eBook Gratuit

APPRENEZ Java Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#java

Table des matières

À propos

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

Chapitre 1: Démarrer avec le langage Java

Remarques

Le langage de programmation Java est ...

- **Usage général** : il est conçu pour être utilisé pour l'écriture de logiciels dans une grande variété de domaines d'application, et ne possède pas de fonctionnalités spécialisées pour un domaine spécifique.
- **Basé sur la classe** : sa structure d'objet est définie dans les classes. Les instances de classe ont toujours les champs et les méthodes spécifiés dans leurs définitions de classe (voir Classes et objets). Cela contraste avec les langages non basés sur des classes tels que JavaScript.
- **Typiquement** : le compilateur vérifie au moment de la compilation que les types de variables sont respectés. Par exemple, si une méthode attend un argument de type `String` , cet argument doit en fait être une chaîne lorsque la méthode est appelée.
- **Orientation objet** : la plupart des choses dans un programme Java sont des instances de classe, c'est-à-dire des paquets d'états (champs) et des comportements (méthodes qui fonctionnent sur les données et forment l' *interface de* l'objet avec le monde extérieur).
- **Portable** : Il peut être compilé sur n'importe quelle plate-forme avec `javac` et les fichiers de classe résultants peuvent s'exécuter sur toute plate-forme dotée d'une machine virtuelle Java.

Java est destiné à permettre aux développeurs d'applications "d'écrire une fois, d'exécuter n'importe où" (WORA), ce qui signifie que le code Java compilé peut s'exécuter sur toutes les plates-formes prenant en charge Java sans avoir besoin de recompiler.

Le code Java est compilé en bytecode (les fichiers `.class`) qui à leur tour sont interprétés par la machine virtuelle Java (JVM). En théorie, le bytecode créé par un compilateur Java devrait fonctionner de la même manière sur n'importe quelle machine virtuelle Java, même sur un autre type d'ordinateur. La JVM peut (et dans les programmes réels) choisir de compiler dans les commandes natives de la machine les parties du bytecode qui sont souvent exécutées. C'est ce qu'on appelle la compilation "Just-in-time (JIT)".

Editions et versions Java

Il existe trois "éditions" de Java définies par Sun / Oracle:

- *Java Standard Edition (SE)* est l'édition conçue pour une utilisation générale.
- *Java Enterprise Edition (EE)* ajoute une gamme de fonctionnalités pour la création de services «d'entreprise» en Java. Java EE est couvert séparément .

- *Java Micro Edition (ME)* est basé sur un sous-ensemble de *Java SE* et est destiné à être utilisé sur de petits périphériques avec des ressources limitées.

Il existe une rubrique distincte sur les éditions Java SE / EE / ME .

Chaque édition a plusieurs versions. Les versions de Java SE sont répertoriées ci-dessous.

Installation de Java

Il existe une rubrique distincte sur l' installation de Java (Standard Edition) .

Compiler et exécuter des programmes Java

Il y a des sujets séparés sur:

- Compilation du code source Java
- Déploiement Java, y compris la création de fichiers JAR
- Exécution d'applications Java
- Le Classpath

Et après?

Voici des liens vers des sujets pour continuer à apprendre et à comprendre le langage de programmation Java. Ces sujets sont les bases de la programmation Java pour vous aider à démarrer.

- Types de données primitifs en Java
- Opérateurs en Java
- Cordes en Java
- Structures de contrôle de base en Java
- Classes et objets en Java
- Tableaux en Java
- Normes de code Java

Essai

Bien que Java ne prenne pas en charge le test dans la bibliothèque standard, il existe des bibliothèques tierces conçues pour prendre en charge les tests. Les deux bibliothèques de tests unitaires les plus populaires sont:

- JUnit (site officiel)
- TestNG (Site Officiel)

Autre

- Les modèles de conception pour Java sont traités dans les modèles de conception .
- La programmation pour Android est couverte par Android .
- Les technologies Java Enterprise Edition sont couvertes dans Java EE .
- Les technologies Oracle JavaFX sont couvertes dans JavaFX .

1. Dans la section **Versions** , la date de *fin de vie (gratuite)* correspond au moment où Oracle cessera de publier d'autres mises à jour de Java SE sur ses sites de téléchargement publics. Les clients qui ont besoin d'un accès continu aux correctifs critiques, aux correctifs de sécurité et à la maintenance générale de Java SE peuvent bénéficier d'une assistance à long terme via Oracle Java SE Support .

Versions

Version Java SE	Nom de code	Fin de vie (gratuit [1])	Date de sortie
Java SE 9 (accès anticipé)	*Aucun*	avenir	2017-07-27
Java SE 8	Araign? e	avenir	2014-03-18
Java SE 7	Dauphin	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	Merlin	avant 2009-11-04	2002-02-06
Java SE 1.3	Crécerelle	avant 2009-11-04	2000-05-08
Java SE 1.2	Cour de récréation	avant 2009-11-04	1998-12-08
Java SE 1.1	*Aucun*	avant 2009-11-04	1997-02-19
Java SE 1.0	Chêne	avant 2009-11-04	1996-01-21

Examples

Création de votre premier programme Java

Créez un nouveau fichier dans votre éditeur de texte ou IDE nommé `HelloWorld.java` . Puis collez ce bloc de code dans le fichier et sauvegardez:

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

```
}
```

Courir en direct sur Ideone

Remarque: Pour que Java puisse reconnaître cela comme une `public class` (et ne pas générer d'erreur de compilation), le nom du fichier doit être identique à celui de la classe (`HelloWorld` dans cet exemple) avec une extension `.java` . Il devrait également y avoir un modificateur d'accès `public` avant lui.

Les conventions de dénomination recommandent que les classes Java commencent par un caractère majuscule et soient au format de casse camel (dans lequel la première lettre de chaque mot est en majuscule). Les conventions recommandent les caractères de soulignement (`_`) et les signes dollar (`$`).

Pour compiler, ouvrez une fenêtre de terminal et accédez au répertoire de `HelloWorld.java` :

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

Note: cd est la commande du terminal pour changer de répertoire.

Entrez `javac` suivi du nom du fichier et de l'extension comme suit:

```
$ javac HelloWorld.java
```

Il est assez courant d'obtenir l'erreur `'javac' is not recognized as an internal or external command, operable program or batch file.` même lorsque vous avez installé le `JDK` et que vous pouvez exécuter le programme à partir d' `IDE` exemple. `eclipse` etc. Comme le chemin n'est pas ajouté à l'environnement par défaut.

Dans le cas où vous obtenez cela sur Windows, pour résoudre, essayez d'abord de naviguer vers votre chemin `javac.exe` , il est très probable qu'il se trouve dans votre `C:\Program Files\Java\jdk(version number)\bin` . Ensuite, essayez de l'exécuter avec ci-dessous.

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

Auparavant, lorsque nous `javac` c'était la même chose que la commande ci-dessus. Seulement dans ce cas, votre `OS` savait où se trouvait `javac` . Disons-le maintenant, de cette façon, vous n'avez pas à taper tout le chemin à chaque fois. Nous devrions ajouter ceci à notre `PATH`

Pour modifier la variable d'environnement `PATH` dans Windows XP / Vista / 7/8/10:

- Panneau de configuration ⇒ Système ⇒ Paramètres système avancés
- Passer à l'onglet "Avancé" ⇒ Variables d'environnement
- Dans "Variables système", faites défiler pour sélectionner "PATH" ⇒ Modifier

Vous ne pouvez pas défaire ceci alors soyez prudent. Commencez par copier votre chemin existant dans le bloc-notes. Ensuite, pour obtenir le PATH exact sur votre `javac` naviguez manuellement dans le dossier où réside `javac` , cliquez sur la barre d'adresse, puis copiez-le. Il

devrait ressembler à `c:\Program Files\Java\jdk1.8.0_xx\bin`

Dans le champ "Valeur de la variable", collez-le **AVANT** de tous les répertoires existants, suivi d'un point-virgule (;). **NE SUPPRIMEZ** aucune entrée existante.

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

Maintenant ceci devrait résoudre.

Pour les systèmes basés sur Linux, essayez ici .

Remarque: *La commande `javac` appelle le compilateur Java.*

Le compilateur générera alors un fichier de bytecode appelé `HelloWorld.class` qui pourra être exécuté dans la machine virtuelle Java (JVM) . Le compilateur de langage de programmation Java, `javac` , lit les fichiers source écrits dans le langage de programmation Java et les compile en fichiers de classe `bytecode` . En option, le compilateur peut également traiter les annotations trouvées dans les fichiers source et de classe à l'aide de l'API `Pluggable` Annotation Processing. Le compilateur est un outil de ligne de commande mais peut également être appelé à l'aide de l'API Java Compiler.

Pour exécuter votre programme, entrez `java` suivi du nom de la classe qui contient la méthode `main` (`HelloWorld` dans notre exemple). Notez que la `.class` est omise:

```
$ java HelloWorld
```

Remarque: *la commande `java` exécute une application Java.*

Cela va sortir sur votre console:

> Bonjour le monde!

Vous avez correctement codé et construit votre tout premier programme Java!

Remarque: pour que les commandes Java (`java` , `javac` , etc.) soient reconnues, vous devez vous assurer que:

- Un JDK est installé (par exemple Oracle , OpenJDK et d'autres sources)
- Vos variables d'environnement sont correctement configurées

Vous devrez utiliser un compilateur (`javac`) et un exécuteur (`java`) fournis par votre JVM. Pour savoir quelles versions vous avez installées, entrez `java -version` et `javac -version` sur la ligne de commande. Le numéro de version de votre programme sera imprimé dans le terminal (par exemple `1.8.0_73`).

Regard sur le programme Hello World

Le programme "Hello World" contient un seul fichier, composé d'une définition de classe `HelloWorld` , d'une méthode `main` et d'une instruction dans la méthode `main` .

```
public class HelloWorld {
```

Le mot `class` clé `class` commence la définition de classe pour une classe nommée `HelloWorld` . Chaque application Java contient au moins une définition de classe (Informations complémentaires sur les classes).

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

Il s'agit d'une méthode de point d'entrée (définie par son nom et par la signature de `public static void main(String[])`) à partir de laquelle la `JVM` peut exécuter votre programme. Chaque programme Java devrait en avoir un. C'est:

- `public` : cela signifie que la méthode peut également être appelée de l'extérieur du programme. Voir Visibilité pour plus d'informations à ce sujet.
- `static` : signifiant qu'il existe et peut être exécuté seul (au niveau de la classe sans créer d'objet).
- `void` : signifiant qu'il ne renvoie aucune valeur. ***Note:*** *Ceci est différent de C et C ++ où un code de retour tel que* `int` *est attendu (le chemin de Java est* `System.exit()` *).*

Cette méthode principale accepte:

- Un tableau (généralement appelé `args`) de `String` est transmis en tant qu'argument à la fonction principale (par exemple à partir d' arguments de ligne de commande).

Presque tout cela est requis pour une méthode de point d'entrée Java.

Pièces non requises:

- Le nom `args` est un nom de variable, il peut donc être appelé tout ce que vous voulez, bien qu'il soit généralement appelé `args` .
- Que son type de paramètre soit un tableau (`String[] args`) ou Varargs (`String... args`) n'a pas d'importance, car les tableaux peuvent être transmis à varargs.

Remarque: Une seule application peut comporter plusieurs classes contenant une méthode de point d'entrée (`main`). Le point d'entrée de l'application est déterminé par le nom de la classe passé en argument à la commande `java` .

A l'intérieur de la méthode principale, nous voyons la déclaration suivante:

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

Décomposons cette déclaration élément par élément:

Élément	Objectif
`System`	Cela signifie que l'expression suivante fera appel à la classe `System` , à partir du

Élément	Objectif
	package `java.lang` .
`.`	c'est un "opérateur de points". Les opérateurs de points vous permettent d'accéder aux membres d'une classe [1] ; c'est-à-dire ses champs (variables) et ses méthodes. Dans ce cas, cet opérateur de points vous permet de référencer le champ `out` statique dans la classe `System` .
`out`	c'est le nom du champ statique de type `PrintStream` dans la classe `System` contenant la fonctionnalité de sortie standard.
`.`	c'est un autre opérateur de points. Cet opérateur de point donne accès à la méthode `println` dans la variable `out` .
`println`	c'est le nom d'une méthode dans la classe PrintStream. Cette méthode en particulier imprime le contenu des paramètres dans la console et insère une nouvelle ligne après.
`(`	cette parenthèse indique qu'une méthode est en cours d'accès (et non un champ) et commence les paramètres passés dans la méthode `println` .
`"Hello, World!"`	c'est le littéral String qui est passé en paramètre dans la méthode `println` . Les guillemets doubles à chaque extrémité délimitent le texte en tant que chaîne.
`)`	cette parenthèse signifie la fermeture des paramètres passés dans la méthode `println` .
`;`	ce point-virgule marque la fin de la déclaration.

Remarque: *Chaque instruction en Java doit se terminer par un point-virgule (;).*

Le corps de la méthode et le corps de la classe sont alors fermés.

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

Voici un autre exemple démontrant le paradigme OO. Modélisons une équipe de football avec un (oui, un!) Membre. Il peut y en avoir plus, mais nous en discuterons quand nous arriverons aux tableaux.

Tout d'abord, définissons notre classe d' `Team` :

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

Maintenant, définissons notre classe de `Member` :

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

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

Pourquoi utilisons-nous `private` ici? Eh bien, si quelqu'un voulait connaître votre nom, il devrait vous le demander directement, au lieu d'aller dans votre poche et de retirer votre carte de sécurité sociale. Ce `private` fait quelque chose comme ça: il empêche les entités extérieures d'accéder à vos variables. Vous ne pouvez renvoyer que `private` membres `private` via les fonctions de lecture (présentées ci-dessous).

Après avoir rassemblé le tout et ajouté les méthodes principales et les méthodes décrites ci-dessus, nous avons:

```
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
    }
}
```

Sortie:

```
Aurieel
light
10
1
```

Courir sur ideone

Encore une fois, la méthode `main` dans la classe `Test` est le point d'entrée de notre programme. Sans la méthode `main` , nous ne pouvons pas dire à la machine virtuelle Java (JVM) où commencer l'exécution du programme.

---

1 - Comme la classe `HelloWorld` a peu de rapport avec la classe `System` , elle ne peut accéder qu'aux données `public` .

Lire Démarrer avec le langage Java en ligne: https://riptutorial.com/fr/java/topic/84/demarrer-avec-le-langage-java

Chapitre 2: Affirmer

Syntaxe

- affirmer l' *expression1* ;
- assert *expression1* : *expression2* ;

Paramètres

Paramètre	Détails
expression1	L'instruction d'assertion renvoie une `AssertionError` si cette expression est `false` .
expression2	Optionnel. Lorsqu'il est utilisé, `AssertionError` émis par l'instruction assert a ce message.

Remarques

Par défaut, les assertions sont désactivées lors de l'exécution.

Pour activer les assertions, vous devez exécuter java avec l' `-ea` .

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

Les assertions sont des instructions qui génèrent une erreur si leur expression est `false` . Les assertions ne doivent être utilisées que pour *tester le* code; ils ne devraient jamais être utilisés en production.

Examples

Vérification de l'arithmétique avec 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;
```

Lire Affirmer en ligne: https://riptutorial.com/fr/java/topic/407/affirmer

Chapitre 3: Agents Java

Examples

Modification de classes avec des agents

Tout d'abord, assurez-vous que l'agent utilisé possède les attributs suivants dans le fichier Manifest.mf:

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

Le démarrage d'un agent Java permettra à l'agent d'accéder à la classe Instrumentation. Avec Instrumentation, vous pouvez appeler *addTransformer (Transformateur ClassFileTransformer)* . ClassFileTransformers vous permettra de réécrire les octets des classes. La classe n'a qu'une seule méthode qui fournit le ClassLoader qui charge la classe, le nom de la classe, une instance java.lang.Class, c'est ProtectionDomain et enfin les octets de la classe elle-même.

Cela ressemble à ceci:

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

La modification d'une classe uniquement à partir d'octets peut prendre des années. Pour remédier à cela, il existe des bibliothèques qui peuvent être utilisées pour convertir les octets de classe en quelque chose de plus utilisable.

Dans cet exemple, je vais utiliser ASM, mais d'autres alternatives comme Javassist et BCEL ont des fonctionnalités similaires.

```
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;
}
```

De là, des modifications peuvent être apportées à l'objet ClassNode. Cela rend incroyablement facile l'accès aux champs / méthodes. De plus, avec l'API Tree d'ASM, modifier le bytecode des méthodes est un jeu d'enfant.

Une fois les modifications terminées, vous pouvez convertir le ClassNode en octets avec la

méthode suivante et les renvoyer dans la méthode de *transformation* :

```
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;
}
```

Ajout d'un agent à l'exécution

Les agents peuvent être ajoutés à une machine virtuelle Java lors de l'exécution. Pour charger un agent, vous devez utiliser *VirtualMachine.attatch (String String) de* l'API Attach. Vous pouvez ensuite charger un jar d'agent compilé avec la méthode suivante:

```
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);
    }
}
```

Cela n'appelle pas *premain ((String agentArgs, Instrumentation inst)* dans l'agent chargé, mais appelle *plutôt agentmain (String agentArgs, instrumentation inst)* . Cela nécessite que la *classe* d' *agent* soit définie dans l'agent Manifest.mf.

Mise en place d'un agent de base

La classe Premain contiendra la méthode *"premain (String agentArgs Instrumentation inst)"*

Voici un exemple:

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

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

Une fois compilé dans un fichier jar, ouvrez le manifeste et assurez-vous qu'il possède l'attribut Premain-Class.

Voici un exemple:

```
Premain-Class: PremainExample
```

Pour utiliser l'agent avec un autre programme Java "myProgram", vous devez définir l'agent dans les arguments JVM:

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

Lire Agents Java en ligne: https://riptutorial.com/fr/java/topic/1265/agents-java

Chapitre 4: Analyse XML à l'aide des API JAXP

Remarques

XML Parsing est l'interprétation des documents XML afin de manipuler leur contenu à l'aide de constructions sensibles, qu'il s'agisse de "nœuds", "attributs", "documents", "espaces de noms" ou d'événements liés à ces constructions.

Java possède une API native pour la gestion des documents XML, appelée JAXP ou API Java pour le traitement XML . JAXP et une implémentation de référence ont été regroupés avec chaque version de Java depuis Java 1.4 (JAXP v1.1) et ont évolué depuis. Java 8 fourni avec JAXP version 1.6.

L'API propose différentes manières d'interagir avec les documents XML, à savoir:

- L'interface DOM (Document Object Model)
- L'interface SAX (API simple pour XML)
- L'interface StAX (API de streaming pour XML)

Principes de l'interface DOM

L'interface DOM a pour but de fournir une manière conforme à l'interprétation du XML du W3C DOM . Diverses versions de JAXP ont pris en charge divers niveaux de spécification DOM (jusqu'au niveau 3).

Dans l'interface du modèle d'objet de document, un document XML est représenté sous la forme d'une arborescence, en commençant par "l'élément de document". Le type de base de l'API est le type `Node` , il permet de naviguer d'un `Node` vers son parent, ses enfants ou ses frères (bien que tous les `Node` ne puissent pas avoir d'enfants, par exemple, les nœuds de `Text` sont définitifs dans l'arborescence). et ne jamais avoir childre). Les balises XML sont représentées sous forme d' `Element` s, qui étendent notamment le `Node` avec des méthodes liées aux attributs.

L’interface DOM est très utile car elle permet d’analyser des documents XML en tant qu’arbres, et de modifier facilement l’arbre construit (ajout de nœud, suppression, copie, etc.) et enfin sa sérialisation (retour au disque).) publier des modifications. Cela a cependant un prix: l'arbre réside dans la mémoire, par conséquent, les arbres DOM ne sont pas toujours pratiques pour les documents XML volumineux. De plus, la construction de l'arborescence n'est pas toujours le moyen le plus rapide de gérer le contenu XML, surtout si toutes les parties du document XML ne l'intéressent pas.

Principes de l'interface SAX

L'API SAX est une API orientée événement pour traiter les documents XML. Dans ce modèle, les composants d'un document XML sont interprétés comme des événements (par exemple, "un tag a été ouvert", "un tag a été fermé", "un noeud de texte a été rencontré", "un commentaire a été rencontré"). ..

L'API SAX utilise une approche "d'analyse par poussée", dans laquelle un `Parser` SAX est chargé d'interpréter le document XML et appelle des méthodes sur un délégué (un `ContentHandler`) pour traiter tout événement trouvé dans le document XML. Généralement, on n'écrit jamais un analyseur, mais on fournit un gestionnaire pour rassembler toutes les informations nécessaires à partir du document XML.

L'interface SAX surmonte les limitations de l'interface DOM en ne conservant que le minimum de données nécessaires au niveau de l'analyseur (ex: contextes des espaces de noms, état de validation), seules les informations conservées par le `ContentHandler` - dont vous êtes responsable tenu en mémoire. Le compromis est qu'il n'y a aucun moyen de "remonter le temps / le document XML" avec une telle approche: alors que DOM permet à un `Node` de revenir à son parent, il n'y a pas de possibilité dans SAX.

Principes de l'interface StAX

L'API StAX adopte une approche similaire au traitement de XML en tant qu'API SAX (c'est-à-dire pilotée par des événements), la seule différence très significative étant que StAX est un analyseur d'extraction (où SAX était un analyseur poussé). Dans SAX, l' `Parser` est en contrôle et utilise des rappels sur le `ContentHandler` . Dans Stax, vous appelez l'analyseur et contrôlez quand / si vous souhaitez obtenir le prochain "événement" XML.

L'API commence par XMLStreamReader (ou XMLEventReader), qui sont les passerelles par lesquelles le développeur peut demander à `nextEvent()` , de manière itérative.

Examples

Analyse et navigation d'un document à l'aide de l'API DOM

Considérant le document suivant:

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

On peut utiliser le code suivant pour construire un arbre DOM à partir d'une `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");
}
}
```

Le code donne les informations suivantes:

```
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
```

Analyse d'un document à l'aide de l'API StAX

Considérant le document suivant:

```
<?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>
```

On peut utiliser le code suivant pour l'analyser et créer une carte des titres de livres par identifiant de livre.

```
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);
}
```

Cela produit:

```
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}
```

Dans cet exemple, il faut faire attention à quelques choses:

1. L'utilisation de `xmlStreamReader.getAttributeValue` fonctionne car nous avons vérifié en premier que l'analyseur est dans l'état `START_ELEMENT` . Dans d' autres Etats Evey (sauf `ATTRIBUTES`), l'analyseur est chargé de lancer `IllegalStateException` , car les attributs ne peuvent apparaître au début des éléments.

2. Il en va de même pour `xmlStreamReader.getTextContent()` , cela fonctionne car nous sommes à un `START_ELEMENT` et nous savons dans ce document que l'élément `<book>` n'a pas de nœud enfant non textuel.

Pour l'analyse de documents plus complexes (éléments plus profonds, imbriqués, ...), il est `BookParser` de "déléguer" l'analyseur à des sous-méthodes ou à d'autres objets, par exemple, avoir une classe ou une méthode `BookParser` et traiter tous les éléments. de START_ELEMENT à END_ELEMENT de la balise XML du livre.

On peut également utiliser un objet `Stack` pour conserver des données importantes dans l'arbre.

Lire Analyse XML à l'aide des API JAXP en ligne: https://riptutorial.com/fr/java/topic/3943/analyse-xml-a-l-aide-des-api-jaxp

Chapitre 5: Annotations

Introduction

En Java, une annotation est une forme de métadonnées syntaxiques pouvant être ajoutées au code source Java. Il fournit des données sur un programme qui ne fait pas partie du programme lui-même. Les annotations n'ont aucun effet direct sur le fonctionnement du code qu'elles annotent. Les classes, méthodes, variables, paramètres et packages peuvent être annotés.

Syntaxe

- @AnnotationName // 'Annotation du marqueur' (pas de paramètres)
- @AnnotationName (someValue) // définit le paramètre avec le nom 'value'
- @AnnotationName (param1 = valeur1) // paramètre nommé
- @AnnotationName (param1 = valeur1, param2 = valeur2) // plusieurs paramètres nommés
- @AnnotationName (param1 = {1, 2, 3}) // paramètre de tableau nommé
- @AnnotationName ({value1}) // tableau avec un seul élément comme paramètre avec le nom 'value'

Remarques

Types de paramètres

Seules les expressions constantes des types suivants sont autorisées pour les paramètres, ainsi que les tableaux de ces types:

- `String`
- `Class`
- types primitifs
- Types enum
- Types d'annotation

Examples

Annotations intégrées

L'édition Standard de Java comporte des annotations prédéfinies. Vous n'avez pas besoin de les définir vous-même et vous pouvez les utiliser immédiatement. Ils permettent au compilateur de permettre une vérification fondamentale des méthodes, des classes et du code.

@Passer outre

Cette annotation s'applique à une méthode et dit que cette méthode doit remplacer une méthode

de superclasse ou implémenter une définition de méthode de superclasse abstraite. Si cette annotation est utilisée avec un autre type de méthode, le compilateur générera une erreur.

Superclasse en béton

```
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 abstraite

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

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

Ne marche pas

```
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);
    }
}
```

L'objectif principal est de détecter les erreurs de frappe, où vous pensez que vous écrasez une méthode, mais en en définissant une nouvelle.

```
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");
    }
}
```

Notez que la signification de `@Override` a changé avec le temps:

- Dans Java 5, cela signifiait que la méthode annotée devait remplacer une méthode non abstraite déclarée dans la chaîne de la superclasse.
- A partir de Java 6, il est *également* satisfait si la méthode annotée implémente une méthode abstraite déclarée dans la hiérarchie des classes / interfaces des classes.

(Cela peut occasionnellement poser problème lors du portage du code vers Java 5.)

@Précis

Cela marque la méthode comme obsolète. Il peut y avoir plusieurs raisons à cela:

- l'API est imparfaite et difficile à réparer,
- l'utilisation de l'API est susceptible de conduire à des erreurs,
- l'API a été remplacée par une autre API,
- l'API est obsolète,
- l'API est expérimentale et est sujette à des modifications incompatibles,
- ou toute combinaison de ce qui précède.

La raison spécifique de l'abandon se trouve généralement dans la documentation de l'API.

L'annotation provoquera une erreur du compilateur si vous l'utilisez. Les IDE peuvent également mettre en évidence cette méthode d'une manière ou d'une autre

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

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

@Supprimer les avertissements

Dans presque tous les cas, lorsque le compilateur émet un avertissement, l'action la plus appropriée consiste à corriger la cause. Dans certains cas (code Generics utilisant du code pré-générique non sécurisé, par exemple), cela peut ne pas être possible et il est préférable de supprimer les avertissements que vous attendez et que vous ne pouvez pas corriger. Vous pouvez donc voir plus clairement les avertissements inattendus.

Cette annotation peut être appliquée à une classe, une méthode ou une ligne entière. Il prend la catégorie d'avertissement comme paramètre.

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

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

Il est préférable de limiter la portée de l'annotation autant que possible pour éviter la suppression des avertissements inattendus. Par exemple, en limitant la portée de l'annotation à une seule ligne:

```
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
```

Les avertissements pris en charge par cette annotation peuvent varier d'un compilateur à l'autre. Seuls les avertissements `unchecked` et `deprecation` sont spécifiquement mentionnés dans le JLS. Les types d'avertissement non reconnus seront ignorés.

@SafeVarargs

En raison de l'effacement de type, la `void method(T... t)` sera convertie en `void method(Object[] t)` ce qui signifie que le compilateur n'est pas toujours en mesure de vérifier que l'utilisation de varargs est de type sécurisé. Par exemple:

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

Il existe des cas où l'utilisation est sûre, auquel cas vous pouvez annoter la méthode avec l'annotation `SafeVarargs` pour supprimer l'avertissement. Cela cache évidemment l'avertissement si votre utilisation est dangereuse aussi.

@Interface fonctionnelle

C'est une annotation facultative utilisée pour marquer une interface fonctionnelle. Cela amènera le compilateur à se plaindre s'il ne se conforme pas à la spécification de FunctionalInterface (possède une seule méthode abstraite)

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

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

Vérification des annotations d'exécution par réflexion

L'API de réflexion de Java permet au programmeur d'effectuer diverses vérifications et opérations sur les champs de classe, les méthodes et les annotations pendant l'exécution. Toutefois, pour qu'une annotation soit visible au moment de l'exécution, `RetentionPolicy` doit être remplacée par `RUNTIME` , comme illustré dans l'exemple ci-dessous:

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

Définition des types d'annotation

Les types d'annotation sont définis avec `@interface` . Les paramètres sont définis de manière similaire aux méthodes d'une interface régulière.

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

Les valeurs par défaut

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

Méta-annotations

Les méta-annotations sont des annotations pouvant être appliquées aux types d'annotation. Une méta-annotation prédéfinie spéciale définit comment les types d'annotation peuvent être utilisés.

@Cible

La méta-annotation `@Target` limite les types `@Target` l'annotation peut être appliquée.

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

Plusieurs valeurs peuvent être ajoutées en utilisant la notation de tableau, par exemple `@Target({ElementType.FIELD, ElementType.TYPE})`

Valeurs disponibles

ElementType	cible	exemple d'utilisation sur l'élément cible
ANNOTATION_TYPE	types d'annotation	`@Retention(RetentionPolicy.RUNTIME)` `@interface MyAnnotation`
CONSTRUCTEUR	constructeurs	`@MyAnnotation` `public MyClass() {}`
CHAMP	champs, constantes enum	`@XmlAttribute` `private int count;`
VARIABLE LOCALE	déclarations de variables à l'intérieur des méthodes	`for (@LoopVariable int i = 0; i < 100; i++) {`

ElementType	cible	exemple d'utilisation sur l'élément cible

		`    @Unused`
		`    String resultVariable;`
		`}`

PAQUET	package (dans `package-info.java`)	`@Deprecated` `package very.old;`
MÉTHODE	méthodes	`@XmlElement` `public int getCount() {...}`

PARAMÈTRE	paramètres de méthode / constructeur	`public Rectangle(`
		`       @NamedArg("width") double width,`
		`       @NamedArg("height") double height) {`
		`    ...`
		`}`

TYPE	classes, interfaces, énumérations	`@XmlRootElement` `public class Report {}`

Java SE 8

ElementType	cible	exemple d'utilisation sur l'élément cible
TYPE_PARAMETER	Type de déclaration de paramètres	`public <@MyAnnotation T> void f(T t) {}`
TYPE_USE	Utilisation d'un type	`Object o = "42";` `String s = (@MyAnnotation String) o;`

@Rétention

La méta-annotation `@Retention` définit la visibilité de l'annotation pendant le processus de compilation ou d'exécution des applications. Par défaut, les annotations sont incluses dans les

fichiers `.class` , mais ne sont pas visibles lors de l'exécution. Pour rendre une annotation accessible au moment de l'exécution, `RetentionPolicy.RUNTIME` doit être défini sur cette annotation.

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

Valeurs disponibles

RetentionPolicy	Effet
CLASSE	L'annotation est disponible dans le fichier `.class` , mais pas à l'exécution
DUREE	L'annotation est disponible au moment de l'exécution et est accessible via la réflexion
LA SOURCE	L'annotation est disponible à la compilation, mais n'est pas ajoutée aux fichiers `.class` . L'annotation peut être utilisée, par exemple, par un processeur d'annotation.

@Documenté

La `@Documented` méta-annotation est utilisée pour marquer des annotations dont l' utilisation doit être documentée par des générateurs de documentation de l' API comme javadoc . Il n'a pas de valeurs. Avec `@Documented` , toutes les classes utilisant l'annotation le listeront sur leur page de documentation générée. Sans `@Documented` , il n'est pas possible de voir quelles classes utilisent l'annotation dans la documentation.

@Hérité

La méta-annotation `@Inherited` concerne les annotations appliquées aux classes. Il n'a pas de valeurs. Marquer une annotation comme `@Inherited` altère le fonctionnement de l'interrogation par annotation.

- Pour une annotation non héritée, la requête examine uniquement la classe en cours d'examen.
- Pour une annotation héritée, la requête vérifie également la chaîne de super-classe (récursivement) jusqu'à ce qu'une instance de l'annotation soit trouvée.

Notez que seules les super-classes sont interrogées: les annotations attachées aux interfaces dans la hiérarchie des classes seront ignorées.

@Repeatable

La méta-annotation `@Repeatable` été ajoutée à Java 8. Elle indique que plusieurs instances de

l'annotation peuvent être attachées à la cible de l'annotation. Cette méta-annotation n'a aucune valeur.

Obtenir des valeurs d'annotation au moment de l'exécution

Vous pouvez extraire les propriétés actuelles de l'annotation en utilisant Reflection pour récupérer la méthode ou le champ ou la classe à laquelle une annotation est appliquée, puis extraire les propriétés souhaitées.

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


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

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

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

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

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

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

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

    public static void main(String[] args) {
        AnnotationExample example = new AnnotationExample();
        try {
            example.testDefaults();
            example.testValues();
        } catch( Exception e ) {
            // Shouldn't throw any Exceptions
            System.err.println("Exception [" + e.getClass().getName() + "] - " +
```

```
e.getMessage());
            e.printStackTrace(System.err);
        }
    }
}
```

La sortie sera

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

Annotations répétées

Jusqu'à Java 8, deux instances de la même annotation ne pouvaient pas être appliquées à un seul élément. La solution de contournement standard consistait à utiliser une annotation de conteneur contenant un tableau d'une autre annotation:

```
// 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 fournit un moyen plus propre et plus transparent d'utiliser les annotations de conteneur, en utilisant l'annotation `@Repeatable` . Tout d'abord, nous ajoutons ceci à la classe `Author` :

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

Ceci indique à Java de traiter plusieurs annotations `@Author` comme si elles étaient entourées par

le conteneur `@Authors` . Nous pouvons également utiliser `Class.getAnnotationsByType()` pour accéder au tableau `@Author` par sa propre classe, plutôt que par son conteneur:

```
@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
        }
    }
}
```

Annotations héritées

Par défaut, les annotations de classe ne s'appliquent pas aux types qui les étendent. Cela peut être modifié en ajoutant l'annotation `@Inherited` à la définition d'annotation

Exemple

Considérez les 2 annotations suivantes:

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

et

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

Si trois classes sont annotées comme ceci:

```
@UninheritedAnnotationType
class A {
}

@InheritedAnnotationType
class B extends A {
}

class C extends B {
}
```

exécuter ce code

```
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));
```

imprimera un résultat similaire à celui-ci (selon les paquets de l'annotation):

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

Notez que les annotations ne peuvent être héritées que des classes, pas des interfaces.

Compiler le traitement du temps à l'aide du processeur d'annotations

Cet exemple montre comment effectuer la vérification du temps de compilation d'un élément annoté.

L'annotation

L'annotation `@Setter` est un marqueur pouvant être appliqué aux méthodes. L'annotation sera supprimée lors de la compilation et ne sera plus disponible par la suite.

```
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 {
}
```

Le processeur d'annotation

La classe `SetterProcessor` est utilisée par le compilateur pour traiter les annotations. Il vérifie si les méthodes annotées avec l'annotation `@Setter` sont `public` , `static` méthodes non `static` avec un nom commençant par `set` et ayant une lettre majuscule comme 4ème lettre. Si l'une de ces conditions n'est pas remplie, une erreur est écrite dans le `Messager` . Le compilateur écrit ceci dans stderr, mais d'autres outils pourraient utiliser ces informations différemment. Par exemple, l'EDI

NetBeans permet à l'utilisateur de spécifier des processeurs d'annotation utilisés pour afficher les messages d'erreur dans l'éditeur.

```
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();
    }

}
```

Emballage

Pour être appliqué par le compilateur, le processeur d'annotation doit être mis à la disposition du SPI (voir ServiceLoader).

Pour ce faire, un fichier texte `META-INF/services/javax.annotation.processing.Processor` doit être ajouté au fichier jar contenant le processeur d'annotation et l'annotation en plus des autres fichiers. Le fichier doit inclure le nom complet du processeur d'annotation, c'est-à-dire qu'il doit ressembler à ceci:

```
annotation.processor.SetterProcessor
```

Nous supposerons que le fichier jar s'appelle `AnnotationProcessor.jar` ci-dessous.

Exemple de classe annotée

La classe suivante est un exemple de classe dans le package par défaut, les annotations étant appliquées aux éléments corrects conformément à la stratégie de rétention. Cependant, seul le processeur d'annotation considère uniquement la seconde méthode comme une cible d'annotation valide.

```
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) {}

}
```

Utilisation du processeur d'annotations avec javac

Si le processeur d'annotations est détecté à l'aide du SPI, il est automatiquement utilisé pour traiter les éléments annotés. Par exemple, compiler la classe `AnnotationProcessorTest` utilisant

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

donne la sortie suivante

```
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
```

au lieu de compiler normalement. Aucun fichier `.class` n'est créé.

Cela pourrait être évité en spécifiant l'option `-proc:none` pour `javac` . Vous pouvez également renoncer à la compilation habituelle en spécifiant `-proc:only` place.

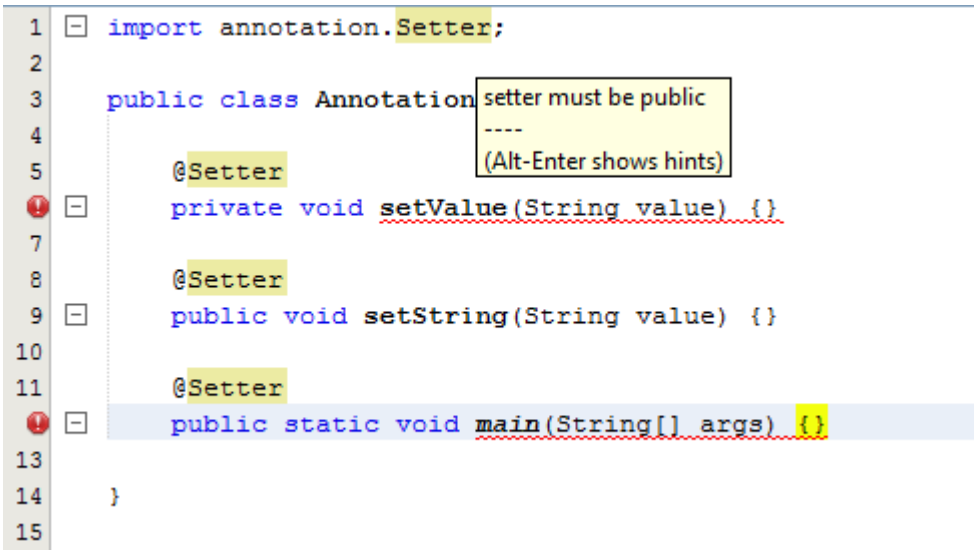
Intégration IDE

Netbeans

Les processeurs d'annotation peuvent être utilisés dans l'éditeur NetBeans. Pour ce faire, le processeur d'annotations doit être spécifié dans les paramètres du projet:

1. allez dans `Project Properties` > `Build` > `Compiling`
2. ajouter des coches pour `Enable Annotation Processing` et `Enable Annotation Processing in Editor`
3. cliquez sur `Add` regard de la liste des processeurs d'annotations
4. Dans la fenêtre qui apparaît, entrez le nom de classe complet du processeur d'annotation et cliquez sur `Ok` .

Résultat



L'idée des annotations

La spécification de langage Java décrit les annotations comme suit:

> Une annotation est un marqueur qui associe des informations à une construction de programme, mais n'a aucun effet au moment de l'exécution.

Les annotations peuvent apparaître avant les types ou les déclarations. Il est possible qu'ils apparaissent dans un endroit où ils pourraient s'appliquer à la fois à un type ou à une déclaration. Ce à quoi s'applique exactement une annotation est régi par la "méta-annotation" `@Target` . Voir "Définition des types d'annotation" pour plus d'informations.

Les annotations sont utilisées à des fins multiples. Les cadres tels que Spring et Spring-MVC utilisent des annotations pour définir où les dépendances doivent être injectées ou où les requêtes doivent être acheminées.

D'autres frameworks utilisent des annotations pour la génération de code. Lombok et JPA sont des exemples principaux, qui utilisent des annotations pour générer du code Java (et SQL).

Ce sujet vise à fournir un aperçu complet de:

- Comment définir vos propres annotations?
- Quelles sont les annotations fournies par le langage Java?
- Comment les annotations sont-elles utilisées dans la pratique?

Annotations pour 'this' et paramètres du récepteur

Lorsque les annotations Java ont été introduites pour la première fois, aucune disposition ne permettait d'annoter la cible d'une méthode d'instance ou le paramètre constructeur caché d'un constructeur de classes internes. Cela a été corrigé dans Java 8 avec l'ajout des déclarations de

paramètres du *récepteur* ; voir JLS 8.4.1 .

> Le paramètre récepteur est un périphérique syntaxique facultatif pour une méthode d'instance ou un constructeur de classe interne. Pour une méthode d'instance, le paramètre récepteur représente l'objet pour lequel la méthode est appelée. Pour un constructeur de classe interne, le paramètre receiver représente l'instance englobante immédiate de l'objet nouvellement construit. De toute façon, le paramètre récepteur existe uniquement pour permettre au type de l'objet représenté d'être désigné dans le code source, de sorte que le type puisse être annoté. Le paramètre du récepteur n'est pas un paramètre formel; plus précisément, il ne s'agit pas d'une déclaration de variable quelconque (§4.12.3), elle n'est jamais liée à une valeur passée en argument dans une expression d'invocation de méthode ou une expression de création d'instance de classe qualifiée, et elle n'a aucun effet sur temps d'exécution

L'exemple suivant illustre la syntaxe des deux types de paramètres du récepteur:

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

Les paramètres du récepteur ont pour seul objectif d'ajouter des annotations. Par exemple, vous pouvez avoir une annotation personnalisée `@IsOpen` dont le but est d'affirmer qu'un objet `Closeable` n'a pas été fermé lorsqu'une méthode est appelée. Par exemple:

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

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

À un `@IsOpen` niveau, l'annotation `@IsOpen` sur `this` `@IsOpen` pourrait simplement servir de documentation. Cependant, nous pourrions potentiellement faire plus. Par exemple:

- Un processeur d'annotations peut insérer un contrôle d'exécution indiquant `this` n'est pas fermé lorsque la `update` à `update` est appelée.
- Un vérificateur de code peut effectuer une analyse de code statique pour trouver des cas où `this` *pourrait être* fermé lorsque la `update` à `update` est appelée.

Ajouter plusieurs valeurs d'annotation

Un paramètre Annotation peut accepter plusieurs valeurs s'il est défini en tant que tableau. Par exemple, l'annotation standard `@SuppressWarnings` est définie comme suit:

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

Le paramètre `value` est un tableau de chaînes. Vous pouvez définir plusieurs valeurs en utilisant une notation similaire aux initialiseurs de tableau:

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

Si vous ne devez définir qu'une seule valeur, les parenthèses peuvent être omises:

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

Lire Annotations en ligne: https://riptutorial.com/fr/java/topic/157/annotations

Chapitre 6: Apache Commons Lang

Examples

Implémenter la méthode equals ()

Pour implémenter facilement la méthode `equals` d'un objet, vous pouvez utiliser la classe `EqualsBuilder` .

Sélection des champs:

```
@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();
}
```

En utilisant la réflexion:

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

le paramètre booléen indique si les égaux doivent vérifier les champs transitoires.

Utiliser la réflexion en évitant certains champs:

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

Implémenter la méthode hashCode ()

Pour implémenter facilement la méthode `hashCode` d'un objet, vous pouvez utiliser la classe `HashCodeBuilder` .

Sélection des champs:

```
@Override
public int hashCode() {

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

    return builder.hashCode();
}
```

En utilisant la réflexion:

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

le paramètre booléen indique s'il doit utiliser des champs transitoires.

Utiliser la réflexion en évitant certains champs:

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

Implémentez la méthode toString ()

Pour implémenter facilement la méthode `toString` d'un objet, vous pouvez utiliser la classe `ToStringBuilder` .

Sélection des champs:

```
@Override
public String toString() {

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

    return builder.toString();
}
```

Exemple de résultat:

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

Donner explicitement des noms aux champs:

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

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

    return builder.toString();
}
```

Exemple de résultat:

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

Vous pouvez changer le style via le paramètre:

```
@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();
}
```

Exemple de résultat:

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

Il y a quelques styles, par exemple JSON, pas de nom de classe, short, etc ...

Par réflexion:

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

Vous pouvez également indiquer le style:

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

Lire Apache Commons Lang en ligne: https://riptutorial.com/fr/java/topic/3338/apache-commons-lang

Chapitre 7: API de réflexion

Introduction

Reflection est couramment utilisé par les programmes qui doivent pouvoir examiner ou modifier le comportement d'exécution des applications exécutées dans la JVM. L'API Java Reflection est utilisée à cette fin, où elle permet d'inspecter les classes, les interfaces, les champs et les méthodes à l'exécution, sans connaître leurs noms au moment de la compilation. Et il permet également d'instancier de nouveaux objets et d'invoquer des méthodes utilisant la réflexion.

Remarques

Performance

Gardez à l'esprit que la réflexion peut diminuer les performances, ne l'utilisez que si votre tâche ne peut être complétée sans réflexion.

A partir du tutoriel Java, l' API Reflection :

> Étant donné que la réflexion implique des types résolus dynamiquement, certaines optimisations de la machine virtuelle Java ne peuvent pas être effectuées. Par conséquent, les opérations de réflexion ont des performances plus lentes que leurs homologues non réfléchissantes et doivent être évitées dans les sections de code appelées fréquemment dans les applications sensibles aux performances.

Examples

introduction

Les bases

L'API de réflexion permet de vérifier la structure de classe du code à l'exécution et d'appeler le code dynamiquement. Ceci est très puissant, mais il est également dangereux car le compilateur n'est pas capable de déterminer statiquement si les appels dynamiques sont valides.

