornire una migliore diagnostica o evitare di gestire eccezioni. Ad esempio, questo metodo tenta di verificare se il path corrisponde a un file leggibile:

```
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;
}
```

Potresti usare il metodo sopra come questo:

```
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.
}
```

Il primo problema è nella firma per FileInputStream(File) perché il compilatore continuerà a insistere sul fatto che intercettiamo IOException qui, o più in alto nello stack.

Il secondo problema è che i controlli eseguiti da getValidatedFile non garantiscono il successo di FileInputStream .

- Condizioni di gara: un altro thread o un processo separato potrebbe rinominare il file, eliminare il file o rimuovere l'accesso in lettura dopo il ritorno di getValidatedFile . Ciò porterebbe a una IOException "semplice" senza il messaggio personalizzato.
- Ci sono casi limite non coperti da questi test. Ad esempio, su un sistema con SELinux in modalità "enforcing", un tentativo di leggere un file può fallire malgrado canRead() restituisce true .

Il terzo problema è che i test sono inefficienti. Ad esempio, il exists , isFile e canRead chiamate saranno ogni effettuare una chiamata di sistema per eseguire il controllo desiderato. Viene quindi eseguito un altro syscall per aprire il file, che ripete gli stessi controlli dietro le quinte.

In breve, metodi come getValidatedFile sono fuorviati. È meglio semplicemente provare ad aprire il file e gestire l'eccezione:

```
try (InputStream is = new FileInputStream("somefile")) {
    // Read data etc.
} catch (IOException ex) {
    System.err.println("IO Error processing 'somefile': " + ex.getMessage());
    return;
}
```

Se si desidera distinguere gli errori IO generati durante l'apertura e la lettura, è possibile utilizzare un try / catch annidato. Se si voleva produrre diagnostiche migliori per i fallimenti aperti, è possibile eseguire le exists , isFile e canRead controlli nel gestore.

Trappola: pensare alle variabili come oggetti

Nessuna variabile Java rappresenta un oggetto.

```
String foo;   // NOT AN OBJECT
```

Nemmeno una matrice Java contiene oggetti.

```
String bar[] = new String[100];  // No member is an object.
```

Se pensi erroneamente alle variabili come oggetti, il comportamento effettivo del linguaggio Java ti sorprenderà.

- Per le variabili Java che hanno un tipo primitivo (come int o float) la variabile contiene una copia del valore. Tutte le copie di un valore primitivo sono indistinguibili; cioè c'è solo un valore int per il numero uno. I valori primitivi non sono oggetti e non si comportano come oggetti.

- Per le variabili Java che hanno un tipo di riferimento (una classe o un tipo di matrice) la variabile contiene un riferimento. Tutte le copie di un riferimento sono indistinguibili. I riferimenti possono indicare oggetti, oppure possono essere null che significa che non puntano a nessun oggetto. Tuttavia, non sono oggetti e non si comportano come oggetti.

Le variabili non sono oggetti in entrambi i casi e non contengono oggetti in entrambi i casi. Possono contenere *riferimenti a oggetti* , ma ciò significa qualcosa di diverso.

Classe di esempio

Gli esempi che seguono utilizzano questa classe, che rappresenta un punto nello spazio 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;
   }
}
```

Un esempio di questa classe è un oggetto che ha due campi x ed y che hanno il tipo int .

Possiamo avere molte istanze della classe MutableLocation . Alcuni rappresenteranno le stesse posizioni nello spazio 2D; ossia i rispettivi valori di x ed y corrisponderanno. Altri rappresenteranno luoghi diversi.

Più variabili possono puntare allo stesso oggetto

```
 MutableLocation here = new MutableLocation(1, 2);
 MutableLocation there = here;
 MutableLocation elsewhere = new MutableLocation(1, 2);
```

In quanto sopra, abbiamo dichiarato here tre variabili, there e elsewhere che possono contenere riferimenti a oggetti MutableLocation .

Se pensate (erroneamente) a queste variabili come ad oggetti, probabilmente avrete erroneamente interpretato le affermazioni come dicendo:

1. Copia la posizione "[1, 2]" here
2. Copia la posizione "[1, 2]" a there
3. Copia la posizione "[1, 2]" in elsewhere

Da ciò, è probabile dedurre che abbiamo tre oggetti indipendenti nelle tre variabili. In effetti ci sono *solo due oggetti creati* da quanto sopra. Le variabili here e there riferiscono effettivamente allo stesso oggetto.

Possiamo dimostrarlo. Assumendo le dichiarazioni variabili come sopra:

```
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);
```

Ciò produrrà il seguente:

```
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
```

Abbiamo assegnato un nuovo valore a here.x e ha cambiato il valore che vediamo tramite there.x . Si riferiscono allo stesso oggetto. Ma il valore che vediamo tramite elsewhere.x non è cambiato, quindi elsewhere deve fare riferimento a un oggetto diverso.

Se una variabile era un oggetto, allora l'assegnazione here.x = 42 non cambierebbe there.x

L'operatore di uguaglianza NON verifica che due oggetti siano uguali

L'applicazione dell'operatore di uguaglianza (==) ai valori di riferimento verifica se i valori si riferiscono allo stesso oggetto. Esso *non* verifica se due (differenti) oggetti sono "uguali" nel senso 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");
}
```

Questo stamperà "qui è lì", ma non stamperà "qui è altrove". (I riferimenti here e elsewhere sono per due oggetti distinti).

Al contrario, se chiamiamo il metodo equals(Object) che abbiamo implementato in precedenza, MutableLocation se due istanze di MutableLocation hanno una posizione uguale.

```
if (here.equals(there)) {
    System.out.println("here equals there");
}
if (here.equals(elsewhere)) {
    System.out.println("here equals elsewhere");
}
```

Questo stamperà entrambi i messaggi. In particolare, here.equals(elsewhere) restituisce true perché i criteri semantici che abbiamo scelto per l'uguaglianza di due oggetti MutableLocation sono stati soddisfatti.

Le chiamate di metodo NON passano affatto gli oggetti

Le chiamate al metodo Java utilizzano il *valore* [1] per passare gli argomenti e restituiscono un risultato.

Quando si passa un valore di riferimento a un metodo, si passa effettivamente un riferimento a un oggetto *per valore* , il che significa che sta creando una copia del riferimento all'oggetto.

Finché entrambi i riferimenti agli oggetti continuano a puntare allo stesso oggetto, puoi modificare quell'oggetto da entrambi i riferimenti, e questo è ciò che causa confusione per alcuni.

Tuttavia, *non* sta passando un oggetto per riferimento [2]. La distinzione è che se la copia di riferimento dell'oggetto viene modificata per puntare a un altro oggetto, il riferimento

all'oggetto originale continuerà a puntare all'oggetto originale.

```
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".
}
```

Né stai passando una copia dell'oggetto.

```
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 - In linguaggi come Python e Ruby, il termine "passa per condivisione" è preferito per "passare per valore" di un oggetto / riferimento.

2 - Il termine "passaggio per riferimento" o "chiamata per riferimento" ha un significato molto specifico nella programmazione della terminologia del linguaggio. In effetti, significa che si passa l'indirizzo *di una variabile o di un elemento dell'array* , in modo che quando il metodo chiamato assegna un nuovo valore all'argomento formale, esso cambia il valore nella variabile originale. Java non supporta questo. Per una descrizione più completa dei diversi meccanismi per il passaggio dei parametri, consultare https://en.wikipedia.org/wiki/Evaluation_strategy .

Trappola: combinazione di incarichi ed effetti collaterali

Occasionalmente vediamo domande Java StackOverflow (e domande C o C ++) che chiedono che cosa tipo questo:

```
i += a[i++] + b[i--];
```

restituisce ... per alcuni stati iniziali note di i , a e b .

Parlando in generale:

- per Java la risposta è sempre specificata [1] , ma non ovvia e spesso difficile da capire
- per C e C ++ la risposta è spesso non specificata.

Tali esempi vengono spesso utilizzati negli esami o nelle interviste di lavoro come tentativo di vedere se lo studente o l'intervistato comprende come la valutazione delle espressioni funzioni realmente nel linguaggio di programmazione Java. Questo è probabilmente legittimo come un "test di conoscenza", ma ciò non significa che dovresti mai farlo in un programma reale.

Per illustrare, il seguente esempio apparentemente semplice è apparso un paio di volte nelle domande StackOverflow (come questo). In alcuni casi, appare come un vero errore nel codice di qualcuno.