Un exemple simple serait d'obtenir les constructeurs publics et les méthodes d'une classe donnée:

```
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
```

Avec cette information, il est possible d'injecter l'objet et d'appeler différentes méthodes dynamiquement.

Réflexion et types génériques

Les informations de type génériques sont disponibles pour:

- paramètres de méthode, en utilisant `getGenericParameterTypes()` .
- types de retour de méthode, en utilisant `getGenericReturnType()` .
- champs **publics** , en utilisant `getGenericType` .

L'exemple suivant montre comment extraire les informations de type générique dans les trois cas:

```
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;
    }
```

```
}
```

Cela se traduit par la sortie suivante:

```
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
```

Invoquer une méthode

En utilisant la réflexion, une méthode d'un objet peut être appelée pendant l'exécution.

L'exemple montre comment appeler les méthodes d'un objet `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!"
```

Champs Obtenir et définir

À l'aide de l'API Reflection, il est possible de modifier ou d'obtenir la valeur d'un champ à l'exécution. Par exemple, vous pouvez l'utiliser dans une API pour extraire différents champs basés sur un facteur, comme le système d'exploitation. Vous pouvez également supprimer des modificateurs comme `final` pour permettre la modification des champs qui sont finaux.

Pour ce faire, vous devrez utiliser la méthode Class # getField () de la manière indiquée ci-dessous:

```
// 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);
```

Obtenir des champs est beaucoup plus facile. Nous pouvons utiliser Field # get () et ses variantes pour obtenir sa valeur:

```
// 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);
```

Notez ceci:

Lorsque vous utilisez Class # getDeclaredField , utilisez-le pour obtenir un champ dans la classe elle-même:

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

class Hacked {
    public String someState;
}
```

Ici, `HackMe#iAmDeclared` est déclaré champ. Cependant, `HackMe#someState` n'est pas un champ déclaré car il est hérité de sa super-classe, Hacked.

Obtenir l'objet constructeur

Vous pouvez obtenir la classe `Constructor` partir de l'objet `Class` comme ceci:

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

Où la variable `constructors` aura une instance de `Constructor` pour chaque constructeur public déclaré dans la classe.

Si vous connaissez les types de paramètres précis du constructeur auquel vous souhaitez accéder, vous pouvez filtrer le constructeur spécifique. L'exemple suivant retourne le constructeur public de la classe donnée qui prend un `Integer` comme paramètre:

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

Si aucun constructeur ne correspond aux arguments du constructeur, une `NoSuchMethodException` est lancée.

Nouvelle instance utilisant un objet constructeur

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

Obtenir les constantes d'une énumération

Donner cette énumération comme exemple:

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

En Java, une classe enum est comme toute autre classe mais possède des constantes définies pour les valeurs enum. De plus, il contient un champ qui est un tableau contenant toutes les valeurs et deux méthodes statiques avec des `values()` name `values()` et `valueOf(String)` .

Nous pouvons le voir si nous utilisons Reflection pour imprimer tous les champs de cette classe

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

le résultat sera:

NORD
EST
SUD
OUEST
degré
ENUM $ VALUES

Nous pourrions donc examiner les classes enum avec Reflection comme toute autre classe. Mais l'API Reflection propose trois méthodes spécifiques à l'énumération.

chèque enum

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

Renvoie true pour les classes qui représentent un type enum.

récupérer des valeurs

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

Retourne un tableau de toutes les valeurs enum comme Compass.values () mais sans avoir besoin d'une instance.

enum contrôle constant

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

Répertorie tous les champs de classe qui sont des valeurs enum.

Obtenir la classe en fonction de son nom (entièrement qualifié)

Étant donné une `String` contenant le nom d'une classe, son objet `Class` est accessible à l'aide de `Class.forName` :

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

Java SE 1.2

Il peut être spécifié si la classe doit être initialisée (deuxième paramètre de `forName`) et quel `ClassLoader` doit être utilisé (troisième paramètre):

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

Appeler les constructeurs surchargés en utilisant la réflexion

Exemple: appel de différents constructeurs en passant des paramètres pertinents

```
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"});

    }
}
```

sortie:

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

Explication:

1. Créer une instance de classe en utilisant `Class.forName` : elle appelle le constructeur par défaut
2. Appelez `getDeclaredConstructor` de la classe en transmettant le type de paramètres en tant que `Class array`
3. Après avoir obtenu le constructeur, créez `newInstance` en transmettant la valeur du paramètre en tant que `Object array`

Mauvaise utilisation de l'API Reflection pour modifier les variables privées et

finales

La réflexion est utile lorsqu'elle est correctement utilisée à bon escient. En utilisant la réflexion, vous pouvez accéder aux variables privées et réinitialiser les variables finales.

Voici l'extrait de code, qui n'est **pas** recommandé.

```
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++;
    }
}
```

Sortie:

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

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

Explication:

Dans un scénario normal, `private` variables `private` ne sont pas accessibles en dehors de la classe déclarée (sans les méthodes getter et setter). `final` variables `final` ne peuvent pas être réaffectées après l'initialisation.

`Reflection` brise les deux obstacles peut être utilisée pour modifier les variables privées et finales comme expliqué ci-dessus.

`field.setAccessible(true)` est la clé pour obtenir la fonctionnalité souhaitée.

Appelez le constructeur de la classe imbriquée

Si vous souhaitez créer une instance d'une classe imbriquée interne, vous devez fournir un objet de classe de la classe englobante en tant que paramètre supplémentaire avec 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"});
    }
}
```

Si la classe imbriquée est statique, vous n'aurez pas besoin de cette instance englobante.

Proxies dynamiques

Les proxys dynamiques n'ont pas grand chose à voir avec Reflection, mais ils font partie de l'API. C'est essentiellement un moyen de créer une implémentation dynamique d'une interface. Cela peut être utile lors de la création de services de maquette.
Un proxy dynamique est une instance d'une interface créée avec un gestionnaire d'invocation appelé qui intercepte tous les appels de méthode et permet la gestion manuelle de leur appel.

```
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();
    }
}
```

Le résultat de ce code est le suivant:

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

Mal Java hacks avec réflexion

L'API de réflexion peut être utilisée pour modifier les valeurs des champs privé et final, même dans la bibliothèque par défaut du JDK. Cela pourrait être utilisé pour manipuler le comportement de certaines classes bien connues comme nous le verrons.

Ce qui n'est pas possible

Laisser commencer par la seule limitation signifie que le seul champ que nous ne pouvons pas modifier avec Reflection. C'est le Java `SecurityManager` . Il est déclaré dans java.lang.System comme

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

Mais il ne sera pas répertorié dans la classe System si nous exécutons ce code

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

C'est à cause de `fieldFilterMap` dans `sun.reflect.Reflection` qui contient la carte elle-même et le champ de sécurité dans `System.class` et les protège contre tout accès avec Reflection. Nous n'avons donc pas pu désactiver le `SecurityManager` .

Cordes folles

Chaque chaîne Java est représentée par la machine virtuelle Java en tant qu'instance de la classe `String` . Toutefois, dans certaines situations, la machine virtuelle Java enregistre l'espace mémoire en utilisant la même instance pour les chaînes qui le sont. Cela se produit pour les littéraux de chaîne, ainsi que pour les chaînes qui ont été "internées" en appelant `String.intern()` . Donc, si vous avez `"hello"` dans votre code plusieurs fois, c'est toujours la même instance d'objet.

Les chaînes sont supposées être immuables, mais il est possible d'utiliser une réflexion "mauvaise" pour les changer. L'exemple ci-dessous montre comment nous pouvons modifier les caractères d'une chaîne en remplaçant son champ de `value` .

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

Donc, ce code imprimera "vous pue!"

1 = 42

La même idée pourrait être utilisée avec 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));
    }
}
```

Tout est vrai

Et selon ce post stackoverflow, nous pouvons utiliser la réflexion pour faire quelque chose de vraiment mal.

```
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);
    }
}
```

Notez que ce que nous faisons ici va provoquer un comportement inexplicable de la JVM. C'est très dangereux.

Lire API de réflexion en ligne: https://riptutorial.com/fr/java/topic/629/api-de-reflexion

Chapitre 8: API Stack-Walking

Introduction

Avant Java 9, l'accès aux cadres de la pile de threads était limité à une classe interne `sun.reflect.Reflection` . Plus précisément, la méthode `sun.reflect.Reflection::getCallerClass` . Certaines bibliothèques s'appuient sur cette méthode qui est déconseillée.

Une API standard de remplacement est maintenant prévue dans 9 JDK via `java.lang.StackWalker` classe, et est conçu pour être efficace en permettant un accès paresseux aux cadres de la pile. Certaines applications peuvent utiliser cette API pour parcourir la pile d'exécution et filtrer les classes.

Examples

Imprimer tous les cadres de pile du thread en cours

Ce qui suit imprime tous les cadres de pile du thread en cours:

```
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 }
```

Sortie:

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

Imprimer la classe d'appelant en cours

Ce qui suit imprime la classe d'appelante actuelle. Notez que dans ce cas, le `StackWalker` doit être créé avec l'option `RETAIN_CLASS_REFERENCE` , afin que `Class` instances de `Class` soient conservées dans les objets `StackFrame` . Sinon, une exception se produirait.

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

Sortie:

```
class test.FooHelper
```

Montrer la réflexion et autres cadres cachés

Quelques autres options permettent aux traces de pile d'inclure des cadres d'implémentation et / ou de réflexion. Cela peut être utile à des fins de débogage. Par exemple, nous pouvons ajouter l'option `SHOW_REFLECT_FRAMES` à l'instance `StackWalker` lors de la création, afin que les cadres des méthodes de réflexion soient également imprimés:

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

Sortie:

```
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
```

Notez que les numéros de ligne de certaines méthodes de réflexion peuvent ne pas être disponibles. `StackFrame.getLineNumber()` peut donc renvoyer des valeurs négatives.

Lire API Stack-Walking en ligne: https://riptutorial.com/fr/java/topic/9868/api-stack-walking

Chapitre 9: AppDynamics et TIBCO BusinessWorks Instrumentation pour une intégration facile

Introduction

Comme AppDynamics vise à fournir un moyen de mesurer les performances des applications, la vitesse de développement, la livraison (déploiement) des applications est un facteur essentiel pour faire des efforts DevOps un véritable succès. La surveillance d'une application TIBCO BW avec AppD est généralement simple et rapide, mais lors du déploiement de grands ensembles d'applications, une instrumentation rapide est essentielle. Ce guide montre comment instrumenter toutes vos applications BW en une seule étape sans modifier chaque application avant le déploiement.

Examples

Exemple d'instrumentation de toutes les applications BW en une seule étape pour Appdynamics

1. Localisez et ouvrez votre fichier TIBCO BW bwengine.tra sous TIBCO_HOME / bw / 5.12 / bin / bwengine.tra (environnement Linux)

2. Recherchez la ligne qui indique:

*** Variables communes. Modifiez-les uniquement. ***

3. Ajoutez la ligne suivante juste après cette section tibco.deployment =% tibco.deployment%

4. Accédez à la fin du fichier et ajoutez (remplacez? Avec vos propres valeurs si nécessaire ou supprimez l'indicateur qui ne s'applique pas): 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. Enregistrez le fichier et redéployez-le. Toutes vos applications doivent maintenant être automatiquement instrumentées au moment du déploiement.

Lire AppDynamics et TIBCO BusinessWorks Instrumentation pour une intégration facile en ligne: https://riptutorial.com/fr/java/topic/10602/appdynamics-et-tibco-businessworks-instrumentation-pour-une-integration-facile

Chapitre 10: Autoboxing

Introduction

La compilation automatique est la conversion automatique effectuée par le compilateur Java entre les types primitifs et leurs classes d'encapsulation d'objet correspondantes. Exemple, conversion int -> Entier, double -> Double ... Si la conversion est inversée, cela s'appelle unboxing. En général, cela est utilisé dans les collections qui ne peuvent contenir que des objets, où les types primitifs de boxe sont nécessaires avant de les définir dans la collection.

Remarques

La programmation automatique peut avoir des problèmes de performance lorsqu'elle est fréquemment utilisée dans votre code.

- http://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html
- Un auto-unboxing entier et une auto-boxing posent des problèmes de performance?

Examples

Utilisation de int et Integer indifféremment

Lorsque vous utilisez des types génériques avec des classes d'utilitaires, vous pouvez souvent constater que les types de nombres ne sont pas très utiles lorsqu'ils sont spécifiés comme types d'objet, car ils ne sont pas égaux à leurs homologues primitifs.

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

Java SE 7

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

Heureusement, les expressions évaluées par `int` peuvent être utilisées à la place d'un `Integer` lorsque cela est nécessaire.

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

Le `ints.add(i);` déclaration équivaut à:

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

Et conserve les propriétés d' `Integer#valueOf` telles que la mise en cache des mêmes objets `Integer` par la JVM lorsqu'elle se trouve dans la plage de mise en cache des nombres.

Cela vaut également pour:

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

Il faut toutefois être prudent dans les situations ambiguës. Considérez le code suivant:

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

L'interface `java.util.List` contient à la fois une méthode `remove(int index)` (méthode d'interface `List`) et une `remove(Object o)` (méthode héritée de `java.util.Collection`). Dans ce cas, aucune boxe n'a lieu et on `remove(int index)` .

Un autre exemple de comportement de code Java étrange causé par des entiers à base de boîtes de vitesses automatiques avec des valeurs comprises entre `-128` et `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
```

Cela se produit car l'opérateur `>=` appelle implicitement `intValue()` qui renvoie `int` alors que `==` compare les **références** , pas les valeurs `int` .

Par défaut, Java met en cache les valeurs dans la plage `[-128, 127]` , de sorte que l'opérateur `==` fonctionne car les `Integers` dans cette plage font référence aux mêmes objets si leurs valeurs sont identiques. La valeur maximale de la plage pouvant être mise en cache peut être définie avec l'option `-XX:AutoBoxCacheMax` JVM. Donc, si vous exécutez le programme avec `-XX:AutoBoxCacheMax=1000` , le code suivant s'imprimera `true` :

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

Utilisation de booléen dans l'instruction if

En raison de la désactivation automatique, on peut utiliser un `Boolean` dans une instruction `if` :

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

Cela fonctionne pour `while` , `do while` et l'état dans le `for` des déclarations aussi bien.

Notez que si le `Boolean` est `null` , une `NullPointerException` sera lancée dans la conversion.

Unboxing automatique peut conduire à NullPointerException

Ce code compile:

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

Mais il se bloquera à l'exécution avec une `java.lang.NullPointerException` sur la deuxième ligne.

Le problème est qu'un `int` primitif ne peut pas avoir de valeur `null` .

C'est un exemple minimaliste, mais dans la pratique, il se manifeste souvent sous des formes plus sophistiquées. Le `NullPointerException` n'est pas très intuitif et aide souvent peu à localiser de tels bogues.

En prenant soin de la mise en boîte automatique et de la désencapsulation automatique avec soin, assurez-vous que les valeurs non débrayées n'auront pas de valeurs `null` à l'exécution.

Mémoire et surcharge de calcul de l'autoboxing

L’autoboxing peut avoir une surcharge de mémoire importante. Par exemple:

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

consommera généralement beaucoup de mémoire (environ 60 kb pour 6 k de données réelles).

De plus, les entiers encadrés nécessitent généralement des allers-retours supplémentaires dans la mémoire, ce qui rend les caches de processeur moins efficaces. Dans l'exemple ci-dessus, la mémoire accédée est répartie sur cinq emplacements différents qui peuvent se trouver dans des régions entièrement différentes de la mémoire: 1. l'objet `HashMap` , 2. l'objet `Entry[] table` , 3. l'objet `Entry` , 4. le entrys `key` object (boxing la clé primitive), 5. l'objet entry `value` (encadrant la `value` primitive).

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

La lecture en `boxed` nécessite deux accès mémoire, n'accédant qu'à une `primitive` .

Lorsque vous obtenez des données de cette carte, le code apparemment innocent

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

est équivalent à:

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

En général, le code ci-dessus provoque la *création et la récupération* de `Integer` un objet `Integer` pour chaque opération `Map#get(Integer)` . (Voir la note ci-dessous pour plus de détails.)

Pour réduire ce surcoût, plusieurs bibliothèques offrent des collections optimisées pour les types primitifs *ne* nécessitant *pas* de boxe. En plus d'éviter la surcharge de boxe, ces collectes nécessiteront environ 4 fois moins de mémoire par entrée. Bien que Java Hotspot *puisse* optimiser l'autoboxing en travaillant avec des objets sur la pile au lieu du tas, il n'est pas possible d'optimiser la surcharge de mémoire et l'indirection de mémoire qui en résulte.

Les flux Java 8 ont également des interfaces optimisées pour les types de données primitifs, tels que `IntStream` qui ne nécessitent pas de boxe.

Remarque: un environnement d'exécution Java classique conserve un cache simple d' `Integer` et des autres objets wrapper primitifs utilisés par les méthodes factory `valueOf` et par la création automatique de boîtes aux lettres. Pour `Integer` , la plage par défaut de ce cache est comprise entre -128 et +127. Certaines machines virtuelles Java offrent une option de ligne de commande JVM pour modifier la taille / plage du cache.

Différents cas où Integer et int peuvent être utilisés indifféremment

Cas 1: En utilisant à la place des arguments de méthode.

Si une méthode nécessite un objet de classe wrapper en tant qu'argument.Alors de manière interchangeable, l'argument peut être passé à une variable du type primitif respectif et vice versa.

Exemple:

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

Cas 2: Lors du passage des valeurs de retour:

Lorsqu'une méthode retourne une variable de type primitive, un objet de la classe wrapper correspondante peut être transmis comme valeur de retour de manière interchangeable et inversement.

Exemple:

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

Cas 3: lors des opérations.

Chaque fois que l'on effectue des opérations sur des nombres, la variable de type primitif et l'objet de la classe d'encapsulation respective peuvent être utilisés indifféremment.

```
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
```

Piège : N'oubliez pas d'initialiser ou d'attribuer une valeur à un objet de la classe wrapper.

Lorsque vous utilisez un objet de classe wrapper et une variable primitive de manière interchangeable, n'oubliez jamais ou omettez d'initialiser ou d'attribuer une valeur à l'objet de classe wrapper. Sinon, cela peut entraîner une exception de pointeur nul lors de l'exécution.

Exemple:

```
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
    }
```

Dans l'exemple ci-dessus, la valeur de l'objet est non affectée et non initialisée. Par conséquent, à l'exécution, le programme s'exécutera avec une exception de pointeur nul. Ainsi, la valeur de l'objet ne devrait jamais être non initialisée et non attribuée.

Lire Autoboxing en ligne: https://riptutorial.com/fr/java/topic/138/autoboxing

Chapitre 11: BigDecimal

Introduction

La classe BigDecimal fournit des opérations pour l'arithmétique (ajouter, soustraire, multiplier, diviser), la manipulation d'échelle, l'arrondissement, la comparaison, le hachage et la conversion de format. Le BigDecimal représente des nombres décimaux signés, immuables et de précision arbitraire. Cette classe doit être utilisée pour effectuer un calcul de haute précision.

Examples

Les objets BigDecimal sont immuables

Si vous voulez calculer avec BigDecimal, vous devez utiliser la valeur renvoyée car les objets BigDecimal sont immuables:

```
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
```

Comparer les BigDecimals

La méthode `compareTo` doit être utilisée pour comparer `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
```

Généralement, vous **ne** devriez **pas** utiliser la méthode des `equals` , car elle considère que deux `BigDecimals` égaux seulement s'ils sont égaux en valeur et aussi en **échelle** :

```
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
```

Opérations mathématiques avec BigDecimal

Cet exemple montre comment effectuer des opérations mathématiques de base en utilisant 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);
```

Résultat: 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);
```

Résultat: -2

3. Multiplication

En multipliant deux `BigDecimal` le résultat aura une échelle égale à la somme des échelles des opérandes.

```
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);
```

Résultat: 36.89931

Pour changer l'échelle du résultat, utilisez la méthode de multiplication surchargée qui permet de passer `MathContext` - un objet décrivant les règles pour les opérateurs, en particulier la précision et le mode d'arrondi du résultat. Pour plus d'informations sur les modes d'arrondi disponibles, reportez-vous à la documentation 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);
```

Résultat: 36.90

4.Division

La division est un peu plus compliquée que les autres opérations arithmétiques, par exemple prenons l'exemple ci-dessous:

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

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

Nous nous attendrions à ce que cela donne quelque chose de similaire à: 0.7142857142857143, mais nous aurions:

> Résultat: **java.lang.ArithmeticException: expansion décimale non terminante; aucun résultat décimal représentable exact.**

Cela fonctionnerait parfaitement lorsque le résultat serait un nombre décimal final si je voulais diviser 5 par 2, mais pour les nombres qui, en se divisant, donneraient un résultat non final, nous aurions une `ArithmeticException` . Dans le scénario réel, on ne peut pas prédire les valeurs qui seraient rencontrées pendant la division. Nous devons donc spécifier l' **échelle** et le **mode d'arrondi** pour la division BigDecimal. Pour plus d'informations sur le mode Scale et Rounding, reportez-vous à la documentation Oracle .

Par exemple, je pourrais faire:

```
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);
```

> Résultat: 0.7142857143

5. Reste ou module

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

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

> Résultat: 5

6. puissance

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

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

Résultat: 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);
```

Résultat: 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);
```

Résultat: 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);
```

Résultat: 52.3449843776

10.Move Point To Right

```
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);
```

Résultat: 5234498.43776

Il y a beaucoup plus d'options et de combinaisons de paramètres pour les exemples mentionnés ci-dessus (par exemple, il existe 6 variantes de la méthode de division), cet ensemble est une liste non exhaustive et couvre quelques exemples de base.

Utiliser BigDecimal au lieu de float

En raison de la représentation du type float dans la mémoire de l'ordinateur, les résultats des opérations utilisant ce type peuvent être inexacts - certaines valeurs sont stockées sous forme d'approximations. Les calculs monétaires en sont de bons exemples. Si une haute précision est nécessaire, d'autres types doivent être utilisés. Par exemple, Java 7 fournit 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));
}
```

Le résultat de ce programme est:

```
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
```

Pour un solde initial de 10 000,00, après 1 000 opérations pour 1,99, le solde devrait être de 8010,00. L'utilisation du type float nous donne une réponse autour de 8009.77, ce qui est très imprécis dans le cas des calculs monétaires. L'utilisation de BigDecimal nous donne le bon résultat.

BigDecimal.valueOf ()

La classe BigDecimal contient un cache interne des nombres fréquemment utilisés, par exemple 0 à 10. Les méthodes BigDecimal.valueOf () sont préférables aux constructeurs avec des paramètres de type similaires, c'est-à-dire que l'exemple ci-dessous est préférable à 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
```

Initialisation de BigDecimals avec la valeur zéro, un ou dix

BigDecimal fournit des propriétés statiques pour les nombres zéro, un et dix. Il est recommandé de les utiliser plutôt que d'utiliser les nombres réels:

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

En utilisant les propriétés statiques, vous évitez une instanciation inutile, vous avez également un littéral dans votre code au lieu d'un «nombre magique».

```
//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;
```

Lire BigDecimal en ligne: https://riptutorial.com/fr/java/topic/1667/bigdecimal

Chapitre 12: BigInteger

Introduction

La classe `BigInteger` est utilisée pour les opérations mathématiques impliquant de grands nombres entiers avec des grandeurs trop grandes pour les types de données primitifs. Par exemple, le facteur 100 correspond à 158 chiffres, ce qui est beaucoup plus grand qu'un `long` peut représenter. `BigInteger` fournit des analogues à tous les opérateurs entiers primitifs de Java et à toutes les méthodes pertinentes de `java.lang.Math` ainsi que de quelques autres opérations.

Syntaxe

- BigInteger nom_variable = nouveau BigInteger ("12345678901234567890"); // un entier décimal sous forme de chaîne
- BigInteger nom_variable = new BigInteger ("10101011010101001010100110001100111010110001111100001010110100 10", 2) // un entier binaire sous forme de chaîne
- BigInteger nom_variable = new BigInteger ("ab54a98ceb1f0800", 16) // un entier hexadécimal sous forme de chaîne
- BigInteger variable_name = new BigInteger (64, nouveau Random ()); // un générateur de nombres pseudo-aléatoires fournissant 64 bits pour construire un entier
- BigInteger nom_variable = nouveau BigInteger (nouvel octet [] {0, -85, 84, -87, -116, -21, 31, 10, -46}); // représentation du complément à deux signé d'un entier (big endian)
- BigInteger nom_variable = nouveau BigInteger (1, nouvel octet [] {- 85, 84, -87, -116, -21, 31, 10, -46}); // la représentation du complément à deux unsigned d'un entier positif (big endian)

Remarques

`BigInteger` est immuable. Par conséquent, vous ne pouvez pas changer son état. Par exemple, les éléments suivants ne fonctionneront pas, car la `sum` ne sera pas mise à jour en raison de l'immuabilité.

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

Attribuez le résultat à la variable `sum` pour la faire fonctionner.

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

Java SE 8

La documentation officielle de `BigInteger` indique que les implémentations `BigInteger` doivent prendre en charge tous les entiers compris entre -2 ^{2147483647} et 2 ^{2147483647} (exclusifs). Cela signifie que `BigInteger` peut avoir plus de 2 *milliards de* bits!

Examples

Initialisation

La classe `java.math.BigInteger` fournit des opérations analogues à tous les opérateurs entiers primitifs de Java et à toutes les méthodes pertinentes de `java.lang.Math` . Comme le package `java.math` n'est pas automatiquement disponible, vous devrez peut-être importer `java.math.BigInteger` avant de pouvoir utiliser le nom de classe simple.

Pour convertir `long` valeurs `long` ou `int` en `BigInteger` utilisez:

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

ou, pour les entiers:

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

ce qui *élargira* le `intValue` entier `intValue` à long, en utilisant l'extension de bit de signe pour les valeurs négatives, afin que les valeurs négatives restent négatives.

Pour convertir une `String` numérique en `BigInteger` utilisez:

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

Le constructeur suivant est utilisé pour traduire la représentation sous forme de chaîne d'un `BigInteger` dans la base spécifiée dans un `BigInteger` .

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

Java prend également en charge la conversion directe des octets en une instance de `BigInteger` . Actuellement, seul le big endian signé et signé peut être utilisé:

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

Cela va générer une instance `BigInteger` avec la valeur -128 car le premier bit est interprété comme le bit de signe.

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

Cela générera une instance `BigInteger` avec la valeur 128 car les octets sont interprétés comme un nombre non signé et le signe est explicitement défini sur 1, un nombre positif.

Il existe des constantes prédéfinies pour les valeurs communes:

- `BigInteger.ZERO` - valeur de "0".
- `BigInteger.ONE` - valeur de "1".
- `BigInteger.TEN` - valeur de "10".

Il y a aussi `BigInteger.TWO` (valeur de "2"), mais vous ne pouvez pas l'utiliser dans votre code car il est `private` .

Comparer les BigIntegers

Vous pouvez comparer `BigIntegers` même manière que vous comparez `String` ou d'autres objets en Java.

Par exemple:

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

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

Sortie:

```
Not Equal
```

Remarque:

En général, n'utilisez **pas** l'opérateur `==` pour comparer BigIntegers

- `==` opérateur: compare les références; c'est-à-dire si deux valeurs se réfèrent au même objet
- méthode `equals()` : compare le contenu de deux BigIntegers.

Par exemple, BigIntegers **ne** doit **pas** être comparé de la manière suivante:

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

Cela peut entraîner un comportement inattendu, car l'opérateur `==` ne vérifie que l'égalité de référence. Si les deux BigIntegers contiennent le même contenu, mais ne font pas référence au même objet, **cela échouera.** Au lieu de cela, comparez BigIntegers en utilisant les méthodes `equals` , comme expliqué ci-dessus.

Vous pouvez également comparer votre `BigInteger` à des valeurs constantes telles que 0,1,10.

par exemple:

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

Vous pouvez également comparer deux BigIntegers en utilisant la méthode `compareTo()` , comme suit: `compareTo()` renvoie 3 valeurs.

- **0:** quand les deux sont **égaux** .
- **1:** Lorsque le premier est **supérieur à la** seconde (celui entre parenthèses).
- **-1:** Quand le premier est **inférieur à la** seconde.

```
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.
}
```

Exemples d'opérations mathématiques BigInteger

BigInteger se trouve dans un objet immuable, vous devez donc affecter les résultats de toute opération mathématique à une nouvelle instance de BigInteger.

Ajout: *10 + 10 = 20*

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

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

sortie: 20

Substraction: *10 - 9 = 1*

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

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

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

sortie: 1

Division: *10/5 = 2*

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

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

sortie: 2

Division: *17/4 = 4*

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

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

sortie: 4

Multiplication: *10 * 5 = 50*

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

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

sortie: 50

Puissance: *10 ^ 3 = 1000*

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

sortie: 1000

Reste: *10% 6 = 4*

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

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

sortie: 4

GCD: Greatest Common Divisor (GCD) pour `12` et `18` est `6` .

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

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

Sortie: 6

Maximum de deux BigIntegers:

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

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

Sortie: 11

Minimum de deux BigIntegers:

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

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

Sortie: 10

Opérations de logique binaire sur BigInteger

BigInteger prend également en charge les opérations logiques binaires disponibles pour les types `Number` . Comme pour toutes les opérations, elles sont implémentées en appelant une méthode.

Binaire ou:

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

val1.or(val2);
```

Sortie: 11 (ce qui équivaut à `10 | 9`)

Binaire Et:

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

val1.and(val2);
```

Sortie: 8 (ce qui équivaut à `10 & 9`)

Binaire Xor:

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

val1.xor(val2);
```

Sortie: 3 (ce qui équivaut à `10 ^ 9`)

RightShift:

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

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

Sortie: 5 (équivalent à `10 >> 1`)

Décalage à gauche:

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

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

Sortie: 20 (équivalent à `10 << 1`)

Inversion binaire (non):

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

val1.not();
```

Sortie: 5

*NAND (And-Not): **

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

val1.andNot(val2);
```

Sortie: 7

Générer des BigIntegers aléatoires

La classe `BigInteger` possède un constructeur dédié à la génération aléatoire de `BigIntegers` , à partir d'une instance de `java.util.Random` et d'un `int` qui spécifie le nombre de bits du `BigInteger` . Son utilisation est assez simple - lorsque vous appelez le constructeur `BigInteger(int, Random)` comme ceci:

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

alors vous vous retrouverez avec un `BigInteger` dont la valeur est comprise entre 0 (inclus) et 2 `bitCount` (exclusif).

Cela signifie également que le `new BigInteger(2147483647, sourceOfRandomness)` peut renvoyer tous les `BigInteger` positifs `BigInteger` suffisamment de temps.

Quelle sera la `sourceOfRandomness` à vous de `sourceOfRandomness` ? Par exemple, un `new Random()` est suffisant dans la plupart des cas:

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

Si vous êtes prêt à abandonner la vitesse pour obtenir des nombres aléatoires de meilleure qualité, vous pouvez utiliser un `new SecureRandom ()` place:

```
import java.security.SecureRandom;

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

Vous pouvez même implémenter un algorithme à la volée avec une classe anonyme! Notez que le **déploiement de votre propre algorithme RNG vous *amènera* à un caractère aléatoire de qualité médiocre** . Veillez donc à toujours utiliser un algorithme qui s'avère être décent, à moins que vous ne souhaitiez que le ou les `BigInteger` soient prévisibles.

```
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;
    }
});
```

Lire BigInteger en ligne: https://riptutorial.com/fr/java/topic/1514/biginteger

Chapitre 13: Bit Manipulation

Remarques

- Contrairement au langage C / C ++, Java est complètement indépendant du matériel informatique sous-jacent. Vous n'obtenez pas de comportement endian grand ou petit par défaut; vous devez spécifier explicitement le comportement que vous voulez.
- Le type d' `byte` est signé, avec une plage comprise entre -128 et +127. Pour convertir une valeur d'octet en son équivalent non signé, masquez-le avec 0xFF comme ceci: `(b & 0xFF)` .

Examples

Emballage / Déballage des valeurs en tant que fragments de bits

Il est courant que les performances de la mémoire compressent plusieurs valeurs en une seule valeur primitive. Cela peut être utile pour transmettre diverses informations dans une seule variable.

Par exemple, on peut emballer 3 octets - tels que le code couleur en RVB - dans un seul int.

Emballer les valeurs

```
// 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;
```

Déballer les valeurs

```
// 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)
};
```

Vérification, configuration, effacement et basculement des bits individuels. Utiliser long comme masque de bits

En supposant que nous voulions modifier le bit `n` d'une primitive entière, `i` (octet, court, char, int ou 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'
```

Utiliser long / int / short / byte comme masque de bits:

```
public class BitMaskExample {
    private static final long FIRST_BIT = 1L << 0;
    private static final long SECOND_BIT = 1L << 1;
    private static final long THIRD_BIT = 1L << 2;
    private static final long FOURTH_BIT = 1L << 3;
    private static final long FIFTH_BIT = 1L << 4;
    private static final long BIT_55 = 1L << 54;

    public static void main(String[] args) {
        checkBitMask(FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55);
    }

    private static void checkBitMask(long bitmask) {
        System.out.println("FIRST_BIT: " + ((bitmask & FIRST_BIT) != 0));
        System.out.println("SECOND_BIT: " + ((bitmask & SECOND_BIT) != 0));
        System.out.println("THIRD_BIT: " + ((bitmask & THIRD_BIT) != 0));
        System.out.println("FOURTh_BIT: " + ((bitmask & FOURTH_BIT) != 0));
        System.out.println("FIFTH_BIT: " + ((bitmask & FIFTH_BIT) != 0));
        System.out.println("BIT_55: " + ((bitmask & BIT_55) != 0));
    }
}
```

Des tirages

```
FIRST_BIT: true
SECOND_BIT: false
THIRD_BIT: true
FOURTh_BIT: false
FIFTH_BIT: true
BIT_55: true
```

qui correspond à ce masque que nous avons passé en tant `checkBitMask` paramètre `FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55` : `FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55` .

Exprimer le pouvoir de 2

Pour exprimer la puissance de 2 (2 ^ n) d'entiers, on peut utiliser une opération bitshift qui permet de spécifier explicitement le `n` .

La syntaxe est fondamentalement:

```
int pow2 = 1<<n;
```

Exemples:

```
int twoExp4 = 1<<4; //2^4
int twoExp5 = 1<<5; //2^5
int twoExp6 = 1<<6; //2^6
...
int twoExp31 = 1<<31; //2^31
```

Ceci est particulièrement utile lors de la définition de valeurs constantes qui doivent indiquer qu'une puissance de 2 est utilisée, au lieu d'utiliser des valeurs hexadécimales ou décimales.

```
int twoExp4 = 0x10; //hexadecimal
int twoExp5 = 0x20; //hexadecimal
int twoExp6 = 64; //decimal
...
int twoExp31 = -2147483648; //is that a power of 2?
```

Une méthode simple pour calculer la puissance int de 2 serait

```
int pow2(int exp){
    return 1<<exp;
}
```

Vérifier si un nombre est une puissance de 2

Si un entier `x` est une puissance de 2, un seul bit est défini, alors que `x-1` a tous les bits définis après cela. Par exemple: `4` est `100` et `3` est `011` comme nombre binaire, ce qui satisfait à la condition susmentionnée. Zéro n'est pas une puissance de 2 et doit être vérifié explicitement.

```
boolean isPowerOfTwo(int x)
{
    return (x != 0) && ((x & (x - 1)) == 0);
}
```

Utilisation pour gauche et droite

Supposons que nous ayons trois types d'autorisations, **READ** , **WRITE** et **EXECUTE** . Chaque autorisation peut varier de 0 à 7. (Supposons un système à 4 bits)

RESOURCE = READ WRITE EXECUTE (numéro de 12 bits)

RESSOURCE = 0100 0110 0101 = 4 6 5 (nombre de 12 bits)

Comment pouvons-nous obtenir les autorisations (nombre 12 bits) définies ci-dessus (nombre 12

bits)?

> 0100 0110 0101
>
> 0000 0000 0111 (&)
>
> 0000 0000 0101 = 5

Donc, voici comment nous pouvons obtenir les autorisations **EXECUTE** de la **ressource** . Maintenant, que faire si nous voulons obtenir les autorisations **READ** de la **ressource** ?

> 0100 0110 0101
>
> 0111 0000 0000 (&)
>
> 0100 0000 0000 = 1024

Droite? Vous supposez probablement cela? Mais, les autorisations sont obtenues dans 1024. Nous voulons obtenir uniquement des autorisations READ pour la ressource. Ne vous inquiétez pas, c'est pourquoi nous avons eu les opérateurs de quarts. Si nous voyons, les autorisations READ sont 8 bits derrière le résultat réel, donc si vous appliquez un opérateur de décalage, ce qui apportera des autorisations READ à droite du résultat? Et si on fait:

> 0100 0000 0000 >> 8 => 0000 0000 0100 (Parce que c'est un nombre positif, donc remplacé par 0, si vous ne vous souciez pas du signe, utilisez simplement l'opérateur de décalage non signé)

Nous avons maintenant les permissions **READ** qui sont 4.

Maintenant, par exemple, on nous donne des autorisations **READ** , **WRITE** , **EXECUTE** pour une **ressource** , que pouvons-nous faire pour obtenir des autorisations pour cette **ressource** ?

Prenons d'abord l'exemple des autorisations binaires. (En supposant toujours un système à 4 bits)

> READ = 0001
>
> WRITE = 0100
>
> EXECUTE = 0110

Si vous pensez que nous allons simplement faire:

`READ | WRITE | EXECUTE` , vous avez un peu raison mais pas exactement. Voyez, que se passera-t-il si nous effectuons READ | ECRIRE | EXÉCUTER

0001 | 0100 | 0110 => 0111

Mais les autorisations sont effectivement représentées (dans notre exemple) sous la forme 0001 0100 0110

Donc, pour ce faire, nous savons que **READ** est placé 8 bits derrière, **WRITE** est placé 4 bits

derrière et **PERMISSIONS** est placé à la dernière place. Le système de numérotation utilisé pour les autorisations **RESOURCE** est en fait 12 bits (dans notre exemple). Il peut (sera) différent dans différents systèmes.

(LIRE << 8) | (ECRIRE << 4) | (EXÉCUTER)

0000 0000 0001 << 8 (READ)

0001 0000 0000 (décalage gauche de 8 bits)

0000 0000 0100 << 4 (WRITE)

0000 0100 0000 (décalage à gauche de 4 bits)

0000 0000 0001 (EXECUTE)

Maintenant, si nous ajoutons les résultats du décalage ci-dessus, ce sera quelque chose comme:

0001 0000 0000 (READ)

0000 0100 0000 (WRITE)

0000 0000 0001 (EXECUTE)

0001 0100 0001 (PERMISSIONS)

Classe java.util.BitSet

Depuis la 1.7, il y a une classe java.util.BitSet qui fournit une interface simple et conviviale de stockage et de manipulation des bits:

```
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` implémente `Clonable` et `Serializable` , et sous le capot, toutes les valeurs de bits sont stockées dans `long[] words` champ `long[] words` , qui se développe automatiquement.

Il prend également en charge les opérations logiques mis ensemble- `and` , `or` , `xor` , `andNot` :

```
bitSet.and(new BitSet(8));
bitSet.or(new BitSet(8));
bitSet.xor(new BitSet(8));
```

```
bitSet.andNot(new BitSet(8));
```

Décalage signé ou non signé

En Java, toutes les primitives numériques sont signées. Par exemple, un int représente toujours des valeurs de [-2 ^ 31 - 1, 2 ^ 31], en gardant le premier bit pour signer la valeur - 1 pour la valeur négative, 0 pour le positif.

Les opérateurs de décalage de base `>>` et `<<` sont des opérateurs signés. Ils conserveront le signe de la valeur.

Mais il est courant que les programmeurs utilisent des nombres pour stocker *des valeurs non signées* . Pour un int, cela signifie qu'il faut déplacer la plage sur [0, 2 ^ 32 - 1], pour avoir deux fois plus de valeur que pour un int signé.

Pour les utilisateurs de puissance, le bit pour le signe n'a aucune signification. C'est pourquoi Java a ajouté `>>>` , un opérateur de gauche, sans tenir compte de ce bit de signe.

```
                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)
```

Pourquoi n'y a-t-il pas `<<<` ?

Cela vient de la définition prévue du passage à droite. Comme il remplit les endroits vides sur la gauche, il n'y a aucune décision à prendre concernant le bit de signe. Par conséquent, 2 opérateurs différents ne sont pas nécessaires.

Voir cette question pour une réponse plus détaillée.

Lire Bit Manipulation en ligne: https://riptutorial.com/fr/java/topic/1177/bit-manipulation

Chapitre 14: BufferedWriter

Syntaxe

- nouveau BufferedWriter (Writer); // Le constructeur par défaut
- BufferedWriter.write (int c); // écrit un seul caractère
- BufferedWriter.write (String str); // écrit une chaîne
- BufferedWriter.newLine (); // écrit un séparateur de ligne
- BufferedWriter.close (); // ferme le BufferedWriter

Remarques

- Si vous essayez d'écrire à partir d'un `BufferedWriter` (en utilisant `BufferedWriter.write()`) après la fermeture de `BufferedWriter` (en utilisant `BufferedWriter.close()`), une `IOException` sera lancée.
- Le constructeur `BufferedWriter(Writer)` ne lance PAS une `IOException` . Cependant, le constructeur `FileWriter(File)` lève une `FileNotFoundException` , qui étend `IOException` . `FileNotFoundException` `IOException` interceptera donc `FileNotFoundException` , il n'y a jamais besoin d'une seconde instruction catch sauf si vous envisagez de faire quelque chose de différent avec `FileNotFoundException` .

Examples

Ecrire une ligne de texte dans File

Ce code écrit la chaîne dans un fichier. Il est important de fermer le graveur, cela se fait donc dans un bloc `finally` .

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

Notez également que `write(String s)` ne place pas de caractère de nouvelle ligne après l'écriture de la chaîne. Pour le mettre, utilisez la méthode `newLine()` .

Java SE 7

Java 7 ajoute le package `java.nio.file` et essaie avec :

```
public void writeLineToFile(String str) throws IOException {
    Path path = Paths.get("file.txt");
    try (BufferedWriter bw = Files.newBufferedWriter(path)) {
        bw.write(str);
    }
}
```

Lire BufferedWriter en ligne: https://riptutorial.com/fr/java/topic/3063/bufferedwriter

Chapitre 15: ByteBuffer

Introduction

La classe `ByteBuffer` été introduite dans java 1.4 pour faciliter le travail sur les données binaires. Il est particulièrement adapté pour être utilisé avec des données de type primitif. Il permet la création, mais aussi la manipulation ultérieure d'un `byte[]` s sur un niveau d'abstraction plus élevé

Syntaxe

- octet [] arr = nouvel octet [1000];
- ByteBuffer buffer = ByteBuffer.wrap (arr);
- ByteBuffer buffer = ByteBuffer.allocate (1024);
- ByteBuffer buffer = ByteBuffer.allocateDirect (1024);
- octet b = buffer.get ();
- octet b = buffer.get (10);
- short s = buffer.getShort (10);
- buffer.put ((octet) 120);
- buffer.putChar ('a');

Examples

Utilisation de base - Création d'un ByteBuffer

Il existe deux manières de créer un `ByteBuffer` , où l'on peut subdiviser à nouveau.

Si vous avez un `byte[]` existant `byte[]` , vous pouvez l' *emballer* dans un `ByteBuffer` pour simplifier le traitement:

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

Ce serait une possibilité pour le code qui gère les interactions de réseau de bas niveau

---

Si vous ne disposez pas d'un `byte[]` existant `byte[]` , vous pouvez créer un `ByteBuffer` sur un tableau spécifiquement alloué au tampon, comme ceci:

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

Si le chemin de code est extrêmement critique et que vous avez besoin d' **un accès direct à la mémoire du système** , le `ByteBuffer` peut même allouer *des* tampons *directs à l'* aide de `#allocateDirect()`

Utilisation de base - Écrire des données dans le tampon

Étant donné un `ByteBuffer` exemple , on peut écrire des données de type primitif à l'aide *relative* et *absolue* `put` . La différence frappante est que le fait de mettre des données en utilisant la méthode *relative* garde la trace de l'index auquel les données sont insérées pour vous, alors que la méthode absolue exige toujours de donner un index pour `put` les données.

Les deux méthodes permettent de *"chaîner" les* appels. Étant donné un tampon de taille suffisante, on peut faire ce qui suit:

```
buffer.putInt(0xCAFEBABE).putChar('c').putFloat(0.25).putLong(0xDEADBEEFCAFEBABE);
```

ce qui équivaut à:

```
buffer.putInt(0xCAFEBABE);
buffer.putChar('c');
buffer.putFloat(0.25);
buffer.putLong(0xDEADBEEFCAFEBABE);
```

Notez que la méthode fonctionnant sur les `byte` n'est pas nommée spécialement. En outre noter qu'il est également valide pour passer à la fois un `ByteBuffer` et un `byte[]` pour `put` . A part cela, tous les types primitifs ont des méthodes de `put` spécialisées.

Une note supplémentaire: L'index donné lors de l'utilisation de `put*` absolu `put*` est toujours compté en `byte` s.

Utilisation de base - Utilisation de DirectByteBuffer

`DirectByteBuffer` est une implémentation spéciale de `ByteBuffer` qui ne comporte aucun `byte[]` dessous.

Nous pouvons allouer un tel ByteBuffer en appelant:

```
ByteBuffer directBuffer = ByteBuffer.allocateDirect(16);
```

Cette opération allouera 16 octets de mémoire. Le contenu des tampons directs *peut* résider en dehors du tas normal récupéré.

Nous pouvons vérifier si ByteBuffer est direct en appelant:

```
directBuffer.isDirect(); // true
```

Les principales caractéristiques de `DirectByteBuffer` sont que JVM essaiera de travailler en mode natif sur la mémoire allouée sans mise en mémoire tampon supplémentaire, de sorte que les opérations effectuées sur celui-ci peuvent être plus rapides que celles effectuées sur ByteBuffer avec des tableaux situés en dessous.

Il est recommandé d'utiliser `DirectByteBuffer` avec des opérations d'E / S lourdes reposant sur la

vitesse d'exécution, comme la communication en temps réel.

Nous devons être conscients que si nous essayons d'utiliser la méthode `array()` , nous obtiendrons une `UnsupportedOperationException` . Il est donc recommandé de vérifier si notre ByteBuffer l'a (tableau d'octets) avant d'essayer d'y accéder:

```
byte[] arrayOfBytes;
if(buffer.hasArray()) {
    arrayOfBytes = buffer.array();
}
```

Une autre utilisation du tampon d'octet direct est l'interopérabilité via JNI. Comme un tampon d'octet direct n'utilise pas d' `byte[]` , mais un bloc de mémoire réel, il est possible d'accéder à cette mémoire directement via un pointeur en code natif. Cela permet d'économiser un peu de temps et d'encombrement lors de la répartition entre Java et la représentation native des données.

L'interface JNI définit plusieurs fonctions pour gérer les tampons d'octets directs: Support NIO .

Lire ByteBuffer en ligne: https://riptutorial.com/fr/java/topic/702/bytebuffer

Chapitre 16: Bytecode Modification

Examples

Qu'est-ce que Bytecode?

Bytecode est l'ensemble des instructions utilisées par la JVM. Pour illustrer cela, prenons ce programme Hello World.

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

C'est ce qu'il devient en compilant en bytecode.

```
public static main([Ljava/lang/String; args)V
    getstatic java/lang/System out Ljava/io/PrintStream;
    ldc "Hello World"
    invokevirtual java/io/PrintStream print(Ljava/lang/String;)V
```

Quelle est la logique derrière cela?

getstatic - Récupère la valeur d'un champ statique d'une classe. Dans ce cas, le *PrintStream "Out"* du *système* .

ldc - Poussez une constante sur la pile. Dans ce cas, la chaîne "Hello World"

invokevirtual - Invoque une méthode sur une référence chargée sur la pile et place le résultat sur la pile. Les paramètres de la méthode sont également extraits de la pile.

Eh bien, il doit y avoir plus de droit?

Il y a 255 opcodes, mais tous ne sont pas encore implémentés. Un tableau avec tous les opcodes actuels peut être trouvé ici: listes d'instructions de bytecode Java .

Comment puis-je écrire / modifier le bytecode?

Il existe plusieurs façons d'écrire et de modifier le bytecode. Vous pouvez utiliser un compilateur, utiliser une bibliothèque ou utiliser un programme.

Pour écrire:

- Jasmin
- Krakatau

Pour l'édition:

- Bibliothèques
  - ASM
  - Javassiste
  - BCEL - *Ne prend pas en charge Java 8+*
- Outils
  - Bytecode-Viewer
  - JBytedit
  - reJ - *Ne supporte pas Java 8+*
  - JBE - *Ne supporte pas Java 8+*

J'aimerais en savoir plus sur le bytecode!

Il y a probablement une page de documentation spécifique pour le bytecode. Cette page se concentre sur la modification de bytecode en utilisant différentes bibliothèques et outils.

Comment éditer des fichiers jar avec ASM

Tout d'abord, les classes du pot doivent être chargées. Nous utiliserons trois méthodes pour ce processus:

- loadClasses (Fichier)
- readJar (JarFile, JarEntry, Map)
- 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;
}
```

Avec ces méthodes, charger et modifier un fichier JAR devient une simple question de changement de ClassNodes dans une carte. Dans cet exemple, nous remplacerons toutes les chaînes du jar par des majuscules à l'aide de 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();
                }
            }
        }
    }
}
```

Maintenant que toutes les chaînes de ClassNode ont été modifiées, nous devons enregistrer les modifications. Pour enregistrer les modifications et avoir une sortie de travail, il faut faire quelques choses:

- Exporter les ClassNodes en octets
- Charger les entrées jar non class *(Ex: Manifest.mf / autres ressources binaires dans jar)* en octets
- Enregistrer tous les octets dans un nouveau fichier jar

À partir de la dernière partie ci-dessus, nous allons créer trois méthodes.

- processNodes (Map <String, ClassNode> noeuds)
- loadNonClasses (File jarFile)
- saveAsJar (Map <String, byte []> outBytes, String nomFichier)

Usage:

```
Map<String, byte[]> out = process(nodes, new HashMap<String, MappedClass>());
out.putAll(loadNonClassEntries(jarFile));
saveAsJar(out, "sample-edit.jar");
```

Les méthodes utilisées:

```
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();
    }
}
```

C'est tout. Toutes les modifications seront enregistrées dans "sample-edit.jar".

Comment charger un ClassNode en tant que 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;
    }
}
```

Comment renommer les classes dans un fichier 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 est une classe existante dans la bibliothèque ASM. Cependant, cela ne permet que de modifier les noms de classe. Si vous souhaitez renommer les champs et les méthodes, vous devez créer votre propre implémentation de la classe Remapper.

Javassist Basique

Javassist est une bibliothèque d'instrumentation de bytecode qui vous permet de modifier le code Java d'injection de bytecode qui sera converti en bytecode par Javassist et ajouté à la classe / méthode instrumentée lors de l'exécution.

Écrivons le premier transformateur qui prend réellement une classe hypothétique "com.my.to.be.instrumented.MyClass" et ajoute aux instructions de chaque méthode un appel de journal.

```
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;
    }
}
```

Maintenant, pour utiliser ce transformateur (afin que notre JVM appelle la méthode transform sur chaque classe au moment du chargement), nous devons ajouter cet instrument avec un agent:

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

public class EasyAgent {

    public static void premain(String agentArgs, Instrumentation inst) {

        // registers the transformer
        inst.addTransformer(new DynamicTransformer());
    }
}
```

La dernière étape pour démarrer notre première expérience avec un instrument consiste à enregistrer cette classe d'agent dans l'exécution de la machine JVM. Le moyen le plus simple est de l'enregistrer avec une option dans la commande shell:

```
java -javaagent:myAgent.jar MyJavaApplication
```

Comme nous pouvons le voir, le projet agent / transformer est ajouté en tant que fichier jar à l'exécution de toute application appelée MyJavaApplication qui est supposée contenir une classe nommée "com.my.to.be.instrumented.MyClass" pour exécuter réellement notre code injecté.

Lire Bytecode Modification en ligne: https://riptutorial.com/fr/java/topic/3747/bytecode-modification

Chapitre 17: Calendrier et ses sous-classes

Remarques

Depuis Java 8, `Calendar` et ses sous-classes ont été remplacés par le package java.time et ses sous-packages. Ils devraient être préférés, sauf si une API héritée nécessite un calendrier.

Examples

Création d'objets de calendrier

`Calendar` objets de `Calendar` peuvent être créés en utilisant `getInstance()` ou en utilisant le constructeur `GregorianCalendar` .

Il est important de noter que les mois dans `Calendar` sont à zéro, ce qui signifie que JANUARY est représenté par une valeur `int` 0. Afin de fournir un meilleur code, utilisez toujours des constantes de `Calendar` , telles que `Calendar.JANUARY` pour éviter les malentendus.

```
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);
```

Remarque : utilisez toujours les constantes de mois: La représentation numérique est trompeuse , par exemple `Calendar.JANUARY` a la valeur `0`

Champs de calendrier croissants / décroissants

`add()` et `roll()` peuvent être utilisés pour augmenter / diminuer les champs du `Calendar` .

```
Calendar calendar = new GregorianCalendar(2016, Calendar.MARCH, 31); // 31 March 2016
```

La méthode `add()` affecte tous les champs et se comporte efficacement si l'on ajoutait ou soustrait des dates réelles du calendrier.

```
calendar.add(Calendar.MONTH, -6);
```

L'opération ci-dessus supprime six mois du calendrier, ce qui nous ramène au 30 septembre 2015.

Pour changer un champ particulier sans affecter les autres champs, utilisez `roll()` .

```
calendar.roll(Calendar.MONTH, -6);
```

L'opération ci-dessus supprime six mois du *mois en cours* , le mois est donc identifié en

septembre. Aucun autre champ n'a été ajusté; l'année n'a pas changé avec cette opération.

Trouver AM / PM

Avec la classe Calendrier, il est facile de trouver AM ou PM.

```
  Calendar cal = Calendar.getInstance();
  cal.setTime(new Date());
  if (cal.get(Calendar.AM_PM) == Calendar.PM)
      System.out.println("It is PM");
```

Retrait des calendriers

Pour obtenir une différence entre deux `Calendar` , utilisez la méthode `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)
```

Lire Calendrier et ses sous-classes en ligne: https://riptutorial.com/fr/java/topic/165/calendrier-et-ses-sous-classes

Chapitre 18: Chargeurs de Classes

Remarques

Un classloader est une classe dont le but principal est d'assurer la médiation de l'emplacement et du chargement des classes utilisées par une application. Un chargeur de classe peut également rechercher et charger des *ressources* .

Les classes de chargeur de classe standard peuvent charger des classes et des ressources à partir de répertoires du système de fichiers et de fichiers JAR et ZIP. Ils peuvent également télécharger et mettre en cache des fichiers JAR et ZIP à partir d'un serveur distant.

Les chargeurs de classe sont normalement enchaînés, de sorte que la JVM essaye de charger les classes des bibliothèques de classes standard de préférence aux sources fournies par l'application. Les chargeurs de classes personnalisés permettent au programmeur de modifier cela. Le peut également faire des choses telles que le décryptage des fichiers bytecode et la modification bytecode.

Examples

Instancier et utiliser un chargeur de classe

Cet exemple de base montre comment une application peut instancier un chargeur de classe et l'utiliser pour charger dynamiquement une classe.

```
URL[] urls = new URL[] {new URL("file:/home/me/extras.jar")};
Classloader loader = new URLClassLoader(urls);
Class<?> myObjectClass = loader.findClass("com.example.MyObject");
```

Le classloader créé dans cet exemple aura le classloader par défaut en tant que parent, et essaiera d'abord de trouver une classe dans le classloader parent avant de chercher dans "extra.jar". Si la classe demandée a déjà été chargée, l'appel `findClass` renverra la référence à la classe précédemment chargée.

L'appel `findClass` peut échouer de différentes manières. Les plus courants sont:

- Si la classe nommée ne peut pas être trouvée, l'appel avec jeter `ClassNotFoundException` .
- Si la classe nommée dépend d'une autre classe introuvable, l'appel lancera `NoClassDefFoundError` .

Implémentation d'un classLoader personnalisé

Chaque chargeur personnalisé doit étendre directement ou indirectement la classe `java.lang.ClassLoader` . Les principaux *points d'extension* sont les méthodes suivantes:

- `findClass(String)` - surcharge cette méthode si votre chargeur de classe suit le modèle de

délégation standard pour le chargement de classe.
- `loadClass(String, boolean)` - surcharge cette méthode pour implémenter un autre modèle de délégation.
- `findResource` et `findResources` - surchargent ces méthodes pour personnaliser le chargement des ressources.

Les méthodes `defineClass` qui sont chargées de charger réellement la classe à partir d'un tableau d'octets sont `final` pour éviter la surcharge. Tout comportement personnalisé doit être effectué avant d'appeler `defineClass` .

Voici un simple qui charge une classe spécifique à partir d'un tableau d'octets:

```
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);
        }
    }
}
```

Comme nous avons uniquement remplacé la méthode `findClass` , ce chargeur de classe personnalisé se comportera comme suit lorsque `loadClass` est appelé.

1. La méthode `loadClass` du chargeur de `loadClass` appelle `findLoadedClass` pour voir si une classe portant ce nom a déjà été chargée par ce chargeur de classe. Si cela réussit, l'objet `Class` résultant est renvoyé au demandeur.
2. La méthode `loadClass` délègue ensuite au chargeur de classes parent en appelant son appel `loadClass` . Si le parent peut traiter la demande, il retournera un objet `Class` qui est ensuite renvoyé au demandeur.
3. Si le chargeur de classe parent ne peut pas charger la classe, `findClass` appelle alors notre méthode `findClass` substitution, en transmettant le nom de la classe à charger.
4. Si le nom demandé correspond à `this.classname` , nous appelons `defineClass` pour charger la classe réelle à partir du `this.classfile` octets `this.classfile` . L'objet `Class` résultant est ensuite renvoyé.
5. Si le nom ne correspond pas, nous lançons `ClassNotFoundException` .

Chargement d'un fichier .class externe

Pour charger une classe, il faut d'abord la définir. La classe est définie par le `ClassLoader` . Il n'y a qu'un seul problème, Oracle n'a pas écrit le `ClassLoader` du `ClassLoader` avec cette fonctionnalité

disponible. Pour définir la classe, nous devrons accéder à une méthode appelée `defineClass()` qui est une méthode privée du `ClassLoader` .

Pour y accéder, nous allons créer une nouvelle classe, `ByteClassLoader` , et l'étendre à `ClassLoader` . Maintenant que nous avons étendu notre classe à `ClassLoader` , nous pouvons accéder aux méthodes privées du `ClassLoader` . Pour rendre `defineClass()` disponible, nous allons créer une nouvelle méthode qui agira comme un miroir pour la `defineClass()` private `defineClass()` . Pour appeler la méthode privée , nous aurons besoin du nom de la classe, le `name` , les octets de classe, `classBytes` , crédits compensatoires du premier octet, qui sera `0` parce que `classBytes` les données commence à `classBytes[0]` , et le décalage de dernier octet, qui sera `classBytes.lenght` car il représente la taille des données, qui sera le dernier décalage.

```
public class ByteClassLoader extends ClassLoader {

    public Class<?> defineClass(String name, byte[] classBytes) {
        return defineClass(name, classBytes, 0, classBytes.length);
    }

}
```

Maintenant, nous avons une `defineClass()` publique `defineClass()` . Il peut être appelé en passant le nom de la classe et les octets de la classe comme arguments.

Disons que nous avons une classe nommée `MyClass` dans le paquet `stackoverflow` ...

Pour appeler la méthode, nous avons besoin des octets de classe, nous créons donc un objet `Path` représentant notre chemin de classe en utilisant la méthode `Paths.get()` et en passant le chemin de la classe binaire en argument. Maintenant, nous pouvons obtenir les octets de classe avec `Files.readAllBytes(path)` . Nous créons donc une instance de `ByteClassLoader` et utilisons la méthode que nous avons créée, `defineClass()` . Nous avons déjà les octets de classe, mais pour appeler notre méthode, nous avons également besoin du nom de classe qui est donné par le nom du paquet (point), le nom canonique de la classe, dans ce cas `stackoverflow.MyClass` .

```
Path path = Paths.get("MyClass.class");

ByteClassLoader loader = new ByteClassLoader();
loader.defineClass("stackoverflow.MyClass", Files.readAllBytes(path);
```

Remarque : La méthode `defineClass()` renvoie un objet `Class<?>` . Vous pouvez le sauvegarder si vous le souhaitez.

Pour charger la classe, il suffit d'appeler `loadClass()` et de passer le nom de la classe. Cette méthode peut lancer une `ClassNotFoundException` ; nous devons donc utiliser un bloc try cath

```
try{
    loader.loadClass("stackoverflow.MyClass");
} catch(ClassNotFoundException e){
    e.printStackTrace();
}
```

Lire Chargeurs de Classes en ligne: https://riptutorial.com/fr/java/topic/5443/chargeurs-de-classes

Chapitre 19: Chiffrement RSA

Examples

Un exemple utilisant un cryptosystème hybride composé de OAEP et GCM

L'exemple suivant crypte les données à l'aide d'un système de cryptage hybride comprenant AES GCM et OAEP, en utilisant leurs tailles de paramètres par défaut et une taille de clé AES de 128 bits.

OAEP est moins vulnérable aux attaques par bourrage que PKCS # 1 v1.5. GCM est également protégé contre les attaques par bourrage.

Le décryptage peut être effectué en récupérant d'abord la longueur de la clé encapsulée, puis en récupérant la clé encapsulée. La clé encapsulée peut ensuite être déchiffrée à l'aide de la clé privée RSA qui forme une paire de clés avec la clé publique. Après cela, le cryptogramme crypté AES / GCM peut être déchiffré dans le texte en clair original.

Le protocole consiste en:

1. un champ de longueur pour la clé `RSAPrivateKey` (`RSAPrivateKey` manque une méthode `getKeySize()`);
2. la clé encapsulée / encapsulée, de la même taille que la taille de la clé RSA en octets;
3. le cryptogramme GCM et le tag d'authentification 128 bits (ajoutés automatiquement par Java).

Remarques:

- Pour utiliser correctement ce code, vous devez fournir une clé RSA d'au moins 2048 bits, la taille la plus grande étant meilleure (mais plus lente, surtout pendant le déchiffrement).
- Pour utiliser AES-256, vous devez d'abord installer les fichiers de stratégie de cryptographie illimités .
- Au lieu de créer votre propre protocole, vous pouvez utiliser un format de conteneur tel que la syntaxe de message cryptographique (CMS / PKCS # 7) ou PGP.

Alors, voici l'exemple:

```
/**
 * 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();
}
```

Bien sûr, le chiffrement n'est pas très utile sans déchiffrement. Notez que cela renverra des informations minimales si le décryptage échoue.

```
/**
 * 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();
}
```

Lire Chiffrement RSA en ligne: https://riptutorial.com/fr/java/topic/1889/chiffrement-rsa

Chapitre 20: Choisir des collections

Introduction

Java offre une grande variété de collections. Choisir quelle collection utiliser peut être difficile. Reportez-vous à la section Exemples pour un organigramme facile à suivre pour choisir la collection appropriée pour le travail.

Examples

Organigramme des collections Java

Utilisez l'organigramme suivant pour choisir la collection appropriée pour le travail.

Cet organigramme était basé sur [http://i.stack.imgur.com/aSDsG.png) .

Lire Choisir des collections en ligne: https://riptutorial.com/fr/java/topic/10846/choisir-des-collections

Chapitre 21: Classe - Réflexion Java

Introduction

La classe java.lang.Class fournit de nombreuses méthodes permettant d'obtenir des métadonnées, d'examiner et de modifier le comportement d'exécution d'une classe.

Les packages java.lang et java.lang.reflect fournissent des classes pour la réflexion java.

Où est-il utilisé

L'API Reflection est principalement utilisée dans:

IDE (Integrated Development Environment), par exemple Eclipse, MyEclipse, NetBeans, etc.
Outils de test du débogueur, etc.

Examples

Méthode getClass () de la 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);
    }
}
```

Lire Classe - Réflexion Java en ligne: https://riptutorial.com/fr/java/topic/10151/classe---reflexion-java

Chapitre 22: Classe de propriétés

Introduction

L'objet properties contient une paire clé et valeur sous forme de chaîne. La classe java.util.Properties est la sous-classe de Hashtable.

Il peut être utilisé pour obtenir une valeur de propriété basée sur la clé de propriété. La classe Properties fournit des méthodes pour extraire des données du fichier de propriétés et stocker des données dans un fichier de propriétés. De plus, il peut être utilisé pour obtenir les propriétés du système.

Avantage du fichier de propriétés

La recompilation n'est pas requise si les informations sont modifiées à partir du fichier de propriétés: Si des informations sont modifiées à partir de

Syntaxe

- Dans un fichier de propriétés:
- clé = valeur
- #commentaire

Remarques

Un objet Propriétés est une carte dont les clés et les valeurs sont des chaînes par convention. Bien que les méthodes de Map puissent être utilisées pour accéder aux données, les méthodes plus sûres getProperty , setProperty et stringPropertyNames sont généralement utilisées à la place.

Les propriétés sont fréquemment stockées dans des fichiers de propriétés Java, qui sont de simples fichiers texte. Leur format est documenté en détail dans la méthode Properties.load . En résumé:

- Chaque paire clé / valeur est une ligne de texte avec un espace, est égale à (`=`) ou deux points (`:`) entre la clé et la valeur. Les égaux ou deux-points peuvent avoir n'importe quelle quantité d'espaces avant et après, ce qui est ignoré.
- Les espaces blancs sont toujours ignorés, les espaces à la fin sont toujours inclus.
- Une barre oblique inverse peut être utilisée pour échapper à n'importe quel caractère (sauf les minuscules `u`).
- Une barre oblique inverse à la fin de la ligne indique que la ligne suivante est une continuation de la ligne en cours. Cependant, comme pour toutes les lignes, les espaces blancs dans la ligne de continuation sont ignorés.
- Tout comme dans le code source Java, `\u` suivi de quatre chiffres hexadécimaux représente un caractère UTF-16.

La plupart des frameworks, y compris les propres installations de Java SE telles que java.util.ResourceBundle, chargent les fichiers de propriétés en tant que InputStreams. Lors du chargement d'un fichier de propriétés à partir d'un InputStream, ce fichier ne peut contenir que des caractères ISO 8859-1 (c'est-à-dire des caractères compris entre 0 et 255). Tout autre caractère doit être représenté comme `\u` s'échappe. Cependant, vous pouvez écrire un fichier texte dans n'importe quel encodage et utiliser l'outil native2ascii (fourni avec chaque JDK) pour le faire.

Si vous chargez un fichier de propriétés avec votre propre code, il peut s'agir d'un encodage quelconque, à condition que vous créiez un Reader (tel qu'un InputStreamReader) basé sur le Charset correspondant. Vous pouvez ensuite charger le fichier en utilisant load (Reader) au lieu de la méthode de chargement héritée (InputStream).

Vous pouvez également stocker des propriétés dans un simple fichier XML, ce qui permet au fichier lui-même de définir le codage. Un tel fichier peut être chargé avec la méthode loadFromXML . La DTD décrivant la structure de ces fichiers XML se trouve à l' adresse http://java.sun.com/dtd/properties.dtd .

Examples

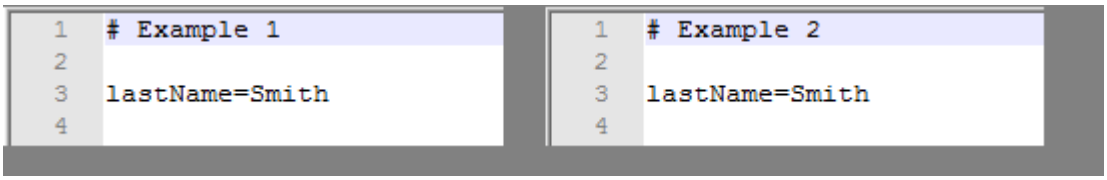
Chargement des propriétés

Pour charger un fichier de propriétés fourni avec votre application:

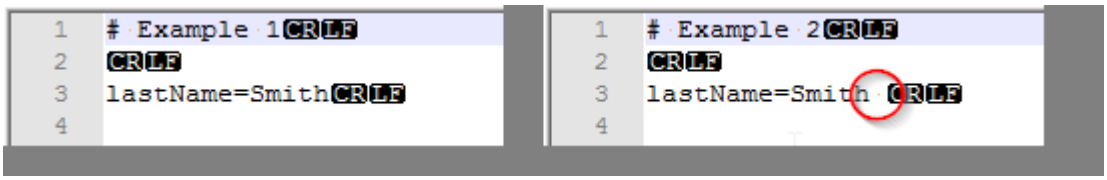
```
public class Defaults {

    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);
        }
    }

}
```

Les fichiers de propriétés indiquent les espaces vides:

Examinez attentivement ces deux fichiers de propriétés qui sont apparemment complètement identiques:



sauf qu'ils ne sont pas vraiment identiques:



(les captures d'écran proviennent de Notepad ++)

Comme les espaces blancs sont conservés, la valeur de `lastName` est `"Smith"` dans le premier cas et `"Smith "` dans le second cas.

Très rarement, c'est ce que les utilisateurs attendent et ils ne peuvent que spéculer sur le comportement par défaut de la classe `Properties` . Il est toutefois facile de créer une version améliorée des `Properties` pour résoudre ce problème. La classe suivante, **TrimmedProperties** , ne fait que cela. C'est un remplacement instantané pour la classe Propriétés 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;
    }
}
```

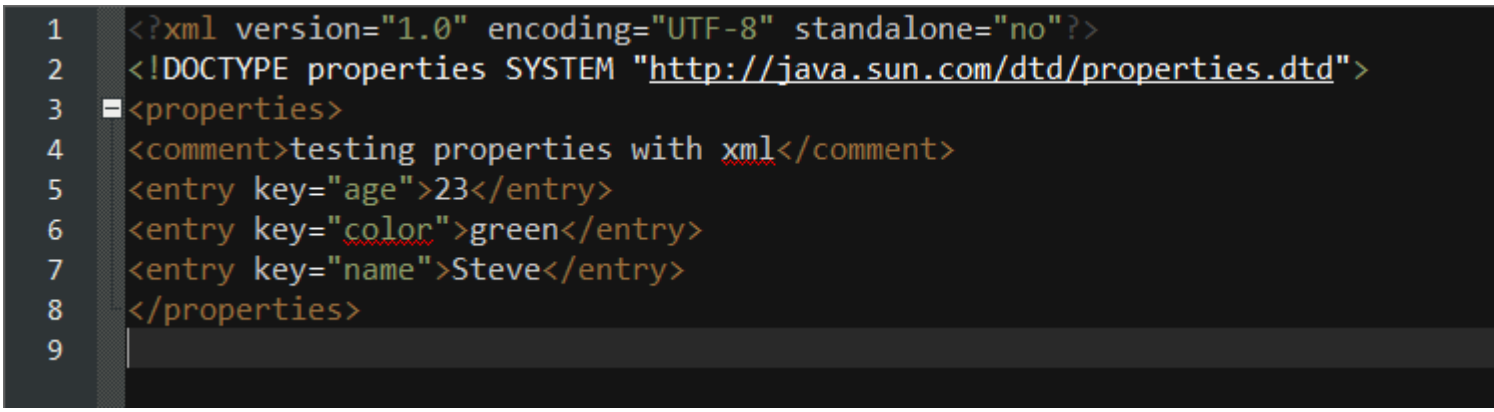
Enregistrement des propriétés en XML

Stockage des propriétés dans un fichier XML

La manière dont vous stockez les fichiers de propriétés en tant que fichiers XML est très similaire à la manière dont vous les `.properties` tant que fichiers `.properties` . Juste au lieu d'utiliser le `store()` vous utiliseriez `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");
}
```

Lorsque vous ouvrez le fichier, cela ressemblera à ceci.



Chargement des propriétés à partir d'un fichier XML

Maintenant, pour charger ce fichier en tant que `properties` vous devez appeler le `loadFromXML()` au lieu du `load()` que vous utiliseriez avec les fichiers `.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);
    }
}
```

Lorsque vous exécutez ce code, vous obtiendrez les éléments suivants dans la console:

```
age=23
color=green
name=Steve
```

Lire Classe de propriétés en ligne: https://riptutorial.com/fr/java/topic/576/classe-de-proprietes

Chapitre 23: Classe EnumSet

Introduction

La classe Java EnumSet est l'implémentation Set spécialisée à utiliser avec les types enum. Il hérite de la classe AbstractSet et implémente l'interface 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());
  }
}
```

Lire Classe EnumSet en ligne: https://riptutorial.com/fr/java/topic/10159/classe-enumset

Chapitre 24: Classe immuable

Introduction

Les objets immuables sont des instances dont l'état ne change pas après son initialisation. Par exemple, String est une classe immuable et, une fois instanciée, sa valeur ne change jamais.

Remarques

Quelques classes immuables en Java:

1. java.lang.String
2. Les classes d'encapsulation pour les types primitifs: 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 plupart des classes d'énumération sont immuables, mais cela dépend en fait du cas concret.
4. java.math.BigInteger et java.math.BigDecimal (au moins les objets de ces classes eux-mêmes)
5. java.io.Fichier. Notez que cela représente un objet externe à la machine virtuelle (un fichier sur le système local), qui peut exister ou non, et comporte certaines méthodes modifiant et interrogeant l'état de cet objet externe. Mais l'objet File lui-même reste immuable.

Examples

Règles pour définir des classes immuables

Les règles suivantes définissent une stratégie simple pour créer des objets immuables.

1. Ne fournissez pas de méthodes "setter" - méthodes qui modifient les champs ou les objets auxquels les champs font référence.
2. Rendez tous les champs définitifs et privés.
3. Ne permettez pas aux sous-classes de remplacer les méthodes. La manière la plus simple de le faire est de déclarer la classe comme finale. Une approche plus sophistiquée consiste à rendre le constructeur privé et à construire des instances dans les méthodes d'usine.
4. Si les champs d'instance contiennent des références à des objets mutables, ne permettez pas que ces objets soient modifiés:
5. Ne fournissez pas de méthodes modifiant les objets mutables.
6. Ne partagez pas les références aux objets mutables. Ne stockez jamais de références à des objets externes, mutables, transmis au constructeur; Si nécessaire, créez des copies et stockez les références aux copies. De même, créez des copies de vos objets internes mutables si nécessaire pour éviter de renvoyer les originaux dans vos méthodes.

Exemple sans références mutables

```
public final class Color {
    final private int red;
    final private int green;
    final private int blue;

    private void check(int red, int green, int blue) {
        if (red < 0 || red > 255 || green < 0 || green > 255 || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public Color(int red, int green, int blue) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    public Color invert() {
        return new Color(255 - red, 255 - green, 255 - blue);
    }
}
```

Exemple avec des références mutables

Dans ce cas, la classe Point est mutable et certains utilisateurs peuvent modifier l'état de l'objet de cette 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;
    }
```

Quel est l'avantage de l'immuabilité?

L'avantage de l'immuabilité vient avec la concurrence. Il est difficile de maintenir une exactitude dans les objets mutables, car plusieurs threads pourraient essayer de changer l'état du même objet, ce qui conduit à ce que certains threads voient un état différent du même objet, en fonction du timing des lectures et des écritures. objet.

En ayant un objet immuable, on peut s'assurer que tous les threads qui regardent l'objet verront le même état, car l'état d'un objet immuable ne changera pas.

Lire Classe immuable en ligne: https://riptutorial.com/fr/java/topic/10561/classe-immuable

Chapitre 25: Classe locale

Introduction

Une classe, c.-à-d. Créée dans une méthode, est appelée classe interne locale en Java. Si vous souhaitez appeler les méthodes de la classe interne locale, vous devez instancier cette classe dans la méthode.

Examples

Classe 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();
 }
}
```

Lire Classe locale en ligne: https://riptutorial.com/fr/java/topic/10160/classe-locale

Chapitre 26: Classes et Objets

Introduction

Les objets ont des états et des comportements. Exemple: un chien a des états - couleur, nom, race ainsi que comportements - qui remue la queue, aboie, mange. Un objet est une instance d'une classe.

Classe - Une classe peut être définie comme un modèle / plan directeur décrivant le comportement / état pris en charge par l'objet de son type.

Syntaxe

- class Example {} // mot-clé de la classe, nom, corps

Examples

Classe la plus simple possible

```
class TrivialClass {}
```

Une classe comprend au minimum le mot `class` clé de `class` , un nom et un corps, qui peuvent être vides.

Vous instanciez une classe avec le `new` opérateur.

```
TrivialClass tc = new TrivialClass();
```

Membre d'objet vs membre statique

Avec cette classe:

```
class ObjectMemberVsStaticMember {

    static int staticCounter = 0;
    int memberCounter = 0;

    void increment() {
        staticCounter ++;
        memberCounter++;
    }
}
```

l'extrait de code suivant:

```
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);
```

produit cette sortie:

```
o1 static counter 3
o1 member counter 1

o2 static counter 3
o2 member counter 2

ObjectMemberVsStaticMember.staticCounter = 3
```

Remarque: Vous ne devez pas appeler `static` membres `static` sur des objets, mais sur des classes. Bien que cela ne fasse aucune différence pour la JVM, les lecteurs humains l'apprécieront.

`static` membres `static` font partie de la classe et n'existent qu'une fois par classe. Les membres non `static` existent sur les instances, il existe une copie indépendante pour chaque instance. Cela signifie également que vous devez accéder à un objet de cette classe pour accéder à ses membres.

Méthodes de surcharge

Parfois, la même fonctionnalité doit être écrite pour différents types d'entrées. A ce moment, on peut utiliser le même nom de méthode avec un ensemble de paramètres différent. Chaque ensemble de paramètres différent est appelé signature de méthode. Comme vu dans l'exemple, une seule méthode peut avoir plusieurs signatures.

```
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"
    }
}
```

L'avantage est que la même fonctionnalité est appelée avec deux nombres d'entrées différents. En invoquant la méthode en fonction de l'entrée que nous transmettons (dans ce cas, une valeur de chaîne ou deux valeurs de chaîne), la méthode correspondante est exécutée.

Les méthodes peuvent être surchargées:

1. Basé sur le **nombre de paramètres** transmis.

   Exemple: `method(String s)` et `method(String s1, String s2)` .

2. Basé sur l' **ordre des paramètres** .

   Exemple: `method(int i, float f)` et `method(float f, int i))` .

***Remarque: Les** méthodes ne peuvent pas être surchargées en modifiant uniquement le type de retour (`int method()` est considéré comme identique à la `String method()` et lancera une `RuntimeException` cas de tentative). Si vous modifiez le type de retour, vous devez également modifier les paramètres pour surcharger.*

Construction et utilisation d'objets de base

Les objets viennent dans leur propre classe, un exemple simple serait une voiture (explications détaillées ci-dessous):

```
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;
    }

}
```

Les objets sont des **instances de** leur classe. La manière dont vous **créeriez un objet** serait donc d'appeler la classe Car **de deux manières différentes** dans votre classe principale (méthode principale en Java ou onCreate sous Android).

Option 1

```
`Car newCar = new Car(30, 10, "Ferrari", "Red");
```

L'option 1 est l'endroit où vous racontez essentiellement au programme tout ce qui concerne la voiture lors de la création de l'objet. Toute modification de toute propriété de la voiture nécessiterait d'appeler l'une des méthodes telles que la méthode `repaintCar` . Exemple:

```
 newCar.repaintCar("Blue");
```

Remarque: Assurez-vous de transmettre le type de données correct à la méthode. Dans l'exemple ci-dessus, vous pouvez également passer une variable à la méthode `repaintCar` **tant que le type de données est correct** .

C'était un exemple de modification des propriétés d'un objet, la réception des propriétés d'un objet nécessiterait l'utilisation d'une méthode de la classe Car ayant une valeur de retour (c'est-à-dire une méthode non `void`). Exemple:

```
String myCarName = newCar.getName();  //returns string "Ferrari"
```

L'option 1 est la **meilleure** option lorsque vous avez **toutes les données de l'objet** au moment de la création.

Option 2

```
`Car newCar = new Car();
```

L'option 2 obtient le même effet mais nécessite plus de travail pour créer un objet correctement. Je veux rappeler ce constructeur dans la classe Car:

```
public void Car(){
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }
```

Notez que vous n'avez pas à transmettre de paramètres à l'objet pour le créer. Ceci est très utile lorsque vous ne possédez pas tous les aspects de l'objet, mais que vous devez utiliser les composants que vous possédez. Cela définit des données génériques dans chacune des variables d'instance de l'objet. Ainsi, si vous appelez une donnée qui n'existe pas, aucune erreur n'est générée.

Remarque: N'oubliez pas que vous devez définir les parties de l'objet plus tard avec lesquelles vous ne l'avez pas initialisé. Par exemple,

```
Car myCar = new Car();
String color = Car.getColor(); //returns empty string
```

C'est une erreur courante parmi les objets qui ne sont pas initialisés avec toutes leurs données.

Les erreurs ont été évitées car il existe un constructeur qui permet de créer un objet Car vide avec **des variables** autonomes (`public Car(){}`), mais aucune partie de myCar n'a été réellement personnalisée. **Exemple correct de création d'objet automobile:**

```
Car myCar = new Car();
myCar.nameCar("Ferrari");
myCar.paintCar("Purple");
myCar.setGallonsInTank(10);
myCar.setMPG(30);
```

Et, pour rappel, obtenez les propriétés d'un objet en appelant une méthode dans votre classe principale. Exemple:

```
String myCarName = myCar.getName(); //returns string "Ferrari"
```

Constructeurs

Les constructeurs sont des méthodes spéciales nommées après la classe et sans type de retour, et sont utilisées pour construire des objets. Les constructeurs, comme les méthodes, peuvent prendre des paramètres d'entrée. Les constructeurs sont utilisés pour initialiser des objets. Les classes abstraites peuvent aussi avoir des constructeurs.

```
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
```

Il est important de comprendre que les constructeurs sont différents des méthodes de plusieurs manières:

1. Les constructeurs peuvent uniquement prendre les modificateurs `public` , `private` et `protected` , et ne peuvent pas être déclarés `abstract` , `final` , `static` ou `synchronized` .
2. Les constructeurs n'ont pas de type de retour.
3. Les constructeurs DOIVENT être nommés comme le nom de la classe. Dans l'exemple `Hello` , le nom du constructeur de l'objet `Hello` est identique au nom de la classe.
4. Le mot `this` clé `this` a une utilisation supplémentaire dans les constructeurs. `this.method(...)` appelle une méthode sur l'instance en cours, alors que `this(...)` fait référence à un autre constructeur de la classe en cours avec des signatures différentes.

Les constructeurs peuvent également être appelés par héritage en utilisant le mot `super` clé `super` .

```
public class SuperManClass{

    public SuperManClass(){
        // some implementation
    }

    // ... methods
}


public class BatmanClass extends SupermanClass{
    public BatmanClass(){
        super();
    }
    //... methods...
}
```

Voir Spécification du langage Java n ° 8.8 et n ° 15.9

Initialisation des champs finaux statiques à l'aide d'un initialiseur statique

Pour initialiser un `static final` nécessitant l'utilisation de plusieurs expressions, un initialiseur `static` peut être utilisé pour affecter la valeur. L'exemple suivant initialise un ensemble de `String` de caractères non modifiable:

```
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);
    }
}
```

Expliquer ce qu'est la surcharge et la dérogation de la méthode.

Le remplacement de méthodes et la surcharge sont deux formes de polymorphisme supportées par Java.

Surcharge de méthode

La surcharge de méthode (également connue sous le nom de polymorphisme statique) est une façon dont vous pouvez avoir deux (ou plusieurs) méthodes (fonctions) avec le même nom dans une seule classe. Oui c'est aussi simple que ça.

```
public class Shape{
    //It could be a circle or rectangle or square
    private String type;
```

```
    //To calculate area of rectangle
    public Double area(Long length, Long breadth){
        return (Double) length * breadth;
    }

     //To calculate area of a circle
     public Double area(Long radius){
        return (Double) 3.14 * r * r;
    }
}
```

De cette façon, l'utilisateur peut appeler la même méthode pour la zone en fonction du type de forme dont il dispose.

Mais la vraie question est maintenant, comment java compiler va-t-il distinguer quel corps de méthode doit être exécuté?

Eh bien, Java a clairement indiqué que même si les **noms de méthodes** (`area()` dans notre cas) **peuvent être identiques, mais que la méthode des arguments est différente, elle devrait être différente.**

> Les méthodes surchargées doivent avoir une liste d'arguments différente (quantité et types).

Cela étant dit, nous ne pouvons pas ajouter une autre méthode pour calculer l'aire d'un carré comme ceci: `public Double area(Long side)` car dans ce cas, elle entrera en conflit avec la méthode de cercle et provoquera une **ambiguïté** pour le compilateur Java.

Dieu merci, il y a des relaxations en écrivant des méthodes surchargées comme

> Peut avoir différents types de retour.
>
> Peut avoir différents modificateurs d'accès.
>
> Peut lancer différentes exceptions.

Pourquoi est-ce ce qu'on appelle le polymorphisme statique?

Eh bien, c'est parce que les méthodes surchargées à invoquer sont décidées au moment de la compilation, en fonction du nombre réel d'arguments et des types de compilation des arguments.

> L'une des raisons courantes de l'utilisation de la surcharge de méthodes est la simplicité du code fourni. Par exemple rappelez-vous `String.valueOf()` qui prend presque n'importe quel type d'argument? Ce qui est écrit derrière la scène est probablement quelque chose comme ceci: -

```
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)
```

Dérogation de méthode

Eh bien, la méthode qui remplace (oui vous le devinez bien, c'est aussi le polymorphisme dynamique) est un sujet un peu plus intéressant et complexe.

En cas de substitution de méthode, nous écrasons le corps de la méthode fourni par la classe parente. Je l'ai? Non? Passons en revue un exemple.

```
public abstract class Shape{

    public abstract Double area(){
        return 0.0;
    }
}
```

Nous avons donc une classe appelée Shape et elle a une méthode appelée zone qui renverra probablement la zone de la forme.

Disons que maintenant nous avons deux classes appelées Circle et 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;
    }
}
```

De même classe de rectangle:

```
 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;
    }
}
```

Donc, maintenant, les deux classes de vos enfants ont mis à jour le corps de la méthode fourni par la classe parent (`Shape`). Maintenant, la question est comment voir le résultat? Eh bien laisse faire le vieux moyen `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());

    }
}
```

Hou la la! n'est-ce pas génial? Deux objets de même type appelant les mêmes méthodes et renvoyant des valeurs différentes. Mon ami, c'est le pouvoir du polymorphisme dynamique.

Voici un tableau pour mieux comparer les différences entre les deux: -

Surcharge de méthode	Dérogation de méthode
La surcharge de la méthode est utilisée pour améliorer la lisibilité du programme.	La substitution de méthode est utilisée pour fournir l'implémentation spécifique de la méthode déjà fournie par sa super classe.
La surcharge de méthode est effectuée dans la classe.	La substitution de méthode se produit dans deux classes ayant une relation IS-A (héritage).
En cas de surcharge de la méthode, le paramètre doit être différent.	En cas de substitution de méthode, le paramètre doit être identique.
La surcharge de méthode est l'exemple du polymorphisme de compilation.	La substitution de méthode est l'exemple du polymorphisme d'exécution.
En Java, la surcharge de la méthode ne peut pas être effectuée en modifiant uniquement le type de retour de la méthode. Le type de retour peut être identique ou différent dans la surcharge de la méthode. Mais vous devez changer le paramètre.	Le type de retour doit être identique ou covariant lors de la substitution de méthode.

Lire Classes et Objets en ligne: https://riptutorial.com/fr/java/topic/114/classes-et-objets

Chapitre 27: Classes imbriquées et internes

Introduction

En utilisant Java, les développeurs ont la possibilité de définir une classe dans une autre classe. Une telle classe s'appelle une classe imbriquée . Les classes imbriquées sont appelées classes internes si elles ont été déclarées comme non statiques, sinon elles sont simplement appelées classes imbriquées statiques. Cette page sert à documenter et à fournir des détails sur les exemples d'utilisation des classes imbriquées et internes Java.

Syntaxe

- Classe publique OuterClass {classe publique InnerClass {}} // Les classes internes peuvent également être privées
- Classe publique OuterClass {classe statique publique StaticNestedClass {}} // Les classes imbriquées statiques peuvent également être privées
- public void method () {classe privée LocalClass {}} // Les classes locales sont toujours privées
- SomeClass anonymousClassInstance = new SomeClass () {}; // Les classes internes anonymes ne peuvent pas être nommées, donc l'accès est sans objet. Si 'SomeClass ()' est abstrait, le corps doit implémenter toutes les méthodes abstraites.
- SomeInterface anonymousClassInstance = new SomeInterface () {}; // Le corps doit implémenter toutes les méthodes d'interface.

Remarques

Terminologie et classification

La spécification de langage Java (JLS) classe les différents types de classe Java comme suit:

> Une *classe de niveau supérieur* est une classe qui n'est pas une classe imbriquée.
>
> Une *classe imbriquée* est une classe dont la déclaration se produit dans le corps d'une autre classe ou interface.
>
> Une *classe interne* est une classe imbriquée qui n'est pas explicitement ou implicitement déclarée statique.
>
> Une classe interne peut être une *classe membre non statique* , une *classe locale* ou une *classe anonyme* . Une classe membre d'une interface est implicitement statique, elle n'est donc jamais considérée comme une classe interne.

En pratique, les programmeurs se réfèrent à une classe de niveau supérieur contenant une classe interne en tant que "classe externe". En outre, il existe une tendance à utiliser "classe imbriquée" pour faire référence uniquement aux classes imbriquées statiques (explicitement ou

implicitement).

Notez qu'il existe une relation étroite entre les classes internes anonymes et les lambda, mais les lambda sont des classes.

Différences sémantiques

- Les classes supérieures sont le "cas de base". Ils sont visibles par d'autres parties d'un programme soumis à des règles de visibilité normales basées sur la sémantique des modificateurs d'accès. S'ils ne sont pas abstraits, ils peuvent être instanciés par tout code où les constructeurs pertinents sont visibles sur la base des modificateurs d'accès.

- Les classes imbriquées statiques suivent les mêmes règles d'accès et d'instanciation que les classes de niveau supérieur, à deux exceptions près:

    - Une classe imbriquée peut être déclarée comme `private` , ce qui la rend inaccessible en dehors de sa classe supérieure.
    - Une classe imbriquée a accès aux membres `private` de la classe de niveau supérieur englobante et de toutes ses classes testées.

    Cela rend les classes imbriquées statiques utiles lorsque vous devez représenter plusieurs "types d'entités" dans une limite d'abstraction étroite; Par exemple, lorsque les classes imbriquées sont utilisées pour masquer les "détails d'implémentation".

- Les classes internes ajoutent la possibilité d'accéder aux variables non statiques déclarées dans des étendues englobantes:

    - Une classe membre non statique peut faire référence à des variables d'instance.
    - Une classe locale (déclarée dans une méthode) peut également faire référence aux variables locales de la méthode, à condition qu'elles soient `final` . (Pour Java 8 et versions ultérieures, ils peuvent être *effectivement définitifs* .)
    - Une classe interne anonyme peut être déclarée dans une classe ou une méthode et peut accéder à des variables selon les mêmes règles.

    Le fait qu'une instance de classe interne puisse faire référence à des variables dans une instance de classe englobante a des implications pour l'instanciation. Plus précisément, une instance englobante doit être fournie, implicitement ou explicitement, lorsqu'une instance d'une classe interne est créée.