```
int a = 1;
a = a++;
System.out.println(a);    // What does this print.
```

La maggior parte dei programmatori (inclusi esperti Java) che leggono *rapidamente* queste affermazioni direbbero che emette 2 . In effetti, emette 1 . Per una spiegazione dettagliata del motivo, leggere questa risposta .

Tuttavia il vero takeaway da questo e gli esempi simili è che *qualsiasi* dichiarazione Java che *sia* assegna da *e* per gli effetti collaterali della stessa variabile sta per essere *nella migliore delle ipotesi* difficile da capire, e *nel peggiore dei casi* addirittura fuorviante. Dovresti evitare di scrivere codice come questo.

---

1 - Modulo potenziali problemi con il modello di memoria Java se le variabili o gli oggetti sono visibili ad altri thread.

Trappola: non capendo che String è una classe immutabile

I nuovi programmatori Java spesso dimenticano, o non riescono a comprendere appieno, che la classe Java String è immutabile. Questo porta a problemi come quello nell'esempio seguente:

```
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();
    }
}
```

Il codice precedente dovrebbe stampare gli argomenti della riga di comando in maiuscolo. Sfortunatamente, non funziona, il caso degli argomenti non è cambiato. Il problema è questa affermazione:

```
s.toUpperCase();
```

Si potrebbe pensare che la chiamata toUpperCase() cambierà s in una stringa maiuscola. Non è così. Non può! String oggetti String sono immutabili. Non possono essere cambiati.

In realtà, il metodo toUpperCase() *restituisce* un oggetto String che è una versione in maiuscolo della String cui viene chiamata. Questo sarà probabilmente un nuovo oggetto String , ma se s era già tutto in maiuscolo, il risultato potrebbe essere la stringa esistente.

Quindi, per utilizzare efficacemente questo metodo, è necessario utilizzare l'oggetto restituito dalla chiamata al metodo; per esempio:

```
s = s.toUpperCase();
```

In effetti, la regola "le stringhe non cambiano mai" si applica a tutti i metodi String . Se lo ricordi, puoi evitare un'intera categoria di errori del principiante.

Leggi Trappole comuni di Java online: https://riptutorial.com/it/java/topic/4388/trappole-comuni-di-java

Capitolo 175: TreeMap e TreeSet

introduzione

TreeMap e TreeSet sono raccolte Java di base aggiunte in Java 1.2. TreeMap è un **mutevole**, **ordinata**, Map implementazione. Allo stesso modo, TreeSet è un'implementazione Set **mutevole** e **ordinata** .

TreeMap è implementato come un albero Red-Black, che fornisce i tempi di accesso O(log n) . TreeSet viene implementato utilizzando una TreeMap con valori fittizi.

Entrambe le collezioni **non** sono thread-safe.

Examples

TreeMap di un semplice tipo Java

Per prima cosa, creiamo una mappa vuota e inseriamo alcuni elementi in essa:

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 volta che abbiamo alcuni elementi nella mappa, possiamo eseguire alcune operazioni:

```
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
```

Possiamo anche scorrere gli elementi della mappa usando un Iterator o un ciclo foreach. Si noti che le voci vengono stampate in base al loro ordinamento naturale , non all'ordine di inserimento:

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 di un semplice tipo Java

Per prima cosa, creiamo un set vuoto e inseriamo alcuni elementi in esso:

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 volta che abbiamo alcuni elementi nel set, possiamo eseguire alcune operazioni:

```
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
```

Possiamo anche scorrere gli elementi della mappa usando un Iterator o un ciclo foreach. Si noti che le voci vengono stampate in base al loro ordinamento naturale , non all'ordine di inserimento:

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 di un tipo Java personalizzato

Poiché TreeMap e TreeSet mantengono le chiavi / gli elementi in base al loro ordinamento naturale . Quindi le chiavi TreeMap e TreeSet elementi TreeSet devono essere paragonabili tra loro.

Supponiamo di avere una classe Person personalizzata:

```
public class Person  {

    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constuctors, getters, setters and various methods
}
```

Se lo memorizziamo così com'è in un TreeSet (o una chiave in una TreeMap):

```
TreeSet<Person2> set = ...
set.add(new Person(1,"first","last",Date.from(Instant.now())));
```

Quindi corriamo in un'eccezione come questa:

```
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)
```

Per risolvere il problema, supponiamo di voler ordinare istanze Person base all'ordine dei loro id (private int id). Potremmo farlo in due modi:

1. Una soluzione è modificare Person modo da implementare l' interfaccia 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. Un'altra soluzione è fornire TreeSet con un comparatore :

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());
    }
});
```

Tuttavia, vi sono due avvertimenti su entrambi gli approcci:

1. È **molto importante** non modificare i campi utilizzati per ordinare una volta che un'istanza è stata inserita in un TreeSet / TreeMap . Nell'esempio precedente, se cambiamo l' id di una persona che è già inserita nella raccolta, potremmo imbattersi in un comportamento imprevisto.

2. È importante implementare il confronto correttamente e coerentemente. Come da Javadoc :

   L'implementatore deve garantire sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) per tutti xey. (Ciò implica che x.compareTo(y) deve generare un'eccezione iff y.compareTo(x) genera un'eccezione).

   L'implementatore deve inoltre garantire che la relazione sia transitiva: (x.compareTo(y)>0 && y.compareTo(z)>0) implica x.compareTo(z)>0 .

   Infine, l'implementatore deve assicurarsi che x.compareTo(y)==0 implichi che sgn(x.compareTo(z)) == sgn(y.compareTo(z)) , per tutti z.

TreeMap e TreeSet Thread Safety

TreeMap e TreeSet **non** sono TreeSet thread-safe, quindi è necessario prestare attenzione per garantire l'utilizzo in programmi multi-thread.

Sia TreeMap che TreeSet sono sicuri quando letti, anche contemporaneamente, da più thread. Quindi, se sono stati creati e popolati da un singolo thread (ad esempio all'inizio del programma) e solo poi letti, ma non modificati da più thread, non c'è motivo per la sincronizzazione o il blocco.

Tuttavia, se letto e modificato contemporaneamente o modificato contemporaneamente da più di un thread, la raccolta potrebbe generare una ConcurrentModificationException o comportarsi in modo imprevisto. In questi casi, è assolutamente necessario sincronizzare / bloccare l'accesso alla raccolta utilizzando uno dei seguenti approcci:

1. Utilizzando Collections.synchronizedSorted.. :

```
SortedSet<Integer> set = Collections.synchronizedSortedSet(new TreeSet<Integer>());
SortedMap<Integer,String> map = Collections.synchronizedSortedMap(new
TreeMap<Integer,String>());
```

Ciò fornirà una SortedSet / SortedMap attuazione sostenuta dalla collezione reale e sincronizzato su qualche oggetto mutex. Si noti che questo sincronizzerà tutti gli accessi in lettura e scrittura alla raccolta su un singolo blocco, quindi anche le letture concorrenti non sarebbero possibili.