Examples

Une pile simple utilisant une classe imbriquée

```
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;
    }
}
```

Et son utilisation, qui (notamment) ne reconnaît pas du tout l'existence de la classe imbriquée.

```
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() + ", ");
        }
    }
}
```

Classes imbriquées statiques et non statiques

Lors de la création d'une classe imbriquée, vous avez le choix entre avoir cette classe imbriquée statique:

```
public class OuterClass1 {

    private static class StaticNestedClass {

    }

}
```

Ou non statique:

```
public class OuterClass2 {
```

```
    private class NestedClass {

    }

}
```

À la base, les classes imbriquées statiques *n'ont pas d'* ***instance*** *environnante* de la classe externe, contrairement aux classes imbriquées non statiques. Cela affecte à la fois où / quand on est autorisé à instancier une classe imbriquée et quelles instances de ces classes imbriquées sont autorisées à accéder. Ajout à l'exemple ci-dessus:

```
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
    }
}
```

Ainsi, votre décision de statique contre non-statique dépend principalement de la nécessité ou non d'accéder directement aux champs et aux méthodes de la classe externe, même si cela a aussi des conséquences sur quand et où vous pouvez construire la classe imbriquée.

En règle générale, faites en sorte que vos classes imbriquées soient statiques sauf si vous devez accéder aux champs et aux méthodes de la classe externe. Comme pour rendre vos champs privés à moins que vous en ayez besoin publiquement, cela réduit la visibilité disponible pour la classe imbriquée (en ne permettant pas l'accès à une instance externe), réduisant ainsi le risque d'erreur.

Modificateurs d'accès pour les classes internes

Une explication complète des modificateurs d'accès en Java peut être trouvée ici . Mais comment interagissent-ils avec les classes internes?

`public` , comme d'habitude, donne un accès illimité à toute portée pouvant accéder au type.

```
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
    }
}
```

Les deux `protected` et le modificateur par défaut (de rien) se comportent aussi bien que prévu, comme ils le font pour les classes non imbriquées.

`private` intéressant, `private` , ne se limite pas à la classe à laquelle il appartient. Au contraire, il se limite à l'unité de compilation - le fichier .java. Cela signifie que les classes externes ont un accès complet aux champs et méthodes de classe interne, même si elles sont marquées comme `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 classe interne elle-même peut avoir une visibilité autre que `public` . En le marquant `private` ou un autre modificateur d'accès restreint, les autres classes (externes) ne seront pas autorisées à importer et assigner le type. Cependant, ils peuvent toujours obtenir des références à des objets de ce type.

```
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
    }
}
```

Classes internes anonymes

Une classe interne anonyme est une forme de classe interne déclarée et instanciée avec une seule instruction. En conséquence, il n'y a pas de nom pour la classe qui peut être utilisé ailleurs dans le programme; c'est à dire qu'il est anonyme.

Les classes anonymes sont généralement utilisées dans les situations où vous devez pouvoir créer une classe légère à transmettre en tant que paramètre. Cela se fait généralement avec une interface. Par exemple:

```
public static Comparator<String> CASE_INSENSITIVE =
        new Comparator<String>() {
            @Override
            public int compare(String string1, String string2) {
```

```
                return string1.toUpperCase().compareTo(string2.toUpperCase());
            }
        };
```

Cette classe anonyme définit un objet `Comparator<String>` (`CASE_INSENSITIVE`) qui compare deux chaînes en ignorant les différences dans la casse.

Les autres interfaces fréquemment implémentées et instanciées à l'aide de classes anonymes sont `Runnable` et `Callable` . Par exemple:

```
// 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"
```

Les classes internes anonymes peuvent également être basées sur des classes. Dans ce cas, la classe anonyme `extends` implicitement la classe existante. Si la classe étendue est abstraite, la classe anonyme doit implémenter toutes les méthodes abstraites. Il peut également remplacer les méthodes non abstraites.

Constructeurs

Une classe anonyme ne peut pas avoir de constructeur explicite. Au lieu de cela, un constructeur implicite est défini qui utilise `super(...)` pour transmettre tous les paramètres à un constructeur de la classe en cours d'extension. Par exemple:

```
SomeClass anon = new SomeClass(1, "happiness") {
            @Override
            public int someMethod(int arg) {
                // do something
            }
        };
```

Le constructeur implicite de notre sous-classe anonyme de `SomeClass` appellera un constructeur de `SomeClass` correspondant à la signature d'appel `SomeClass(int, String)` . Si aucun constructeur n'est disponible, vous obtiendrez une erreur de compilation. Toutes les exceptions lancées par le constructeur correspondant sont également renvoyées par le constructeur implicite.

Naturellement, cela ne fonctionne pas lors de l'extension d'une interface. Lorsque vous créez une classe anonyme à partir d'une interface, la superclasse de classes est `java.lang.Object` qui ne possède qu'un constructeur no-args.

Méthode Classes internes

Une classe écrite dans une méthode appelée **méthode classe interne locale** . Dans ce cas, la

portée de la classe interne est restreinte dans la méthode.

Une classe interne à la méthode locale ne peut être instanciée que dans la méthode où la classe interne est définie.

L'exemple de l'utilisation de la classe interne locale method:

```
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'exécution donnera une sortie: `Method local inner class 1` .

Accéder à la classe externe à partir d'une classe interne non statique

La référence à la classe externe utilise le nom de la classe et `this`

```
public class OuterClass {
    public class InnerClass {
        public void method() {
            System.out.println("I can access my enclosing class: " + OuterClass.this);
        }
    }
}
```

Vous pouvez accéder directement aux champs et aux méthodes de la classe externe.

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        public void method() {
            System.out.println("I can access " + counter);
        }
    }
}
```

Mais en cas de collision de noms, vous pouvez utiliser la référence de classe externe.

```
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;
        }
    }
}
```

Créer une instance de classe interne non statique depuis l'extérieur

Une classe interne visible par toute classe externe peut également être créée à partir de cette classe.

La classe interne dépend de la classe externe et nécessite une référence à une instance de celle-ci. Pour créer une instance de la classe interne, l'opérateur `new` doit uniquement être appelé sur une instance de la classe externe.

```
class OuterClass {

    class InnerClass {
    }
}

class OutsideClass {

    OuterClass outer = new OuterClass();

    OuterClass.InnerClass createInner() {
        return outer.new InnerClass();
    }
}
```

Notez l'utilisation en tant que `outer.new` .

Lire Classes imbriquées et internes en ligne: https://riptutorial.com/fr/java/topic/3317/classes-imbriquees-et-internes

Chapitre 28: Clonage d'objets

Remarques

Le clonage peut être délicat, en particulier lorsque les champs de l'objet contiennent d'autres objets. Il existe des situations où vous souhaitez effectuer une copie complète , au lieu de copier uniquement les valeurs de champ (c.-à-d. Les références aux autres objets).

La ligne du bas est clone est cassé , et vous devriez réfléchir à deux fois avant d'implémenter l'interface `Cloneable` et de `Cloneable` la méthode de `clone` . Le `clone` méthode est déclarée dans l' `Object` classe et non dans la `Cloneable` interface, donc `Cloneable` ne fonctionne pas comme une interface , car il ne dispose pas d' un public `clone` méthode. Le résultat est que le contrat d'utilisation du `clone` est peu documenté et faiblement appliqué. Par exemple, une classe qui remplace le `clone` s'appuie parfois sur toutes ses classes parentes qui ont également la priorité sur le `clone` . Ils ne sont pas obligés de le faire, et s'ils ne le font pas, votre code peut générer des exceptions.

Une solution bien meilleure pour fournir une fonctionnalité de clonage consiste à fournir un *constructeur de copie* ou une *fabrique de copies* . Reportez-vous à l' article 11 de Java efficace de Joshua Bloch : remplacer judicieusement le clone.

Examples

Clonage à l'aide d'un constructeur de copie

Un moyen simple de cloner un objet consiste à implémenter un constructeur de 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
```

Clonage en implémentant l'interface Clonable

Clonage d'un objet en implémentant l'interface 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
```

Cloner en effectuant une copie superficielle

Le comportement par défaut lors du clonage d'un objet consiste à effectuer une copie superficielle des champs de l'objet. Dans ce cas, l'objet d'origine et l'objet cloné contiennent des références aux mêmes objets.

Cet exemple montre ce comportement.

```
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));
}
```

Cloner en effectuant une copie en profondeur

Pour copier des objets imbriqués, une copie en profondeur doit être effectuée, comme illustré dans cet exemple.

```
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));
}
```

Clonage à l'aide d'une fabrique de copies

```
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)
    }

}
```

Lire Clonage d'objets en ligne: https://riptutorial.com/fr/java/topic/2830/clonage-d-objets

Chapitre 29: Code officiel Oracle standard

Introduction

Le guide de style officiel d'Oracle pour Java Programming Language est une norme suivie par les développeurs d'Oracle et recommandée par tout autre développeur Java. Il couvre les noms de fichiers, l'organisation des fichiers, l'indentation, les commentaires, les déclarations, les instructions, les espaces blancs, les conventions de dénomination, les pratiques de programmation et inclut un exemple de code.

Remarques

- Les exemples ci-dessus suivent strictement le nouveau guide de style officiel d'Oracle. En d'autres termes, ils ne sont *pas* subjectivement constitués par les auteurs de cette page.
- Le guide de style officiel a été soigneusement écrit pour être compatible avec le guide de style original et la majorité du code dans la nature.
- Le guide de style officiel a été pairs examiné par entre autres, Brian Goetz (langage Java Architect) et Mark Reinhold (architecte en chef de la plate - forme Java).
- Les exemples ne sont pas normatifs. Bien qu'ils aient l'intention d'illustrer la manière correcte de formater le code, il peut exister d'autres moyens de formater correctement le code. C'est un principe général: il peut y avoir plusieurs manières de formater le code, toutes respectant les directives officielles.

Examples

Conventions de nommage

Noms de paquets

- Les noms de paquets doivent être en minuscules, sans caractères de soulignement ou autres caractères spéciaux.
- Les noms de package commencent par la partie d'autorité inversée de l'adresse Web de la société du développeur. Cette partie peut être suivie d'une sous-structure de package dépendant de la structure de projet / programme.
- N'utilisez pas le pluriel. Suivez la convention de l'API standard qui utilise par exemple `java.lang.annotation` et non `java.lang.annotations` .
- **Exemples:** `com.yourcompany.widget.button` , `com.yourcompany.core.api`

Noms de classe, d'interface et d'énumération

- Les noms de classe et d'énumération doivent généralement être des noms.
- Les noms d'interface doivent généralement être des noms ou des adjectifs se terminant par… able.
- Utilisez la casse mixte avec la première lettre de chaque mot en majuscule (c.-à- d . CamelCase).
- Correspond à l'expression régulière `^[AZ][a-zA-Z0-9]*$` .
- Utilisez des mots entiers et évitez d'utiliser des abréviations à moins que l'abréviation ne soit plus utilisée que la forme longue.
- Formatez une abréviation en tant que mot si elle fait partie d'un nom de classe plus long.
- **Exemples:** `ArrayList` , `BigInteger` , `ArrayIndexOutOfBoundsException` , `Iterable` .

Noms de méthode

Les noms de méthode doivent généralement être des verbes ou d'autres descriptions d'actions

- Ils doivent correspondre à l'expression régulière `^[az][a-zA-Z0-9]*$` .
- Utilisez la casse mixte avec la première lettre en minuscule.
- **Exemples:** `toString` , `hashCode`

Les variables

Les noms de variables doivent être en casse mixte avec la première lettre en minuscule

- Correspond à l'expression régulière `^[az][a-zA-Z0-9]*$`
- Recommandation supplémentaire: Variables
- **Exemples:** `elements` , `currentIndex`

Variables de type

Pour les cas simples où peu de variables de type impliquent, utilisez une seule lettre majuscule.

- Correspond à l'expression régulière `^[AZ][0-9]?$`
- Si une lettre est plus descriptive qu'une autre (comme `K` et `V` pour les clés et les valeurs dans les cartes ou `R` pour un type de retour de fonction), utilisez-la, sinon utilisez `T`
- Pour les cas complexes où les variables de type lettre unique deviennent confuses, utilisez des noms plus longs écrits dans toutes les majuscules et utilisez le trait de soulignement (_) pour séparer les mots.
- **Exemples:** `T` , `V` , `SRC_VERTEX`

Les constantes

Les constantes (champs `static final` dont le contenu est immuable, par règles de langage ou par convention) doivent être nommées avec toutes les majuscules et le trait de soulignement (_) pour

séparer les mots.

- Correspond à l'expression régulière `^[AZ][A-Z0-9]*(_[A-Z0-9]+)*$`
- **Exemples:** `BUFFER_SIZE` , `MAX_LEVEL`

Autres directives sur le nommage

- Évitez de masquer / masquer les méthodes, les variables et les variables de type dans les étendues externes.
- Laisser la verbosité du nom correspondre à la taille de la portée. (Par exemple, utilisez des noms descriptifs pour les champs des grandes classes et des noms abrégés pour les variables locales à courte durée de vie.)
- Lorsque vous nommez des membres statiques publics, laissez l'identificateur être auto-descriptif si vous pensez qu'ils seront importés de manière statique.
- Lectures complémentaires: Section de nommage (dans le Guide de style Java officiel)

Source: Directives de style Java d'Oracle

Fichiers source Java

- Toutes les lignes doivent être terminées par un caractère de saut de ligne (LF, valeur ASCII 10) et non par exemple CR ou CR + LF.
- Il ne peut y avoir aucun espace blanc à la fin d'une ligne.
- Le nom d'un fichier source doit être égal au nom de la classe qu'il contient, suivi de l'extension `.java` , même pour les fichiers ne contenant qu'une classe privée de package. Cela ne s'applique pas aux fichiers ne contenant aucune déclaration de classe, tels que `package-info.java` .

Caractères spéciaux

- Outre LF, le seul espace blanc autorisé est Space (valeur ASCII 32). Notez que cela implique que les autres caractères d'espace blanc (in, par exemple, les littéraux de chaîne et de caractère) doivent être écrits sous une forme échappée.
- `\'` , `\"` , `\\` , `\t` , `\b` , `\r` , `\f` et `\n` devraient être préférés aux caractères octaux correspondants (par exemple `\047`) ou Unicode (par exemple `\u0027`).
- Au cas où il serait nécessaire de respecter les règles ci-dessus à des fins de test, le test devrait *générer* l'entrée requise par programmation.

Déclaration de colis

```
package com.example.my.package;
```

La déclaration de package ne doit pas être encapsulée, qu'elle dépasse ou non la longueur

maximale recommandée d'une ligne.

Déclarations d'importation

```
// 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;
```

- Les déclarations d'importation doivent être triées…
    - … Principalement par non statique / statique avec les importations non statiques en premier.
    - … Secondairement par origine du colis selon l'ordre suivant
        - paquets java
        - paquets javax
        - packages externes (ex: org.xml)
        - packages internes (ex: com.sun)
    - … Tertiaire par colis et identifiant de classe par ordre lexicographique
- Les instructions d'importation ne doivent pas être mises en ligne, même si elles dépassent la longueur maximale recommandée d'une ligne.
- Aucune importation inutilisée ne devrait être présente.

Importations de caractères génériques

- Les importations de caractères génériques ne doivent généralement pas être utilisées.
- Lors de l'importation d'un grand nombre de classes étroitement liées (par exemple, l'implémentation d'un visiteur sur une arborescence comportant des dizaines de classes de «nœuds» distinctes), une importation de caractères génériques peut être utilisée.
- Dans tous les cas, vous ne devez pas utiliser plus d'une importation de caractères génériques par fichier.

Structure de classe

Ordre des membres de la classe

Les membres du groupe doivent être commandés comme suit:

1. Champs (par ordre public, protégé et privé)
2. Constructeurs
3. Méthodes d'usine
4. Autres méthodes (par ordre public, protégé et privé)

Les champs et méthodes de commande principalement par leurs modificateurs d'accès ou leur identifiant ne sont pas requis.

Voici un exemple de cet ordre:

```
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 + "]";
    }

}
```

Groupement des membres du groupe

- Les champs connexes doivent être regroupés.
- Un type imbriqué peut être déclaré juste avant sa première utilisation; sinon il devrait être déclaré avant les champs.
- Les constructeurs et les méthodes surchargées doivent être regroupés par fonctionnalité et classés avec une arité croissante. Cela implique que la délégation entre ces constructions circule vers le bas dans le code.
- Les constructeurs doivent être regroupés sans autres membres entre eux.
- Les variantes surchargées d’une méthode doivent être regroupées sans autres membres.

Modificateurs

```
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();
}
```

- Les modificateurs doivent aller dans l'ordre suivant
    - Modificateur d'accès (`public` / `private` / `protected`)
    - `abstract`
    - `static`
    - `final`
    - `transient`
    - `volatile`
    - `default`
    - `synchronized`
    - `native`
    - `strictfp`
- Les modificateurs ne doivent pas être écrits lorsqu'ils sont implicites. Par exemple, les méthodes d'interface ne doivent ni être déclarées `public` , ni `abstract` , et les énumérations et les interfaces imbriquées ne doivent pas être déclarés statiques.
- Les paramètres de la méthode et les variables locales ne doivent pas être déclarés `final` moins d'améliorer la lisibilité ou de documenter une décision de conception réelle.
- Les champs doivent être déclarés `final` moins qu'il y ait une raison impérieuse de les rendre mutables.

Échancrure

- Le niveau d'indentation est de **quatre espaces** .
- Seuls les caractères d'espacement peuvent être utilisés pour l'indentation. **Pas d'onglets.**
- Les lignes vides ne doivent pas être en retrait. (Ceci est impliqué par la règle de non-fin des espaces blancs.)
- `case` lignes de `case` doivent être séparées par quatre espaces, et les instructions dans la casse doivent comporter quatre espaces supplémentaires.

```
switch (var) {
    case TWO:
        setChoice("two");
        break;
    case THREE:
        setChoice("three");
        break;
    default:
        throw new IllegalArgumentException();
}
```

Reportez-vous aux instructions d'emballage pour obtenir des instructions sur la mise en retrait des lignes de continuation.

Déclarations d'emballage

- Le code source et les commentaires ne doivent généralement pas dépasser 80 caractères par ligne et rarement, voire jamais, 100 caractères par ligne, y compris l'indentation.

La limite de caractères doit être évaluée au cas par cas. Ce qui compte vraiment, c'est la «densité» sémantique et la lisibilité de la ligne. Rendre les lignes trop longues les rend difficiles à lire; De même, faire des «tentatives héroïques» pour les adapter à 80 colonnes peut aussi les rendre difficiles à lire. La flexibilité décrite ici vise à permettre aux développeurs d'éviter ces extrêmes, et non de maximiser l'utilisation du moniteur immobilier.

- Les URL ou les exemples de commandes ne doivent pas être encapsulés.

```
// 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(", "));
```

- L'emballage à un niveau syntaxique supérieur est préférable à l'emballage à un niveau syntaxique inférieur.

- Il devrait y avoir au plus 1 déclaration par ligne.

- Une ligne de continuation doit être indentée de l'une des quatre manières suivantes

    - **Variante 1** : Avec 8 espaces supplémentaires par rapport à l'indentation de la ligne précédente.
    - **Variante 2** : Avec 8 espaces supplémentaires par rapport à la colonne de départ de l'expression enveloppée.
    - **Variante 3** : Alignée avec l'expression fraternelle précédente (tant qu'il est clair qu'il s'agit d'une ligne de continuation)
    - **Variante 4** : Alignée avec l'appel de méthode précédent dans une expression chaînée.

Déclarations de méthode d'emballage

```
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 {
    …
}
```

- Les déclarations de méthode peuvent être formatées en répertoriant les arguments verticalement, ou par une nouvelle ligne et +8 espaces supplémentaires
- Si une clause throws doit être encapsulée, placez le saut de ligne devant la clause throws et assurez-vous qu'elle se démarque de la liste des arguments, soit en indentant +8 par rapport à la déclaration de fonction, soit +8 par rapport à la ligne précédente.

Expressions d'emballage

- Si une ligne se rapproche de la limite maximale de caractères, envisagez toujours de la décomposer en plusieurs instructions / expressions au lieu de la limiter.
- Casser avant les opérateurs.
- Pause avant le. dans les appels de méthode chaînés.

```
popupMsg("Inbox notification: You have "
        + newMsgs + " new messages");

// Don't! Looks like two arguments
popupMsg("Inbox notification: You have " +
         newMsgs + " new messages");
```

Espace blanc

Espaces verticaux

- Une seule ligne blanche doit être utilisée pour séparer…
    - Déclaration de colis
    - Déclarations de classe
    - Constructeurs
    - Les méthodes

    - Initialisateurs statiques
    - Initialiseurs d'instance

- … Et peuvent être utilisés pour séparer des groupes logiques de

    - déclarations d'importation
    - des champs
    - déclarations

- Plusieurs lignes vierges consécutives ne doivent être utilisées que pour séparer des groupes de membres apparentés et non comme l'espacement standard entre membres.

Espaces horizontaux

- Un seul espace doit être utilisé…

    - Pour séparer les mots-clés des crochets et des accolades voisins
    - Avant et après tous les opérateurs binaires et les opérateurs comme les symboles tels que les flèches dans les expressions lambda et les deux points dans les expressions améliorées pour les boucles (mais pas avant les deux points d'une étiquette)
    - Après `//` commence un commentaire.
    - Après les virgules séparant les arguments et les points-virgules séparant les parties d'une boucle for.
    - Après la parenthèse de fermeture d'une distribution.

- Dans les déclarations de variables, il n'est pas recommandé d'aligner les types et les variables.

Déclarations variables

- Une variable par déclaration (et au plus une déclaration par ligne)
- Les crochets des tableaux doivent être du type (`String[] args`) et non de la variable (`String args[]`).
- Déclarez une variable locale juste avant sa première utilisation et initialisez-la le plus près possible de la déclaration.

Annotations

Les annotations de déclaration doivent être placées sur une ligne distincte de la déclaration en cours d'annotation.

```
@SuppressWarnings("unchecked")
public T[] toArray(T[] typeHolder) {
    ...
}
```

Cependant, peu ou pas d'annotations annotant une méthode à une seule ligne peuvent être

placées sur la même ligne que la méthode si elle améliore la lisibilité. Par exemple, on peut écrire:

```
@Nullable String getName() { return name; }
```

Pour des raisons de cohérence et de lisibilité, toutes les annotations doivent être placées sur la même ligne ou chaque annotation doit être placée sur une ligne distincte.

```
// 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) {
    ...
}
```

Expressions lambda

```
Runnable r = () -> System.out.println("Hello World");

Supplier<String> c = () -> "Hello World";

// Collection::contains is a simple unary method and its behavior is
// clear from the context. A method reference is preferred here.
appendFilter(goodStrings::contains);

// A lambda expression is easier to understand than just tempMap::put in this case
trackTemperature((time, temp) -> tempMap.put(time, temp));
```

- Les lambdas d'expression sont préférés aux lambda de bloc à une seule ligne.
- Les références de méthode doivent généralement être préférées aux expressions lambda.
- Pour les références de méthode d'instance liées, ou les méthodes avec une arité supérieure à une, une expression lambda peut être plus facile à comprendre et donc préférable. Surtout si le comportement de la méthode n'est pas clair dans le contexte.
- Les types de paramètres doivent être omis, sauf s'ils améliorent la lisibilité.
- Si une expression lambda s'étend sur plusieurs lignes, envisagez de créer une méthode.

Parenthèses redondantes

```
return flag ? "yes" : "no";

String cmp = (flag1 != flag2) ? "not equal" : "equal";

// Don't do this
return (flag ? "yes" : "no");
```

- Les parenthèses de regroupement redondantes (parenthèses n'affectant pas l'évaluation) peuvent être utilisées si elles améliorent la lisibilité.
- Les parenthèses de regroupement redondantes doivent généralement être omises dans des expressions plus courtes impliquant des opérateurs communs, mais incluses dans des expressions plus longues ou des expressions impliquant des opérateurs dont la priorité et l'associativité ne sont pas claires sans parenthèses. Les expressions ternaires avec des conditions non triviales appartiennent à cette dernière.
- L'expression entière qui suit un mot-clé `return` ne doit pas être entourée de parenthèses.

Littéraux

```
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` littéraux `long` doivent utiliser le suffixe `L` majuscule.
- Les littéraux hexadécimaux doivent utiliser les lettres majuscules `A` - `F`
- Tous les autres préfixes numériques, infixes et suffixes doivent utiliser des lettres minuscules.

Bretelles

```
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");
        }
    }
}
```

- Les accolades doivent être placées à la fin de la ligne en cours plutôt que sur une ligne.

- Il devrait y avoir une nouvelle ligne devant une accolade à moins que le bloc ne soit vide (voir les formulaires abrégés ci-dessous)

- Les accolades sont recommandées même si le langage les rend facultatives, comme les corps de ligne et les boucles en une seule ligne.
    - Si un bloc s'étend sur plusieurs lignes (commentaires compris), il doit comporter des accolades.
    - Si l'un des blocs d'une instruction `if` / `else` a des accolades, l'autre bloc doit également l'être.
    - Si le bloc arrive en dernier dans un bloc englobant, il doit avoir des accolades.
- Le mot-clé `else` , `catch` et the `while` in `do…while` boucles vont sur la même ligne que l'accolade de fermeture du bloc précédent.

Formes courtes

```
enum Response { YES, NO, MAYBE }
public boolean isReference() { return true; }
```

Les recommandations ci-dessus visent à améliorer l'uniformité (et donc à accroître la familiarité et la lisibilité). Dans certains cas, les «formulaires abrégés» qui s'écartent des directives ci-dessus sont tout aussi lisibles et peuvent être utilisés à la place. Ces cas incluent, par exemple, des déclarations d'énumération simples et des méthodes triviales et des expressions lambda.

Lire Code officiel Oracle standard en ligne: https://riptutorial.com/fr/java/topic/2697/code-officiel-oracle-standard

Chapitre 30: Collections

Introduction

Le framework de collections dans `java.util` fournit un certain nombre de classes génériques pour des ensembles de données avec des fonctionnalités qui ne peuvent pas être fournies par des tableaux réguliers.

Le framework de collections contient des interfaces pour `Collection<O>` , avec les sous-interfaces principales `List<O>` et `Set<O>` , ainsi que la collection de mappages `Map<K,V>` . Les collections sont l'interface racine et sont implémentées par de nombreux autres cadres de collecte.

Remarques

Les collections sont des objets pouvant stocker des collections d'autres objets à l'intérieur d'eux. Vous pouvez spécifier le type de données stockées dans une collection à l'aide de Generics .

Les collections utilisent généralement les espaces de noms `java.util` ou `java.util.concurrent` .

Java SE 1.4

Java 1.4.2 et les versions ultérieures ne prennent pas en charge les génériques. En tant que tel, vous ne pouvez pas spécifier les paramètres de type que contient une collection. En plus de ne pas avoir de type de sécurité, vous devez également utiliser des moulages pour récupérer le type correct dans une collection.

Outre `Collection<E>` , il existe plusieurs types principaux de collections, dont certains comportent des sous-types.

- `List<E>` est une collection ordonnée d'objets. Il est similaire à un tableau, mais ne définit pas de limite de taille. La taille des implémentations augmentera généralement en interne pour accueillir de nouveaux éléments.
- `Set<E>` est une collection d'objets qui n'autorise pas les doublons.
    - `SortedSet<E>` est un `Set<E>` qui spécifie également l'ordre des éléments.
- `Map<K,V>` est une collection de paires clé / valeur.
    - `SortedMap<K,V>` est une `Map<K,V>` qui spécifie également l'ordre des éléments.

Java SE 5

Java 5 ajoute un nouveau type de collection:

- `Queue<E>` est un ensemble d'éléments destinés à être traités dans un ordre spécifique. L'implémentation spécifie s'il s'agit de FIFO ou LIFO. Cela rend obsolète la classe `Stack` .

Java SE 6

Java 6 ajoute de nouveaux sous-types de collections.

- `NavigableSet<E>` est un `Set<E>` avec des méthodes de navigation spéciales intégrées.
- `NavigableMap<K,V>` est une `Map<K,V>` avec des méthodes de navigation spéciales intégrées.
- `Deque<E>` est une `Queue<E>` pouvant être lue de chaque côté.

Notez que les éléments ci-dessus sont tous des interfaces. Pour les utiliser, vous devez rechercher les classes d'implémentation appropriées, telles que `ArrayList` , `HashSet` , `HashMap` ou `PriorityQueue` .

Chaque type de collection a plusieurs implémentations qui ont des mesures de performance et des cas d'utilisation différents.

Notez que le principe de substitution Liskov s'applique aux sous-types de collection. En d'autres `SortedSet<E>` , un `SortedSet<E>` peut être transmis à une fonction qui attend un `Set<E>` . Il est également utile de lire les Paramètres liés dans la section Génériques pour plus d'informations sur l'utilisation des collections avec héritage de classe.

Si vous voulez créer vos propres collections, il peut être plus facile d'hériter l'une des classes abstraites (comme `AbstractList`) au lieu d'implémenter l'interface.

Java SE 1.2

Avant la version 1.2, vous deviez utiliser les classes / interfaces suivantes:

- `Vector` au lieu de `ArrayList`
- `Dictionary` au lieu de `Map` . Notez que Dictionary est aussi une classe abstraite plutôt qu'une interface.
- `Hashtable` au lieu de `HashMap`

Ces classes sont obsolètes et ne doivent pas être utilisées dans le code moderne.

Examples

Déclaration d'un ArrayList et ajout d'objets

Nous pouvons créer une `ArrayList` (en suivant l'interface `List`):

```
List aListOfFruits = new ArrayList();
```

Java SE 5

```
List<String> aListOfFruits = new ArrayList<String>();
```

Java SE 7

```
List<String> aListOfFruits = new ArrayList<>();
```

Maintenant, utilisez la méthode `add` pour ajouter une `String` :

```
aListOfFruits.add("Melon");
```

```
aListOfFruits.add("Strawberry");
```

Dans l'exemple ci-dessus, `ArrayList` contiendra la `String` "Melon" à l'index 0 et la `String` "Strawberry" à l'index 1.

On peut aussi ajouter plusieurs éléments avec `addAll(Collection<? extends E> c)`

```
List<String> aListOfFruitsAndVeggies = new ArrayList<String>();
aListOfFruitsAndVeggies.add("Onion");
aListOfFruitsAndVeggies.addAll(aListOfFruits);
```

Maintenant, "Onion" est placé à 0 index dans `aListOfFruitsAndVeggies` , "Melon" est à index 1 et "Strawberry" à index 2.

Construire des collections à partir de données existantes

Collections standard

Framework Java Collections

Un moyen simple de construire une `List` partir de valeurs de données individuelles consiste à utiliser la méthode `java.utils.Arrays` `Arrays.asList` :

```
List<String> data = Arrays.asList("ab", "bc", "cd", "ab", "bc", "cd");
```

Toutes les implémentations de collection standard fournissent des constructeurs qui prennent une autre collection comme argument en ajoutant tous les éléments à la nouvelle collection au moment de la construction:

```
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
```

Cadre des collections Google Goyave

Un autre excellent `Google Guava` est `Google Guava` , une classe d'utilitaires étonnante (fournissant des méthodes statiques pratiques) pour la construction de différents types de collections standard: `Lists` et `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");
```

Collections de cartographie

Framework Java Collections

De même pour les cartes, si une `Map<String, Object> map` une nouvelle carte peut être construite avec tous les éléments comme suit:

```
Map<String, Object> map1 = new HashMap<>(map);
SortedMap<String, Object> map2 = new TreeMap<>(map);
```

Cadre Apache Commons Collections

En utilisant `Apache Commons` vous pouvez créer Map en utilisant array dans `ArrayUtils.toMap` ainsi que `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"}});
```

Chaque élément du tableau doit être un Map.Entry ou un Array, contenant au moins deux éléments, le premier élément étant utilisé comme clé et le second comme valeur.

Cadre des collections Google Goyave

La classe d'utilitaires du framework `Google Guava` est nommée `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

Utiliser `Stream` ,

```
Stream.of("xyz", "abc").collect(Collectors.toList());
```

ou

```
Arrays.stream("xyz", "abc").collect(Collectors.toList());
```

Joindre des listes

Les méthodes suivantes peuvent être utilisées pour joindre des listes sans modifier les listes de sources.

Première approche. A plus de lignes mais facile à comprendre

```
List<String> newList = new ArrayList<String>();
newList.addAll(listOne);
newList.addAll(listTwo);
```

Deuxième approche A une ligne de moins mais moins lisible.

```
List<String> newList = new ArrayList<String>(listOne);
newList.addAll(listTwo);
```

Troisième approche Nécessite une bibliothèque tierce de collections de documents Apache .

```
ListUtils.union(listOne,listTwo);
```

Java SE 8

L'utilisation de Streams peut être réalisée de la même manière par

```
List<String> newList = Stream.concat(listOne.stream(),
listTwo.stream()).collect(Collectors.toList());
```

Les références. Liste d'interfaces

Supprimer des éléments d'une liste dans une boucle

Il est difficile de supprimer des éléments d'une liste dans une boucle, ceci étant dû au fait que l'index et la longueur de la liste sont modifiés.

Compte tenu de la liste suivante, voici quelques exemples qui donneront un résultat inattendu et d'autres qui donneront le résultat correct.

```
List<String> fruits = new ArrayList<String>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Strawberry");
```

INCORRECT

Suppression en itération de `for` la déclaration *Ignore « Banana »:*

L'échantillon de code n'imprimera que `Apple` et `Strawberry` . `Banana` est ignorée , car il se déplace à l' index `0` une fois d' `Apple` est supprimé, mais en même temps `i` s'incrémenté à `1` .

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

Retrait dans l'instruction améliorée `for` *Throws Exception:*

En raison de l'itération sur la collecte et la modification en même temps.

> Jette: java.util.ConcurrentModificationException

```
for (String fruit : fruits) {
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruits.remove(fruit);
    }
}
```

CORRECT

Retrait en boucle en utilisant 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'interface `Iterator` a une méthode `remove()` intégrée uniquement pour ce cas. Cependant, cette méthode est marquée comme "facultative" dans la documentation et peut générer une `UnsupportedOperationException` .

> Lève: une exception UnsupportedOperationException - si l'opération de suppression

n'est pas prise en charge par cet itérateur

Par conséquent, il est conseillé de vérifier la documentation pour vous assurer que cette opération est prise en charge (en pratique, à moins que la collection ne soit immuable via une bibliothèque tierce ou l'utilisation de la méthode `Collections.unmodifiable...()` . l'opération est presque toujours supportée).

Lors de l'utilisation d'un `Iterator` une `Iterator` `ConcurrentModificationException` est `modCount` lorsque le `modCount` de la `List` est modifié depuis la création de l' `Iterator` . Cela aurait pu se produire dans le même thread ou dans une application multi-thread partageant la même liste.

Un `modCount` est une variable `int` qui compte le nombre de fois où cette liste a été structurellement modifiée. Une modification structurelle signifie essentiellement une opération `add()` ou `remove()` appelée sur l'objet `Collection` (les modifications apportées par `Iterator` ne sont pas comptabilisées). Lorsque l' `Iterator` est créé, il stocke ce `modCount` et à chaque itération de la `List` , si le `modCount` actuel est identique à celui où l' `Iterator` été créé. S'il y a un changement dans la `modCount` valeur qu'il jette un `ConcurrentModificationException` .

Par conséquent, pour la liste ci-dessus, une opération comme ci-dessous ne lancera aucune exception:

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.set(0, "Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());
}
```

Mais ajouter un nouvel élément à la `List` après l'initialisation d'un `Iterator` une `Iterator` `ConcurrentModificationException` :

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.add("Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());    //ConcurrentModificationException here
}
```

Itérer en arrière

```
for (int i = (fruits.size() - 1); i >=0; i--) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

Cela ne saute rien L'inconvénient de cette approche est que la sortie est inverse. Cependant, dans la plupart des cas, vous supprimez les éléments qui ne comptent pas. Vous ne devriez jamais faire cela avec `LinkedList` .

Itérer en avant, ajuster l'indice de boucle

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
         i--;
    }
}
```

Cela ne saute rien. Lorsque le `i` ième élément est retiré de la `List` , l'élément placé à l' origine à l' indice `i+1` devient le nouveau `i` ème élément. Par conséquent, la boucle peut décrémenter `i` pour que la prochaine itération traite l'élément suivant sans sauter.

Utiliser une liste "à supprimer"

```
ArrayList shouldBeRemoved = new ArrayList();
for (String str : currentArrayList) {
    if (condition) {
        shouldBeRemoved.add(str);
    }
}
currentArrayList.removeAll(shouldBeRemoved);
```

Cette solution permet au développeur de vérifier si les éléments corrects sont supprimés de manière plus propre.

Java SE 8

Dans Java 8, les alternatives suivantes sont possibles. Celles-ci sont plus propres et plus simples si le retrait ne doit pas nécessairement se faire en boucle.

Filtrage d'un flux

Une `List` peut être diffusée et filtrée. Un filtre approprié peut être utilisé pour supprimer tous les éléments indésirables.

```
List<String> filteredList =
    fruits.stream().filter(p -> !"Apple".equals(p)).collect(Collectors.toList());
```

Notez que, contrairement à tous les autres exemples, cet exemple produit une nouvelle instance de `List` et conserve la `List` origine inchangée.

Utiliser `removeIf`

Enregistre la surcharge de la construction d'un flux si tout ce qui est nécessaire est de supprimer un ensemble d'éléments.

```
fruits.removeIf(p -> "Apple".equals(p));
```

Collection non modifiable

Parfois, il n’est pas une bonne pratique d’exposer une collection interne car cela peut conduire à une vulnérabilité de code malveillant en raison de sa caractéristique mutable. Afin de fournir des collections "en lecture seule", java fournit ses versions non modifiables.

Une collection non modifiable est souvent une copie d'une collection modifiable qui garantit que la collection elle-même ne peut pas être modifiée. Les tentatives de modification entraîneront une exception UnsupportedOperationException.

Il est important de noter que les objets présents dans la collection peuvent toujours être modifiés.

```
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);
    }
}
```

La tentative suivante de modifier une collection non modifiable lancera une exception:

```
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);
    }
}
```

sortie:

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
    at App.main(App.java:12)
```

Itération sur les collections

Itérer sur la liste

```
List<String> names  = new ArrayList<>(Arrays.asList("Clementine", "Duran", "Mike"));
```

Java SE 8

```
names.forEach(System.out::println);
```

Si nous avons besoin du parallélisme

```
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());
}
```

Itération sur 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());
}
```

Itérer sur la carte

```
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());
}
```

Collections vides immuables

Parfois, il est approprié d'utiliser une collection vide immuable. La classe `Collections` fournit des méthodes pour obtenir de telles collections de manière efficace:

```
List<String> anEmptyList = Collections.emptyList();
Map<Integer, Date> anEmptyMap = Collections.emptyMap();
Set<Number> anEmptySet = Collections.emptySet();
```

Ces méthodes sont génériques et convertissent automatiquement la collection renvoyée au type auquel elle est affectée. C'est-à-dire qu'une invocation de par exemple `emptyList()` peut être affectée à n'importe quel type de `List` , de même que `emptySet()` et `emptyMap()` .

Les collections renvoyées par ces méthodes sont immuables dans la mesure où elles lancent une `UnsupportedOperationException` si vous tentez d'appeler des méthodes qui modifieraient leur contenu (`add` , `put` , etc.). Ces collections sont principalement utiles en tant que substituts de résultats de méthode vides ou d'autres valeurs par défaut, au lieu d'utiliser `null` ou de créer des objets avec `new` .

Collections et valeurs primitives

Les collections en Java ne fonctionnent que pour les objets. Ie il n'y a pas de `Map<int, int>` en Java. Au lieu de cela, les valeurs primitives doivent être *placées* dans des objets, comme dans `Map<Integer, Integer>` . L’auto-boxing Java permettra une utilisation transparente de ces collections:

```
Map<Integer, Integer> map = new HashMap<>();
map.put(1, 17); // Automatic boxing of int to Integer objects
int a = map.get(1); // Automatic unboxing.
```

Malheureusement, les frais généraux sont *considérables* . Un `HashMap<Integer, Integer>` nécessitera environ 72 octets par entrée (par exemple, sur une JVM 64 bits avec des pointeurs compressés et en supposant des entiers supérieurs à 256 et en supposant une charge de 50% de la map). Étant donné que les données réelles ne sont que de 8 octets, cela génère une surcharge considérable. De plus, il nécessite deux niveaux d’indirection (Map -> Entry -> Value) qui sont inutilement lents.

Il existe plusieurs bibliothèques avec des collections optimisées pour les types de données primitifs (qui nécessitent seulement ~ 16 octets par entrée à une charge de 50%, soit quatre fois moins de mémoire et un niveau d’indirection inférieur), valeurs en Java.

Suppression des éléments correspondants des listes à l'aide de l'itérateur.

Au-dessus, j'ai remarqué un exemple pour supprimer des éléments d'une liste dans une boucle et j'ai pensé à un autre exemple qui pourrait s'avérer utile cette fois-ci en utilisant l'interface `Iterator` .

Ceci est une démonstration d'une astuce qui peut s'avérer utile pour traiter les éléments en double dans les listes dont vous souhaitez vous débarrasser.

Remarque: Ceci ne fait qu'ajouter aux **éléments Suppression d'une liste dans un** exemple de

boucle :

Définissons donc nos listes comme d'habitude

```
    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));
```

La méthode suivante utilise deux objets Collection et effectue la magie de la suppression des éléments de notre `removeNameList` qui correspondent aux éléments de `nameList` .

```
private static void removeNames(Collection<String> collection1, Collection<String>
collection2) {
      //get Iterator.
    Iterator<String> iterator = collection1.iterator();

    //Loop while collection has items
    while(iterator.hasNext()){
        if (collection2.contains(iterator.next()))
            iterator.remove(); //remove the current Name or Item
    }
}
```

Appel de la méthode et passage de la liste de `nameList` et de la `removeNameList` comme suit
`removeNames(nameList,removeNameList);`
Produira la sortie suivante:

> Liste des tableaux avant de supprimer les noms: **James Smith Sonny Huckle Berry Finn Allan**
> Liste de matrices après suppression des noms: **James Smith Finn Allan**

Une utilisation simple et ordonnée pour les collections qui peuvent être utiles pour supprimer des éléments répétés dans les listes.

Créer votre propre structure Iterable à utiliser avec Iterator ou pour chaque boucle.

Pour que notre collection puisse être itérée en utilisant un itérateur ou pour chaque boucle, nous devons prendre en charge les étapes suivantes:

1. Les éléments sur lesquels nous voulons effectuer une itération doivent être `Iterable` et exposer l' `iterator()` .
2. Concevez un `java.util.Iterator` en `hasNext()` , `next()` et `remove()` .

J'ai ajouté une implémentation de liste chaînée générique simple ci-dessous qui utilise les entités ci-dessus pour rendre la liste chaînée itérable.

```
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 + " ");
        }
    }
}
```

Sortie

```
using Iterator:
1 2 4 3
using for-each:
1 2 4 3
```

Cela fonctionnera dans Java 7+. Vous pouvez le faire fonctionner sur Java 5 et Java 6 également en remplaçant:

```
LinkedList<Integer> list = new LinkedList<>(1);
```

avec

```
LinkedList<Integer> list = new LinkedList<Integer>(1);
```

ou juste toute autre version en incorporant les modifications compatibles.

Piège: exceptions de modification concurrentes

Cette exception se produit lorsqu'une collection est modifiée lors d'une itération utilisant des méthodes autres que celles fournies par l'objet itérateur. Par exemple, nous avons une liste de chapeaux et nous voulons supprimer tous ceux qui ont des oreillettes:

```
List<IHat> hats = new ArrayList<>();
hats.add(new Ushanka()); // that one has ear flaps
hats.add(new Fedora());
```

```
hats.add(new Sombrero());
for (IHat hat : hats) {
    if (hat.hasEarFlaps()) {
        hats.remove(hat);
    }
}
```

Si nous exécutons ce code, ***ConcurrentModificationException*** sera levée car le code modifie la collection tout en l'itérant. La même exception peut se produire si l'un des threads multiples travaillant avec la même liste tente de modifier la collection alors que d'autres la parcourent. La modification simultanée de collections dans plusieurs threads est une chose naturelle, mais devrait être traitée avec les outils habituels de la boîte à outils de programmation simultanée tels que les verrous de synchronisation, les collections spéciales adoptées pour la modification simultanée, la modification de la collection clonée, etc.

Sous collections

List subList (int fromIndex, int toIndex)

Ici, d'index est inclusif et toIndex est exclusif.

```
List list = new ArrayList();
List list1 = list.subList(fromIndex,toIndex);
```

1. Si la liste n'existe pas dans la plage de valeurs, elle génère une exception IndexOutofBoundException.
2. Les modifications apportées à la liste1 auront un impact sur les mêmes modifications de la liste. Ce sont les collections sauvegardées.
3. Si le fromnIndex est supérieur à toIndex (fromIndex> toIndex), il génère une exception IllegalArgumentException.

Exemple:

```
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);
```

Sortie:
Avant la sous-liste [Hello1, Hello2]
Après les changements de sous-liste [Hello1, Hello3, Hello2]

Définir le sous-ensemble (fromIndex, toIndex)

Ici, d'index est inclusif et toIndex est exclusif.

```
Set set = new TreeSet();
Set set1 = set.subSet(fromIndex,toIndex);
```

L'ensemble renvoyé lancera une exception IllegalArgumentException lors d'une tentative d'insertion d'un élément en dehors de sa plage.

Map subMap (fromKey, toKey)

fromKey est inclusif et toKey est exclusif

```
Map map = new TreeMap();
Map map1 = map.get(fromKey,toKey);
```

Si fromKey est supérieur à toKey ou si cette carte elle-même a une plage restreinte, et fromKey ou toKey se situe en dehors des limites de la plage, elle lève une exception IllegalArgumentException.

Toutes les collections prenant en charge les collections sauvegardées signifient que les modifications apportées à la sous-collection auront les mêmes modifications sur la collection principale.

Lire Collections en ligne: https://riptutorial.com/fr/java/topic/90/collections

Chapitre 31: Collections alternatives

Remarques

Ce sujet sur les collections Java à partir de guava, apache, eclipse: Multiset, Bag, Multimap, utils fonctionne à partir de cette lib, etc.

Examples

Apache HashBag, Guava HashMultiset et Eclipse HashBag

Un bag / ultiset stocke chaque objet de la collection avec un nombre d'occurrences. Des méthodes supplémentaires sur l'interface permettent à plusieurs copies d'un objet d'être ajoutées ou supprimées à la fois. JDK analog est HashMap <T, Integer>, lorsque les valeurs sont le nombre de copies de cette clé.

Type	Goyave	Collections Apache Commons	Collections GS	JDK
Ordre non défini	HashMultiset	HashBag	HashBag	HashM
Trié	TreeMultiset	TreeBag	TreeBag	TreeMa
Ordre d'insertion	LinkedHashMultiset	-	-	LinkedH
Variante concurrente	ConcurrentHashMultiset	SynchronizedBag	SynchronizedBag	Collect
Concurrentes et triées	-	SynchronizedSortedBag	SynchronizedSortedBag	Collect
Collection immuable	ImmutableMultiset	UnmodifiableBag	UnmodifiableBag	Collect
Immuable et trié	ImmutableSortedMultiset	UnmodifiableSortedBag	UnmodifiableSortedBag	Collect)

Exemples :

1. Utilisation de SynchronizedSortedBag d'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. En utilisant TreeBag d'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. Utilisation de LinkedHashMultiset depuis 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
```

Plus d'exemples:

I. Collection Apache:

1. HashBag - ordre non défini
2. SynchronizedBag - concurrent et ordre non défini
3. SynchronizedSortedBag - - Ordre simultané et trié
4. TreeBag - ordre trié

II. Collection GS / Eclipse

5. MutableBag - commande non définie
6. MutableSortedBag - ordre trié

III. Goyave

7. HashMultiset - ordre non défini
8. TreeMultiset - ordre trié
9. LinkedHashMultiset - ordre d'insertion
10. ConcurrentHashMultiset - concurrent et ordre non défini

Multimap dans les collections Goyave, Apache et Eclipse

Ce multimap permet des paires clé-valeur en double. Les analogues JDK sont HashMap <K, List>, HashMap <K, Set>, etc.

Commande de Key	Commande de valeur	Dupliquer	Clé analogique	Valeur analogique	Goyave	A
non défini	Ordre d'insertion	Oui	HashMap	ArrayList	ArrayListMultimap	Mu
non défini	non défini	non	HashMap	HashSet	HashMultimap	Mu mu Ha Ha
non défini	trié	non	HashMap	TreeSet	Multimaps. newMultimap(HashMap, Supplier <TreeSet>)	Mu ne Tr
Ordre d'insertion	Ordre d'insertion	Oui	LinkedHashMap	ArrayList	LinkedListMultimap	M m Li ()
Ordre d'insertion	Ordre d'insertion	non	LinkedHashMap	LinkedHashSet	LinkedHashMultimap	Mu mu Li Li
trié	trié	non	TreeMap	TreeSet	TreeMultimap	Mu mu Tr Se

Exemples utilisant Multimap

Tâche : Analyser "Bonjour tout le monde! Bonjour à tous! Bonjour à tous!" chaîne pour séparer les mots et imprimer tous les index de chaque mot en utilisant MultiMap (par exemple, Hello = [0, 2], World! = [1, 5], etc.)

1. MultiValueMap d'Apache

```
    String INPUT_TEXT = "Hello World! Hello All! Hi World!";
    // Parse text to words and index
    List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
    // Create Multimap
    MultiMap<String, Integer> multiMap = new MultiValueMap<String, Integer>();


    // Fill Multimap
    int i = 0;
    for(String word: words) {
        multiMap.put(word, i);
        i++;
    }

    // Print all words
    System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} -
in random orders
    // Print all unique words
    System.out.println(multiMap.keySet());    // print [Hi, Hello, World!, All!] - in random
```

```
orders

    // Print all indexes
    System.out.println("Hello = " + multiMap.get("Hello"));    // print [0, 2]
    System.out.println("World = " + multiMap.get("World!"));    // print [1, 5]
    System.out.println("All = " + multiMap.get("All!"));    // print [3]
    System.out.println("Hi = " + multiMap.get("Hi"));    // print [4]
    System.out.println("Empty = " + multiMap.get("Empty"));    // print null

    // Print count unique words
    System.out.println(multiMap.keySet().size());    //print 4
```

2. HashBiMap de la collection 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 de goyave

```
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
```

Nore exemples:

I. Collection Apache:

1. MultiValueMap
2. MultiValueMapLinked
3. MultiValueMapTree

II. Collection GS / Eclipse

1. FastListMultimap
2. HashBagMultimap
3. TreeSortedSetMultimap
4. UnifiedSetMultimap

III. Goyave

1. HashMultiMap
2. LinkedHashMultimap
3. LinkedListMultimap
4. TreeMultimap
5. ArrayListMultimap

Comparer les opérations avec les collections - Créer des collections

Comparer les opérations avec les collections - Créer des collections

1. Créer une liste

La description	JDK	goyave	gs-collections
Créer une liste vide	`new ArrayList<> ()`	`Lists.newArrayList()`	`FastList.newList()`
Créer une liste à partir de valeurs	`Arrays.asList("1", "2", "3")`	`Lists.newArrayList("1", "2", "3")`	`FastList.newListWith("1", "2", "3")`
Créer une liste avec une capacité = 100	`new ArrayList<>(100)`	`Lists.newArrayListWithCapacity(100)`	`FastList.newList(100)`
Créer une liste à partir de n'importe quelle collection	`new ArrayList<>(collection)`	`Lists.newArrayList(collection)`	`FastList.newList(collecti`
Créer une liste à partir de tout Iterable	-	`Lists.newArrayList(iterable)`	`FastList.newList(iterable`
Créer une liste depuis Iterator	-	`Lists.newArrayList(iterator)`	-
Créer une liste à partir d'un tableau	`Arrays.asList(array)`	`Lists.newArrayList(array)`	`FastList.newListWith(arra`
Créer une liste en utilisant l'usine	-	-	`FastList.newWithNValues(1 () -> "1")`

Exemples:

```
    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 Créer un ensemble

La description	JDK	goyave	gs-collections
Créer un ensemble vide	`new HashSet<>()`	`Sets.newHashSet()`	`UnifiedSet.newSet()`
Créatrice définie à partir de valeurs	`new HashSet<>(Arrays.asList("alpha", "beta", "gamma"))`	`Sets.newHashSet("alpha", "beta", "gamma")`	`UnifiedSet.newSetWith("a "beta", "gamma")`
Créer un ensemble à partir de n'importe quelle collection	`new HashSet<>(collection)`	`Sets.newHashSet(collection)`	`UnifiedSet.newSet(collec`
Créer un ensemble à partir de n'importe quelle Iterable	-	`Sets.newHashSet(iterable)`	`UnifiedSet.newSet(iterab`
Créer un ensemble à partir de n'importe quel itérateur	-	`Sets.newHashSet(iterator)`	-
Créer un ensemble depuis Array	`new HashSet<>(Arrays.asList(array))`	`Sets.newHashSet(array)`	`UnifiedSet.newSetWith(ar`

Exemples:

```
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 Créer une carte

La description	JDK	goyave	gs-collections
Créer une carte vide	`new HashMap<>()`	`Maps.newHashMap()`	`UnifiedMap.newMap()`
Créer une carte avec une capacité = 130	`new HashMap<>(130)`	`Maps.newHashMapWithExpectedSize(100)`	`UnifiedMap.newMap(130)`
Créer une	`new HashMap<>(map)`	`Maps.newHashMap(map)`	`UnifiedMap.newMap(map)`

La description	JDK	goyave	gs-collections
carte à partir d'une autre carte			
Créer une carte à partir des clés	-	-	`UnifiedMap.newWithKeysValues("1", "a", "2", "b")`

Exemples:

```
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");
```

Plus d'exemples: CreateCollectionTest

1. CollectionCompare
2. CollectionSearch
3. JavaTransform

Lire Collections alternatives en ligne: https://riptutorial.com/fr/java/topic/2958/collections-alternatives

Chapitre 32: Collections concurrentes

Introduction

Une *collection concurrente* est une [collection] [1] qui permet l'accès par plusieurs threads en même temps. Différents threads peuvent généralement parcourir le contenu de la collection et ajouter ou supprimer des éléments. La collection est chargée de veiller à ce que la collection ne soit pas corrompue. [1]: http://stackoverflow.com/documentation/java/90/collections#t=20161222193649729848 4

Examples

Collections à filetage sécurisé

Par défaut, les différents types de collections ne sont pas compatibles avec les threads.

Cependant, il est assez facile de rendre une collection thread-safe.

```
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>());
```

Lorsque vous créez une collection thread-safe, vous ne devez jamais y accéder via la collection d'origine, uniquement via l'encapsuleur thread-safe.

Java SE 5

À partir de Java 5, `java.util.collections` possède plusieurs nouvelles collections sécurisées pour les threads qui n'ont pas besoin des diverses méthodes `Collections.synchronized` .

```
List<String> threadSafeList = new CopyOnWriteArrayList<String>();
Set<String> threadSafeSet = new ConcurrentHashSet<String>();
Map<String, String> threadSafeMap = new ConcurrentHashMap<String, String>();
```

Collections concurrentes

Les collections simultanées sont une généralisation des collections thread-safe, ce qui permet une utilisation plus large dans un environnement concurrent.

Bien que les collections thread-safe disposent d'un ajout ou d'une suppression d'éléments sûrs dans plusieurs threads, elles n'ont pas nécessairement une itération sécurisée dans le même contexte (il est possible de ne pas itérer en toute sécurité dans un thread, tandis qu'un autre le modifie en ajoutant / enlever des éléments).

C'est là que les collections concurrentes sont utilisées.

Comme l'itération est souvent l'implémentation de base de plusieurs méthodes en vrac dans des collections, comme `addAll` , `removeAll` ou également la copie de collection (via un constructeur ou un autre moyen), le tri ...

Par exemple, le fichier Java SE 5 `java.util.concurrent.CopyOnWriteArrayList` est une implémentation `Lis` t sécurisée et simultanée de threads, dont le javadoc stipule:

> La méthode d'itérateur de style "instantané" utilise une référence à l'état du tableau au moment où l'itérateur a été créé. Ce tableau ne change jamais pendant la durée de vie de l'itérateur, de sorte que l'interférence est impossible et que l'itérateur est assuré de ne pas lancer l'exception ConcurrentModificationException.

Par conséquent, le code suivant est sécurisé:

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

`ConcurrentLinkedQueue` , qui indique:

> Les itérateurs sont faiblement cohérents, renvoyant des éléments reflétant l'état de la file d'attente à un moment donné ou depuis la création de l'itérateur. Ils ne lancent pas java.util.ConcurrentModificationException et peuvent continuer simultanément avec d'autres opérations. Les éléments contenus dans la file d'attente depuis la création de l'itérateur seront renvoyés exactement une fois.

On devrait vérifier les javadocs pour voir si une collection est concurrente ou non. Les attributs de l'itérateur renvoyés par la méthode `iterator()` ("échec rapide", "peu cohérent", ...) constituent l'attribut le plus important à rechercher.

Thread safe mais exemples non concurrents

Dans le code ci-dessus, changer la déclaration `LIST` en

```
public static final List<Integer> LIST = Collections.synchronizedList(new ArrayList<>());
```

Pourrait (et statistiquement sera sur la plupart des architectures multi-CPU / core modernes) conduire à des exceptions.

Les collections synchronisées à partir des méthodes de l'utilitaire `Collections` sont sécurisées pour l'ajout / suppression d'éléments, mais pas pour l'itération (à moins que la collection sous-jacente ne lui soit déjà transmise).

Insertion dans 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.
       }
    }
```

```
}
```

Lire Collections concurrentes en ligne: https://riptutorial.com/fr/java/topic/8363/collections-concurrentes

Chapitre 33: Commandes d'exécution

Examples

Ajout de crochets d'arrêt

Parfois, vous avez besoin d'un morceau de code à exécuter lorsque le programme s'arrête, par exemple en libérant des ressources système que vous ouvrez. Vous pouvez exécuter un thread lorsque le programme s'arrête avec la méthode `addShutdownHook` :

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    ImportantStuff.someImportantIOStream.close();
}));
```

Lire Commandes d'exécution en ligne: https://riptutorial.com/fr/java/topic/7304/commandes-d-execution

Chapitre 34: Comparable et Comparateur

Syntaxe

- la classe publique MyClass implémente Comparable `<MyClass>`
- classe publique MyComparator implémente Comparator `<SomeOtherClass>`
- public int compareTo (MyClass autre)
- public int compare (SomeOtherClass o1, SomeOtherClass o2)

Remarques

Lorsque vous implémentez une `compareTo(..)` qui dépend d'un `double` , **ne procédez pas** comme suit:

```
public int comareTo(MyClass other) {
    return (int)(doubleField - other.doubleField); //THIS IS BAD
}
```

La troncature provoquée par le `(int)` coulée provoque parfois la méthode à retourner de façon incorrecte `0` au lieu d'un nombre positif ou négatif, et peut ainsi conduire à des comparaisons et des bogues de tri.

Au lieu de cela, l'implémentation correcte la plus simple consiste à utiliser Double.compare , en tant que tel:

```
public int comareTo(MyClass other) {
    return Double.compare(doubleField,other.doubleField); //THIS IS GOOD
}
```

Une version non générique de `Comparable<T>` , simplement `Comparable` , existe depuis Java 1.2 . Outre l'interfaçage avec le code existant, il est toujours préférable d'implémenter la version générique `Comparable<T>` , car elle ne nécessite pas de conversion par comparaison.

Il est très courant qu'une classe soit comparable à elle-même, comme dans:

```
public class A implements Comparable<A>
```

Bien qu'il soit possible de rompre avec ce paradigme, soyez prudent lorsque vous le faites.

Un `Comparator<T>` peut toujours être utilisé sur des instances d'une classe si cette classe implémente `Comparable<T>` . Dans ce cas, la logique du `Comparator` sera utilisée; l'ordre naturel spécifié par l'implémentation `Comparable` sera ignoré.

Examples

Trier une liste en utilisant Comparable ou un comparateur

Supposons que nous travaillons sur une classe représentant une personne par son nom et son prénom. Nous avons créé une classe de base pour ce faire et implémenté des méthodes `equals` et `hashCode` appropriées.

```
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);
    }
}
```

Maintenant, nous aimerions trier une liste d'objets `Person` par leur nom, comme dans le scénario suivant:

```
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.
}
```

Malheureusement, comme indiqué ci-dessus, ce qui précède ne sera pas compilé.

`Collections.sort(..)` sait seulement trier une liste si les éléments de cette liste sont comparables ou si une méthode de comparaison personnalisée est fournie.

Si on vous demandait de trier la liste suivante: `1,3,5,4,2` , vous n'auriez aucun problème à dire que la réponse est `1,2,3,4,5` . Cela est dû au fait que les entiers (à la fois en Java et en mathématiques) ont un *classement naturel* , un classement de base de comparaison standard par défaut. Pour donner à notre classe Person un ordre naturel, nous implémentons `Comparable<Person>` , qui nécessite l'implémentation de la méthode `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);
        }
    }
}
```

Maintenant, la méthode principale donnée fonctionnera correctement

```
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
}
```

Si, toutefois, vous ne souhaitez pas ou ne pouvez pas modifier la classe `Person` , vous pouvez fournir un `Comparator<T>` personnalisé `Comparator<T>` qui gère la comparaison de deux objets `Person` . Si on vous demandait de trier la liste suivante: `circle, square, rectangle, triangle, hexagon` vous ne le pouviez pas, mais si vous deviez trier cette liste en *fonction du nombre de coins* , vous pourriez le faire. De même, fournir un comparateur indique à Java comment comparer deux objets normalement non comparables.

```
public class PersonComparator implements Comparator<Person> {

    public int compare(Person p1, Person p2) {
        // If p1's lastName and p2's lastName are not comparably equivalent,
        // Compare p1 to p2 by comparing their last names.
        // Otherwise, compare p1 to p2 by comparing their first names
        if (p1.getLastName().compareTo(p2.getLastName()) != 0) {
            return p1.getLastName().compareTo(p2.getLastName());
        } else {
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
}

//Assume the first version of Person (that does not implement Comparable) is used here
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                        new Person("Bob", "Dole"),
                                        new Person("Ronald", "McDonald"),
                                        new Person("Alice", "McDonald"),
                                        new Person("Jill", "Doe"));
    Collections.sort(people); //Illegal, Person doesn't implement Comparable.
    Collections.sort(people, new PersonComparator()); //Legal

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

Les comparateurs peuvent également être créés / utilisés en tant que classe interne anonyme

```
//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

Comparateurs basés sur l'expression lambda

A partir de Java 8, les comparateurs peuvent également être exprimés en expressions lambda

```
    //Lambda
    Collections.sort(people, (p1, p2) -> { //Legal
        //Method code....
    });
```

Méthodes par défaut du comparateur

De plus, il existe des méthodes par défaut intéressantes sur l'interface Comparator pour construire des comparateurs: les éléments suivants construisent un comparateur comparant `lastName` et `firstName` .

```
Collections.sort(people, Comparator.comparing(Person::getLastName)
                                .thenComparing(Person::getFirstName));
```

Inverser l'ordre d'un comparateur

Tout comparateur peut également être facilement inversé à l'aide de la méthode `reversedMethod` qui changera l'ordre croissant en ordre décroissant.

Les méthodes de comparaison et de comparaison

L'interface `Comparable<T>` nécessite une méthode:

```
public interface Comparable<T> {

    public int compareTo(T other);

}
```

Et l'interface `Comparator<T>` nécessite une méthode:

```
public interface Comparator<T> {

    public int compare(T t1, T t2);

}
```

Ces deux méthodes font essentiellement la même chose, avec une différence mineure: `compareTo` compare `this` à l' `other` , alors que `compare` compare `t1` à `t2` , ne pas se soucier du tout `this` .

Outre cette différence, les deux méthodes ont des exigences similaires. Spécifiquement (pour compareTo), compare cet objet avec l'objet spécifié pour l'ordre. Renvoie un entier négatif, zéro ou un entier positif car cet objet est inférieur, égal ou supérieur à l'objet spécifié. Ainsi, pour la comparaison de `a` et `b` :

- Si `a < b` , `a.compareTo(b)` et `compare(a,b)` doivent renvoyer un entier négatif, et que `b.compareTo(a)` et `compare(b,a)` doivent retourner un entier positif
- Si `a > b` , `a.compareTo(b)` et `compare(a,b)` devraient renvoyer un entier positif, et `b.compareTo(a)` et `compare(b,a)` devraient retourner un entier négatif
- Si `a` égale à `b` pour la comparaison, toutes les comparaisons doivent renvoyer `0` .

Tri naturel (comparable) vs explicite (comparateur)

Il existe deux méthodes `Collections.sort()` :

- Celui qui prend `List<T>` comme paramètre où `T` doit implémenter Comparable et remplacer la méthode `compareTo()` qui détermine l'ordre de tri.
- L'un qui prend les listes et les comparateurs comme arguments, où le comparateur détermine l'ordre de tri.

Tout d'abord, voici une classe Person qui implémente 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();
    }

}
```

Voici comment utiliser la classe ci-dessus pour trier une liste dans l'ordre naturel de ses éléments, définie par la méthode `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);
```

Voici comment vous utiliseriez un comparateur en ligne anonyme pour trier une liste qui n'implémente pas Comparable ou, dans ce cas, pour trier une liste dans un ordre autre que le classement naturel:

```
            //-- 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);
```

Tri des entrées de la carte

A partir de Java 8, il existe des méthodes par défaut sur l'interface `Map.Entry` pour permettre le tri des itérations de carte.

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
```

Bien sûr, ceux-ci peuvent également être utilisés en dehors du flux api:

Java SE 8

```
List<Map.Entry<String, Integer>> entries = new ArrayList<>(numberOfEmployees.entrySet());
Collections.sort(entries, Map.Entry.comparingByValue());
```

Création d'un comparateur à l'aide de la méthode de comparaison

```
Comparator.comparing(Person::getName)
```

Cela crée un comparateur pour la classe `Person` qui utilise ce nom de personne comme source de comparaison. Il est également possible d'utiliser la version de méthode pour comparer long, int et double. Par exemple:

```
Comparator.comparingInt(Person::getAge)
```

Ordre inversé

Pour créer un comparateur qui impose la méthode d'inversion en ordre `reversed()` :

```
Comparator.comparing(Person::getName).reversed()
```

Chaîne de comparateurs

```
Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstName)
```

Cela va créer un comparateur comparé au nom de famille avec le prénom. Vous pouvez enchaîner autant de comparateurs que vous le souhaitez.

Lire Comparable et Comparateur en ligne: https://riptutorial.com/fr/java/topic/3137/comparable-et-comparateur

Chapitre 35: Comparaison C ++

Introduction

Java et C ++ sont des langages similaires. Cette rubrique sert de guide de référence rapide pour les ingénieurs Java et C ++.

Remarques

Classes définies dans d'autres constructions

Défini dans une autre classe

C ++

Classe imbriquée [ref] (nécessite une référence à la classe englobante)

```
class Outer {
   class Inner {
      public:
         Inner(Outer* o) :outer(o) {}

      private:
         Outer*  outer;
   };
};
```

Java

[non statique] classe imbriquée (alias classe interne ou classe de membre)

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Définie statiquement dans une autre classe

C ++

Classe imbriquée statique

```
class Outer {
   class Inner {
      ...
   };
};
```

Java

Classe imbriquée statique (aka Classe de membre statique) [ref]

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

Défini dans une méthode

(par exemple, gestion d'événements)

C ++

Classe locale [ref]

```
void fun() {
   class Test {
      /* members of Test class */
   };
}
```

Voir aussi les expressions Lambda

Java

Classe locale [ref]

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
            }
        }).start();
```

```
    }
}
```

Dérogation vs surcharge

Les points de substitution et de surcharge suivants s'appliquent à C ++ et à Java:

- Une méthode remplacée a le même nom et les mêmes arguments que sa méthode de base.
- Une méthode surchargée porte le même nom mais des arguments différents et ne repose pas sur l'héritage.
- Deux méthodes avec le même nom et les mêmes arguments mais avec un type de retour différent sont illégales. Voir les questions relatives à Stackoverflow relatives à la "surcharge avec un type de retour différent dans Java" - Question 1 ; question 2

Polymorphisme

Le polymorphisme est la capacité des objets de différentes classes liés par héritage à répondre différemment au même appel de méthode. Voici un exemple:

- Classe de base Forme avec zone comme méthode abstraite
- deux classes dérivées, Square et Circle, implémentent des méthodes de zone
- La référence de forme pointe sur Carré et la zone est appelée

En C ++, le polymorphisme est activé par les méthodes virtuelles. En Java, les méthodes sont virtuelles par défaut.

Ordre de Construction / Destruction

Nettoyage d'objet

En C ++, il est judicieux de déclarer un destructeur comme virtuel pour s'assurer que le destructeur de la sous-classe sera appelé si le pointeur de la classe de base est supprimé.

En Java, une méthode de finalisation est similaire à un destructeur en C ++; cependant, les finaliseurs sont imprévisibles (ils dépendent du GC). Meilleure pratique - utilisez une méthode "close" pour un nettoyage explicite.

```
protected void close() {
    try {
       // do subclass cleanup
    }
    finally {
       isClosed = true;
       super.close();
```

```
    }
}

protected void finalize() {
    try {
       if(!isClosed) close();
    }
    finally {
       super.finalize();
    }
}
```

Méthodes et classes abstraites

Concept	C ++	Java
Méthode abstraite déclaré sans implémentation	méthode virtuelle pure `virtual void eat(void) = 0;`	méthode abstraite `abstract void draw();`
Classe abstraite ne peut pas être instancié	ne peut pas être instancié; a au moins 1 méthode virtuelle pure `class AB {public: virtual void f() = 0;};`	ne peut pas être instancié; peut avoir des méthodes non abstraites `abstract class GraphicObject {}`
Interface aucun champs d'instance	pas de mot-clé "interface", mais peut imiter une interface Java avec les fonctionnalités d'une classe abstraite	très similaire à la classe abstraite, mais 1) prend en charge l'héritage multiple; 2) aucun champs d'instance `interface TestInterface {}`

Modificateurs d'accessibilité

Modificateur	C ++	Java
Public - accessible à tous	*pas de notes spéciales*	*pas de notes spéciales*
Protégé - accessible par sous-classes	également accessible par des amis	également accessible dans le même paquet
Privé - accessible par les membres	également accessible par des amis	*pas de notes spéciales*
défaut	le défaut de classe est privé; struct default est public	accessible par toutes les classes du même

Modificateur	C ++	Java
		package
autre	Friend - un moyen d'accorder l'accès aux membres privés et protégés sans héritage (voir ci-dessous)	

Exemple C ++ Friend

```
class Node {
  private:
    int key;  Node *next;
    // LinkedList::search() can access "key" & "next"
    friend int LinkedList::search();
};
```

Le problème des diamants redoutés

> Le problème du diamant est une ambiguïté qui se produit lorsque deux classes B et C héritent de A et que la classe D hérite à la fois de B et de C. S'il existe une méthode dans A que B et C ont remplacée et D ne la remplace pas quelle version de la méthode D hérite: celle de B ou celle de C? (de Wikipedia)

Bien que C ++ ait toujours été sensible au problème des diamants, Java était sensible à Java 8. À l'origine, Java ne supportait pas l'héritage multiple, mais avec l'arrivée des méthodes d'interface par défaut, les classes Java ne peuvent pas hériter de plusieurs implémentations .

Classe java.lang.Object

En Java, toutes les classes héritent, implicitement ou explicitement, de la classe Object. Toute référence Java peut être convertie en type d'objet.

C ++ n'a pas de classe "Object" comparable.

Collections Java & Conteneurs C ++

Les collections Java sont synonymes de conteneurs C ++.

Organigramme des collections Java

Organigramme des conteneurs C ++

Types entiers

Morceaux	Min	Max	Type C ++ (sur LLP64 ou LP64)	Type Java
8	-2 (8-1) = -128	2 (8-1) -1 = 127	carboniser	octet
8	0	2 (8) -1 = 255	char non signé	-
16	-2 (16-1) = -32,768	2 (16-1) -1 = 32 767	court	court
16	0 (\ u0000)	2 (16) -1 = 65,535 (\ uFFFF)	non signé court	char (non signé)
32	-2 (32-1) = -2,147 milliards	2 (32-1) -1 = 2,147 milliards	int	int
32	0	2 (32) -1 = 4,295 milliards	unsigned int	-
64	-2 (64-1)	2 (16-1) - 1	longue*	long long
64	0	2 (16) -1	non signé long * non signé longtemps	-

* API Win64 n'est que 32 bits

Beaucoup plus de types C ++

Examples

Membres de la classe statique

Les membres statiques ont une portée de classe par opposition à la portée d'objet

Exemple C ++

```
// define in header
class Singleton {
   public:
      static Singleton *getInstance();

   private:
      Singleton() {}
      static Singleton *instance;
```

```
};

// initialize in .cpp
Singleton* Singleton::instance = 0;
```

Exemple Java

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Classes définies dans d'autres constructions

Défini dans une autre classe

C ++

Classe imbriquée [ref] (nécessite une référence à la classe englobante)

```
class Outer {
   class Inner {
      public:
         Inner(Outer* o) :outer(o) {}

      private:
         Outer*  outer;
   };
};
```

Java

[non statique] classe imbriquée (alias classe interne ou classe de membre)

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Définie statiquement dans une autre classe

C ++

Classe imbriquée statique

```
class Outer {
   class Inner {
      ...
   };
};
```

Java

Classe imbriquée statique (aka Classe de membre statique) [ref]

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

Défini dans une méthode

(par exemple, gestion d'événements)

C ++

Classe locale [ref]

```
void fun() {
   class Test {
      /* members of Test class */
   };
}
```

Java

Classe locale [ref]

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
```

```
            }
        }).start();
    }
}
```

Pass-by-value & Pass-by-reference

Beaucoup soutiennent que Java est SEULEMENT pass-by-value, mais il est plus nuancé que cela. Comparez les exemples suivants de C ++ et de Java pour voir les nombreuses variantes de pass-by-value (aka copy) et de pass-by-reference (alias alias).

Exemple C ++ (code complet)

```
// 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;
}
```

Exemple Java (code complet)

```
// 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);
   }
```

Héritage vs composition

C ++ et Java sont tous deux des langages orientés objet, ainsi le diagramme suivant s'applique aux deux.

Downcasting Outcast

Méfiez-vous de l'utilisation du "downcasting" - Le downcasting réduit la hiérarchie d'héritage d'une classe de base à une sous-classe (c.-à-d. L'opposé du polymorphisme). En général, utilisez le polymorphisme et la substitution au lieu de l'instance et du downcasting.

Exemple C ++

```
// explicit type case required
Child *pChild =  (Child *) &parent;
```

Exemple Java

```
if(mySubClass instanceof SubClass) {
   SubClass mySubClass = (SubClass)someBaseClass;
   mySubClass.nonInheritedMethod();
}
```

Méthodes et classes abstraites

Méthode abstraite

déclaré sans implémentation

C ++

méthode virtuelle pure

```
virtual void eat(void) = 0;
```

Java

méthode abstraite

```
abstract void draw();
```

Classe abstraite

ne peut pas être instancié

C ++

ne peut pas être instancié; a au moins 1 méthode virtuelle pure

```
class AB {public: virtual void f() = 0;};
```

Java

ne peut pas être instancié; peut avoir des méthodes non abstraites

```
abstract class GraphicObject {}
```

Interface

aucun champs d'instance

C ++

rien de comparable à Java

Java

très similaire à la classe abstraite, mais 1) prend en charge l'héritage multiple; 2) aucun champs d'instance

```
interface TestInterface {}
```

Lire Comparaison C ++ en ligne: https://riptutorial.com/fr/java/topic/10849/comparaison-c-plusplus

Chapitre 36: Compilateur Java - 'javac'

Remarques

La commande `javac` est utilisée pour compiler les fichiers source Java en fichiers bytecode. Les fichiers Bytecode sont indépendants de la plate-forme. Cela signifie que vous pouvez compiler votre code sur un type de matériel et de système d'exploitation, puis exécuter le code sur toute autre plate-forme prenant en charge Java.

La commande `javac` est incluse dans les distributions Java Development Kit (JDK).

Le compilateur Java et le reste de la chaîne d'outils Java standard imposent les restrictions suivantes au code:

- Le code source est contenu dans les fichiers avec le suffixe ".java"
- Les bytecodes sont contenus dans les fichiers avec le suffixe ".class"
- Pour les fichiers source et bytecode du système de fichiers, les chemins d'accès des fichiers doivent refléter le nom du package et de la classe.

Remarque: Le compilateur `javac` ne doit pas être confondu avec le compilateur Just in Time (JIT) qui compile les bytecodes en code natif.

Examples

La commande 'javac' - démarrer

Exemple simple

En supposant que "HelloWorld.java" contient la source Java suivante:

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

(Pour une explication du code ci-dessus, reportez-vous à la section Démarrer avec le langage Java .)

Nous pouvons compiler le fichier ci-dessus en utilisant cette commande:

```
$ javac HelloWorld.java
```

Cela produit un fichier appelé "HelloWorld.class", que nous pouvons alors exécuter comme suit:

```
$ java HelloWorld
```

```
Hello world!
```

Les points clés à noter dans cet exemple sont les suivants:

1. Le nom de fichier source "HelloWorld.java" doit correspondre au nom de la classe dans le fichier source ... qui est `HelloWorld` . S'ils ne correspondent pas, vous obtiendrez une erreur de compilation.
2. Le nom de fichier du bytecode "HelloWorld.class" correspond au nom de la classe. Si vous deviez renommer "HelloWorld.class", vous obtiendriez une erreur lorsque vous avez essayé de l'exécuter.
3. Lorsque vous exécutez une application Java utilisant `java` , vous indiquez le nom de classe NOT le nom de fichier du bytecode.

Exemple avec des packages

Le code Java le plus pratique utilise des packages pour organiser l'espace de noms des classes et réduire le risque de collision de noms de classe accidentels.

Si nous voulions déclarer la classe `HelloWorld` dans un paquet appelé `com.example` , le fichier "HelloWorld.java" contiendrait la source Java suivante:

```
package com.example;

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

Ce fichier de code source doit être stocké dans une arborescence de répertoires dont la structure correspond au nom du package.

```
.    # the current directory (for this example)
|
 ----com
     |
      ----example
          |
           ----HelloWorld.java
```

Nous pouvons compiler le fichier ci-dessus en utilisant cette commande:

```
$ javac com/example/HelloWorld.java
```

Cela produit un fichier appelé "com / example / HelloWorld.class"; c'est-à-dire qu'après la compilation, la structure du fichier devrait ressembler à ceci:

```
.    # the current directory (for this example)
|
 ----com
```

```
    |
     ----example
         |
          ----HelloWorld.java
          ----HelloWorld.class
```

Nous pouvons alors exécuter l'application comme suit:

```
$ java com.example.HelloWorld
Hello world!
```

Les points supplémentaires à noter de cet exemple sont les suivants:

1. La structure de répertoire doit correspondre à la structure du nom du package.
2. Lorsque vous exécutez la classe, le nom complet de la classe doit être fourni. c'est-à-dire "com.example.HelloWorld" pas "HelloWorld".
3. Vous n'avez pas besoin de compiler et d'exécuter du code Java à partir du répertoire en cours. Nous le faisons juste ici pour l'illustrer.

Compiler plusieurs fichiers à la fois avec 'javac'.

Si votre application se compose de plusieurs fichiers de code source (et la plupart le font!), Vous pouvez les compiler un par un. Alternativement, vous pouvez compiler plusieurs fichiers en même temps en listant les noms de chemins:

```
$ javac Foo.java Bar.java
```

ou en utilisant la fonctionnalité générique de nom de fichier de votre shell de commande ....

```
$ javac *.java
$ javac com/example/*.java
$ javac */**/*.java #Only works on Zsh or with globstar enabled on your shell
```

Cela compilera tous les fichiers source Java dans le répertoire courant, dans le répertoire "com / example" et récursivement dans les répertoires enfants respectivement. Une troisième alternative consiste à fournir une liste des noms de fichiers source (et des options du compilateur) en tant que fichier. Par exemple:

```
$ javac @sourcefiles
```

où le fichier `sourcefiles` contient:

```
Foo.java
Bar.java
com/example/HelloWorld.java
```

Note: compiler du code comme celui-ci est approprié pour les petits projets d'une personne et pour les programmes ponctuels. Au-delà, il est conseillé de sélectionner et d'utiliser un outil de

compilation Java. Alternativement, la plupart des programmeurs utilisent un IDE Java (par exemple NetBeans , eclipse , IntelliJ IDEA) qui offre un compilateur intégré et la création incrémentale de "projets".

Options «javac» couramment utilisées

Voici quelques options pour la commande `javac` susceptibles de vous être utiles

- L'option `-d` définit un répertoire de destination pour l'écriture des fichiers ".class".
- L'option `-sourcepath` définit un chemin de recherche de code source.
- Le `-cp` ou `-classpath` option permet de définir le chemin de recherche pour trouver des classes externes et compilées précédemment. Pour plus d'informations sur le chemin de classe et sa spécification, reportez-vous à la rubrique The Classpath Topic.
- L'option `-version` imprime les informations de version du compilateur.

Une liste plus complète des options du compilateur sera décrite dans un exemple distinct.

Les références

La référence définitive pour la commande `javac` est la page de manuel Oracle pour `javac` .

Compiler pour une version différente de Java

Le langage de programmation Java (et son exécution) a subi de nombreuses modifications depuis sa sortie depuis sa publication initiale. Ces changements incluent:

- Changements dans la syntaxe et la sémantique du langage de programmation Java
- Modifications apportées aux API fournies par les bibliothèques de classes standard Java.
- Modifications apportées au jeu d'instructions Java (bytecode) et au format classfile.

À quelques exceptions près (par exemple, le mot clé `enum` , les modifications apportées à certaines classes "internes", etc.), ces modifications sont rétrocompatibles.

- Un programme Java compilé à l'aide d'une ancienne version de la chaîne d'outils Java s'exécute sur une plate-forme Java plus récente sans recompilation.
- Un programme Java écrit dans une version antérieure de Java sera compilé avec un nouveau compilateur Java.

Compiler l'ancien Java avec un compilateur plus récent

Si vous devez (re) compiler un code Java plus ancien sur une plate-forme Java plus récente pour qu'il s'exécute sur la nouvelle plate-forme, vous n'avez généralement pas besoin d'indiquer de compilation spéciale. Dans quelques cas (par exemple, si vous avez utilisé `enum` comme identifiant), vous pouvez utiliser l'option `-source` pour désactiver la nouvelle syntaxe. Par exemple, compte tenu de la classe suivante:

```
public class OldSyntax {
    private static int enum;  // invalid in Java 5 or later
}
```

les éléments suivants sont requis pour compiler la classe à l'aide d'un compilateur Java 5 (ou ultérieur):

```
$ javac -source 1.4 OldSyntax.java
```

Compiler pour une plate-forme d'exécution plus ancienne

Si vous devez compiler Java pour qu'il fonctionne sur une plate-forme Java plus ancienne, l'approche la plus simple consiste à installer un JDK pour la version la plus ancienne à prendre en charge et à utiliser le compilateur JDK dans vos versions.

Vous pouvez également compiler avec un compilateur Java plus récent, mais il y a des complications. Tout d'abord, il existe des conditions préalables importantes qui doivent être satisfaites:

- Le code que vous compilez ne doit pas utiliser les constructions de langage Java qui n'étaient pas disponibles dans la version de Java que vous ciblez.
- Le code ne doit pas dépendre des classes Java standard, des champs, des méthodes, etc., qui n'étaient pas disponibles sur les anciennes plates-formes.
- Les bibliothèques tierces dont dépend le code doivent également être construites pour l'ancienne plate-forme et disponibles à la compilation et à l'exécution.

Étant donné que les conditions préalables sont remplies, vous pouvez recompiler le code pour une plate-forme plus ancienne en utilisant l'option `-target` . Par exemple,

```
$ javac -target 1.4 SomeClass.java
```

compilera la classe ci-dessus pour produire des bytecodes compatibles avec JVM Java 1.4 ou ultérieur. (En fait, l'option `-source` implique un `-target` compatible, donc `javac -source 1.4 ...` aurait le même effet. La relation entre `-source` et `-target` est décrite dans la documentation Oracle.)

Cela dit, si vous utilisez simplement `-target` ou `-source` , vous compilerez toujours avec les bibliothèques de classes standard fournies par le JDK du compilateur. Si vous ne faites pas attention, vous pouvez vous retrouver avec des classes avec la version correcte du bytecode, mais avec des dépendances sur des API non disponibles. La solution consiste à utiliser l'option `-bootclasspath` . Par exemple:

```
$ javac -target 1.4 --bootclasspath path/to/java1.4/rt.jar SomeClass.java
```

compilera avec un autre ensemble de bibliothèques d'exécution. Si la classe compilée a des dépendances (accidentelles) sur les nouvelles bibliothèques, cela vous donnera des erreurs de compilation.

Lire Compilateur Java - 'javac' en ligne: https://riptutorial.com/fr/java/topic/4478/compilateur-java---javac-

Chapitre 37: Compilateur Just in Time (JIT)

Remarques

Histoire

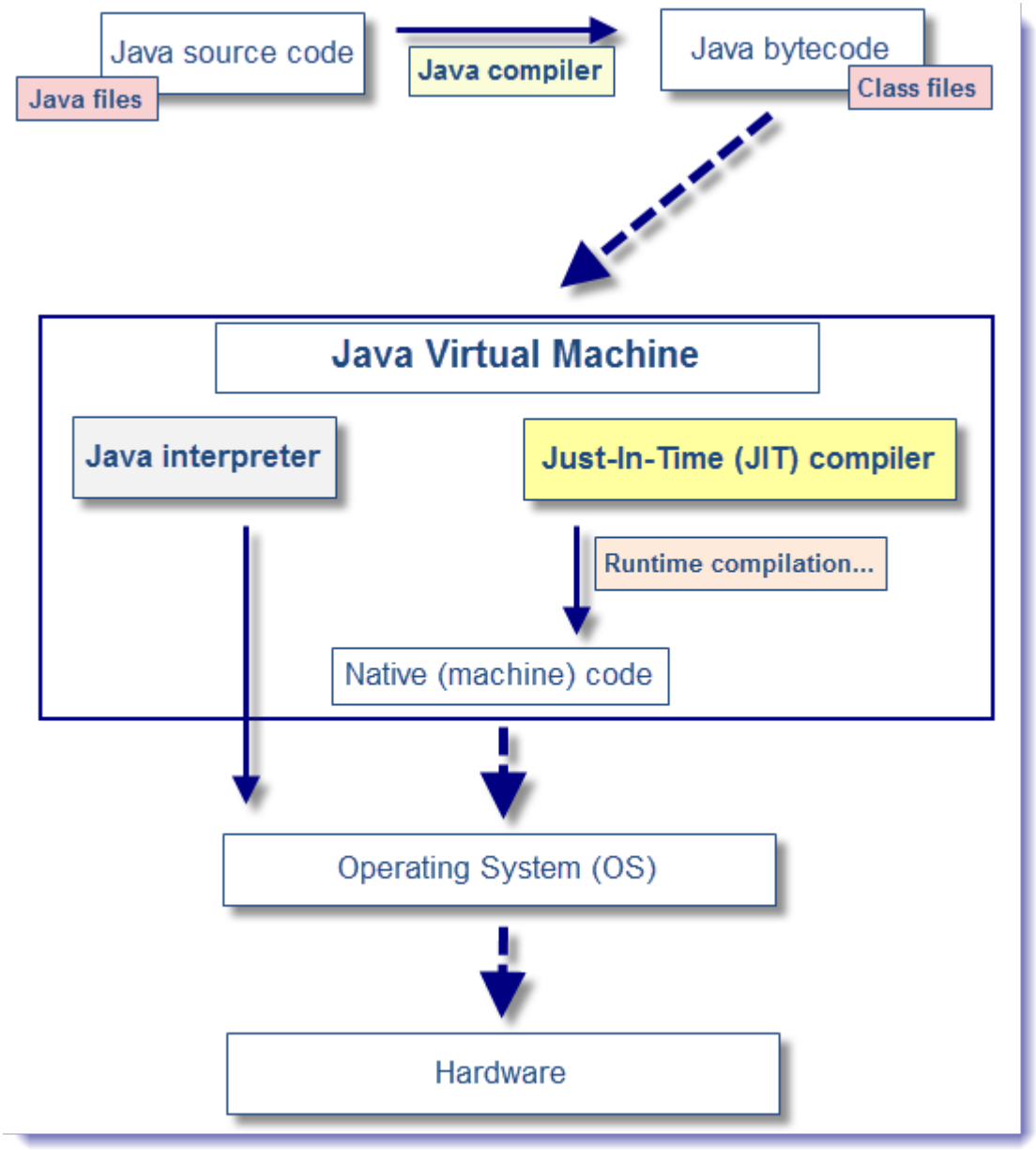
Le compilateur Symantec JIT était disponible dans Sun Java à partir de la version 1.1.5, mais il avait des problèmes.

Le compilateur Hotspot JIT a été ajouté à Sun Java dans la version 1.2.2 en tant que plug-in. Dans Java 1.3, JIT était activé par défaut.

(Source: Quand Java a-t-il un compilateur JIT?

Examples

Vue d'ensemble



Le compilateur Just-In-Time (JIT) est un composant de l'environnement d'exécution Java ™ qui améliore les performances des applications Java au moment de l'exécution.

- Les programmes Java sont constitués de classes, qui contiennent des bytecodes indépendants de la plate-forme pouvant être interprétés par une JVM sur de nombreuses architectures informatiques différentes.
- Lors de l'exécution, la machine virtuelle Java charge les fichiers de classe, détermine la sémantique de chaque bytecode individuel et effectue le calcul approprié.

  L'utilisation supplémentaire du processeur et de la mémoire lors de l'interprétation signifie qu'une application Java fonctionne plus lentement qu'une application native.

  Le compilateur JIT améliore les performances des programmes Java en compilant les codes d'octets dans le code machine natif au moment de l'exécution.

Le compilateur JIT est activé par défaut et est activé lors de l'appel d'une méthode Java. Le compilateur JIT compile les bytecodes de cette méthode en code machine natif, en le compilant `"just in time"` pour s'exécuter.

Lorsqu'une méthode a été compilée, la machine virtuelle Java appelle directement le code compilé de cette méthode au lieu de l'interpréter. Théoriquement, si la compilation ne nécessitait pas de temps processeur et d'utilisation de la mémoire, la compilation de chaque méthode pourrait permettre à la vitesse du programme Java de s'approcher de celle d'une application native.

La compilation JIT nécessite du temps processeur et de la mémoire. Au démarrage de la JVM, des milliers de méthodes sont appelées. La compilation de toutes ces méthodes peut affecter de manière significative le temps de démarrage, même si le programme atteint finalement de très bonnes performances de pointe.

- En pratique, les méthodes ne sont pas compilées la première fois qu'elles sont appelées. Pour chaque méthode, la machine virtuelle Java conserve un `call count` qui est incrémenté chaque fois que la méthode est appelée.
- JVM interprète une méthode jusqu'à ce que son nombre d'appels dépasse un seuil de compilation JIT.
- Par conséquent, les méthodes souvent utilisées sont compilées peu après le démarrage de la JVM et les méthodes moins utilisées sont compilées beaucoup plus tard ou pas du tout.
- Le seuil de compilation JIT aide la JVM à démarrer rapidement et à améliorer les performances.
- Le seuil a été soigneusement sélectionné pour obtenir un équilibre optimal entre les temps de démarrage et les performances à long terme.
- Une fois qu'une méthode est compilée, son nombre d'appels est remis à zéro et les appels ultérieurs à la méthode continuent à incrémenter son nombre.
- Lorsque le nombre d'appels d'une méthode atteint un seuil de recompilation JIT, le compilateur JIT le compile une seconde fois, en appliquant une plus grande sélection d'optimisations que la compilation précédente.
- Ce processus est répété jusqu'à ce que le niveau d'optimisation maximal soit atteint.

  Les méthodes les plus utilisées d'un programme Java sont toujours optimisées de manière plus agressive, optimisant ainsi les avantages de l'utilisation du compilateur JIT.

Le compilateur JIT peut également mesurer `operational data at run time` et utiliser ces données pour améliorer la qualité des recompilations ultérieures.

> Le compilateur JIT peut être désactivé, auquel cas le programme Java entier sera interprété. La désactivation du compilateur JIT n'est pas recommandée, sauf pour diagnostiquer ou contourner les problèmes de compilation JIT.

Lire Compilateur Just in Time (JIT) en ligne: https://riptutorial.com/fr/java/topic/5152/compilateur-just-in-time--jit-

Chapitre 38: CompletableFuture

Introduction

CompletableFuture est une classe ajoutée à Java SE 8 qui implémente l'interface Future de Java SE 5. Outre la prise en charge de l'interface Future, elle ajoute de nombreuses méthodes permettant un rappel asynchrone lorsque le futur est terminé.

Examples

Convertir la méthode de blocage en asynchrone

La méthode suivante prendra une seconde ou deux selon votre connexion pour récupérer une page Web et compter la longueur du texte. Quel que soit le thread qui l'appelle, il sera bloqué pour cette période. En outre, il relance une exception utile par la suite.