2. Sincronizzando manualmente su alcuni oggetti, come la raccolta stessa:

```
 TreeSet<Integer> set = new TreeSet<>();
```

...

```
//Thread 1
synchronized (set) {
    set.add(4);
}
```

...

```
//Thread 2
synchronized (set) {
    set.remove(5);
}
```

3. Utilizzando un blocco, come un 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 differenza dei precedenti metodi di sincronizzazione, l'utilizzo di ReadWriteLock consente a più thread di leggere contemporaneamente dalla mappa.

Leggi TreeMap e TreeSet online: https://riptutorial.com/it/java/topic/9905/treemap-e-treeset

Capitolo 176: Utilizzando la parola chiave statica

Sintassi

- public static int myVariable; // Dichiarazione di una variabile statica
- public static myMethod () {} // Dichiarazione di un metodo statico
- public static final double MY_CONSTANT; // Dichiarazione di una variabile costante condivisa tra tutte le istanze della classe
- doppia finale pubblica MY_CONSTANT; // Dichiarazione di una variabile costante specifica per questa istanza della classe (utilizzata al meglio in un costruttore che genera una costante diversa per ogni istanza)

Examples

Uso statico per dichiarare le costanti

Poiché la parola chiave static viene utilizzata per accedere a campi e metodi senza una classe istanziata, può essere utilizzata per dichiarare le costanti da utilizzare in altre classi. Queste variabili rimarranno costanti in ogni istanza della classe. Per convenzione, static variabili static sono sempre ALL_CAPS e usano caratteri di sottolineatura piuttosto che cammello. ex:

```
 static E STATIC_VARIABLE_NAME
```

Poiché le costanti non possono cambiare, è possibile utilizzare anche la static con il modificatore final :

Ad esempio, per definire la costante matematica di pi:

```
public class MathUtilities {

    static final double PI = 3.14159265358

}
```

Quale può essere utilizzato in qualsiasi classe come costante, ad esempio:

```
public class MathCalculations {

    //Calculates the circumference of a circle
    public double calculateCircumference(double radius) {
        return (2 * radius * MathUtilities.PI);
    }

}
```

Usando statico con questo

Statico fornisce un metodo o una memoria variabile che *non* è allocata per ogni istanza della classe. Piuttosto, la variabile statica è condivisa tra tutti i membri della classe. Per inciso, cercando di trattare la variabile statica come un membro dell'istanza della classe si otterrà un avvertimento:

```
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
```

I metodi dichiarati statici si comportano più o meno allo stesso modo, ma con una restrizione aggiuntiva:

- Non è possibile utilizzare la this parola chiave in loro!

```
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
    }

}
```

In generale, è meglio dichiarare static metodi che si applicano a diverse istanze di una classe (come i metodi clone) static , mantenendo metodi come equals() come non statici. Il main metodo di un programma Java è sempre statica, il che significa che la parola this non può essere utilizzato all'interno main() .

Riferimento a membri non statici dal contesto statico

Le variabili e i metodi statici non fanno parte di un'istanza. Ci sarà sempre una singola copia di quella variabile, indipendentemente dal numero di oggetti creati da una determinata classe.

Ad esempio potresti voler avere una lista immutabile di costanti, sarebbe una buona idea tenerla statica e inizializzarla solo una volta all'interno di un metodo statico. Ciò ti darebbe un significativo guadagno di prestazioni se crei diverse istanze di una particolare classe su base regolare.

Inoltre puoi anche avere un blocco statico in una classe. Puoi usarlo per assegnare un valore predefinito a una variabile statica. Vengono eseguiti solo una volta quando la classe viene caricata in memoria.

Le variabili di istanza come suggeriscono il nome dipendono da un'istanza di un oggetto particolare, vivono per servire i capricci di esso. Puoi giocare con loro durante un particolare ciclo di vita di un oggetto.

Tutti i campi e i metodi di una classe utilizzati all'interno di un metodo statico di quella classe devono essere statici o locali. Se si tenta di utilizzare variabili o metodi di istanza (non statici), il codice non verrà compilato.

```
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
    }

}
```

Leggi Utilizzando la parola chiave statica online:
https://riptutorial.com/it/java/topic/2253/utilizzando-la-parola-chiave-statica

Capitolo 177: Utilizzo di altri linguaggi di scripting in Java

introduzione

Java in sé è un linguaggio estremamente potente, ma la sua potenza può essere ulteriormente estesa Grazie a JSR223 (Java Specification Request 223) che introduce un motore di script

Osservazioni

L'API Java Scripting consente agli script esterni di interagire con Java

L'API Scripting può abilitare l'interazione tra lo script e java. Le lingue di scripting devono avere un'implementazione di Script Engine nel classpath.

Per default JavaScript (noto anche come ECMAScript) viene fornito da Nashorn per impostazione predefinita. Ogni motore di script ha un contesto di script in cui tutte le variabili, le funzioni, i metodi sono memorizzati in associazioni. A volte è possibile che si desideri utilizzare più contesti poiché supportano il reindirizzamento dell'output a un writer bufferizzato e un errore a un altro.

Ci sono molte altre librerie di motori di script come Jython e JRuby. Finché sono sul classpath è possibile eseguire il codice di valutazione.

Possiamo usare i binding per esporre le variabili nello script. Abbiamo bisogno di legami multipli in alcuni casi poiché l'esposizione di variabili al motore fondamentalmente espone le variabili solo a quel motore, a volte richiediamo di esporre determinate variabili come l'ambiente di sistema e il percorso che è lo stesso per tutti i motori dello stesso tipo. In tal caso, è necessaria un'associazione che è un ambito globale. Esporre le variabili a quelle esposte a tutti i motori di script creati dallo stesso EngineFactory

Examples

Valutazione di un file javascript in modalità -scripting di 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;
    }

}
```

Ora il metodo principale

```
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);");
}
```

Il tuo output dovrebbe essere simile a questo
hello world

Come puoi vedere la variabile esposta x è stata stampata. Ora prova con un file.

Qui abbiamo test.js

```
print(x);
function test(){
```

```
    print("hello test.js:test");
}
test();
```

E il metodo principale aggiornato

```
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");
}
```

Supponendo che test.js si trovi nella stessa directory dell'applicazione Dovresti avere un output simile a questo

```
hello world
hello test.js:test
```

Leggi Utilizzo di altri linguaggi di scripting in Java online:
https://riptutorial.com/it/java/topic/9926/utilizzo-di-altri-linguaggi-di-scripting-in-java

Capitolo 178: Utilizzo di ThreadPoolExecutor in applicazioni MultiThreaded.

introduzione

Quando si crea un'applicazione performante e guidata dai dati, può essere molto utile eseguire attività che richiedono molto tempo in modo asincrono e far eseguire contemporaneamente più attività. Questo argomento introdurrà il concetto di utilizzo di ThreadPoolExecutors per completare contemporaneamente più attività asincrone.

Examples

Esecuzione di attività asincrone in cui non è necessario alcun valore di ritorno utilizzando un'istanza di classe eseguibile

Alcune applicazioni potrebbero voler creare attività cosiddette "Fire & Forget" che possono essere attivate periodicamente e non devono restituire alcun tipo di valore restituito al completamento dell'attività assegnata (ad esempio, eliminazione di vecchi file temporanei, registri in rotazione, salvataggio automatico stato).

In questo esempio, creeremo due classi: una che implementa l'interfaccia Runnable e una che contiene un metodo 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();
    }
}
```

L'esecuzione di AsyncExample1.main () ha prodotto il seguente output:

```
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
```

Osservazioni sulla nota: ci sono diverse cose da notare nell'output sopra,

1. Le attività non sono state eseguite in un ordine prevedibile.
2. Dato che ogni attività stava dormendo per una quantità (pseudo) casuale di tempo, non necessariamente completavano nell'ordine in cui erano invocati.