```
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);
    }
}
```

Cela le convertit en une méthode qui retournera immédiatement en déplaçant l'appel de la méthode de blocage vers un autre thread. Par défaut, la méthode supplyAsync exécutera le fournisseur sur le pool commun. Pour une méthode de blocage, ce n'est probablement pas un bon choix car on peut épuiser les threads de ce pool, c'est pourquoi j'ai ajouté le paramètre de service facultatif.

```
static private ExecutorService service = Executors.newCachedThreadPool();

static public CompletableFuture<Long> asyncGetWebPageLength(String url) {
    return CompletableFuture.supplyAsync(() -> blockingGetWebPageLength(url), service);
}
```

Pour utiliser la fonction de manière asynchrone, il faut utiliser les méthodes qui acceptent l’appel d’une lamda avec le résultat du fournisseur lorsque celui-ci se termine comme thenAccept. Il est également important d'utiliser exceptionnellement ou de gérer la méthode pour consigner les éventuelles exceptions.

```
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;
            });

}
```

Exemple simple de CompletableFuture

Dans l'exemple ci - dessous, la `calculateShippingPrice` méthode calcule les coûts d'expédition, ce qui prend un certain temps de traitement. Dans un exemple concret, il s'agirait par exemple de contacter un autre serveur qui renvoie le prix en fonction du poids du produit et du mode de livraison.

En modélisant cela de manière asynchrone via `CompletableFuture` , nous pouvons continuer différents travaux dans la méthode (c.-à-d. Calculer les coûts d'emballage).

```
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!
    });
}
```

Lire CompletableFuture en ligne: https://riptutorial.com/fr/java/topic/10935/completablefuture

Chapitre 39: Console I / O

Examples

Lecture des entrées utilisateur depuis la console

Utilisation de `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());
}
```

Les importations suivantes sont nécessaires pour ce code:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

Utilisation du `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 + "!");
```

L'importation suivante est nécessaire pour cet exemple:

```
import java.util.Scanner;
```

Pour lire plusieurs lignes, `scanner.nextLine()` plusieurs fois `scanner.nextLine()` :

```
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 + "!");
```

Il existe deux méthodes pour obtenir des `Strings` , `next()` et `nextLine()` . `next()` renvoie le texte jusqu'au premier espace (également appelé "jeton"), et `nextLine()` renvoie tout le texte `nextLine()` par l'utilisateur jusqu'à ce que vous `nextLine()` sur enter.

`Scanner` fournit également des méthodes utilitaires pour lire des types de données autres que `String` . Ceux-ci inclus:

```
scanner.nextByte();
scanner.nextShort();
scanner.nextInt();
scanner.nextLong();
scanner.nextFloat();
scanner.nextDouble();
scanner.nextBigInteger();
scanner.nextBigDecimal();
```

Si vous préfixez l'une de ces méthodes avec `has` (comme dans `hasNextLine()` , `hasNextInt()`) renvoie `true` si le type de requête est plus `hasNextInt()` dans le flux. Remarque: Ces méthodes planteront le programme si l'entrée n'est pas du type demandé (par exemple, en tapant "a" pour `nextInt()`). Vous pouvez utiliser un `try {} catch() {}` pour éviter cela (voir: Exceptions)

```
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");
}
```

En utilisant `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();
```

Avantages :

- Les méthodes de lecture sont synchronisées
- La syntaxe de la chaîne de format peut être utilisée

Remarque : Cela ne fonctionnera que si le programme est exécuté à partir d'une ligne de commande réelle sans rediriger les flux d'entrée et de sortie standard. Il ne fonctionne pas lorsque le programme est exécuté à partir de certains IDE, tels que Eclipse. Pour le code qui fonctionne

dans les IDE et avec la redirection de flux, voir les autres exemples.

Implémentation du comportement de base de la ligne de commande

Pour les prototypes de base ou le comportement de base de la ligne de commande, la boucle suivante est 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");
```

```
    }
}
```

Alignement des chaînes dans la console

La méthode `PrintWriter.format` (appelée via `System.out.format`) peut être utilisée pour imprimer des chaînes alignées dans la console. La méthode reçoit une `String` avec les informations de format et une série d'objets à formater:

```
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();
}
```

Sortie:

```
1   1          1
1234 1234    1234
1234567 1234567 123456
123456789 12345678 123456
```

L'utilisation de chaînes de format avec une taille fixe permet d'imprimer les chaînes dans un aspect de type tableau avec des colonnes de taille fixe:

```
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();
}
```

Sortie:

```
1   1             1
123 1234       1234
```

```
123 1234567  123456
123 12345678 123456
```

Exemples de chaînes de format

- `%s` : juste une chaîne sans formatage
- `%5s` : formate la chaîne avec un **minimum** de 5 caractères; si la chaîne est plus courte, elle sera **remplacée** par 5 caractères et alignée à **droite**
- `%-5s` : `%-5s` la chaîne avec un **minimum** de 5 caractères; si la chaîne est plus courte, elle sera **remplie** à 5 caractères et alignée à **gauche**
- `%5.10s` : `%5.10s` la chaîne avec un **minimum** de 5 caractères et un **maximum** de 10 caractères; si la chaîne est inférieure à 5, elle sera **remplacée** par 5 caractères et alignée à **droite** ; si la chaîne est plus longue que 10, elle sera **tronquée** à 10 caractères et alignée à **droite**
- `%-5.5s` : `%-5.5s` la chaîne avec une taille **fixe** de 5 caractères (minimum et maximum sont égaux); si la chaîne est inférieure à 5, elle sera **remplacée** par 5 caractères et alignée à **gauche** ; si la chaîne est plus longue que 5, elle sera **tronquée** à 5 caractères et alignée à **gauche**

Lire Console I / O en ligne: https://riptutorial.com/fr/java/topic/126/console-i---o

Chapitre 40: Constructeurs

Introduction

Bien que cela ne soit pas obligatoire, les compilateurs en Java sont des méthodes reconnues par le compilateur pour instancier des valeurs spécifiques pour la classe, qui peuvent être essentielles au rôle de l'objet. Cette rubrique illustre l'utilisation correcte des constructeurs de classes Java.

Remarques

La spécification de langage Java traite en détail de la nature exacte de la sémantique du constructeur. Ils peuvent être trouvés dans JLS §8.8

Examples

Constructeur par défaut

La "valeur par défaut" pour les constructeurs est qu'ils n'ont aucun argument. Si vous ne spécifiez **aucun** constructeur, le compilateur générera un constructeur par défaut pour vous.
Cela signifie que les deux extraits suivants sont sémantiquement équivalents:

```
public class TestClass {
    private String test;
}
```

```
public class TestClass {
    private String test;
    public TestClass() {

    }
}
```

La visibilité du constructeur par défaut est la même que celle de la classe. Ainsi, une classe définie package-private a un constructeur par défaut package-private

Cependant, si vous avez un constructeur autre que celui par défaut, le compilateur ne générera pas de constructeur par défaut pour vous. Donc, ils ne sont pas équivalents:

```
public class TestClass {
    private String test;
    public TestClass(String arg) {
    }
}
```

```
public class TestClass {
    private String test;
    public TestClass() {
```

```
    }
    public TestClass(String arg) {
    }
}
```

Attention, le constructeur généré n'effectue aucune initialisation non standard. Cela signifie que tous les champs de votre classe auront leur valeur par défaut, à moins qu'ils n'aient un initialiseur.

```
public class TestClass {

    private String testData;

    public TestClass() {
        testData = "Test"
    }
}
```

Les constructeurs sont appelés comme ceci:

```
TestClass testClass = new TestClass();
```

Constructeur avec arguments

Les constructeurs peuvent être créés avec n'importe quel type d'arguments.

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
        this.testData = testData;
    }
}
```

Appelé comme ceci:

```
TestClass testClass = new TestClass("Test Data");
```

Une classe peut avoir plusieurs constructeurs avec des signatures différentes. Pour enchaîner les appels de constructeur (appeler un constructeur différent de la même classe lors de l'instanciation), utilisez `this()` .

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
        this.testData = testData;
    }

    public TestClass() {
        this("Test"); // testData defaults to "Test"
    }
```

```
}
```

Appelé comme ceci:

```
TestClass testClass1 = new TestClass("Test Data");
TestClass testClass2 = new TestClass();
```

Appeler le constructeur parent

Disons que vous avez une classe Parent et une classe Enfant. Pour construire une instance Child, vous devez toujours exécuter un constructeur Parent au sein même du constructeur Child. Nous pouvons sélectionner le constructeur Parent que nous voulons en appelant explicitement `super(...)` avec les arguments appropriés dans notre première déclaration de constructeur Child. Cela vous fait gagner du temps en réutilisant le constructeur de la classe Parent au lieu de réécrire le même code dans le constructeur de la classe Child.

Sans `super(...)` méthode:

(implicitement, la version no-args `super()` est appelée de manière invisible)

```
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;
    }
}
```

Avec la méthode `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
    }
}
```

Remarque: Les appels à un autre constructeur (chaînage) ou au super constructeur **DOIVENT** être la première instruction à l'intérieur du constructeur.

Si vous appelez explicitement le constructeur `super(...)` , un constructeur parent correspondant doit exister (c'est simple, n'est-ce pas?).

Si vous n'appelez aucun constructeur `super(...)` explicitement, votre classe parent doit avoir un constructeur no-args - et ceci peut être écrit explicitement ou créé par défaut par le compilateur si la classe parente ne fournit pas n'importe quel constructeur.

```
class Parent{
    public Parent(String tName, int tAge) {}
}

class Child extends Parent{
    public Child(){}
}
```

La classe Parent n'a pas de constructeur par défaut, le compilateur ne peut donc pas ajouter de `super` dans le constructeur Child. Ce code ne sera pas compilé. Vous devez changer les constructeurs pour les adapter des deux côtés, ou écrire votre propre `super` appel, comme ça:

```
class Child extends Parent{
    public Child(){
          super("",0);
    }
}
```

Lire Constructeurs en ligne: https://riptutorial.com/fr/java/topic/682/constructeurs

Chapitre 41: Conversion de type

Syntaxe

- TargetType target = (SourceType) source;

Examples

Coulée primitive non numérique

Le type `boolean` ne peut être converti en / à partir d'aucun autre type primitif.

Une `char` peut être coulé à / de tout type numérique en utilisant les mises en correspondance de point de code spécifié par Unicode. Une `char` est représenté dans la mémoire en tant que valeur non signé Entier 16 bits (2 octets), de sorte que la coulée d' `byte` (1 octet) va baisser 8 de ces bits (ce qui est sans danger pour les caractères ASCII). Les méthodes utilitaires de la classe `Character` utilisent `int` (4 octets) pour transférer vers / des valeurs de point de code, mais un `short` (2 octets) suffirait également pour stocker un point de code 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
```

Coulée primitive numérique

Les primitives numériques peuvent être converties de deux manières. *La* diffusion *implicite* se produit lorsque le type de source a une portée inférieure à celle du type cible.

```
//Implicit casting
byte byteVar = 42;
short shortVar = byteVar;
int intVar = shortVar;
long longVar = intvar;
float floatVar = longVar;
double doubleVar = floatVar;
```

La diffusion *explicite* doit être effectuée lorsque le type de source est plus large que le type cible.

```
//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;
```

Lors de la conversion de primitives en virgule flottante (`float` , `double`) en primitives de nombres entiers, le nombre est **arrondi** au nombre **inférieur** .

Coulée d'objets

Comme pour les primitives, les objets peuvent être exprimés explicitement et implicitement.

La diffusion implicite se produit lorsque le type de source étend ou implémente le type cible (conversion vers une super-classe ou une interface).

Le transtypage explicite doit être effectué lorsque le type de source est étendu ou implémenté par le type cible (conversion en sous-type). Cela peut générer une exception d'exécution (`ClassCastException`) lorsque l'objet en cours de diffusion n'est pas du type cible (ou du sous-type de la cible).

```
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)
```

Promotion numérique de 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.
}
```

Tester si un objet peut être converti en utilisant instanceof

Java fournit l'opérateur `instanceof` pour tester si un objet est d'un certain type ou une sous-classe de ce type. Le programme peut alors choisir de lancer ou non cet objet en conséquence.

```
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.
}
```

Lire Conversion de type en ligne: https://riptutorial.com/fr/java/topic/1392/conversion-de-type

Chapitre 42: Conversion vers et à partir de chaînes

Examples

Conversion d'autres types de données en chaîne

- Vous pouvez obtenir la valeur d'autres types de données primitifs en tant que chaîne en utilisant l'une des méthodes `valueOf` la classe String.

  Par exemple:

  ```
  int i = 42;
  String string = String.valueOf(i);
  //string now equals "42”.
  ```

  Cette méthode est également surchargée pour d'autres types de données, tels que `float` , `double` , `boolean` et même `Object` .

- Vous pouvez également obtenir tout autre objet (toute instance de n'importe quelle classe) en tant que chaîne en appelant `.toString` dessus. Pour que cela donne des résultats utiles, la classe doit remplacer `toString()` . La plupart des classes de bibliothèques Java standard le font, telles que `Date` et autres.

  Par exemple:

  ```
  Foo foo = new Foo(); //Any class.
  String stringifiedFoo = foo.toString().
  ```

  Ici `stringifiedFoo` contient une représentation de `foo` tant que chaîne.

Vous pouvez également convertir n'importe quel type de nombre en chaîne avec une notation courte comme ci-dessous.

```
int i = 10;
String str = i + "";
```

Ou simplement simple est

```
String str = 10 + "";
```

Conversion en / des octets

Pour encoder une chaîne dans un tableau d'octets, vous pouvez simplement utiliser la méthode `String#getBytes()` , avec l'un des jeux de caractères standard disponibles sur tout environnement

d'exécution Java:

```
byte[] bytes = "test".getBytes(StandardCharsets.UTF_8);
```

et décoder:

```
String testString = new String(bytes, StandardCharsets.UTF_8);
```

vous pouvez simplifier davantage l'appel en utilisant une importation statique:

```
import static java.nio.charset.StandardCharsets.UTF_8;
...
byte[] bytes = "test".getBytes(UTF_8);
```

Pour les jeux de caractères moins courants, vous pouvez indiquer le jeu de caractères avec une chaîne:

```
byte[] bytes = "test".getBytes("UTF-8");
```

et l'inverse:

```
String testString = new String (bytes, "UTF-8");
```

Cela signifie toutefois que vous devez gérer l'exception `UnsupportedCharsetException` vérifiée.

L'appel suivant utilisera le jeu de caractères par défaut. Le jeu de caractères par défaut est spécifique à la plate-forme et diffère généralement entre les plates-formes Windows, Mac et Linux.

```
byte[] bytes = "test".getBytes();
```

et l'inverse:

```
String testString = new String(bytes);
```

Notez que les caractères et octets invalides peuvent être remplacés ou ignorés par ces méthodes. Pour plus de contrôle, par exemple pour valider les entrées, vous êtes invité à utiliser les classes `CharsetEncoder` et `CharsetDecoder` .

Base64 Encoding / Decoding

Parfois, vous aurez besoin d'encoder des données binaires sous la forme d'une chaîne codée en *base64* .

Pour cela , nous pouvons utiliser la `DatatypeConverter` classe du `javax.xml.bind` package:

```
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

Alternativement, nous pouvons utiliser `Base64` d' Apache commons-codec .

```
import org.apache.commons.codec.binary.Base64;

// your blob of binary as a byte array
byte[] blob = "someBinaryData".getBytes();

// use the Base64 class to encode
String binaryAsAString = Base64.encodeBase64String(blob);

// use the Base64 class to decode
byte[] blob2 = Base64.decodeBase64(binaryAsAString);

// assert that the two blobs are equal
System.out.println("Equal : " + Boolean.toString(Arrays.equals(blob, blob2)));
```

Si vous inspectez ce programme en cours d'exécution, vous verrez que `someBinaryData` encode vers `c29tZUJpbmFyeURhdGE=` , un objet String *UTF-8* très `c29tZUJpbmFyeURhdGE=` à `c29tZUJpbmFyeURhdGE=` .

Java SE 8

Les détails pour les mêmes peuvent être trouvés à 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);
```

Référence

Analyse de chaînes à une valeur numérique

Chaîne à un type numérique primitif ou un type d'encapsuleur numérique:

Chaque classe d'encapsuleur numérique fournit une méthode `parseXxx` qui convertit une `String` dans le type primitif correspondant. Le code suivant convertit un `String` en `int` à l'aide de la méthode `Integer.parseInt` :

```
String string = "59";
int primitive = Integer.parseInteger(string);
```

Pour convertir en une `String` une instance d'une classe d'encapsuleur numérique, vous pouvez soit utiliser une surcharge de la méthode des classes wrapper `valueOf` :

```
String string = "59";
Integer wrapper = Integer.valueOf(string);
```

ou compter sur la boxe automatique (Java 5 et versions ultérieures):

```
String string = "59";
Integer wrapper = Integer.parseInteger(string);  // 'int' result is autoboxed
```

Le modèle ci-dessus fonctionne pour les `byte` , les `short` , les `int` , les `long` , les `float` et les `double` et les classes de wrappers correspondantes (`Byte` , `Short` , `Integer` , `Long` , `Float` et `Double`).

Chaîne à Entier à l'aide de 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
```

Des exceptions

L'exception NumberFormatException non vérifiée sera déclenchée si une méthode `valueOf(String)` ou `parseXxx(...)` numérique est appelée pour une chaîne qui n'est pas une représentation numérique acceptable ou qui représente une valeur hors limites.

Obtenir un `String` à partir d'un` InputStream`

Une `String` peut être lue depuis un `InputStream` à l'aide du constructeur de tableau d'octets.

```
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);
}
```

Cela utilise le jeu de caractères par défaut du système, bien qu'un autre jeu de caractères puisse

être spécifié:

```
return new String(bytes, Charset.forName("UTF-8"));
```

Conversion d'une chaîne en d'autres types de données

Vous pouvez convertir une chaîne **numérique** en divers types numériques Java comme suit:

String à int:

```
String number = "12";
int num = Integer.parseInt(number);
```

Chaîne à flotter:

```
String number = "12.0";
float num = Float.parseFloat(number);
```

Chaîne à doubler:

```
String double = "1.47";
double num = Double.parseDouble(double);
```

Chaîne à booléen:

```
String falseString = "False";
boolean falseBool = Boolean.parseBoolean(falseString);   // falseBool = false

String trueString = "True";
boolean trueBool = Boolean.parseBoolean(trueString);     // trueBool = true
```

String à long:

```
String number = "47";
long num = Long.parseLong(number);
```

Chaîne à BigInteger:

```
String bigNumber = "21";
BigInteger reallyBig = new BigInteger(bigNumber);
```

Chaîne à BigDecimal:

```
String bigFraction = "17.21455";
BigDecimal reallyBig = new BigDecimal(bigFraction);
```

Exceptions à la conversion:

Les conversions numériques ci-dessus `NumberFormatException` toutes une `NumberFormatException`

(non contrôlée) si vous tentez d'analyser une chaîne qui n'est pas un nombre correctement formaté ou qui est hors de portée pour le type cible. La rubrique Exceptions explique comment gérer ces exceptions.

Si vous voulez tester que vous pouvez analyser une chaîne, vous pouvez implémenter une méthode `tryParse...` comme ceci:

```
boolean tryParseInt (String value) {
    try {
        String somechar = Integer.parseInt(value);
        return true;
     } catch (NumberFormatException e) {
        return false;
     }
}
```

Cependant, appeler cette méthode `tryParse...` immédiatement avant l'analyse est (sans doute) une mauvaise pratique. Il serait préférable d'appeler la méthode `parse...` et de traiter l'exception.

Lire Conversion vers et à partir de chaînes en ligne:
https://riptutorial.com/fr/java/topic/6678/conversion-vers-et-a-partir-de-chaines

Chapitre 43: Cordes

Introduction

Les chaînes (`java.lang.String`) sont des morceaux de texte stockés dans votre programme. Les chaînes **ne** sont **pas** un type de données primitif en Java , mais elles sont très courantes dans les programmes Java.

En Java, les chaînes sont immuables, ce qui signifie qu'elles ne peuvent pas être modifiées. (Cliquez ici pour une explication plus approfondie de l'immuabilité.)

Remarques

Les chaînes Java étant immuables , toutes les méthodes qui manipulent une `String` **renvoient un nouvel objet `String`** . Ils ne changent pas la `String` origine. Cela inclut les méthodes de sous-chaîne et de remplacement que les programmeurs C et C ++ pourraient modifier l'objet `String` cible.

Utilisez `StringBuilder` au lieu de `String` si vous souhaitez concaténer plus de deux objets `String` dont les valeurs ne peuvent pas être déterminées au moment de la compilation. Cette technique est plus performante que la création de nouveaux objets `String` et leur concaténation, car `StringBuilder` est mutable.

`StringBuffer` peut également être utilisé pour concaténer des objets `String` . Cependant, cette classe est moins performante car elle est conçue pour être thread-safe et acquiert un mutex avant chaque opération. Comme vous n'avez presque jamais besoin de sécurité des threads lors de la concaténation de chaînes, il est préférable d'utiliser `StringBuilder` .

Si vous pouvez exprimer une concaténation de chaîne en tant qu'expression unique, il est préférable d'utiliser l'opérateur `+` . Le compilateur Java convertira une expression contenant `+` concaténations en une séquence efficace d'opérations à l'aide de `String.concat(...)` ou `StringBuilder` . Le conseil d'utilisation de `StringBuilder` ne s'applique explicitement que lorsque la concaténation implique plusieurs expressions.

Ne stockez pas d'informations sensibles dans des chaînes. Si quelqu'un est capable d'obtenir un vidage de mémoire de votre application en cours d'exécution, il pourra alors trouver tous les objets `String` existants et lire leur contenu. Cela inclut les objets `String` inaccessibles et en attente de récupération de place. Si cela pose un problème, vous devrez effacer les données de chaîne sensibles dès que vous en avez terminé. Vous ne pouvez pas faire cela avec des objets `String` car ils sont immuables. Par conséquent, il est conseillé d'utiliser des objets `char[]` pour stocker des données de caractères sensibles et de les effacer (par exemple, les remplacer par des `'\000'` caractères) lorsque vous avez terminé.

Toutes les instances de `String` sont créées sur le tas, même les instances qui correspondent aux littéraux de chaîne. La particularité des littéraux de chaîne est que la machine virtuelle Java garantit que tous les littéraux égaux (c'est-à-dire constitués des mêmes caractères) sont représentés par un seul objet `String` (ce comportement est spécifié dans JLS). Ceci est implémenté par les chargeurs de classes JVM. Lorsqu'un chargeur de classe charge une classe, il recherche les littéraux de chaîne utilisés dans la définition de classe, chaque fois qu'il en voit un, il vérifie s'il existe déjà un enregistrement dans le pool de chaînes pour ce littéral (en utilisant le littéral comme clé). . S'il existe déjà une entrée pour le littéral, la référence à une instance `String` stockée en tant que paire pour ce littéral est utilisée. Sinon, une nouvelle instance `String` est créée et une référence à l'instance est stockée pour le littéral (utilisé comme clé) dans le pool de chaînes. (Voir aussi l' internat de chaînes).

Le pool de chaînes est contenu dans le segment de mémoire Java et est soumis à une récupération de place normale.

Java SE 7

Dans les versions de Java antérieures à Java 7, le pool de chaînes était contenu dans une partie spéciale du tas appelée "PermGen". Cette partie n'a été collectée qu'occasionnellement.

Java SE 7

Dans Java 7, le pool de chaînes a été retiré de "PermGen".

Notez que les littéraux de chaîne sont implicitement accessibles depuis n'importe quelle méthode qui les utilise. Cela signifie que les objets `String` correspondants ne peuvent être récupérés que si le code lui-même est récupéré.

Jusqu'à Java 8, les objets `String` sont implémentés sous la forme d'un tableau de caractères UTF-16 (2 octets par caractère). Il existe une proposition dans Java 9 d'implémenter `String` comme un tableau d'octets avec un champ d'indicateur de codage à noter si la chaîne est codée sous forme d'octets (LATIN-1) ou de caractères (UTF-16).

Examples

Comparaison de chaînes

Afin de comparer les chaînes pour l'égalité, vous devez utiliser les méthodes `equals` ou `equalsIgnoreCase` l'objet String.

Par exemple, l'extrait suivant déterminera si les deux instances de `String` sont égales sur tous les caractères:

```
String firstString = "Test123";
String secondString = "Test" + 123;

if (firstString.equals(secondString)) {
   // Both Strings have the same content.
```

```
}
```

Démo en direct

Cet exemple les comparera, indépendamment de leur cas:

```
String firstString = "Test123";
String secondString = "TEST123";

if (firstString.equalsIgnoreCase(secondString)) {
    // Both Strings are equal, ignoring the case of the individual characters.
}
```

Démo en direct

Notez que `equalsIgnoreCase` ne vous permet pas de spécifier une `Locale` . Par exemple, si vous comparez les deux mots `"Taki"` et `"TAKI"` en anglais, ils sont égaux; Cependant, ils sont différents en turc (en turc, le minuscule `I` est `ı`). Pour les cas comme celui - ci, les deux chaînes de conversion en minuscules (ou majuscules) avec des `Locale` , puis en comparant avec `equals` est la solution.

```
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
```

Démo en direct

N'utilisez pas l'opérateur == pour comparer des chaînes

Sauf si vous pouvez garantir que toutes les chaînes ont été internées (voir ci-dessous), vous **ne** devez **pas** utiliser les opérateurs `==` ou `!=` Pour comparer des chaînes. Ces opérateurs testent en fait des références, et comme plusieurs objets `String` peuvent représenter la même chaîne, cela risque de donner une mauvaise réponse.

Utilisez `String.equals(Object)` la `String.equals(Object)` , qui comparera les objets String en fonction de leurs valeurs. Pour une explication détaillée, veuillez vous référer à Pitfall: utiliser == pour comparer des chaînes .

Comparaison de chaînes dans une déclaration de commutateur

Java SE 7

À partir de Java 1.7, il est possible de comparer une variable String à des littéraux dans une instruction `switch` . Assurez-vous que la chaîne n'est pas nulle, sinon elle lancera toujours une `NullPointerException` . Les valeurs sont comparées à l'aide de `String.equals` , c'est-à-dire, sensibles à la casse.

```
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;
}
```

Démo en direct

Comparaison de chaînes avec des valeurs constantes

Lorsque vous comparez une `String` à une valeur constante, vous pouvez placer la valeur constante sur le côté gauche des valeurs `equals` pour vous assurer de ne pas obtenir d'exception `NullPointerException` si l'autre chaîne est `null` .

```
"baz".equals(foo)
```

Alors que `foo.equals("baz")` lancera une `NullPointerException` si `foo` est `null` , `"baz".equals(foo)` sera évalué à `false` .

Java SE 7

Une alternative plus lisible consiste à utiliser `Objects.equals()` , qui effectue une vérification null sur les deux paramètres: `Objects.equals(foo, "baz")` .

(**Remarque:** il est discutable de savoir s'il vaut mieux éviter les `NullPointerExceptions` en général,

ou les laisser se produire, puis corriger la cause première; voir ici et ici . L'appel à la stratégie d'évitement n'est pas justifiable.)

Classement des chaînes

La classe `String` implémente `Comparable<String>` avec la méthode `String.compareTo` (comme décrit au début de cet exemple). Cela rend l'ordre naturel des objets `String` ordre sensible à la casse. La classe `String` fournit une constante `Comparator<String>` appelée `CASE_INSENSITIVE_ORDER` adaptée au tri non sensible à la casse.

Comparaison avec des chaînes internes

La spécification de langage Java (JLS 3.10.6) indique ce qui suit:

> "De plus, un littéral de chaîne fait toujours référence à la même instance de la classe `String` . En effet, les littéraux de chaîne - ou plus généralement les chaînes d'expressions constantes - sont *internés* afin de partager des instances uniques, en utilisant la méthode `String.intern` . "

Cela signifie qu'il est prudent de comparer des références à deux *littéraux de* chaîne utilisant `==` . De plus, il en va de même pour les références aux objets `String` produits à l'aide de la méthode `String.intern()` .

Par exemple:

```
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");
}
```

Dans les coulisses, le mécanisme d'internement maintient une table de hachage qui contient toutes les chaînes internes encore *accessibles* . Lorsque vous appelez `intern()` sur une `String` , la méthode recherche l'objet dans la table de hachage:

- Si la chaîne est trouvée, cette valeur est renvoyée sous la forme de la chaîne interne.
- Sinon, une copie de la chaîne est ajoutée à la table de hachage et cette chaîne est renvoyée en tant que chaîne interne.

Il est possible d'utiliser l'internement pour permettre de comparer des chaînes en utilisant `==` . Cependant, cela pose des problèmes importants. voir Pitfall - Interner des chaînes pour pouvoir utiliser == est une mauvaise idée pour les détails. Ce n'est pas recommandé dans la plupart des cas.

Changer la casse des caractères dans une chaîne

Le type `String` fournit deux méthodes pour convertir des chaînes entre majuscules et minuscules:

- `toUpperCase` pour convertir tous les caractères en majuscules
- `toLowerCase` pour convertir tous les caractères en minuscules

Ces méthodes renvoient toutes les deux les chaînes converties en tant que nouvelles instances `String` : les objets `String` origine ne sont pas modifiés car `String` est immuable dans Java. Voir cela pour plus sur l'immutabilité: Immuabilité des chaînes en Java

```
String string = "This is a Random String";
String upper = string.toUpperCase();
String lower = string.toLowerCase();

System.out.println(string);   // prints "This is a Random String"
System.out.println(lower);    // prints "this is a random string"
System.out.println(upper);    // prints "THIS IS A RANDOM STRING"
```

Les caractères non alphabétiques, tels que les chiffres et les signes de ponctuation, ne sont pas affectés par ces méthodes. Notez que ces méthodes peuvent également traiter de manière incorrecte certains caractères Unicode sous certaines conditions.

Remarque : Ces méthodes sont *sensibles aux paramètres régionaux* et peuvent produire des résultats inattendus si elles sont utilisées sur des chaînes destinées à être interprétées indépendamment des paramètres régionaux. Des exemples sont les identificateurs de langage de programmation, les clés de protocole et `HTML` balises `HTML` .

Par exemple, `"TITLE".toLowerCase()` dans une locale turque retourne " `tıtle` ", où `ı (\u0131)` est le caractère LATIN SMALL LETTER DOTLESS I. Pour obtenir des résultats corrects pour les chaînes insensibles aux paramètres régionaux, transmettez `Locale.ROOT` tant que paramètre à la méthode de conversion de casse correspondante (par exemple, `toLowerCase(Locale.ROOT)` ou `toUpperCase(Locale.ROOT)`).

Bien que l'utilisation de `Locale.ENGLISH` soit également correcte dans la plupart des cas, la méthode **invariante de** la **langue** est `Locale.ROOT` .

Une liste détaillée des caractères Unicode nécessitant un boîtier spécial est disponible sur le site Web du Consortium Unicode .

Modification de la casse d'un caractère spécifique dans une chaîne ASCII:

Pour modifier la casse d'un caractère spécifique d'une chaîne ASCII, l'algorithme suivant peut être utilisé:

Pas:

1. Déclarez une chaîne.
2. Entrez la chaîne.
3. Convertissez la chaîne en un tableau de caractères.
4. Entrez le caractère à rechercher.
5. Recherchez le caractère dans le tableau de caractères.
6. Si trouvé, vérifiez si le caractère est en minuscule ou en majuscule.
    - Si majuscule, ajoutez 32 au code ASCII du caractère.
    - Si en minuscule, soustrayez 32 du code ASCII du caractère.
7. Modifiez le caractère d'origine du tableau Caractère.
8. Convertissez le tableau de caractères dans la chaîne.

Voila, la caisse du personnage est changée.

Un exemple de code pour l'algorithme est:

```
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);
```

Recherche d'une chaîne dans une autre chaîne

Pour vérifier si une chaîne particulière `a` est contenue dans une chaîne `b` ou non, nous pouvons utiliser la méthode `String.contains()` avec la syntaxe suivante:

```
b.contains(a); // Return true if a is contained in b, false otherwise
```

La méthode `String.contains()` peut être utilisée pour vérifier si une `CharSequence` peut être trouvée

dans la chaîne. La méthode recherche la chaîne `a` dans la chaîne `b` de manière sensible à la casse.

```
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
```

Démo en direct sur Ideone

Pour rechercher la position exacte où une chaîne commence dans une autre chaîne, utilisez `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
```

Démonstration en direct sur Ideone

Le `String.indexOf()` méthode renvoie le premier indice d'une `char` ou `String` dans une autre `String` . La méthode renvoie `-1` si elle n'est pas trouvée.

Remarque : La méthode `String.indexOf()` est sensible à la casse.

Exemple de recherche en ignorant la casse:

```
String str1 = "Hello World";
String str2 = "wOr";
str1.indexOf(str2);                                 // -1
str1.toLowerCase().contains(str2.toLowerCase());  // true
str1.toLowerCase().indexOf(str2.toLowerCase());   // 6
```

Démo en direct sur Ideone

Obtenir la longueur d'une chaîne

Pour obtenir la longueur d'un objet `String` , appelez la méthode `length()` . La longueur est égale au nombre d'unités de code UTF-16 (caractères) dans la chaîne.

```
String str = "Hello, World!";
System.out.println(str.length()); // Prints out 13
```

Démo en direct sur Ideone

Une `char` dans une chaîne est la valeur UTF-16. Les points de code Unicode dont les valeurs sont ≥ 0x1000 (par exemple, la plupart des émoticônes) utilisent deux positions de caractère. Pour

compter le nombre de points de code Unicode dans une chaîne, que chaque point de code correspond à une UTF-16 `char` valeur, vous pouvez utiliser la `codePointCount` méthode:

```
int length = str.codePointCount(0, str.length());
```

Vous pouvez également utiliser un flux de points de code, à partir de Java 8:

```
int length = str.codePoints().count();
```

Sous-supports

```
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")
```

Les sous-chaînes peuvent également être appliquées pour trancher et ajouter / remplacer un caractère dans sa chaîne d'origine. Par exemple, vous avez fait face à une date chinoise contenant des caractères chinois, mais vous souhaitez la stocker en tant que chaîne de format au format bien.

```
String datestring = "2015□11□17□"
datestring = datestring.substring(0, 4) + "-" + datestring.substring(5,7) + "-" +
datestring.substring(8,10);
//Result will be 2015-11-17
```

La méthode de sous-chaîne extrait un morceau d'une `String` . Lorsqu'il est fourni un paramètre, le paramètre est le début et le morceau s'étend jusqu'à la fin de la `String` . Lorsque deux paramètres sont donnés, le premier paramètre est le caractère de départ et le second paramètre est l'index du caractère immédiatement après la fin (le caractère à l'index n'est pas inclus). Un moyen simple de vérifier est la soustraction du premier paramètre de la seconde qui doit donner la longueur attendue de la chaîne.

Java SE 7

Dans les versions JDK <7u6, la méthode de `substring` - `substring` instancie une `String` qui partage le même caractère de sauvegarde `char[]` que la `String` origine et dont les champs `offset` et `count` internes sont définis au début et à la longueur du résultat. Un tel partage peut provoquer des fuites de mémoire, ce qui peut être évité en appelant `new String(s.substring(...))` pour forcer la création d'une copie, après quoi le `char[]` peut être récupéré.

Java SE 7

De la 7u6 JDK `substring` méthode copie toujours l'ensemble sous-jacent `char[]` tableau, ce qui rend le linéaire de la complexité par rapport à une constante précédente , mais qui garantit l'absence de fuites de mémoire en même temps.

Obtenir le nième personnage dans une chaîne

```
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"
```

Pour obtenir le nième caractère d'une chaîne, appelez simplement `charAt(n)` sur une `String` , où `n` est l'index du caractère que vous souhaitez récupérer.

NOTE: l' indice `n` commence à `0` , donc le premier élément est à n = 0.

Séparateur de lignes indépendant de la plate-forme

Étant donné que le nouveau séparateur de ligne varie d'une plate-forme à l'autre (par exemple, `\n` sur les systèmes de type Unix ou `\r\n` sous Windows), il est souvent nécessaire d'avoir un moyen indépendant d'accéder à la plate-forme. En Java, il peut être extrait d'une propriété système:

```
System.getProperty("line.separator")
```

Java SE 7

Comme le nouveau séparateur de ligne est si souvent nécessaire, à partir de Java 7, une méthode de raccourci renvoyant exactement le même résultat que le code ci-dessus est disponible:

```
System.lineSeparator()
```

Remarque : Comme il est très improbable que le nouveau séparateur de ligne change pendant l'exécution du programme, il est préférable de le stocker dans une variable finale statique au lieu de la récupérer à chaque fois que cela est nécessaire.

Lorsque vous utilisez `String.format` , utilisez `%n` plutôt que `\n` ou '\ r \ n' pour générer un nouveau séparateur de ligne indépendant de la plate-forme.

```
System.out.println(String.format('line 1: %s.%nline 2: %s%n', lines[0],lines[1]));
```

Ajout de la méthode toString () pour les objets personnalisés

Supposons que vous ayez défini la classe `Person` suivante:

```
public class Person {

    String name;
    int age;

    public Person (int age, String name) {
```

```
        this.age = age;
        this.name = name;
    }
}
```

Si vous instanciez un nouvel objet `Person` :

```
Person person = new Person(25, "John");
```

et plus tard dans votre code, vous utilisez l'instruction suivante pour imprimer l'objet:

```
System.out.println(person.toString());
```

Démo en direct sur Ideone

vous obtiendrez une sortie similaire à la suivante:

```
Person@7ab89d
```

Ceci est le résultat de l'implémentation de la `toString()` définie dans la classe `Object` , une super-classe de `Person` . La documentation de `Object.toString()` indique:

> La méthode toString de la classe Object renvoie une chaîne composée du nom de la classe dont l'objet est une instance, du caractère "@" et de la représentation hexadécimale non signée du code de hachage de l'objet. En d'autres termes, cette méthode renvoie une chaîne égale à la valeur de:
>
> ```
> getClass().getName() + '@' + Integer.toHexString(hashCode())
> ```

Donc, pour une sortie significative, vous devrez **remplacer** la `toString()` :

```
@Override
public String toString() {
    return "My name is " + this.name + " and my age is " + this.age;
}
```

Maintenant, le résultat sera:

```
My name is John and my age is 25
```

Vous pouvez aussi écrire

```
System.out.println(person);
```

Démo en direct sur Ideone

En fait, `println()` invoque implicitement la méthode `toString` sur l'objet.

Fendre les cordes

Vous pouvez fractionner une `String` sur un caractère de délimitation particulier ou une expression régulière , vous pouvez utiliser la méthode `String.split()` avec la signature suivante:

```
public String[] split(String regex)
```

Notez que le caractère de délimitation ou l'expression régulière sont supprimés du tableau de chaînes résultant.

Exemple utilisant un caractère de délimitation:

```
String lineFromCsvFile = "Mickey;Bolton;12345;121216";
String[] dataCells = lineFromCsvFile.split(";");
// Result is dataCells = { "Mickey", "Bolton", "12345", "121216"};
```

Exemple utilisant une expression régulière:

```
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?"};
```

Vous pouvez même séparer directement un littéral `String` :

```
String[] firstNames = "Mickey, Frank, Alicia, Tom".split(", ");
// Result is firstNames = {"Mickey", "Frank", "Alicia", "Tom"};
```

Attention : n'oubliez pas que le paramètre est toujours traité comme une expression régulière.

```
"aaa.bbb".split("."); // This returns an empty array
```

Dans l'exemple précédent `.` est traité comme le caractère générique d'expression régulière qui correspond à n'importe quel caractère, et comme chaque caractère est un délimiteur, le résultat est un tableau vide.

Fractionnement basé sur un délimiteur qui est un méta-caractère regex

Les caractères suivants sont considérés comme spéciaux (aka méta-caractères) dans regex

```
< > - = ! ( ) [ ] { } \ ^ $ | ? * + .
```

Pour diviser une chaîne en fonction de l'un des délimiteurs ci-dessus, vous devez soit y *échapper* en utilisant `\\` soit utiliser `Pattern.quote()` :

- En utilisant `Pattern.quote()` :

  ```
  String s = "a|b|c";
  ```

```
String regex = Pattern.quote("|");
String[] arr = s.split(regex);
```

- Fuyant les caractères spéciaux:

```
String s = "a|b|c";
String[] arr = s.split("\\|");
```

Split supprime les valeurs vides

`split(delimiter)` supprime par défaut les chaînes vides du tableau de résultats. Pour désactiver ce mécanisme, nous devons utiliser une version surchargée de `split(delimiter, limit)` avec une limite définie sur une valeur négative telle que

```
String[] split = data.split("\\|", -1);
```

`split(regex)` renvoie en interne le résultat du `split(regex, 0)` .

Le paramètre limit contrôle le nombre de fois où le motif est appliqué et affecte donc la longueur du tableau résultant.
Si la limite *n* est supérieure à zéro, alors le motif sera appliqué au maximum *n* - *1* fois, la longueur du tableau ne sera pas supérieure à *n* et la dernière entrée du tableau contiendra toutes les entrées au-delà du dernier délimiteur correspondant.
Si *n* est négatif, le motif sera appliqué autant de fois que possible et le tableau pourra avoir n'importe quelle longueur.
Si *n* est égal à zéro, le modèle sera appliqué autant de fois que possible, le tableau peut avoir n'importe quelle longueur et les chaînes vides à la fin seront supprimées.

Fractionnement avec un `StringTokenizer`

Outre la méthode `split()` , les chaînes peuvent également être divisées en utilisant un `StringTokenizer` .

`StringTokenizer` est encore plus restrictif que `String.split()` , et un peu plus difficile à utiliser. Il est essentiellement conçu pour extraire des jetons délimités par un ensemble fixe de caractères (donnés sous forme de `String`). Chaque personnage agira comme un séparateur. A cause de cette restriction, il est environ deux fois plus rapide que `String.split()` .

Les jeux de caractères par défaut sont des espaces vides (`\t\n\r\f`). L'exemple suivant imprimera chaque mot séparément.

```
String str = "the lazy fox jumped over the brown fence";
StringTokenizer tokenizer = new StringTokenizer(str);
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

Cela va imprimer:

```
the
lazy
fox
jumped
over
the
brown
fence
```

Vous pouvez utiliser différents jeux de caractères pour la séparation.

```
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());
}
```

Cela va imprimer:

```
j
mp
d ov
r
```

Joindre des chaînes avec un délimiteur

Java SE 8

Un tableau de chaînes peut être joint en utilisant la méthode statique `String.join()` :

```
String[] elements = { "foo", "bar", "foobar" };
String singleString = String.join(" + ", elements);

System.out.println(singleString); // Prints "foo + bar + foobar"
```

De même, il existe une méthode `String.join()` surchargée pour les `Iterable` s.

Pour avoir un contrôle précis sur la jointure, vous pouvez utiliser 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]"
```

Pour joindre un flux de chaînes, vous pouvez utiliser le collecteur de jointure :

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", "));
System.out.println(joined); // Prints "foo, bar, foobar"
```

Il y a une option pour définir le préfixe et le suffixe ici aussi:

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", ", "{", "}"));
System.out.println(joined); // Prints "{foo, bar, foobar}"
```

Cordes d'inversion

Il y a plusieurs manières d'inverser une chaîne pour la faire reculer.