Esecuzione di attività asincrone in cui è necessario un valore restituito utilizzando un'istanza di classe richiamabile

Spesso è necessario eseguire un'attività di lunga durata e utilizzare il risultato di tale attività una volta completata.

In questo esempio, creeremo due classi: Una che implementa l'interfaccia Callable <T> (dove T è il tipo che si desidera restituire) e una che contiene un metodo 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());
                    }
                }
            }
        }
    }
}
```

L'esecuzione di AsyncExample2.main () ha prodotto il seguente output:

```
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
```

Osservazioni di nota:

Ci sono diverse cose da notare nell'output sopra,

1. Ogni chiamata a ExecutorService.submit () ha restituito un'istanza di Future, che è stata memorizzata in un elenco per un uso futuro
2. Future contiene un metodo chiamato isDone () che può essere usato per verificare se la nostra attività è stata completata prima di provare a controllare il suo valore di ritorno. Chiamare il metodo Future.get () su un Future che non è stato ancora eseguito bloccherà il thread corrente fino al completamento dell'attività, annullando potenzialmente molti benefici ottenuti dall'esecuzione dell'attività in modo asincrono.
3. Il metodo executorService.shutdown () è stato chiamato prima di controllare i valori di ritorno degli oggetti Future. Questo non è richiesto, ma è stato fatto in questo modo per dimostrare che è possibile. Il metodo executorService.shutdown () non impedisce il completamento delle attività che sono già state inoltrate a ExecutorService, ma impedisce che nuove attività vengano aggiunte alla coda.

Definizione delle attività asincrone in linea utilizzando Lambdas

Mentre una buona progettazione del software spesso massimizza la riusabilità del codice, a volte può essere utile definire attività asincrone in linea nel codice tramite espressioni Lambda per massimizzare la leggibilità del codice.

In questo esempio, creeremo una singola classe che contiene un metodo main (). All'interno di questo metodo, utilizzeremo espressioni Lambda per creare ed eseguire istanze di Callable e 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);
    }
}
```

Osservazioni di nota:

Ci sono diverse cose da notare nell'output sopra,

1. Le espressioni Lambda hanno accesso a variabili e metodi disponibili per l'ambito in cui sono definiti, ma tutte le variabili devono essere definitive (o effettivamente definitive) per l'uso all'interno di un'espressione lambda.
2. Non è necessario specificare se la nostra espressione Lambda è un riscattabile o una <T> eseguibile in modo esplicito, il tipo restituito viene inferito automaticamente dal tipo restituito.

Leggi Utilizzo di ThreadPoolExecutor in applicazioni MultiThreaded. online: https://riptutorial.com/it/java/topic/8646/utilizzo-di-threadpoolexecutor-in-applicazioni-multithreaded-

Capitolo 179: Valuta e denaro

Examples

Aggiungi valuta personalizzata

JAR richiesti sul classpath:

- javax.money:money-api:1.0 (JSR354 money and currency api)
- org.javamoney: moneta: 1.0 (implementazione di riferimento)
- javax: annotation-api: 1.2. (Annotazioni comuni utilizzate dall'implementazione di riferimento)

```
// 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
```

Leggi Valuta e denaro online: https://riptutorial.com/it/java/topic/8359/valuta-e-denaro

Capitolo 180: Varargs (argomento variabile)

Osservazioni

Un argomento del metodo "varargs" consente ai chiamanti di quel metodo di specificare più argomenti del tipo designato, ciascuno come argomento separato. Viene specificato nella dichiarazione del metodo da tre periodi ASCII (...) dopo il tipo base.

Il metodo stesso riceve quegli argomenti come un singolo array, il cui tipo di elemento è il tipo dell'argomento varargs. La matrice viene creata automaticamente (sebbene ai chiamanti sia ancora permesso di passare una matrice esplicita invece di passare più valori come argomenti separati del metodo).

Regole per vararg:

1. Varargs deve essere l'ultimo argomento.
2. Ci possono essere solo un Vararg nel metodo.

Devi seguire le regole precedenti altrimenti il programma darà un errore di compilazione.

Examples

Specifica di un parametro varargs

```
void doSomething(String... strings) {
    for (String s : strings) {
        System.out.println(s);
    }
}
```

I tre punti dopo il tipo del parametro finale indicano che l'argomento finale può essere passato come una matrice o come una sequenza di argomenti. Varargs può essere utilizzato solo nella posizione dell'argomento finale.

Lavorare con i parametri di Vararg

Usando varargs come parametro per una definizione di metodo, è possibile passare una matrice o una sequenza di argomenti. Se viene passata una sequenza di argomenti, vengono convertiti automaticamente in una matrice.

Questo esempio mostra sia una matrice che una sequenza di argomenti passati nel metodo printVarArgArray() e come vengono trattati in modo identico nel codice all'interno del metodo:

```
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);
    }
}
```

Produzione:

```
10,20,
5,6,5,8,6,31
```

Se si definisce il metodo in questo modo, si otterranno errori in fase di compilazione.

```
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
```

Leggi Varargs (argomento variabile) online: https://riptutorial.com/it/java/topic/1948/varargs--argomento-variabile-

Capitolo 181: Visibilità (controllo dell'accesso ai membri di una classe)

Sintassi

- nome del tipo pubblico [= valore];
- nome del tipo privato [= valore];
- nome del tipo protetto [= valore];
- nome del tipo [= valore];
- nome di classe pubblica {
- nome della classe{

Osservazioni

Dal tutorial di Java :

I modificatori del livello di accesso determinano se altre classi possono utilizzare un particolare campo o richiamare un particolare metodo. Esistono due livelli di controllo degli accessi:

- Al livello più alto: public o *package-private* (nessun modificatore esplicito).
- A livello di membro: public , private , protected o *privato del pacchetto* (nessun modificatore esplicito).

Una classe può essere dichiarata con il modificatore public , nel qual caso tale classe è visibile a tutte le classi ovunque. Se una classe non ha alcun modificatore (l'impostazione predefinita, nota anche come *pacchetto-privato*), è visibile solo all'interno del proprio pacchetto.

A livello di membro, puoi anche usare il modificatore public o nessun modificatore (*pacchetto-privato*) proprio come con le classi di primo livello e con lo stesso significato. Per i membri, ci sono due ulteriori modificatori di accesso: private e protected . Il modificatore private specifica che è possibile accedere al membro solo nella sua classe. Il modificatore protected specifica che è possibile accedere al membro solo all'interno del proprio pacchetto (come con *package-private*) e, inoltre, da una sottoclasse della sua classe in un altro pacchetto.

La seguente tabella mostra l'accesso ai membri consentiti da ciascun modificatore.