1. StringBuilder / StringBuffer:

```
 String code = "code";
 System.out.println(code);

 StringBuilder sb = new StringBuilder(code);
 code = sb.reverse().toString();

 System.out.println(code);
```

2. Tableau de caractères:

```
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));
```

Compter les occurrences d'une sous-chaîne ou d'un caractère dans une chaîne

`countMatches` méthode `countMatches` d' org.apache.commons.lang3.StringUtils est généralement utilisée pour compter les occurrences d'une sous-chaîne ou d'un caractère dans une `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
```

Sinon, pour faire la même chose avec les API Java standard, vous pouvez utiliser des expressions régulières:

```
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;
}
```

Concaténation de chaînes et StringBuilders

La concaténation de chaînes peut être effectuée à l'aide de l'opérateur `+` . Par exemple:

```
String s1 = "a";
String s2 = "b";
String s3 = "c";
String s = s1 + s2 + s3; // abc
```

Normalement, une implémentation du compilateur effectuera la concaténation ci-dessus en utilisant des méthodes impliquant un `StringBuilder` sous le capot. Une fois compilé, le code ressemblerait à celui ci-dessous:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append("c").toString();
```

`StringBuilder` dispose de plusieurs méthodes surchargées pour ajouter différents types, par exemple, pour ajouter un `int` au lieu d'une `String` . Par exemple, une implémentation peut convertir:

```
String s1 = "a";
String s2 = "b";
String s = s1 + s2 + 2; // ab2
```

à ce qui suit:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append(2).toString();
```

Les exemples ci-dessus illustrent une opération de concaténation simple effectuée efficacement à un seul endroit du code. La concaténation implique une seule instance de `StringBuilder` . Dans certains cas, une concaténation est effectuée de manière cumulative, par exemple dans une boucle:

```
String result = "";
for(int i = 0; i < array.length; i++) {
    result += extractElement(array[i]);
}
return result;
```

Dans de tels cas, l'optimisation du compilateur n'est généralement pas appliquée et chaque itération crée un nouvel objet `StringBuilder` . Cela peut être optimisé en transformant explicitement le code pour utiliser un seul `StringBuilder` :

```
StringBuilder result = new StringBuilder();
for(int i = 0; i < array.length; i++) {
    result.append(extractElement(array[i]));
}
return result.toString();
```

Un `StringBuilder` sera initialisé avec un espace vide de 16 caractères seulement. Si vous savez à l'avance que vous allez créer des chaînes plus grandes, il peut être utile de l'initialiser suffisamment à l'avance pour que le tampon interne n'ait pas besoin d'être redimensionné:

```
StringBuilder buf = new StringBuilder(30); // Default is 16 characters
buf.append("0123456789");
buf.append("0123456789"); // Would cause a reallocation of the internal buffer otherwise
String result = buf.toString(); // Produces a 20-chars copy of the string
```

Si vous produisez de nombreuses chaînes, il est conseillé de réutiliser `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());
}
```

Si (et seulement si) plusieurs threads écrivent dans le *même* tampon, utilisez StringBuffer , qui est une version `synchronized` de `StringBuilder` . Mais comme un seul thread écrit généralement dans un tampon, il est généralement plus rapide d'utiliser `StringBuilder` sans synchronisation.

Utilisation de la méthode concat ():

```
String string1 = "Hello ";
```

```
String string2 = "world";
String string3 = string1.concat(string2);  // "Hello world"
```

Cela retourne une nouvelle chaîne qui est string1 avec string2 ajouté à la fin. Vous pouvez également utiliser la méthode concat () avec des littéraux de chaîne, comme dans:

```
"My name is ".concat("Buyya");
```

Remplacement de pièces de chaînes

Deux manières de remplacer: par regex ou par correspondance exacte.

Remarque: l'objet String d'origine sera inchangé, la valeur de retour contient la chaîne modifiée.

Correspondance exacte

Remplacez un seul caractère par un autre caractère unique:

```
String replace(char oldChar, char newChar)
```

> Renvoie une nouvelle chaîne résultant du remplacement de toutes les occurrences de oldChar dans cette chaîne par newChar.

```
String s = "popcorn";
System.out.println(s.replace('p','W'));
```

Résultat:

```
WoWcorn
```

Remplacez la séquence de caractères par une autre séquence de caractères:

```
String replace(CharSequence target, CharSequence replacement)
```

> Remplace chaque sous-chaîne de cette chaîne qui correspond à la séquence cible littérale par la séquence de remplacement littérale spécifiée.

```
String s = "metal petal et al.";
System.out.println(s.replace("etal","etallica"));
```

Résultat:

```
metallica petallica et al.
```

Regex

Remarque : le regroupement utilise le caractère `$` pour référencer les groupes, par exemple `$1` .

Remplacer toutes les correspondances:

```
String replaceAll(String regex, String replacement)
```

> Remplace chaque sous-chaîne de cette chaîne qui correspond à l'expression régulière donnée avec le remplacement donné.

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Résultat:

```
spiral metallica petallica et al.
```

Remplacez le premier match uniquement:

```
String replaceFirst(String regex, String replacement)
```

> Remplace la première sous-chaîne de cette chaîne qui correspond à l'expression régulière donnée avec le remplacement donné

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Résultat:

```
spiral metallica petal et al.
```

Supprimer les espaces au début et à la fin d'une chaîne

La méthode `trim()` renvoie une nouvelle chaîne avec les espaces blancs de début et de fin supprimés.

```
String s = new String("   Hello World!!  ");
String t = s.trim();  // t = "Hello World!!"
```

Si vous `trim` une chaîne qui n'a pas d'espace à supprimer, vous recevrez la même instance String.

Notez que la méthode `trim()` a sa propre notion d'espace , qui diffère de la notion utilisée par la méthode `Character.isWhitespace()` :

- Tous les caractères de contrôle ASCII avec les codes `U+0000` à `U+0020` sont considérés comme des espaces et sont supprimés par `trim()` . Cela inclut `U+0020 'SPACE'` , `U+0009 'CHARACTER TABULATION'` , `U+000A 'LINE FEED'` et `U+000D 'CARRIAGE RETURN'` , mais aussi les caractères comme `U+0007 'BELL'` .

- Les espaces blancs Unicode tels que `U+00A0 'NO-BREAK SPACE'` ou `U+2003 'EM SPACE'` *ne* sont *pas* reconnus par `trim()` .

Stockage de pool de chaînes et de tas

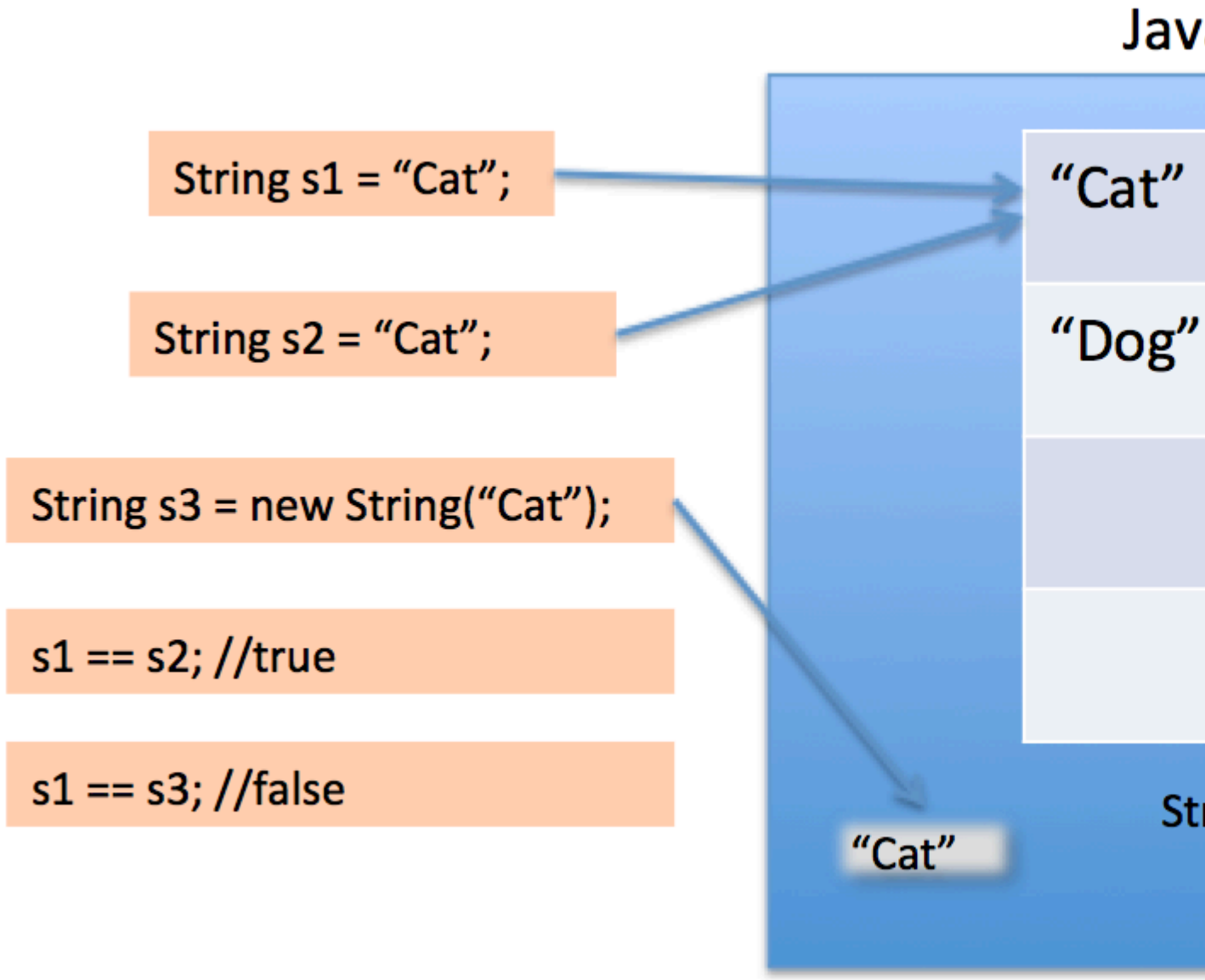
Comme beaucoup d'objets Java, **toutes les** instances de `String` sont créées sur le tas, même les littéraux. Lorsque la machine JVM trouve un littéral `String` qui n'a pas de référence équivalente dans le tas, la machine virtuelle Java crée une instance `String` correspondante sur le segment de mémoire **et** stocke également une référence à l'instance `String` nouvellement créée dans le pool String. Toute autre référence au même littéral `String` est remplacée par l'instance `String` précédemment créée dans le tas.

Regardons l'exemple suivant:

```
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);
    }
}
```

Le résultat de ce qui précède est:

```
true
true
true
true
```



Lorsque nous utilisons des guillemets doubles pour créer une chaîne, celle-ci recherche d'abord la chaîne avec la même valeur dans le pool de chaînes. Si elle est trouvée, elle renvoie simplement la référence, sinon elle crée une nouvelle chaîne dans le pool, puis renvoie la référence.

Cependant, en utilisant un nouvel opérateur, nous forçons la classe String à créer un nouvel objet String dans l’espace du tas. Nous pouvons utiliser la méthode intern () pour la placer dans le pool ou faire référence à un autre objet String d'un pool de chaînes ayant la même valeur.

Le pool de chaînes lui-même est également créé sur le tas.

Java SE 7

Avant Java 7, les **littéraux de** `String` étaient stockés dans le pool de constante d'exécution dans la zone de méthode de `PermGen` , qui avait une taille fixe.

Le pool de chaînes réside également dans `PermGen` .

Java SE 7

RFC: 6962931

> Dans JDK 7, les chaînes internes ne sont plus allouées dans la génération permanente du segment de mémoire Java, mais sont plutôt allouées dans la partie principale du segment de mémoire Java (appelées générations jeunes et anciennes), avec les autres objets créés par l'application. . Cette modification se traduira par plus de données résidant dans le segment de mémoire principal Java et moins de données dans la génération permanente, ce qui peut nécessiter un ajustement de la taille des segments. La plupart des applications ne verront que des différences relativement faibles dans l'utilisation du tas en raison de ce changement, mais les applications plus volumineuses qui chargent de nombreuses classes ou qui utilisent `String.intern()` méthode `String.intern()` verront des différences plus significatives.

Commutateur insensible à la casse

Java SE 7

`switch` lui-même ne peut pas être paramétré pour être insensible à la casse, mais s'il est absolument nécessaire, peut se comporter de manière insensible à la chaîne d'entrée en utilisant `toLowerCase()` ou `toUpperCase` :

```
switch (myString.toLowerCase()) {
     case "case1" :
        ...
     break;
     case "case2" :
        ...
     break;
}
```

Il faut se méfier

- `Locale` peuvent affecter la manière dont les cas changent !
- Il faut prendre soin de ne pas avoir de majuscules dans les étiquettes - celles-ci ne seront jamais exécutées!

Lire Cordes en ligne: https://riptutorial.com/fr/java/topic/109/cordes

Chapitre 44: Créer des images par programme

Remarques

`BufferedImage.getGraphics()` renvoie toujours `Graphics2D` .

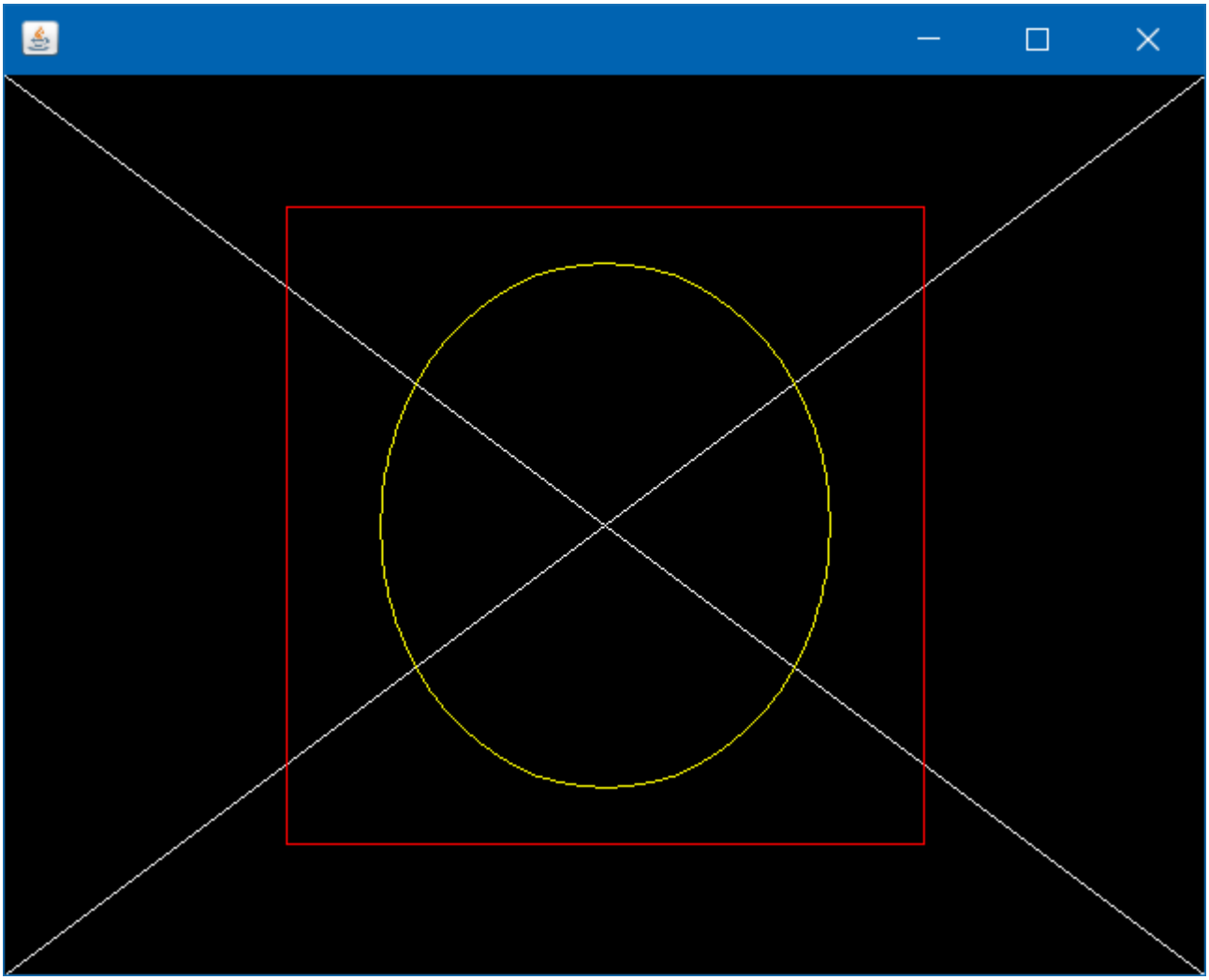
L'utilisation de `VolatileImage` peut considérablement améliorer la vitesse des opérations de dessin, mais présente également des inconvénients: son contenu peut être perdu à tout moment et il peut être nécessaire de le redessiner à partir de zéro.

Examples

Créer une image simple par programmation et l'afficher

```
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);
    }
}
```



Enregistrer une image sur le disque

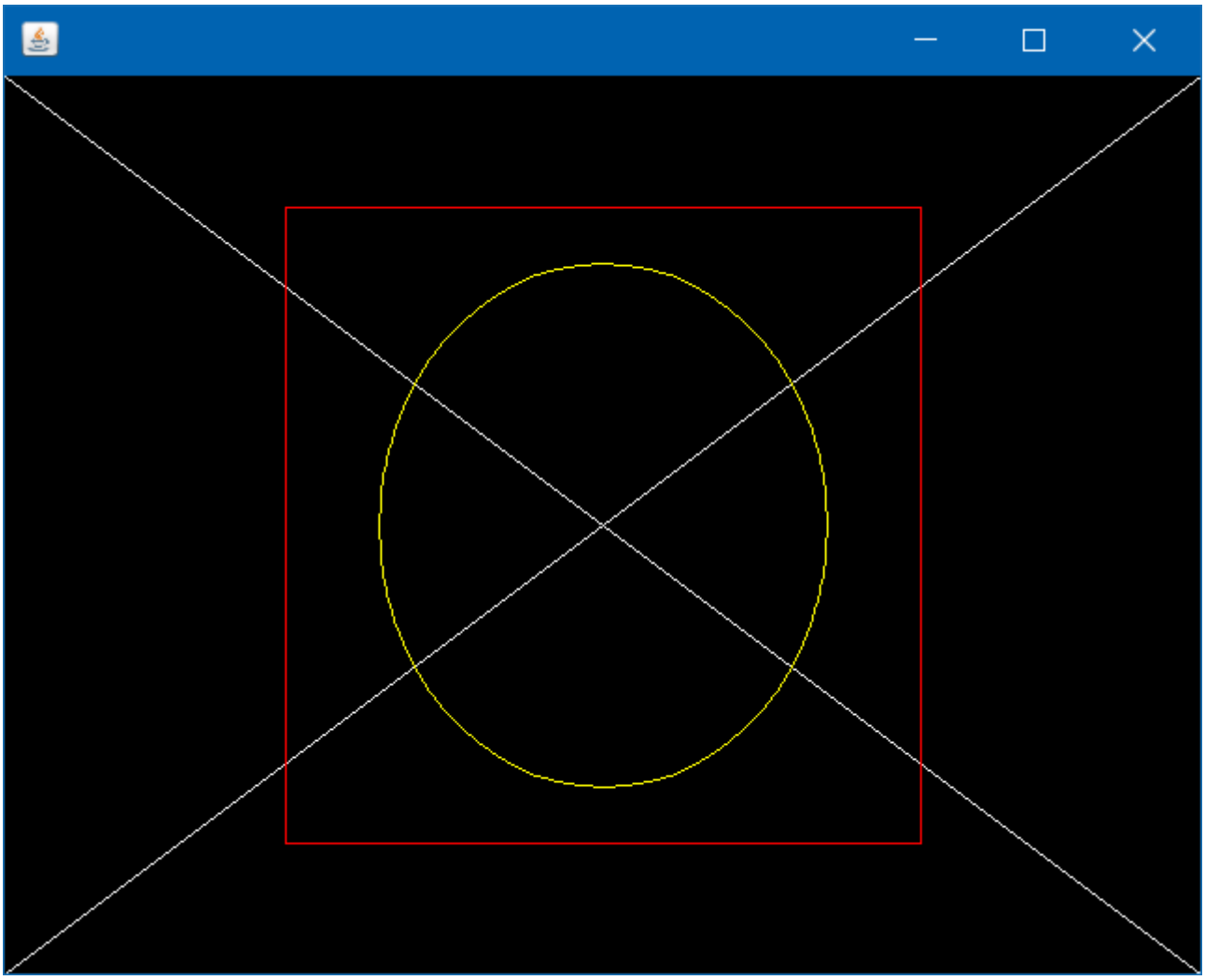
```
public static void saveImage(String destination) throws IOException {
    // method implemented in "Creating a simple image Programmatically and displaying it"
example
    BufferedImage img = createSampleImage();

    // ImageIO provides several write methods with different outputs
    ImageIO.write(img, "png", new File(destination));
}
```

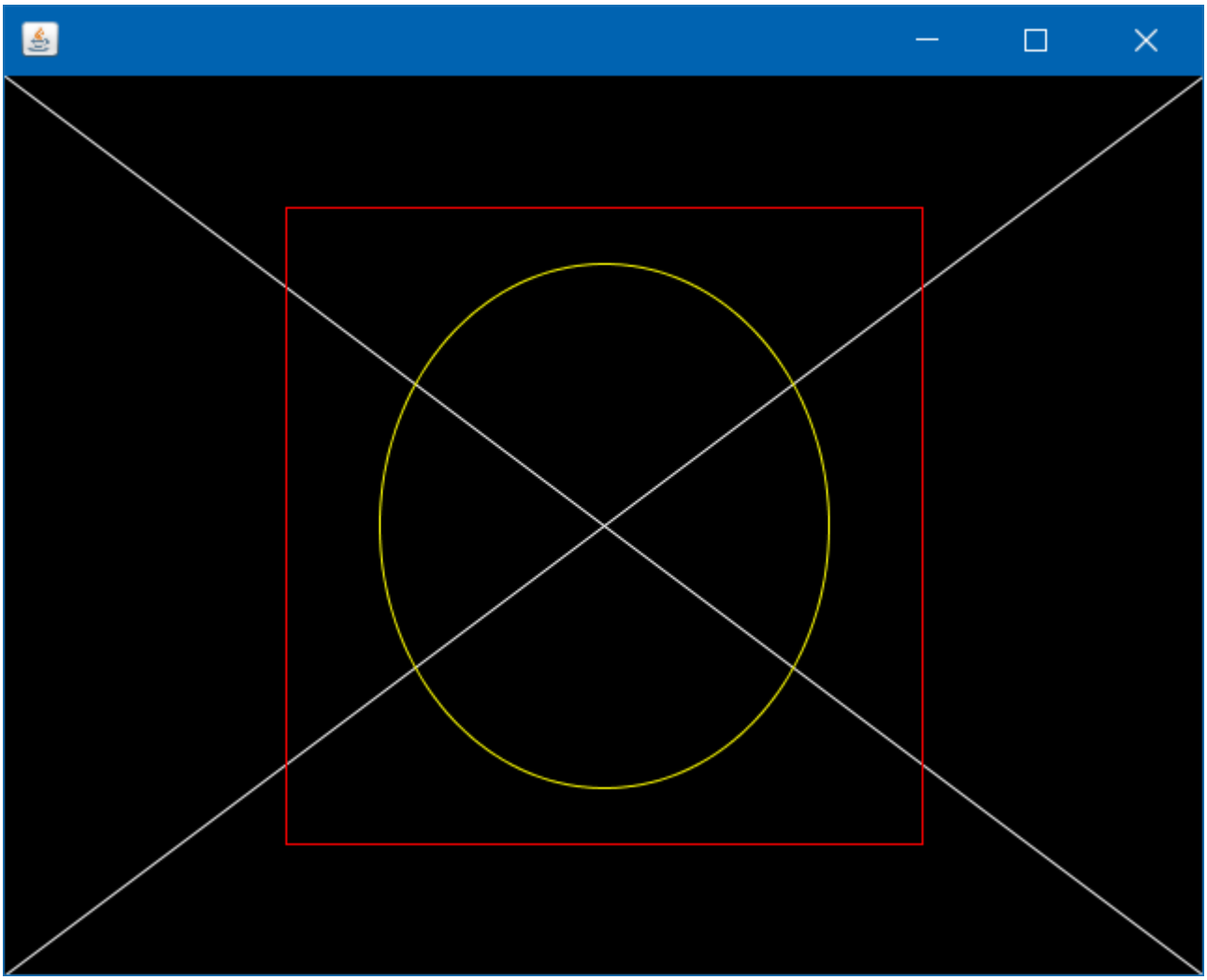
Spécification de la qualité de rendu de l'image

```
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);
}
```

Une comparaison du rendu QUALITE et VITESSE de l'image échantillon:





Création d'une image avec 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"));
```

Sortie:

Modification et réutilisation d'image avec 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"));
```

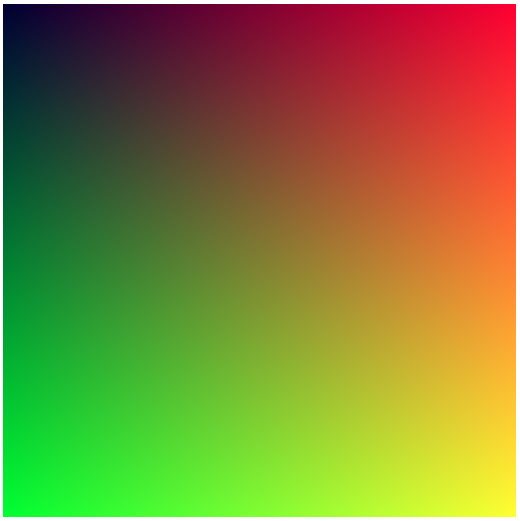
Fichier de chat original:

Fichier produit:

Définition de la couleur d'un pixel individuel dans 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"));
```

Sortie:



Comment mettre à l'échelle une image tamponnée

```
/**
* 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;
}
```

Lire Créer des images par programme en ligne: https://riptutorial.com/fr/java/topic/5482/creer-des-images-par-programme

Chapitre 45: Date classe

Syntaxe

- `Date object = new Date();`
- `Date object = new Date(long date);`

Paramètres

Paramètre	Explication
Aucun paramètre	Crée un nouvel objet Date en utilisant l'heure d'allocation (à la milliseconde près)
longue date	Crée un nouvel objet Date avec l'heure définie en nombre de millisecondes depuis "l'époque" (1er janvier 1970, 00:00:00 GMT)

Remarques

Représentation

En interne, un objet Java Date est représenté comme un objet long; c'est le nombre de millisecondes depuis une heure spécifique (appelée *époque*). La classe Java Date originale avait des méthodes pour gérer les fuseaux horaires, mais celles-ci étaient déconseillées en faveur de la nouvelle classe Calendar.

Donc, si tout ce que vous voulez faire dans votre code est de représenter une heure spécifique, vous pouvez créer une classe Date et la stocker, etc. Toutefois, si vous souhaitez imprimer une version lisible par l'homme, vous créez une classe Calendar. et utiliser son formatage pour produire des heures, des minutes, des secondes, des jours, des fuseaux horaires, etc. N'oubliez pas qu'une milliseconde spécifique est affichée sous forme d'heures différentes dans des fuseaux horaires différents; normalement, vous voulez en afficher un dans le fuseau horaire "local", mais les méthodes de formatage doivent tenir compte du fait que vous souhaiterez peut-être l'afficher pour un autre.

Sachez également que les horloges utilisées par les machines virtuelles Java n'ont généralement pas d'exactitude à la milliseconde; l'horloge peut seulement "cocher" toutes les 10 millisecondes et, par conséquent, si vous chronométrez les choses, vous ne pouvez pas compter sur des mesures précises à ce niveau.

Déclaration d'importation

```
import java.util.Date;
```

La classe `Date` peut être importée à partir du package `java.util` .

Mise en garde

`Date` instances de `Date` étant mutables, leur utilisation peut rendre difficile l'écriture de code adapté aux threads ou donner accidentellement un accès en écriture à l'état interne. Par exemple, dans la classe ci-dessous, la méthode `getDate()` permet à l'appelant de modifier la date de la transaction:

```
public final class Transaction {
  private final Date date;

  public Date getTransactionDate() {
    return date;
  }
}
```

La solution consiste à renvoyer une copie du champ de `date` ou à utiliser les nouvelles API de `java.time` introduites dans Java 8.

La plupart des méthodes de constructeur de la classe `Date` sont obsolètes et ne doivent pas être utilisées. Dans presque tous les cas, il est conseillé d'utiliser la classe `Calendar` pour les opérations de date.

Java 8

Java 8 introduit une nouvelle API de date et heure dans le package `java.time` , y compris LocalDate et LocalTime . Les classes du package `java.time` fournissent une API révisée plus facile à utiliser. Si vous écrivez sur Java 8, nous vous encourageons vivement à utiliser cette nouvelle API. Voir Dates et heure (java.time. *) .

Examples

Créer des objets Date

```
Date date = new Date();
System.out.println(date); // Thu Feb 25 05:03:59 IST 2016
```

Ici, cet objet `Date` contient la date et l'heure actuelles de création de cet objet.

```
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` objets de `Date` sont mieux créés via une instance de `Calendar` , car l'utilisation des constructeurs de données est déconseillée et déconseillée. Pour ce faire, nous devons obtenir une instance de la classe `Calendar` partir de la méthode factory. Nous pouvons ensuite définir l'année, le mois et le jour du mois en utilisant des nombres ou, en cas de mois, des constantes fournies par la classe Calendar pour améliorer la lisibilité et réduire les erreurs.

```
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
```

Avec la date, nous pouvons également passer le temps dans l'ordre des heures, des minutes et des secondes.

Comparaison d'objets Date

Calendrier, Date et LocalDate

Java SE 8

avant, après, compareTo et égale les méthodes

```
//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 et est égal à des méthodes

```
//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)));
```

Comparaison de dates avant Java 8

Avant Java 8, les dates pouvaient être comparées à l'aide des classes java.util.Calendar et java.util.Date . La classe de date offre 4 méthodes pour comparer les dates:

- après (date quand)
- avant (date quand)
- compareTo (Date anotherDate)
- est égal à (objet obj)

`after` , `before` , les méthodes `compareTo` et `equals` comparent les valeurs renvoyées par la méthode getTime () pour chaque date.

`compareTo` méthode `compareTo` renvoie un entier positif.

- Valeur supérieure à 0: lorsque la date est postérieure à l'argument Date
- Valeur supérieure à 0: lorsque la date est antérieure à l'argument Date
- La valeur est égale à 0: lorsque la date est égale à l'argument Date

`equals` résultat peut être surprenant, comme indiqué dans l'exemple, car les valeurs, comme les millisecondes, ne sont pas initialisées avec la même valeur si elles ne sont pas explicitement données.

Depuis Java 8

Avec Java 8, un nouvel objet à utiliser avec Date est disponible java.time.LocalDate . `LocalDate` implémente ChronoLocalDate , la représentation abstraite d'une date où la chronologie, ou le système de calendrier, est enfichable.

Pour avoir la précision de la date et de l'heure, l'objet java.time.LocalDateTime doit être utilisé. `LocalDate` et `LocalDateTime` utilisent le même nom de méthode pour la comparaison.

La comparaison de dates à l'aide d'un `LocalDate` est différente de l'utilisation de `ChronoLocalDate` car la chronologie ou le système de calendrier ne sont pas pris en compte pour le premier.

Comme la plupart des applications doivent utiliser `LocalDate` , `ChronoLocalDate` n'est pas inclus dans les exemples. Plus de lecture ici .

> La plupart des applications doivent déclarer les signatures de méthode, les champs et les variables en tant que LocalDate, et non cette interface [ChronoLocalDate].

`LocalDate` a 5 méthodes pour comparer les dates:

- isAfter (ChronoLocalDate autre)
- isBefore (ChronoLocalDate autre)
- isEqual (ChronoLocalDate autre)
- compareTo (ChronoLocalDate autre)
- est égal à (objet obj)

Dans le cas du paramètre `LocalDate` , `isAfter` , `isBefore` , `isEqual` , `equals` et `compareTo` utilise maintenant cette méthode:

```
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` méthode `equals` vérifie si le paramètre reference est égal à la date en premier, alors que `isEqual` appelle directement `compareTo0` .

Dans le cas d'une autre instance de classe de `ChronoLocalDate` les dates sont comparées à l'aide du `Epoch Day` . Le nombre de jours Epoch est un nombre simple de jours où le jour 0 est 1970-01-01 (ISO).

Conversion de date en un certain format de chaîne

`format()` de la classe `SimpleDateFormat` permet de convertir un objet `Date` un objet `String` format `String` à l'aide de la *chaîne de motif* fournie.

```
Date today = new Date();

SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MMM-yy"); //pattern is specified here
System.out.println(dateFormat.format(today)); //25-Feb-16
```

Les patterns peuvent être appliqués à nouveau en utilisant `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
```

Note: Ici `mm` (petite lettre m) indique les minutes et `MM` (majuscule M) indique le mois. Portez une

attention particulière lors du formatage des années: le "Y" (`Y`) en majuscule indique la "semaine de l'année" tandis que "y" en minuscule (`y`) indique l'année.

Conversion de chaîne en date

`parse()` de la classe `SimpleDateFormat` permet de convertir un modèle `String` en objet `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
```

Il existe 4 styles différents pour le format de texte, `SHORT` , `MEDIUM` (il s'agit de la valeur par défaut), `LONG` et `FULL` , qui dépendent tous des paramètres régionaux. Si aucun paramètre régional n'est spécifié, les paramètres régionaux par défaut du système sont utilisés.

Style	Locale.US	Locale.France
COURT	30/06/09	30/06/09
MOYEN	30 juin 2009	30 juin 2009
LONGUE	30 juin 2009	30 juin 2009
PLEIN	Mardi 30 juin 2009	mardi 30 juin 2009

Une sortie de date de base

En utilisant le code suivant avec la chaîne de format `yyyy/MM/dd hh:mm.ss` , nous recevrons la sortie suivante

2016/04/19 11: 45.36

```
// define the format to use
String formatString = "yyyy/MM/dd hh:mm.ss";

// get a current date object
Date date = Calendar.getInstance().getTime();

// create the formatter
SimpleDateFormat simpleDateFormat = new SimpleDateFormat(formatString);

// format the date
String formattedDate = simpleDateFormat.format(date);

// print it
System.out.println(formattedDate);


// single-line version of all above code
System.out.println(new SimpleDateFormat("yyyy/MM/dd
hh:mm.ss").format(Calendar.getInstance().getTime()));
```

Convertir la représentation de chaîne formatée de la date en objet Date

Cette méthode peut être utilisée pour convertir une représentation de chaîne formatée d'une date en objet `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;
    }
```

Créer une date spécifique

Bien que la classe Java Date ait plusieurs constructeurs, vous remarquerez que la plupart sont obsolètes. La seule façon acceptable de créer directement une instance Date est d'utiliser le constructeur vide ou de le faire en un temps long (nombre de millisecondes depuis le temps de base standard). Ni l'un ni l'autre n'est pratique à moins que vous recherchiez la date actuelle ou que vous ayez déjà une autre instance de date en main.

Pour créer une nouvelle date, vous aurez besoin d'une instance de calendrier. De là, vous pouvez définir l'instance du calendrier sur la date dont vous avez besoin.

```
Calendar c = Calendar.getInstance();
```

Cela retourne une nouvelle instance de calendrier définie à l'heure actuelle. Le calendrier a de nombreuses méthodes pour modifier sa date et son heure ou pour le configurer directement. Dans ce cas, nous allons définir une date spécifique.

```
c.set(1974, 6, 2, 8, 0, 0);
Date d = c.getTime();
```

La méthode `getTime` renvoie l'instance Date dont nous avons besoin. Gardez à l'esprit que les méthodes du jeu de calendriers ne définissent qu'un ou plusieurs champs, mais ne les définissent pas tous. En d'autres termes, si vous définissez l'année, les autres champs restent inchangés.

PIÈGE

Dans de nombreux cas, cet extrait de code remplit son objectif, mais gardez à l'esprit que deux parties importantes de la date / heure ne sont pas définies.

- les paramètres `(1974, 6, 2, 8, 0, 0)` sont interprétés dans le fuseau horaire par défaut, défini ailleurs,
- les millisecondes ne sont pas définies sur zéro, mais remplies à partir de l'horloge système au moment de la création de l'instance de calendrier.

Objets Java 8 LocalDate et LocalDateTime

Les objets Date et LocalDate **ne peuvent pas** être convertis *exactement* entre eux car un objet Date représente à la fois un jour et une heure spécifiques, tandis qu'un objet LocalDate ne contient pas d'informations sur l'heure ou le fuseau horaire. Cependant, il peut être utile de convertir entre les deux si vous ne vous souciez que des informations de date réelles et non des informations de temps.

Crée une 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());
```

Crée 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 à date et vice-versa

```
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 et vice-versa

```
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());
```

Fuseaux horaires et java.util.Date

Un objet `java.util.Date` *n'a pas de* concept de fuseau horaire.

- Il est impossible de **définir** un fuseau horaire pour une date
- Il est impossible de **modifier** le fuseau horaire d'un objet Date
- Un objet Date créé avec le `new Date()` constructeur par défaut `new Date()` sera initialisé avec l'heure actuelle dans le fuseau horaire par défaut du système

Cependant, il est possible d'afficher la date représentée par le moment décrit par l'objet Date dans un fuseau horaire différent en utilisant, par exemple, `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));
```

Sortie:

```
Central European Time
2016-07-21 22:50:56
2016-07-21 21:50:56
```

Convertissez java.util.Date en java.sql.Date

`java.sql.Date` conversion `java.util.Date` vers `java.sql.Date` est généralement nécessaire lorsqu'un objet Date doit être écrit dans une base de données.

`java.sql.Date` est un wrapper de valeur en millisecondes et est utilisé par `JDBC` pour identifier un type de `SQL DATE`

Dans l'exemple ci-dessous, nous utilisons le constructeur `java.util.Date()` , qui crée un objet Date et l'initialise pour représenter l'heure à la milliseconde près. Cette date est utilisée dans la méthode `convert(java.util.Date utilDate)` pour renvoyer un objet `java.sql.Date`

Exemple

```
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;
    }

}
```

Sortie

```
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` a à la fois des informations de date et d'heure, alors que `java.sql.Date` n'a que des informations de date

Heure locale

Pour utiliser uniquement la partie heure d'une Date, utilisez LocalTime. Vous pouvez instancier un objet LocalTime de deux manières

1. `LocalTime time = LocalTime.now();`
2. `time = LocalTime.MIDNIGHT;`
3. `time = LocalTime.NOON;`
4. `time = LocalTime.of(12, 12, 45);`

`LocalTime` également une méthode intégrée à toString qui affiche très bien le format.

```
System.out.println(time);
```

vous pouvez également obtenir, ajouter et soustraire des heures, des minutes, des secondes et des nanosecondes de l'objet LocalTime, c'est-à-dire

```
time.plusMinutes(1);
time.getMinutes();
time.minusMinutes(1);
```

Vous pouvez le transformer en un objet Date avec le code suivant:

```
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);
```

cette classe fonctionne très bien dans une classe de minuterie pour simuler un réveil.

Lire Date classe en ligne: https://riptutorial.com/fr/java/topic/164/date-classe

Chapitre 46: Dates et heure (java.time. *)

Examples

Manipulations de date simples

Obtenez la date actuelle

```
LocalDate.now()
```

Obtenez la date d'hier.

```
LocalDate y = LocalDate.now().minusDays(1);
```

Obtenez la date de demain

```
LocalDate t = LocalDate.now().plusDays(1);
```

Obtenez une date précise.

```
LocalDate t = LocalDate.of(1974, 6, 2, 8, 30, 0, 0);
```

Outre les méthodes `plus` et `minus` , il existe un ensemble de méthodes "with" qui peuvent être utilisées pour définir un champ particulier sur une instance de `LocalDate` .

```
LocalDate.now().withMonth(6);
```

L'exemple ci-dessus renvoie une nouvelle instance avec le mois défini en juin (cela diffère de `java.util.Date` où `setMonth` été indexé à 0 le 5 juin).

Comme les manipulations de LocalDate renvoient des instances LocalDate immuables, ces méthodes peuvent également être chaînées.

```
LocalDate ld = LocalDate.now().plusDays(1).plusYears(1);
```

Cela nous donnerait la date de demain dans un an.

Date et l'heure

Date et heure sans informations de fuseau horaire

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 27, 8, 0);
LocalDateTime now = LocalDateTime.now();
LocalDateTime parsed = LocalDateTime.parse("2016-07-27T07:00:00");
```

Date et heure avec informations de fuseau horaire

```
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]");
```

Date et heure avec des informations de décalage (c.-à-d. Aucune modification de l'heure d'été prise en compte)

```
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");
```

Opérations sur les dates et les heures

```
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]"))
```

Instant

Représente un instant dans le temps. Peut être considéré comme un wrapper autour d'un horodatage 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
```

Utilisation de différentes classes d'API Date Time

L'exemple suivant comporte également des explications nécessaires à la compréhension de l'exemple.

```
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);

}
}
```

Formatage de la date et de l'heure

Avant Java 8, il y avait des classes `DateFormat` et `SimpleDateFormat` dans le package `java.text` et ce code hérité continuerait à être utilisé pendant un `java.text` temps.

Mais Java 8 offre une approche moderne du traitement du formatage et de l'analyse.

Lors du formatage et de l'analyse, vous transmettez d'abord un objet `String` à `DateTimeFormatter` et l'utilisez à son tour pour le formatage ou l'analyse.