Livelli di accesso:

Modificatore	Classe	Pacchetto	sottoclasse	Mondo
public	Y	Y	Y	Y
protected	Y	Y	Y	N
nessun modificatore	Y	Y	N	N
private	Y	N	N	N

Examples

Membri dell'interfaccia

```
public interface MyInterface {
    public void foo();
```

```
    int bar();

    public String TEXT = "Hello";
    int ANSWER = 42;

    public class X {
    }

    class Y {
    }
}
```

I membri dell'interfaccia hanno sempre visibilità pubblica, anche se la parola chiave public viene omessa. Quindi sia foo() , bar() , TEXT , ANSWER , X e Y hanno visibilità pubblica. Tuttavia, l'accesso può essere ancora limitato dall'interfaccia contenente - poiché MyInterface ha visibilità pubblica, i suoi membri possono essere accessibili da qualsiasi luogo, ma se MyInterface avesse avuto visibilità del pacchetto, i suoi membri sarebbero stati accessibili solo all'interno dello stesso pacchetto.

Visibilità pubblica

Visibile alla classe, al pacchetto e alla sottoclasse.

Vediamo un esempio con il test di classe.

```
public class Test{
    public int number = 2;

    public Test(){

    }
}
```

Ora proviamo a creare un'istanza della classe. In questo esempio, **possiamo** accedere al number perché è public .

```
public class Other{

    public static void main(String[] args){
        Test t = new Test();
        System.out.println(t.number);
    }

}
```

Visibilità privata

private visibilità private consente a una variabile di accedere solo alla sua classe. Essi sono spesso usati in **combinazione** con public getter e setter.

```
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());
    }
}
```

Visibilità del pacchetto

Senza **alcun modificatore** , l'impostazione predefinita è la visibilità del pacchetto. *Dalla documentazione Java,* "[visibilità pacchetto] indica se le classi nello stesso pacchetto della classe (indipendentemente dalla loro parentela) hanno accesso al membro." In questo esempio di javax.swing ,

```
package javax.swing;
public abstract class JComponent extends Container … {
    …
    static boolean DEBUG_GRAPHICS_LOADED;
    …
}
```

DebugGraphics trova nello stesso pacchetto, quindi DEBUG_GRAPHICS_LOADED è accessibile.

```
package javax.swing;
public class DebugGraphics extends Graphics {
    …
    static {
        JComponent.DEBUG_GRAPHICS_LOADED = true;
    }
    …
}
```

Questo articolo fornisce alcune informazioni sull'argomento.

Visibilità protetta

Cause di visibilità protetta significa che questo membro è visibile per il suo pacchetto, insieme a una qualsiasi delle sue sottoclassi.

Come esempio:

```
package com.stackexchange.docs;
public class MyClass{
    protected int variable; //This is the variable that we are trying to access
    public MyClass(){
        variable = 2;
    };
```

```
}
```

Ora estenderemo questa classe e proveremo ad accedere a uno dei suoi membri protected .

```
package some.other.pack;
import com.stackexchange.docs.MyClass;
public class SubClass extends MyClass{
    public SubClass(){
        super();
        System.out.println(super.variable);
    }
}
```

Sarai anche in grado di accedere a un membro protected senza estenderlo se accederai dallo stesso pacchetto.

Nota che questo modificatore funziona solo sui membri di una classe, non sulla classe stessa.

Riepilogo dei modificatori di accesso ai membri della classe

Modificatore d'accesso	Visibilità	Eredità
Privato	Solo la lezione	Non può essere ereditato
Nessun modificatore / pacchetto	Nel pacchetto	Disponibile se sottoclasse nel pacchetto
protetta	Nel pacchetto	Disponibile in sottoclasse
Pubblico	Ovunque	Disponibile in sottoclasse

C'era una volta un modificatore private protected (entrambe le parole chiave contemporaneamente) che poteva essere applicato a metodi o variabili per renderli accessibili da una sottoclasse esterna al pacchetto, ma renderli privati delle classi in quel pacchetto. Tuttavia, questo è stato rimosso nella versione di Java 1.0 .

Leggi Visibilità (controllo dell'accesso ai membri di una classe) online: https://riptutorial.com/it/java/topic/134/visibilita--controllo-dell-accesso-ai-membri-di-una-classe-

Capitolo 182: WeakHashMap

introduzione

Concetti di Hashmap debole

Examples

Concetti di WeakHashmap

Punti chiave:-

- Implementazione della mappa.
- memorizza solo riferimenti deboli alle sue chiavi.

Riferimenti deboli : gli oggetti a cui si fa riferimento solo con riferimenti deboli sono garbage collection desiderati; il GC non aspetterà finché non avrà bisogno di memoria in quel caso.

Diffrenza tra Hashmap e WeakHashMap: -

Se il gestore della memoria Java non ha più un riferimento forte all'oggetto specificato come chiave, la voce nella mappa verrà rimossa in WeakHashMap.

Esempio :-

```
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"));
    }
```

Differenze di dimensione (HashMap vs WeakHashMap):

Il metodo calling size () sull'oggetto HashMap restituirà lo stesso numero di coppie chiave-valore. la dimensione diminuirà solo se il metodo remove () viene chiamato esplicitamente sull'oggetto HashMap.

Poiché il garbage collector può scartare le chiavi in qualsiasi momento, una WeakHashMap può comportarsi come se un thread sconosciuto stia rimuovendo silenziosamente le voci. Pertanto, è possibile che il metodo di ridimensionamento restituisca valori più piccoli nel tempo. Quindi, nella **riduzione della dimensione di WeakHashMap avviene automaticamente** .

Leggi WeakHashMap online: https://riptutorial.com/it/java/topic/10749/weakhashmap

Capitolo 183: XJC

introduzione

XJC è uno strumento Java SE che compila un file di schema XML in classi Java completamente annotate.

È distribuito all'interno del pacchetto JDK e si trova in /bin/xjc path.

Sintassi

- xjc [opzioni] file di schema / URL / dir / jar ... [-b bindinfo] ...

Parametri

Parametro	Dettagli
file di schema	Il file dello schema xsd da convertire in java

Osservazioni

Lo strumento XJC è disponibile come parte del JDK. Permette di creare codice java annotato con annotazioni JAXB adatto per (un) marshalling.

Examples

Generazione di codice Java da semplice file XSD

Schema XSD (schema.xsd)

Il seguente schema xml (xsd) definisce un elenco di utenti con name e reputation attributi.

```
<?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

Ciò richiede il percorso dello strumento xjc (binari JDK) nella variabile del percorso del sistema operativo.

La generazione del codice può essere avviata usando

```
xjc schema.xsd
```

Questo genererà file java nella directory di lavoro.

File di risultati

Ci saranno alcuni commenti aggiuntivi, ma fondamentalmente i file java generati hanno il seguente aspetto:

```
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;
```

Leggi XJC online: https://riptutorial.com/it/java/topic/4538/xjc

Capitolo 184: XML XPath Evaluation

Osservazioni

Le espressioni XPath vengono utilizzate per navigare e selezionare uno o più nodi all'interno di un documento ad albero XML, ad esempio selezionando un determinato elemento o nodo di attributo.

Vedi questa raccomandazione del W3C per un riferimento su questa lingua.

Examples

Valutazione di un NodeList in un documento XML

Dato il seguente 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>
```

Quanto segue recupera tutti i nodi di example per il tag Java (Utilizzare questo metodo se si valuta XPath solo in XML una volta. Vedere altro esempio per quando più chiamate XPath vengono valutate nello stesso file 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
...
```

Analisi di più espressioni XPath in un singolo XML

Utilizzando lo stesso esempio di **Evaluating a NodeList in un documento XML** , ecco come si renderebbero più chiamate XPath in modo efficiente:

Dato il seguente 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>
```

Ecco come useresti XPath per valutare più espressioni in 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

...
```

Analisi di singoli XPath Expression più volte in un XML

In questo caso, si desidera che l'espressione sia compilata prima delle valutazioni, in modo che ogni chiamata da evaluate non compile la stessa espressione. La semplice sintassi sarebbe:

```
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
```

Complessivamente, due chiamate a XPathExpression.evaluate() saranno molto più efficienti di due

chiamate a XPath.evaluate() .

Leggi XML XPath Evaluation online: https://riptutorial.com/it/java/topic/4148/xml-xpath-evaluation

Capitolo 185: XOM - Modello a oggetti XML

Examples

Leggere un file XML

Per caricare i dati XML con XOM dovrai creare un Builder da cui puoi creare un Document .

```
Builder builder = new Builder();
Document doc = builder.build(file);
```

Per ottenere l'elemento root, il genitore più alto nel file xml, è necessario utilizzare getRootElement() nell'istanza Document .

```
Element root = doc.getRootElement();
```

Ora la classe Element ha molti metodi a portata di mano che rendono la lettura di xml davvero facile. Alcuni dei più utili sono elencati di seguito:

- getChildElements(String name) : restituisce un'istanza di Elements che funge da matrice di elementi
- getFirstChildElement(String name) - restituisce il primo elemento figlio con quel tag.
- getValue() - restituisce il valore all'interno dell'elemento.
- getAttributeValue(String name) - restituisce il valore di un attributo con il nome specificato.

Quando chiami getChildElements() ottieni un'istanza di Elements . Da questo puoi scorrere e chiamare il metodo get(int index) su di esso per recuperare tutti gli elementi all'interno.

```
Elements colors = root.getChildElements("color");
for (int q = 0; q < colors.size(); q++){
    Element color = colors.get(q);
}
```

Esempio: ecco un esempio di lettura di un file XML:

File XML:

```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>
```

Codice per leggerlo e stamparlo:

```
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();
            }
        }
    }

}
```

Questo verrà stampato nella console:

```
Name: Smith, Dan
Age: 23 (years)
Favorite Color: green
----------------
Name: Autry, Bob
Age: 3 (months)
Favorite Color: N/A
----------------
```

Scrivere in un file XML

Scrivere su un file XML usando XOM è molto simile a leggerlo tranne che in questo caso stiamo creando le istanze invece di recuperarle dalla radice.

Per creare un nuovo elemento utilizzare l' Element(String name) costruttore Element(String name) . Dovrai creare un elemento radice in modo che tu possa facilmente aggiungerlo a un Document .

```
Element root = new Element("root");
```

La classe Element ha alcuni metodi utili per la modifica degli elementi. Sono elencati di seguito:

- appendChild(String name) - questo in pratica imposta il valore dell'elemento da nominare.
- appendChild(Node node) - questo renderà il node gli elementi padre. (Gli elementi sono nodi in modo da poter analizzare gli elementi).
- addAttribute(Attribute attribute) - aggiungerà un attributo all'elemento.

La classe Attribute ha un paio di diversi costruttori. Il più semplice è Attribute(String name, String value) .

Dopo aver aggiunto tutti gli elementi al tuo elemento radice, puoi trasformarlo in un Document . Document prenderà un Element come argomento nel suo costruttore.

È possibile utilizzare un Serializer per scrivere il tuo XML in un file. Sarà necessario creare un nuovo flusso di output da analizzare nel costruttore di Serializer .

```
FileOutputStream fileOutputStream = new FileOutputStream(file);
Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
serializer.setIndent(4);
serializer.write(doc);
```

Esempio

Codice:

```
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; }
    }

}
```

Questo sarà il contenuto di "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>

```

Leggi XOM - Modello a oggetti XML online: https://riptutorial.com/it/java/topic/5091/xom---modello-a-oggetti-xml

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Java Language	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, Confiqure, 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, ıɯɐɓoɟ əızuəʞ, 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, Sнађошfa□, 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	Accesso nativo Java	Ezekiel Baniaga, Stephan, Stephen C
3	affermare	Jonathan, Makoto, rajah9, RamenChef, The Guy with The Hat, Uri Agassi
4	Agenti Java	Display Name, mnoronha
5	Analisi XML usando le API JAXP	GPI
6	annotazioni	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 □ Перeúpa ツ
7	Apache Commons Lang	Jonathan Barbero
8	API di Reflection	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 Пepeúpa ツ
9	API Stack-Walking	manouti
10	AppDynamics e TIBCO BusinessWorks Instrumentation per una facile integrazione	Alexandre Grimaud
11	applet	ArcticLord, Enigo, MadProgrammer, ppeterka
12	Array	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, noɥʎdʎzɐɔ, 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, ɟolɐəz əɥɟ 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, ΦXocę □ Перейра ツ
13	Audio	Dac Saunders, Petter Friberg, RamenChef, TNT, tonirush, Tot Zam, Vogel612
14	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
15	Bandiere JVM	Configure, RamenChef
16	benchmark	esin88
17	BigDecimal	alain.janinm, Christian, Dth, Enigo, ggolding, Harish Gyanani, John Nash, Loris Securo, Łukasz Piaszczyk, Manish Kothari, mszymborski, RamenChef, sudo, xwoker
18	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
19	BufferedWriter	Andrii Abramov, fabian, Jorn Vernee, Robin, VatsalSura
20	ByteBuffer	Community, Jon Ericson, Jorn Vernee, Tarık Yılmaz, Tomasz Bawor, victorantunes, Vogel612
21	Calendario e le sue sottoclassi	Bob Rivers, cdm, kann, Makoto, mnoronha, ppeterka, Ram, VGR
22	Classe - Java Reflection	gobes, KIRAN KUMAR MATAM
23	Classe EnumSet	KIRAN KUMAR MATAM
24	Classe immutabile	Mykola Yashchenko
25	Classe interna locale	KIRAN KUMAR MATAM
26	Classi e oggetti	Community, Configure, 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
27	Classi nidificate e interiori	ChemicalFlash, DimaSan, fgb, hd84335, Mshnik, RamenChef, Sandesh, sargue, Slava Babin, Stephen C, tynn
28	classloader	FFY00, Flow, Holger, Makoto, Stephen C
29	Clonazione dell'oggetto	Ayush Bansal, Christophe Weis, Jonathan
30	Code e Deques	Ad Infinitum, Alek Mieczkowski, Androbin, DimaSan,

		engineercoding, ppeterka, RamenChef, rd22, Samk, Stephen C
31	Codifica dei caratteri	Ilya
32	collezioni	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
33	Collezioni alternative	mnoronha, ppeterka, Viacheslav Vedenin
34	Collezioni simultanee	GPI, Kenster, Powerlord, user2296600
35	Comandi di runtime	RamenChef
36	Comparabile e comparatore	Andrii Abramov, Conrad.Dean, Daniel Nugent, fabian, GPI, Hazem Farahat, JAVAC, Mshnik, Nolequen, Petter Friberg, Prateek Agarwal, sebkur, Stephen C
37	Compilatore Java - 'javac'	CraftedCart, Jatin Balodhi, Mark Stewart, nishizawa23, Stephen C, Снађошfa□, Tom Gijselinck
38	Compilatore Just in Time (JIT)	Liju Thomas, Stephen C
39	CompletableFuture	Adowrath, Kishore Tulsiani, WillShackleford
40	Confronto C ++	John DiFini
41	Console I / O	Aaron Franke, Ani Menon, Erkan Haspulat, Francesco Menzani, jayantS, Lankymart, Loris Securo, manetsus, Olivier Grégoire, Petter Friberg, rolve, Saagar Jha, Stephen C
42	Conversione da e verso le stringhe	Chirag Parmar, DarkV1, Gihan Chathuranga, Jabir, JonasCz, Kaushal28, Lachlan Dowding, Laurel, Maarten Bodewes, Matt Clark, PSo, RamenChef, Shaan, Stephen C, still_learning
43	Costruttori	Andrii Abramov, Asiat, BrunoDM, ced-b, Codebender, Dylan, George Bailey, Jeremy, Ralf Kleberhoff, RamenChef, Thomas Gerot, tynn, Vogel612
44	Creazione di immagini a livello di codice	alain.janinm, Dariusz, kajacx, Kenster, mnoronha
45	Crittografia RSA	Dennis Kriechel, Drunix, iqbal_cs, Maarten Bodewes, Nicktar, Shog9
46	Data di lezione	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

47	Date e ora (java.time.*)	Bilbo Baggins, bowmore, Michael Piefel, Miles, mnoronha, Simon , Squidward, Tarun Maganti, Vogel612, ΦXocę Пepeúpa ツ
48	Disassemblare e decompilare	ipsi, mnoronha
49	Divisione di una stringa in parti di lunghezza fissa	Bohemian
50	Documentazione del codice 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
51	Eccezioni e gestione delle eccezioni	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
52	Edizioni, versioni, rilasci e distribuzioni Java	Gal Dreiman, screab, Stephen C
53	Elaborazione degli argomenti della riga di comando	Burkhard, Michael von Wenckstern, Stephen C
54	elenchi	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 Пepeúpa ツ
55	Enum che inizia con il numero	Sugan
56	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 Пepeúpa ツ
57	Eredità	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
58	espressioni	1d0m3n30, Andreas, EJP, Li357, RamenChef, shmosel, Stephen C, Stephen Leppik
59	Espressioni regolari	Amani Kilumanga, Andy Thomas, Asaph, ced-b, Daniel M., fabian, hd84335, intboolstring, kaotikmynd, Laurel, Makoto, nhahtdh, ppeterka, Ram, RamenChef, Saif, Tot Zam, Unihedron, Vogel612
60	File 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
61	File JAR multi-release	manouti
62	FileUpload in AWS	Amit Gujarathi
63	Fluent Interface	bn., noscreenname, P.J.Meisch, RamenChef, TuringTux
64	FTP (File Transfer Protocol)	Kelvin Kellner
65	Funzionalità Java SE 7	compuhosny, RamenChef
66	Funzionalità Java SE 8	compuhosny, RamenChef, sun-solar-arrow
67	Generazione del codice Java	Tony
68	Generazione di numeri casuali	Arthur, David Grant, David Soroko, dorukayhan, F. Stephen Q, Kichiin, MasterBlaster, michaelbahr, rokonoid, Stephen C, Thodgnir
69	Generics	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 əuʇ qoq, Unihedron, Vlad-HC, Wesley, Wilson, yiwei, Yury Fedorov
70	Gestione della memoria Java	Daniel M., engineercoding, fgb, John Nash, jwd630, mnoronha, OverCoder, padippist, RamenChef, Squidward, Stephen C
71	Getter e setter	Fildor, Ironcache, Krös, martin, Petter Friberg, Stephen C, Sujith Niraikulathan, Thisaru Guruge, uzaif
72	Grafica 2D in Java	17slim, ABDUL KHALIQ
73	HttpURLConnection	Community, Datagrammar, EJP, Inzimam Tariq IT, JonasCz, kiedysktos, Mureinik, NageN, Stephen C, still_learning
74	I flussi	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
75	Il comando Java - 'java' e 'javaw'	4444, Ben, mnoronha, Stephen C, Vogel612
76	Il percorso di classe	Aaron Digulla, GPI, K'', Kenster, Ruslan Ulanov, Stephen C, trashgod
77	Implementazione Java	garg10may, nishizawa23, Pseudonym Patel, RamenChef, Smit, Stephen C
78	Implementazioni del sistema di plugin Java	Alexiy
79	Imposta	A_Arnold, atom, ced-b, Chirag Parmar, Daniel Stradowski, demongolem, DimaSan, fabian, Kaushal28, Kenster
80	incapsulamento	Adam Ratzman, Adil, Daniel M., Drayke, VISHWANATH N P
81	InputStreams e	akgren_soar, EJP, Gubbel, J Atkin, Jens Schauder, John Nash,

	OutputStreams	Kip, KIRAN KUMAR MATAM, Matt Clark, Michael, RamenChef, Stephen C, Vogel612
82	Insidie di Java - Nulls e NullPointerException	17slim, Andrii Abramov, Daniel Nugent, dorukayhan, fabian, François Cassin, Miles, Stephen C, Zircon
83	Insidie di Java - Problemi di prestazioni	Dorian, GPI, John Starich, Jorn Vernee, Michał Rybak, mnoronha, ppeterka, Sharon Rozinsky, steffen, Stephen C, xTrollxDudex
84	Insidie di Java - Thread e concorrenza	dorukayhan, james large, Stephen C
85	Insidie di Java - Utilizzo delle eccezioni	Bhoomika, bruno, dimo414, Gal Dreiman, hd84335, SachinSarawgi, scorpp, Stephen C, Stephen Leppik, user3105453
86	Insidie Java - Sintassi della lingua	Alex T., Cody Gray, Enwired, Friederike, Gal Dreiman, hd84335, Hiren, Peter Rader, piyush_baderia, RamenChef, Ravindra HV, RudolphEst, Stephen C, Todd Sewell, user3105453
87	Installazione di Java (Standard Edition)	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, Snađoшfa□, tonirush, Walery Strauch, Ze Rubeus
88	interfacce	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
89	Interfacce funzionali	Andreas
90	Interfaccia Dequeue	Suketu Patel
91	Invio di metodi dinamici	Jeet
92	Iteratore e Iterable	Abubakkar, Comic Sans, Dariusz, Hulk, Lukas Knuth, RamenChef, Stephen C, user1121883, WillShackleford
93	Java Native Interface	Coffee Ninja, Fjoni Yzeiri, Jorn Vernee, RamenChef, Stephen C, user1803551
94	Java Performance Tuning	Gene Marin, jatanp, Stephen C, Vogel612
95	Java Virtual Machine (JVM)	Dushman, RamenChef, Rory McCrossan, Stephen Leppik
96	JavaBean	foxt7ot, J. Pichardo, James Fry, SaWo, Stephen C
97	JAXB	Dariusz, Drunix, fabian, hd84335, Jabir, ppeterka, Ram,

		Stephan, Thomas Fritsch, vallismortis, Walery Strauch
98	JAX-WS	ext1812, Jonathan Barbero, Stephen Leppik
99	JMX	esin88
100	JNDI	EJP, neohope, RamenChef
101	JShell	ostrichofevil, Sudip Bhandari
102	JSON in 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
103	JVM Tool Interface	desilijic
104	La classe java.util.Objects	mnoronha, RamenChef, Stephen C
105	Lambda Expressions	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
106	letterali	1d0m3n30, EJP, ParkerHalo, Stephen C, ThePhantomGamer
107	Lettori e scrittori	JD9999, KIRAN KUMAR MATAM, Mureinik, Stephen C, VatsalSura
108	LinkedHashMap	Amit Gujarathi, KIRAN KUMAR MATAM
109	Lista vs SET	KIRAN KUMAR MATAM
110	Localizzazione e internazionalizzazione	Code.IT, dimo414, Eduard Wirch, emotionlessbananas, Squidward, sun-solar-arrow
111	log4j / log4j2	Daniel Wild, Fildor, HCarrasko, hd84335, Mrunal Pagnis, Rens van der Heijden
112	Manager della sicurezza	alphaloop, hexafraction, Uux
113	Manipolazione bit	Aimee Borda, Blubberguy22, dosdebug, esin88, Gerald Mücke, Jorn Vernee, Kineolyan, mnoronha, Nayuki, Rednivrug, Ryan Hilbert, Stephen C, thatguy
114	Mappa Enum	KIRAN KUMAR MATAM
115	Mappe	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
116	Metodi di classe oggetto e costruttore	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
117	Metodi di raccolta della fabbrica	Jacob G.
118	Metodi predefiniti	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
119	Modello di memoria Java	Shree, Stephen C, Suminda Sirinath S. Dharmasena
120	Modifica del codice byte	bloo, Display Name, rakwaht, Squidward
121	Modificatori di non accesso	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
122	moduli	Jonathan, user140547
123	Motore JavaScript Nashorn	ben75, ekaerovets, Francesco Menzani, hd84335, Ilya, InitializeSahib, kasperjj, VatsalSura
124	Networking	Arthur, Burkhard, devnull69, DonyorM, glee8e, Grayson Croom, Ilya, Malt, Matej Kormuth, Matthieu, Mine_Stone, ppeterka, RamenChef, Stephen C, Tot Zam, vsav
125	NIO - Networking	Matthieu, mnoronha
126	NumberFormat	arpit pandey, John Nash, RamenChef, ΦXocę Пepeúpa ツ
127	Nuovo I / O file	dorukayhan, niheno, TuringTux
128	Oggetti immutabili	1d0m3n30, Bohemian, Holger, Idcmp, Jon Ericson, kristyna, Michael Piefel, Stephen C, Vogel612
129	Oggetti sicuri	Ankit Katiyar
130	operatori	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
131	Operazioni Java Floating Point	Dariusz, hd84335, HTNW, Ilya, Mr. P, Petter Friberg, ravthiru, Stephen C, Stephen Leppik, Vogel612
132	Opzionale	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
133	Ora locale	100rabh, A_Arnold, Alex, Andrii Abramov, Bob Rivers, Cache Staheli, DimaSan, Jasper, Kakarot, Kuroda, Manuel Vieda, Michael Piefel, phatfingers, RamenChef, Skylar Sutton, Vivek Anoop
134	Oracle Official Code Standard	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
135	Pacchi	JamesENL, KIRAN KUMAR MATAM
136	Polimorfismo	Adrian Krebs, Amani Kilumanga, Daniel LIn, Dushman, Kakarot, Lernkurve, Markus L, NageN, Pawan, Ravindra babu, Saiful Azad, Stephen C
137	Pool Executor, ExecutorService e Thread	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
138	Preferenze	RAnders00
139	Processi	Andy Thomas, Bob Rivers, ppeterka, vorburger, yitzih
140	Programmazione parallela con il framework Fork / Join	Community, Joe C
141	Programmazione simultanea (thread)	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
142	Proprietà Classe	17slim, Arthur, J Atkin, Jabir, KIRAN KUMAR MATAM, Marvin, peterh, Stephen C, VGR, vorburger
143	Registrazione (java.util.logging)	bn., Christophe Weis, Emil Sierżęga, P.J.Meisch, vallismortis
144	Remote Method Invocation (RMI)	RamenChef, smichel, Stephen C, user1803551, Vasiliy Vlasov
145	ricorsione	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
146	Riferimenti dell'oggetto	Andrii Abramov, arcy, Vasiliy Vlasov
147	Risorse (sul classpath)	Androbin, Christian, Emily Mabrey, Enwired, fabian, Gerald Mücke, Jesse van Bekkum, Kenster, Stephen C, timbooo, VGR, vorburger
148	Scanner	Alek Mieczkowski, Chirag Parmar, Community, Jon Ericson, JonasCz, Ram, RamenChef, Redterd, Stephen C, sun-solar-arrow, ΦΧοςę □ Περεύρα ツ
149	Scegliere le collezioni	John DiFini
150	serializzazione	akhilsk, Batty, Bilesh Ganguly, Burkhard, EJP, emotionlessbananas, faraa, GradAsso, KIRAN KUMAR MATAM, noscreenname, Onur, rokonoid, Siva Sainath Reddy Bandi, Vasilis Vasilatos, Vasiliy Vlasov
151	ServiceLoader	fabian, Florian Genser, Gerald Mücke
152	Servizio di stampa Java	Danilo Guimaraes, Leonardo Pina
153	Sicurezza e crittografia	John Nash, shibli049
154	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
155	Socket Java	Nikhil R
156	Sockets	Ordiel
157	SortedMap	Amit Gujarathi
158	StringBuffer	Amit Gujarathi
159	StringBuilder	Andrii Abramov, Cache Staheli, David Soroko, Enigo, fabian, fgb, JudgingNotJudging, KIRAN KUMAR MATAM, Nicktar, P.J.Meisch

		, Stephen C
160	stringhe	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, Snađoшfa□, 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 □ Перeúpa ツ
161	Strutture di controllo di base	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
162	sun.misc.Unsafe	4444, Daniel Nugent, Grexis, Stephen C, Suminda Sirinath S. Dharmasena
163	super parola chiave	Abhijeet
164	tabella hash	KIRAN KUMAR MATAM
165	Test unitario	Ironcache
166	ThreadLocal	Dariusz, Liju Thomas, Manish Kothari, Nithanim, taer
167	Tipi atomici	Daniel Nugent, Stephen C, Suminda Sirinath S. Dharmasena, xTrollxDudex
168	Tipi di dati di riferimento	Do Nhu Vy, giucal, Jorn Vernee, Lord Farquaad, Yohanes Khosiawan □□□

169	Tipi di dati primitivi	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
170	Tipi di riferimento	EJP, NageN, Thisaru Guruge
171	Tipo di conversione	4castle, Filip Smola, Joshua Carmody, Nick Donnelly, RamenChef , Squidward
172	Tokenizer di stringa	M M
173	Trappole comuni di 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
174	TreeMap e TreeSet	Malt, Stephen C
175	Utilizzando la parola chiave statica	17slim, Amir Rachum, Andrew Brooke, Arthur, ben75, CarManuel, Daniel Nugent, EJP, Hi I'm Frogatto, Mark Yisri, Sadiq Ali, Skepter, Squidward
176	Utilizzo di altri linguaggi di scripting in Java	Nikhil R
177	Utilizzo di ThreadPoolExecutor in applicazioni MultiThreaded.	Brendon Dugan
178	Valuta e denaro	Alexey Lagunov
179	Varargs (argomento variabile)	Daniel Nugent, Dushman, Omar Ayala, Rafael Pacheco, RamenChef, VGR, xsami
180	Visibilità (controllo dell'accesso ai membri di una classe)	Aasmund Eldhuset, Abhishek Balaji R, Catalina Island, Daniel M., intboolstring, Jonathan, Mark Yisri, Mureinik, NageN, ParkerHalo, Stephen C, Vogel612
181	WeakHashMap	Amit Gujarathi, KIRAN KUMAR MATAM
182	XJC	Danilo Guimaraes, fabian
183	XML XPath Evaluation	17slim, manouti
184	XOM - Modello a oggetti XML	Arthur, Makoto