```
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));
    }
}
```

Une remarque importante, au lieu d'utiliser des modèles personnalisés, est une bonne pratique d'utiliser des formateurs prédéfinis. Votre code semble plus clair et l'utilisation d'ISO8061 vous aidera certainement à long terme.

Calculer la différence entre 2 LocalDates

Utilisez `LocalDate` et `ChronoUnit` :

```
LocalDate d1 = LocalDate.of(2017, 5, 1);
LocalDate d2 = LocalDate.of(2017, 5, 18);
```

maintenant, puisque la méthode `between` l'énumérateur `ChronoUnit` prend 2 paramètres `Temporal` pour que vous puissiez transmettre sans problème les instances de `LocalDate`

```
long days = ChronoUnit.DAYS.between(d1, d2);
System.out.println( days );
```

Lire Dates et heure (java.time. *) en ligne: https://riptutorial.com/fr/java/topic/4813/dates-et-heure--java-time----

Chapitre 47: Démonter et décompiler

Syntaxe

- javap [options] <classes>

Paramètres

prénom	La description
`<classes>`	Liste des classes à démonter. Peut - être dans les deux `package1.package2.Classname` format *ou* `package1/package2/Classname` format. N'incluez **pas** l'extension `.class` .
`-help` , `--help` , `-?`	Imprimer ce message d'utilisation
`-version`	Information sur la version
`-v` , `-verbose`	Imprimer des informations supplémentaires
`-l`	Imprimer le numéro de ligne et les tables de variables locales
`-public`	Afficher uniquement les classes publiques et les membres
`-protected`	Afficher les classes et les membres protégés / publics
`-package`	Afficher les paquetages / protected / classes publiques et les membres (par défaut)
`-p` , `-private`	Afficher toutes les classes et tous les membres
`-c`	Démonter le code
`-s`	Imprimer les signatures de type internes
`-sysinfo`	Afficher les informations système (chemin, taille, date, hachage MD5) de la classe en cours de traitement
`-constants`	Afficher les constantes finales
`-classpath <path>`	Indiquez l'emplacement des fichiers de classe d'utilisateurs
`-cp <path>`	Indiquez l'emplacement des fichiers de classe d'utilisateurs
`-bootclasspath <path>`	Remplacer l'emplacement des fichiers de classe bootstrap

Examples

Affichage du bytecode avec javap

Si vous voulez voir le bytecode généré pour un programme Java, vous pouvez utiliser la commande `javap` fournie pour l'afficher.

En supposant que nous ayons le fichier source Java suivant:

```
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");
    }
}
```

Après la compilation du fichier source, l'utilisation la plus simple est la suivante:

```
cd <directory containing classes> (e.g. target/classes)
javap com/stackoverflow/documentation/SpringExample
```

Qui produit la sortie

```
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();
}
```

Cela répertorie toutes les méthodes non privées de la classe, mais cela n'est pas particulièrement utile dans la plupart des cas. La commande suivante est beaucoup plus utile:

```
javap -p -c -s -constants -l -v com/stackoverflow/documentation/HelloWorldService
```

Qui produit la sortie:

```
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()
```

Lire Démonter et décompiler en ligne: https://riptutorial.com/fr/java/topic/2318/demonter-et-decompiler

Chapitre 48: Déploiement Java

Introduction

Il existe une variété de technologies pour «empaqueter» les applications Java, les applications Web, etc., en vue de leur déploiement sur la plate-forme sur laquelle elles s'exécuteront. Ils vont de la simple bibliothèque ou des fichiers `JAR` exécutables, des fichiers `WAR` et `EAR` aux programmes d'installation et aux exécutables autonomes.

Remarques

Au niveau le plus fondamental, un programme Java peut être déployé en copiant une classe compilée (un fichier ".class") ou une arborescence de répertoires contenant des classes compilées. Cependant, Java est normalement déployé de l'une des manières suivantes:

- En copiant un fichier JAR ou une collection de fichiers JAR sur le système où ils seront exécutés; par exemple en utilisant `javac` .
- En copiant ou en téléchargeant un fichier WAR, EAR ou un fichier similaire dans un "conteneur de servlets" ou un "serveur d'applications".
- En exécutant une sorte d'installateur d'application qui automatise ce qui précède. Le programme d'installation peut également installer un environnement JRE intégré.
- En plaçant les fichiers JAR de l'application sur un serveur Web pour permettre leur lancement à l'aide de Java WebStart.

L'exemple de création de fichiers JAR, WAR et EAR résume les différentes façons de créer ces fichiers.

Il existe de nombreux outils open source et commerciaux "installer generator" et "EXE generator" pour Java. De même, il existe des outils pour masquer les fichiers de classe Java (pour rendre le reverse engineering plus difficile) et pour ajouter une vérification de licence d'exécution. Tout cela est hors de portée pour la documentation "Java Programming Language".

Examples

Créer un fichier JAR exécutable à partir de la ligne de commande

Pour créer un pot, vous avez besoin d'un ou plusieurs fichiers de classe. Cela devrait avoir une méthode principale si elle doit être exécutée par un double clic.

Pour cet exemple, nous utiliserons:

```
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);
    }

}
```

Il a été nommé HelloWorld.java

Ensuite, nous voulons compiler ce programme.

Vous pouvez utiliser n'importe quel programme que vous voulez faire. Pour exécuter à partir de la ligne de commande, consultez la documentation sur la compilation et l'exécution de votre premier programme Java.

Une fois que vous avez HelloWorld.class, créez un nouveau dossier et appelez-le comme vous voulez.

Créez un autre fichier appelé manifest.txt et collez-le

```
Main-Class: HelloWorld
Class-Path: HelloWorld.jar
```

Placez-le dans le même dossier avec HelloWorld.class
Utilisez la ligne de commande pour créer votre répertoire actuel (`cd C:\Your\Folder\Path\Here` sur Windows).

Utilisez Terminal et changez de répertoire vers le répertoire (`cd /Users/user/Documents/Java/jarfolder` sur Mac) votre dossier

Lorsque cela est fait, tapez `jar -cvfm HelloWorld.jar manifest.txt HelloWorld.class` et appuyez sur Entrée. Cela crée un fichier jar (dans le dossier avec votre manifeste et HelloWorld.class) en utilisant les fichiers .class spécifiés et nommés HelloWorld.jar. Voir la section Syntaxe pour plus d'informations sur les options (comme -m et -v).
Après ces étapes, accédez à votre répertoire avec le fichier manifeste et vous devriez trouver HelloWorld.jar
Cliquer dessus devrait afficher `Hello, World` dans une zone de texte.

Création de fichiers JAR, WAR et EAR

Les types de fichiers JAR, WAR et EAR sont fondamentalement des fichiers ZIP avec un fichier "manifest" et (pour les fichiers WAR et EAR) une structure de répertoire / fichier interne particulière.

La méthode recommandée pour créer ces fichiers consiste à utiliser un outil de génération spécifique à Java qui «comprend» les exigences pour les types de fichiers respectifs. Si vous n'utilisez pas d'outil de construction, l'IDE "export" est la prochaine option à essayer.

(*Note éditoriale: les descriptions de la façon de créer ces fichiers sont mieux placées dans la documentation des outils respectifs. Placez-les là-bas. Veuillez faire preuve de retenue et ne pas les casser dans cet exemple!*)

Création de fichiers JAR et WAR à l'aide de Maven

Créer un fichier JAR ou WAR à l'aide de Maven consiste simplement à placer l'élément `<packaging>` correct dans le fichier POM; par exemple,

```
<packaging>jar</packaging>
```

ou

```
<packaging>war</packaging>
```

Pour plus de détails. Maven peut être configuré pour créer des fichiers JAR "exécutables" en ajoutant les informations requises sur la classe du point d'entrée et les dépendances externes en tant que propriétés du plug-in maven jar. Il existe même un plugin pour créer des fichiers "uberJAR" qui combinent une application et ses dépendances en un seul fichier JAR.

S'il vous plaît se référer à la documentation Maven (http://www.riptutorial.com/topic/898) pour plus d'informations.

Création de fichiers JAR, WAR et EAR à l'aide d'Ant

L'outil de génération Ant a des "tâches" distinctes pour la construction de JAR, WAR et EAR. S'il vous plaît se référer à la documentation Ant (http://www.riptutorial.com/topic/4223) pour plus d'informations.

Création de fichiers JAR, WAR et EAR à l'aide d'un IDE

Les trois IDE Java les plus populaires ont tous un support intégré pour la création de fichiers de déploiement. La fonctionnalité est souvent décrite comme "exportation".

- Eclipse - http://www.riptutorial.com/topic/1143
- NetBeans - http://www.riptutorial.com/topic/5438
- Intellij-IDEA - Exportation

Création de fichiers JAR, WAR et EAR à l'aide de la commande `jar` .

Il est également possible de créer ces fichiers "à la main" en utilisant la commande `jar` . Il s'agit simplement d'assembler une arborescence de fichiers avec les fichiers de composants appropriés au bon endroit, de créer un fichier manifeste et d'exécuter `jar` pour créer le fichier JAR.

Reportez-vous à la rubrique Commande `jar` (Création et modification de fichiers JAR) pour plus d'informations.

Introduction à Java Web Start

Les tutoriels Oracle Java résument Web Start comme suit:

> Le logiciel Java Web Start permet de lancer des applications complètes en un seul clic. Les utilisateurs peuvent télécharger et lancer des applications, telles qu'un tableur complet ou un client de discussion sur Internet, sans passer par de longues procédures d'installation.

Les autres avantages de Java Web Start sont la prise en charge du code signé et la déclaration explicite des dépendances de plate-forme, ainsi que la prise en charge de la mise en cache du code et du déploiement des mises à jour des applications.

Java Web Start est également appelé JavaWS et JAWS. Les principales sources d'information sont:

- Les tutoriels Java - Leçon: Java Web Start
- Guide de démarrage Java Web
- FAQ Java Web Start
- Spécification JNLP
- `javax.jnlp` API Documentation
- Site des développeurs Java Web Start

Conditions préalables

À un niveau élevé, Web Start fonctionne en distribuant des applications Java empaquetées en tant que fichiers JAR à partir d'un serveur Web distant. Les prérequis sont:

- Une installation Java préexistante (JRE ou JDK) sur la machine cible sur laquelle l'application doit s'exécuter. Java 1.2.2 ou supérieur est requis:
    - A partir de Java 5.0, la prise en charge de Web Start est incluse dans JRE / JDK.
    - Pour les versions antérieures, la prise en charge de Web Start est installée séparément.
    - L'infrastructure Web Start inclut du Javascript qui peut être inclus dans une page Web pour aider l'utilisateur à installer les logiciels nécessaires.
- Le serveur Web qui héberge le logiciel doit être accessible à la machine cible.
- Si l'utilisateur va lancer une application Web Start en utilisant un lien dans une page Web, alors:

- ils ont besoin d'un navigateur Web compatible, et
- pour les navigateurs modernes (sécurisés), ils doivent savoir comment indiquer au navigateur d'autoriser l'exécution de Java ... sans compromettre la sécurité du navigateur Web.

Un exemple de fichier JNLP

L'exemple suivant est destiné à illustrer les fonctionnalités de base de JNLP.

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="https://www.example.com/demo"
    href="demo_webstart.jnlp">
    <information>
        <title>Demo</title>
        <vendor>The Example.com Team</vendor>
    </information>
    <resources>
        <!-- Application Resources -->
        <j2se version="1.7+" href="http://java.sun.com/products/autodl/j2se"/>
        <jar href="Demo.jar" main="true"/>
    </resources>
    <application-desc
         name="Demo Application"
         main-class="com.example.jwsdemo.Main"
         width="300"
         height="300">
     </application-desc>
     <update check="background"/>
</jnlp>
```

Comme vous pouvez le voir, un fichier JNLP basé sur XML et les informations sont toutes contenues dans l'élément `<jnlp>` .

- L'attribut `spec` donne la version de la spécification JNPL à laquelle ce fichier est conforme.
- L'attribut `codebase` donne l'URL de base pour la résolution des URL `href` relatives dans le reste du fichier.
- L'attribut `href` donne l'URL définitive pour ce fichier JNLP.
- L'élément `<information>` contient les métadonnées de l'application, y compris son titre, ses auteurs, sa description et son site Web d'aide.
- L'élément `<resources>` décrit les dépendances de l'application, y compris la version Java requise, la plate-forme OS et les fichiers JAR.
- L'élément `<application-desc>` (ou `<applet-desc>`) fournit les informations nécessaires au lancement de l'application.

Configuration du serveur Web

Le serveur Web doit être configuré pour utiliser le `application/x-java-jnlp-file` comme type MIME pour les fichiers `.jnlp` .

Le fichier JNLP et les fichiers JAR de l'application doivent être installés sur le serveur Web afin qu'ils soient disponibles à l'aide des URL indiquées par le fichier JNLP.

Activation du lancement via une page Web

Si l'application doit être lancée via un lien Web, la page contenant le lien doit être créée sur le serveur Web.

- Si vous pouvez supposer que Java Web Start est déjà installé sur l'ordinateur de l'utilisateur, la page Web doit simplement contenir un lien permettant de lancer l'application. Par exemple.

```
<a href="https://www.example.com/demo_webstart.jnlp">Launch the application</a>
```

- Sinon, la page doit également inclure des scripts pour détecter le type de navigateur utilisé par l'utilisateur et demander à télécharger et installer la version requise de Java.

REMARQUE: Il est déconseillé d'encourager les utilisateurs à encourager l'installation de Java de cette manière ou même d'activer Java dans leurs navigateurs Web pour que le lancement de la page Web JNLP fonctionne.

Lancer des applications Web Start à partir de la ligne de commande

Les instructions pour lancer une application Web Start à partir de la ligne de commande sont simples. En supposant que l'utilisateur dispose d'un JRE ou JDK Java 5.0, il suffit de l'exécuter:

```
$ javaws <url>
```

où `<url>` est l'URL du fichier JNLP sur le serveur distant.

Créer un UberJAR pour une application et ses dépendances

Une exigence commune pour une application Java est qu'elle peut être déployée en copiant un seul fichier. Pour les applications simples qui ne dépendent que des bibliothèques de classes Java SE standard, cette exigence est satisfaite en créant un fichier JAR contenant toutes les classes d'applications (compilées).

Les choses ne sont pas si simples si l'application dépend de bibliothèques tierces. Si vous placez simplement des fichiers JAR de dépendance dans un JAR d'application, le chargeur de classe Java standard ne pourra pas trouver les classes de bibliothèque et votre application ne démarrera pas. Au lieu de cela, vous devez créer un fichier JAR unique contenant les classes d'application et les ressources associées avec les classes et les ressources de dépendance. Celles-ci doivent être organisées en un seul espace de noms pour le chargeur de classe.

Un tel fichier JAR est souvent appelé UberJAR.

Créer un UberJAR en utilisant la commande

"jar"

La procédure de création d'un UberJAR est simple. (J'utiliserai les commandes Linux pour plus de simplicité. Les commandes doivent être identiques pour Mac OS et similaires pour Windows.)

1. Créez un répertoire temporaire et modifiez-le.

   ```
   $ mkdir tempDir
   $ cd tempDir
   ```

2. Pour chaque fichier JAR dépendant, *dans l'ordre inverse* où ils doivent apparaître dans le chemin de classe de l'application, utilisez la commande `jar` pour décompresser le fichier JAR dans le répertoire temporaire.

   ```
   $ jar -xf <path/to/file.jar>
   ```

   Faire cela pour plusieurs fichiers JAR *superposera le* contenu des fichiers JAR.

3. Copiez les classes d'application de l'arborescence de construction dans le répertoire temporaire

   ```
   $ cp -r path/to/classes .
   ```

4. Créez le UberJAR à partir du contenu du répertoire temporaire:

   ```
   $ jar -cf ../myApplication.jar
   ```

   Si vous créez un fichier JAR exécutable, incluez un fichier MANIFEST.MF approprié, comme décrit ici.

Création d'un UberJAR à l'aide de Maven

Si votre projet est construit avec Maven, vous pouvez le faire pour créer un UberJAR en utilisant les plugins "maven-assembly" ou "maven-shade". Voir la rubrique Maven Assembly (dans la documentation Maven) pour plus de détails.

Les avantages et inconvénients des UberJAR

Certains avantages des UberJAR sont évidents:

- Un UberJAR est facile à distribuer.
- Vous ne pouvez pas casser les dépendances de bibliothèque pour un UberJAR, car les bibliothèques sont autonomes.

De plus, si vous utilisez un outil approprié pour créer UberJAR, vous aurez la possibilité d'exclure les classes de bibliothèque qui ne sont pas utilisées à partir du fichier JAR. Cependant, cela se fait généralement par une analyse statique des classes. Si votre application utilise la réflexion, le traitement des annotations et des techniques similaires, vous devez faire attention à ce que les classes ne soient pas exclues de manière incorrecte.

Les UberJAR présentent également certains inconvénients:

- Si vous avez beaucoup de UberJAR avec les mêmes dépendances, chacune contiendra une copie des dépendances.
- Certaines bibliothèques open source ont des licences qui *peuvent* empêcher [1] leur utilisation dans un UberJAR.

---

1 - Certaines licences de bibliothèque open source vous permettent d'utiliser uniquement la bibliothèque de l'utilisateur final qui peut remplacer une version de la bibliothèque par une autre. Les UberJAR peuvent rendre difficile le remplacement des dépendances de version.

Lire Déploiement Java en ligne: https://riptutorial.com/fr/java/topic/6840/deploiement-java

Chapitre 49: Des applets

Introduction

Les applets font partie de Java depuis sa sortie officielle et ont été utilisés pour enseigner Java et la programmation pendant plusieurs années.

Au cours des dernières années, on s'est efforcé de s'éloigner d'Applets et d'autres plug-ins de navigateur, certains navigateurs les bloquant ou ne les supportant pas activement.

En 2016, Oracle a annoncé son intention de déprécier le plug-in, en passant à un Web sans plug-in.

Des API plus récentes et meilleures sont maintenant disponibles

Remarques

Une applet est une application Java qui s'exécute normalement dans un navigateur Web. L'idée de base est d'interagir avec l'utilisateur sans avoir besoin d'interagir avec le serveur et de transférer des informations. Ce concept a connu un grand succès en 2000, lorsque la communication par Internet était lente et coûteuse.

Une applet propose cinq méthodes pour contrôler leur cycle de vie.

nom de la méthode	la description
`init()`	est appelé une fois lorsque l'applet est chargé
`destroy()`	est appelé une fois lorsque l'applet est retiré de la mémoire
`start()`	est appelé chaque fois que l'applet est visible
`stop()`	est appelé chaque fois que l'applet est recouvert par d'autres fenêtres
`paint()`	est appelée en cas de besoin ou déclenchée manuellement en appelant `repaint()`

Examples

Applet Minimal

Une applet très simple dessine un rectangle et imprime quelque chose à l'écran.

```
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 d'une applet s'étend depuis `javax.swing.JApplet` .

Java SE 1.2

Avant Java 1.2 et l'introduction du swing, les applets API s'étaient étendus à partir de `java.applet.Applet` .

Les applets ne nécessitent pas de méthode principale. Le point d'entrée est contrôlé par le cycle de vie. Pour les utiliser, ils doivent être incorporés dans un document HTML. C'est aussi le point où leur taille est définie.

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

Créer une interface graphique

Les applets peuvent facilement être utilisés pour créer une interface graphique. Ils agissent comme un `Container` et ont une méthode `add()` qui prend tout composant `awt` ou `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);
    }
}
```

Ouvrir des liens depuis l'applet

Vous pouvez utiliser la méthode `getAppletContext()` pour obtenir un objet `AppletContext` qui vous permet de demander au navigateur d'ouvrir un lien. Pour cela, vous utilisez la méthode `showDocument()` . Son second paramètre indique au navigateur d'utiliser une nouvelle fenêtre `_blank` ou celle qui affiche 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);
    }
}
```

Chargement d'images, audio et autres ressources

Les applets Java peuvent charger différentes ressources. Mais comme ils s'exécutent dans le navigateur Web du client, vous devez vous assurer que ces ressources sont accessibles. Les applets ne peuvent pas accéder aux ressources client en tant que système de fichiers local.

Si vous souhaitez charger des ressources à partir de la même URL que l'applet est stockée, vous pouvez utiliser la méthode `getCodeBase()` pour récupérer l'URL de base. Pour charger des ressources, les applets proposent les méthodes `getImage()` et `getAudioClip()` pour charger des

images ou des fichiers audio.

Charger et afficher une image

```
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);
        }
}
```

Charger et lire un fichier 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();
    }
}
```

Charger et afficher un fichier texte

```
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 */ }
    }
}
```

Lire Des applets en ligne: https://riptutorial.com/fr/java/topic/5503/des-applets

Chapitre 50: Des listes

Introduction

Une *liste* est une collection *ordonnée* de valeurs. En Java, les listes font partie de Java Collections Framework . Les listes implémentent l'interface `java.util.List` , qui étend `java.util.Collection` .

Syntaxe

- ls.add (élément E); // Ajoute un élément
- ls.remove (élément E); // Supprime un élément
- for (élément E: ls) {} // Itère sur chaque élément
- ls.toArray (new String [ls.length]); // Convertit une liste de chaînes en un tableau de chaînes
- ls.get (int index); // Renvoie l'élément à l'index spécifié.
- ls.set (int index, E element); // Remplace l'élément à une position spécifiée.
- ls.isEmpty (); // Renvoie true si le tableau ne contient aucun élément, sinon il retourne false.
- ls.indexOf (objet o); // Renvoie l'index du premier emplacement de l'élément spécifié o ou, s'il n'est pas présent, renvoie -1.
- ls.lastIndexOf (objet o); // Renvoie l'index du dernier emplacement de l'élément spécifié o ou, s'il n'est pas présent, renvoie -1.
- ls.size (); // Renvoie le nombre d'éléments de la liste.

Remarques

Une liste est un objet qui stocke une collection ordonnée de valeurs. "Ordonné" signifie que les valeurs sont stockées dans un ordre particulier - un élément vient en premier, un vient en second, et ainsi de suite. Les valeurs individuelles sont communément appelées "éléments". Les listes Java fournissent généralement ces fonctionnalités:

- Les listes peuvent contenir zéro ou plusieurs éléments.
- Les listes peuvent contenir des valeurs en double. En d'autres termes, un élément peut être inséré dans une liste plus d'une fois.
- Les listes stockent leurs éléments dans un ordre particulier, ce qui signifie qu'un élément vient en premier, un vient ensuite, etc.
- Chaque élément a un *index* indiquant sa position dans la liste. Le premier élément a l'index 0, le prochain a l'index 1, etc.
- Les listes permettent d'insérer des éléments au début, à la fin ou à n'importe quel index de la liste.
- Tester si une liste contient une valeur particulière signifie généralement examiner chaque élément de la liste. Cela signifie que le temps pour effectuer cette vérification est O (n) , proportionnel à la taille de la liste.

L'ajout d'une valeur à une liste autre que la fin déplace tous les éléments suivants "vers le bas" ou "vers la droite". En d'autres termes, l'ajout d'un élément à l'index *n* déplace l'élément qui était à

l'index *n* pour indexer *n + 1* , et ainsi de suite. Par exemple:

```
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

Trier une liste générique

La classe `Collections` propose deux méthodes statiques standard pour trier une liste:

- `sort(List<T> list)` applicable aux listes où `T extends Comparable<? super T>` , et
- `sort(List<T> list, Comparator<? super T> c)` applicable aux listes de tout type.

Appliquer le premier nécessite de modifier la classe des éléments de liste en cours de tri, ce qui n'est pas toujours possible. Cela peut également être indésirable car, bien qu'il fournisse le tri par défaut, d'autres ordres de tri peuvent être requis dans différentes circonstances, ou le tri n'est qu'une tâche ponctuelle.

Considérons que nous avons pour tâche de trier les objets qui sont des instances de la classe suivante:

```
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);
    }
}
```

Pour utiliser `Collections.sort(List<User> list)` nous devons modifier la classe `User` pour implémenter l'interface `Comparable` . Par exemple

```
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);
    }
}
```

(À part: de nombreuses classes Java standard telles que `String` , `Long` , `Integer` implémentent l'interface `Comparable` . Cela rend les listes de ces éléments triables par défaut et simplifie l'implémentation de `compare` ou `compareTo` dans d'autres classes.)

Avec la modification ci-dessus, nous pouvons facilement trier une liste d'objets `User` fonction de l' *ordre naturel des* classes. (Dans ce cas, nous avons défini cela comme étant basé sur les valeurs d' `id`). Par exemple:

```
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]
```

Cependant, supposons que nous voulions trier `User` objets `User` par `name` plutôt que par `id` . Supposons maintenant que nous n'avions pas été en mesure de changer la classe pour le faire mettre en œuvre `Comparable` .

C'est là que la méthode de `sort` avec l'argument `Comparator` est 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

Dans Java 8, vous pouvez utiliser un *lambda* au lieu d'une classe anonyme. Ce dernier se réduit à une seule ligne:

```
Collections.sort(users, (l, r) -> l.username.compareTo(r.username));
```

De plus, Java 8 ajoute une méthode de `sort` par défaut sur l'interface `List` , ce qui simplifie encore

le tri.

```
users.sort((l, r) -> l.username.compareTo(r.username))
```

Créer une liste

Donner un type à votre liste

Pour créer une liste, vous avez besoin d'un type (n'importe quelle classe, par exemple `String`). C'est le type de votre `List` . La `List` ne stockera que les objets du type spécifié. Par exemple:

```
List<String> strings;
```

Peut stocker `"string1"` , `"hello world!"` , `"goodbye"` , etc., mais il ne peut pas stocker `9.2` , cependant:

```
List<Double> doubles;
```

Peut stocker `9.2` , mais pas `"hello world!"` .

Initialiser votre liste

Si vous essayez d'ajouter quelque chose aux listes ci-dessus, vous obtiendrez une exception NullPointerException, car les `strings` et les `doubles` toutes deux **nulles** !

Il existe deux manières d'initialiser une liste:

Option 1: Utiliser une classe qui implémente List

`List` est une interface, ce qui signifie qu'elle n'a pas de constructeur, plutôt que des méthodes qu'une classe doit remplacer. `ArrayList` est la plus couramment utilisée `List` , bien que `LinkedList` est également commune. Donc, nous initialisons notre liste comme ceci:

```
List<String> strings = new ArrayList<String>();
```

ou

```
List<String> strings = new LinkedList<String>();
```

Java SE 7

À partir de Java SE 7, vous pouvez utiliser un *opérateur de diamant* :

```
List<String> strings = new ArrayList<>();
```

ou

```
List<String> strings = new LinkedList<>();
```

Option 2: Utiliser la classe Collections

La classe `Collections` fournit deux méthodes utiles pour créer des listes sans variable `List` :

- `emptyList()` : renvoie une liste vide.
- `singletonList(T)` : crée une liste de type T et ajoute l'élément spécifié.

Et une méthode qui utilise une `List` existante pour remplir des données dans:

- `addAll(L, T...)` : ajoute tous les éléments spécifiés à la liste passée en tant que premier paramètre.

Exemples:

```
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);
```

Opérations d'accès positionnel

L'API de liste a huit méthodes pour les opérations d'accès positionnel:

- `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)`

Donc, si nous avons une liste:

```
List<String> strings = new ArrayList<String>();
```

Et nous voulions ajouter les chaînes "Hello world!" et "Au revoir le monde!" à cela, nous le ferions en tant que tel:

```
strings.add("Hello world!");
strings.add("Goodbye world!");
```

Et notre liste comprendrait les deux éléments. Maintenant, disons que nous voulions ajouter "Programme de démarrage!" en **tête** de liste. Nous ferions ceci comme ceci:

```
strings.add(0, "Program starting!");
```

REMARQUE: le premier élément est 0.

Maintenant, si nous voulions supprimer le "Monde Au revoir!" ligne, nous pourrions le faire comme ceci:

```
strings.remove("Goodbye world!");
```

Et si nous voulions supprimer la première ligne (qui dans ce cas serait "Programme en cours!", Nous pourrions le faire comme ceci:

```
strings.remove(0);
```

Remarque:

1. L'ajout et la suppression d'éléments de liste modifient la liste, ce qui peut entraîner une `ConcurrentModificationException` si la liste est itérée simultanément.
2. L'ajout et la suppression d'éléments peuvent être `O(1)` ou `O(N)` selon la classe de liste, la méthode utilisée et si vous ajoutez / supprimez un élément au début, à la fin ou au milieu de la liste.

Afin de récupérer un élément de la liste à une position spécifiée, vous pouvez utiliser le `E get(int index);` méthode de l'API List. Par exemple:

```
strings.get(0);
```

renverra le premier élément de la liste.

Vous pouvez remplacer n'importe quel élément à une position spécifiée en utilisant l' `set(int index, E element);` . Par exemple:

```
strings.set(0,"This is a replacement");
```

Cela définira la chaîne "Ceci est un remplacement" comme premier élément de la liste.

Remarque: La méthode set remplacera l'élément à la position 0. Il n'ajoutera pas la nouvelle chaîne à la position 0 et poussera l'ancienne à la position 1.

Le `int indexOf(Object o);` renvoie la position de la première occurrence de l'objet passé en argument. S'il n'y a pas d'occurrences de l'objet dans la liste, la valeur -1 est renvoyée. Dans la suite de l'exemple précédent, si vous appelez:

```
strings.indexOf("This is a replacement")
```

le 0 devrait être renvoyé lorsque nous définissons la chaîne "Ceci est un remplacement" dans la position 0 de notre liste. Dans le cas où il y a plus d'une occurrence dans la liste quand `int indexOf(Object o);` est appelé alors comme mentionné, l'index de la première occurrence sera renvoyé. En appelant `int lastIndexOf(Object o)` vous pouvez récupérer l'index de la dernière occurrence dans la liste. Donc, si nous ajoutons un autre "Ceci est un remplacement":

```
strings.add("This is a replacement");
strings.lastIndexOf("This is a replacement");
```

Cette fois, le 1 sera renvoyé et non le 0;

Itérer sur les éléments d'une liste

Pour l'exemple, disons que nous avons une liste de type String qui contient quatre éléments: "bonjour", "comment", "sont", "vous?"

La meilleure façon de parcourir chaque élément est d'utiliser une boucle for-each:

```
public void printEachElement(List<String> list){
    for(String s : list){
        System.out.println(s);
    }
}
```

Qui imprimerait:

```
hello,
how
are
you?
```

Pour les imprimer tous dans la même ligne, vous pouvez utiliser un StringBuilder:

```
public void printAsLine(List<String> list){
    StringBuilder builder = new StringBuilder();
    for(String s : list){
        builder.append(s);
    }
    System.out.println(builder.toString());
}
```

Imprimera:

```
hello, how are you?
```

Vous pouvez également utiliser l'indexation d'élément (comme décrit dans Accès à l'élément à partir d'Index de ArrayList) pour itérer une liste. Attention: cette approche est inefficace pour les listes liées.

Suppression d'éléments de la liste B présents dans la liste A

Supposons que vous ayez 2 listes A et B, et que vous voulez supprimer de **B** tous les éléments que vous avez dans **A** la méthode dans ce cas est

```
 List.removeAll(Collection c);
```

#Exemple:

```
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);
    }
```

cela va imprimer

A: [1, 3, 4, 7, 5, 2]

B: [13, 32, 533, 3, 4, 2]

B effacé: [13, 32, 533]

Recherche d'éléments communs entre 2 listes

Supposons que vous ayez deux listes: A et B, et que vous devez trouver les éléments qui existent dans les deux listes.

Vous pouvez le faire en invoquant simplement la méthode `List.retainAll()` .

Exemple:

```
public static void main(String[] args) {
    List<Integer> numbersA = new ArrayList<>();
    List<Integer> numbersB = new ArrayList<>();
    numbersA.addAll(Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 }));
    numbersB.addAll(Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 }));

    System.out.println("A: " + numbersA);
    System.out.println("B: " + numbersB);
    List<Integer> numbersC = new ArrayList<>();
    numbersC.addAll(numbersA);
    numbersC.retainAll(numbersB);

    System.out.println("List A : " + numbersA);
    System.out.println("List B : " + numbersB);
    System.out.println("Common elements between A and B: " + numbersC);

}
```

Convertir une liste d'entiers en une liste de chaînes

```
List<Integer> nums = Arrays.asList(1, 2, 3);
List<String> strings = nums.stream()
    .map(Object::toString)
```

```
    .collect(Collectors.toList());
```

C'est:

1. Créer un flux à partir de la liste
2. Mapper chaque élément en utilisant `Object::toString`
3. Collectez les valeurs de `String` dans une `List` aide de `Collectors.toList()`

Créer, ajouter et supprimer un élément d'une liste de tableaux

`ArrayList` est l'une des structures de données intégrées à Java. C'est un tableau dynamique (où la taille de la structure de données n'a pas besoin d'être déclarée en premier) pour stocker des éléments (objets).

Il étend la classe `AbstractList` et implémente l'interface `List` . Un `ArrayList` peut contenir des éléments en double où il maintient l'ordre d'insertion. Il convient de noter que la classe `ArrayList` n'est pas synchronisée, il faut donc faire attention lors de la gestion de la concurrence avec `ArrayList` . `ArrayList` permet un accès aléatoire car le tableau fonctionne sur la base de l'index. La manipulation est lente dans `ArrayList` raison du décalage qui se produit souvent lorsqu'un élément est supprimé de la liste de tableaux.

Un `ArrayList` peut être créé comme suit:

```
List<T> myArrayList = new ArrayList<>();
```

Où `T` (Generics) est le type qui sera stocké dans `ArrayList` .

Le type de `ArrayList` peut être n'importe quel objet. Le type ne peut pas être un type primitif (utilisez plutôt leurs *classes wrapper*).

Pour ajouter un élément à `ArrayList` , utilisez la méthode `add()` :

```
myArrayList.add(element);
```

Ou pour ajouter un élément à un certain index:

```
myArrayList.add(index, element); //index of the element should be an int (starting from 0)
```

Pour supprimer un élément de `ArrayList` , utilisez la méthode `remove()` :

```
myArrayList.remove(element);
```

Ou pour supprimer un élément d'un certain index:

```
myArrayList.remove(index); //index of the element should be an int (starting from 0)
```

Remplacement sur place d'un élément List

Cet exemple concerne le remplacement d'un élément `List` tout en veillant à ce que l'élément de remplacement soit à la même position que l'élément remplacé.

Cela peut être fait en utilisant ces méthodes:

- set (int index, type T)
- int indexOf (type T)

Considérons un `ArrayList` contenant les éléments "Programme démarrant!", "Bonjour tout le monde!" et "Au revoir le monde!"

```
List<String> strings = new ArrayList<String>();
strings.add("Program starting!");
strings.add("Hello world!");
strings.add("Goodbye world!");
```

Si nous connaissons l'indice de l'élément que nous voulons remplacer, nous pouvons simplement utiliser `set` comme suit:

```
strings.set(1, "Hi world");
```

Si nous ne connaissons pas l'index, nous pouvons le rechercher en premier. Par exemple:

```
int pos = strings.indexOf("Goodbye world!");
if (pos >= 0) {
    strings.set(pos, "Goodbye cruel world!");
}
```

Remarques:

1. L'opération `set` ne provoquera pas une `ConcurrentModificationException` .
2. L'opération `set` est rapide (`O(1)`) pour `ArrayList` mais lente (`O(N)`) pour une `LinkedList` .
3. Une recherche `indexOf` sur une `ArrayList` ou `LinkedList` est lente (`O(N)`).

Rendre une liste non modifiable

La classe Collections fournit un moyen de rendre une liste non modifiable:

```
List<String> ls = new ArrayList<String>();
List<String> unmodifiableList = Collections.unmodifiableList(ls);
```

Si vous souhaitez une liste non modifiable avec un élément, vous pouvez utiliser:

```
List<String> unmodifiableList = Collections.singletonList("Only string in the list");
```

Déplacement d'objets dans la liste

La classe Collections vous permet de déplacer des objets dans la liste en utilisant différentes méthodes (ls est la liste):

Inverser une liste:

```
Collections.reverse(ls);
```

Rotation des positions des éléments dans une liste

La méthode rotate nécessite un argument entier. C'est le nombre de points à déplacer le long de la ligne. Un exemple de ceci est ci-dessous:

```
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();
```

Cela va imprimer "bonjour, comment vas-tu?"

Mélanger les éléments dans une liste

En utilisant la même liste ci-dessus, nous pouvons mélanger les éléments dans une liste:

```
Collections.shuffle(ls);
```

On peut aussi lui donner un objet java.util.Random qu'il utilise pour placer aléatoirement des objets dans les taches:

```
Random random = new Random(12);
Collections.shuffle(ls, random);
```

Liste d'implémentation des classes - Avantages et inconvénients

L'interface `List` est implémentée par différentes classes. Chacun d'entre eux a sa propre façon de le mettre en œuvre avec différentes stratégies et de fournir des avantages et des inconvénients différents.

Classes implémentant la liste

Ce sont toutes les classes `public` de Java SE 8 qui implémentent l'interface `java.util.List` :

1. **Classes abstraites:**
    - AbstractList
    - AbstractSequentialList
2. **Classes de béton:**
    - ArrayList

- AttributeList
- CopyOnWriteArrayList
- LinkedList
- Liste de rôles
- RoleUnresolvedList
- Empiler
- Vecteur

Avantages et inconvénients de chaque mise en œuvre en termes de complexité temporelle

ArrayList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

ArrayList est une implémentation de tableau redimensionnable de l'interface List. En stockant la liste dans un tableau, *ArrayList* fournit des méthodes (en plus des méthodes implémentant l'interface *List*) pour manipuler la taille du tableau.

Initialiser ArrayList d'Integer avec la taille 100

```
List<Integer> myList = new ArrayList<Integer>(100); // Constructs an empty list with the
specified initial capacity.
```

- AVANTAGES:

Les opérations size, isEmpty, ***get*** , ***set*** , iterator et listIterator s'exécutent en temps constant. Ainsi, obtenir et définir chaque élément de la liste a le même *coût* :

```
int e1 = myList.get(0);  //   \
int e2 = myList.get(10); //    | => All the same constant cost => O(1)
myList.set(2,10);        //   /
```

- LES INCONVÉNIENTS:

Étant implémenté avec un tableau (structure statique), l'ajout d'éléments sur la taille du tableau a un coût important, car une nouvelle allocation doit être effectuée pour tout le tableau. Cependant, à partir de la documentation :

> L'opération d'ajout s'exécute dans un temps constant amorti, c'est-à-dire que l'ajout de n éléments nécessite l'heure O (n)

Supprimer un élément nécessite O (n) temps.

AttributeList

En venant

CopyOnWriteArrayList

En venant

LinkedList

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

LinkedList est implémentée par une liste à double liaison, une structure de données liée qui consiste en un ensemble d'enregistrements liés de manière séquentielle appelés nœuds.

Iitialize LinkedList d'Integer

```
List<Integer> myList = new LinkedList<Integer>(); // Constructs an empty list.
```

- AVANTAGES:

L'ajout ou la suppression d'un élément au début de la liste ou à la fin a un temps constant.

```
myList.add(10);  // \
myList.add(0,2); //  | => constant time => O(1)
myList.remove(); // /
```

- CONS: De la documentation :

> Les opérations indexées dans la liste traverseront la liste depuis le début ou la fin, selon ce qui est le plus proche de l'index spécifié.

Opérations telles que:

```
myList.get(10);    // \
myList.add(11,25); //  | => worst case done in O(n/2)
myList.set(15,35); // /
```

Liste de rôles

En venant

RoleUnresolvedList

En venant

Empiler

En venant

Vecteur

En venant

Lire Des listes en ligne: https://riptutorial.com/fr/java/topic/2989/des-listes

Chapitre 51: Documentation du code Java

Introduction

La documentation du code Java est souvent générée à l'aide de javadoc . Javadoc a été créé par Sun Microsystems dans le but de générer une documentation API au format HTML à partir du code source Java. L'utilisation du format HTML permet de créer des liens hypertexte entre des documents associés.

Syntaxe

- / ** - début de JavaDoc sur une classe, un champ, une méthode ou un package
- @author // Pour nommer l'auteur de la classe, de l'interface ou de l'énumération. C'est requis.
- @version // La version de cette classe, interface ou enum. C'est requis. Vous pouvez utiliser des macros telles que% I% ou% G% pour votre logiciel de contrôle de code source pour remplir la vérification.
- @param // Affiche les arguments (paramètres) d'une méthode ou d'un constructeur. Spécifiez une balise @param pour chaque paramètre.
- @return // Affiche les types de retour pour les méthodes non vides.
- @exception // Indique quelles exceptions peuvent être émises par la méthode ou le constructeur. Les exceptions qui **DOIVENT** être prises doivent être listées ici. Si vous le souhaitez, vous pouvez également inclure ceux qui n'ont pas besoin d'être interceptés, comme ArrayIndexOutOfBoundsException. Spécifiez une exception @ pour chaque exception pouvant être levée.
- @throws // Identique à @exception.
- @see // Liens vers une méthode, un champ, une classe ou un package. Utiliser sous la forme de package.Class # quelque chose.
- @since // Lorsque cette méthode, champ ou classe a été ajouté. Par exemple, JDK-8 pour une classe comme java.util.Optional <T> .
- @serial, @serialField, @serialData // Utilisé pour afficher le serialVersionUID.
- @deprecated // Pour marquer une classe, une méthode ou un champ comme étant obsolète. Par exemple, un serait java.io.StringBufferInputStream . Voir une liste complète des classes obsolètes existantes ici .
- {@link} // Similaire à @see, mais peut être utilisé avec du texte personnalisé: {@link #setDefaultCloseOperation (int closeOperation) voir JFrame # setDefaultCloseOperation pour plus d'informations}.
- {@linkplain} // Similaire à {@link}, mais sans la police de code.
- {@code} // Pour le code littéral, tel que les balises HTML. Par exemple: {@code <html> </ html>}. Cependant, cela utilisera une police à espacement fixe. Pour obtenir le même résultat sans la police monospace, utilisez {@literal}.
- {@literal} // Identique à {@code}, mais sans la police monospace.
- {@value} // Affiche la valeur d'un champ statique: la valeur de JFrame # EXIT_ON_CLOSE est {@value}. Vous pouvez également créer un lien vers un champ donné: Utilise le nom de

l'application {@value AppConstants # APP_NAME}.
- {@docRoot} // Le dossier racine du HTML JavaDoc relatif au fichier en cours. Exemple: Crédits .
- HTML est autorisé: <code> "Salut les cookies" .substring (3) </ code>.
- * / - fin de la déclaration JavaDoc

Remarques

Javadoc est un outil fourni avec le JDK qui permet de convertir les commentaires dans le code en une documentation HTML. La spécification de l'API Java a été générée à l'aide de Javadoc. La même chose est vraie pour une grande partie de la documentation des bibliothèques tierces.

Examples

Documentation de classe

Tous les commentaires Javadoc commencent par un commentaire de bloc suivi d'un astérisque (`/**`) et se terminent lorsque le commentaire de bloc fait (`*/`). Facultativement, chaque ligne peut commencer par des espaces blancs arbitraires et un seul astérisque; ceux-ci sont ignorés lorsque les fichiers de documentation sont générés.

```
/**
 * 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 {
}
```

Les mêmes balises et formats utilisés pour les `Classes` peuvent également être utilisés pour les `Enums` et les `Interfaces` .

Méthode Documentation

Tous les commentaires Javadoc commencent par un commentaire de bloc suivi d'un astérisque (`/**`) et se terminent lorsque le commentaire de bloc fait (`*/`). Facultativement, chaque ligne peut commencer par des espaces blancs arbitraires et un seul astérisque; ceux-ci sont ignorés lorsque les fichiers de documentation sont générés.

```
/**
 * 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;
}
```

Documentation de terrain

Tous les commentaires Javadoc commencent par un commentaire de bloc suivi d'un astérisque (`/**`) et se terminent lorsque le commentaire de bloc fait (`*/`). Facultativement, chaque ligne peut

commencer par des espaces blancs arbitraires et un seul astérisque; ceux-ci sont ignorés lorsque les fichiers de documentation sont générés.

```
/**
 * 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";
```

Documentation du package

Java SE 5

Il est possible de créer une documentation au niveau du package dans Javadocs à l'aide d'un fichier appelé `package-info.java` . Ce fichier doit être formaté comme ci-dessous. Les espaces blancs et les astérisques sont facultatifs, généralement présents dans chaque ligne pour la raison du formatage

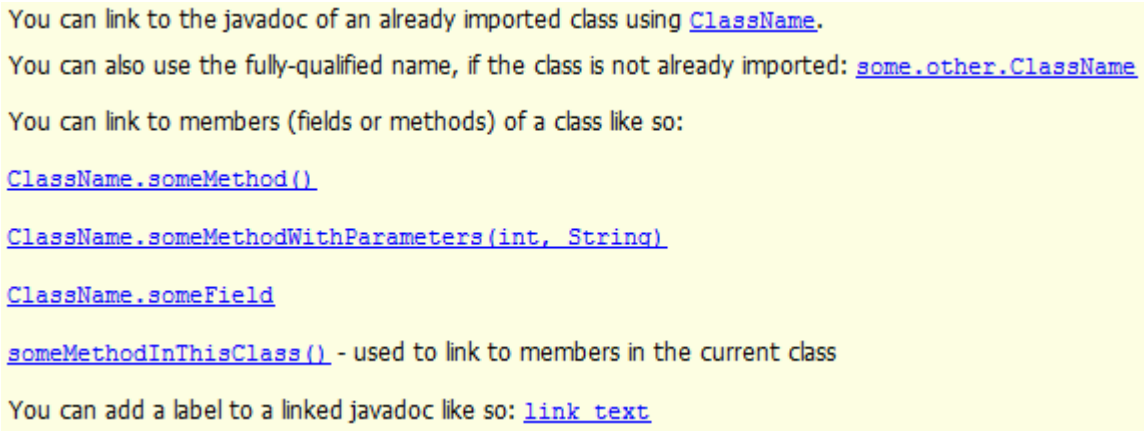
```
/**
 * Package documentation goes here; any documentation before the first period will
 * be used as a summary.
 *
 * It is common practice to leave a blank line between the summary and the rest
 * of the documentation; use this space to describe the package in as much detail
 * as is appropriate.
 *
 * 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.
```

Dans le cas ci-dessus, vous devez placer ce fichier `package-info.java` dans le dossier du package Java `com.example.foo` .

Liens

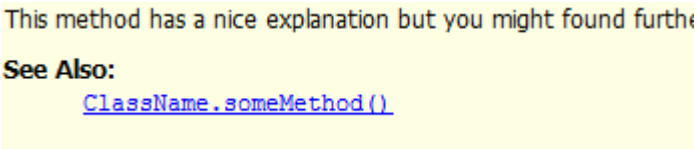
La liaison à d'autres Javadocs se fait avec la balise `@link` :

```
/**
 * 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}
 */
```



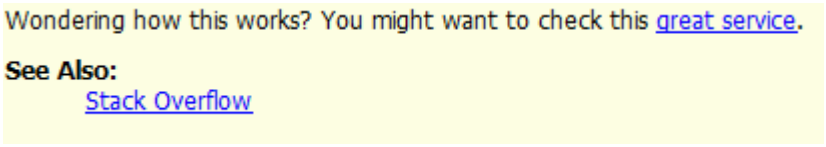
Avec le tag `@see` vous pouvez ajouter des éléments à la section *Voir aussi* . Comme `@param` ou `@return` l'endroit où ils apparaissent n'est pas pertinent. La spécification dit que vous devriez l'écrire après `@return` .

```
/**
 * This method has a nice explanation but you might found further
 * information at the bottom.
 *
 * @see ClassName#someMethod()
 */
```



Si vous souhaitez ajouter des **liens vers des ressources externes,** vous pouvez simplement utiliser la balise HTML `<a>` . Vous pouvez l'utiliser en ligne n'importe où ou à l'intérieur des balises `@link` et `@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>
 */
```



Construction de Javadocs à partir de la ligne de commande

De nombreux IDE prennent en charge la génération automatique de HTML à partir de Javadocs; certains outils de construction (Maven et Gradle , par exemple) ont également des plugins capables de gérer la création HTML.

Cependant, ces outils ne sont pas requis pour générer le HTML Javadoc; Cela peut être fait en utilisant l'outil de ligne de commande `javadoc` .

L'utilisation la plus élémentaire de l'outil est la suivante:

```
javadoc JavaFile.java
```

Qui générera du HTML à partir des commentaires Javadoc dans `JavaFile.java` .

Une utilisation plus pratique de l'outil en ligne de commande, qui lit tous les fichiers Java dans `[source-directory]` manière récursive, crée la documentation pour `[package.name]` et tous les sous-packages, et place le code HTML généré dans le `[docs-directory]` est:

```
javadoc -d [docs-directory] -subpackages -sourcepath [source-directory] [package.name]
```

Documentation du code en ligne

Outre la documentation Javadoc, le code peut être documenté en ligne.

Les commentaires sur une seule ligne sont lancés par `//` et peuvent être positionnés après une instruction sur la même ligne, mais pas avant.

```
public void method() {

  //single line comment
  someMethodCall(); //single line comment after statement

}
```

Les commentaires multi-lignes sont définis entre `/*` et `*/` . Ils peuvent couvrir plusieurs lignes et peuvent même être positionnés entre les instructions.

```
public void method(Object object) {

  /*
    multi
    line
    comment
  */
  object/*inner-line-comment*/.method();
```

```
}
```

JavaDocs est une forme spéciale de commentaires sur plusieurs lignes, commençant par `/**` .

Comme un trop grand nombre de commentaires en ligne peut nuire à la lisibilité du code, ils doivent être utilisés de manière limitée au cas où le code ne serait pas suffisamment explicite ou que la décision de conception n'est pas évidente.

Un autre cas d'utilisation des commentaires sur une seule ligne est l'utilisation de TAG, qui sont des mots-clés courts, basés sur des conventions. Certains environnements de développement reconnaissent certaines conventions pour de tels commentaires uniques. Des exemples communs sont

- `//TODO`
- `//FIXME`

Ou émettre des références, par exemple pour Jira

- `//PRJ-1234`

Extraits de code dans la documentation

La manière canonique d'écrire du code dans la documentation est la construction `{@code }` . Si vous avez du code multiligne, retournez-le dans `<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 {
```

Parfois, vous devrez peut-être mettre du code complexe dans le commentaire javadoc. Le signe `@` est particulièrement problématique. L'utilisation de l'ancienne `<code>` côté de la construction `{@literal }` résout le problème.

```
/**
 * 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 { }
```

Lire Documentation du code Java en ligne: https://riptutorial.com/fr/java/topic/140/documentation-du-code-java

Chapitre 52: Douilles

Introduction

Un socket est l'un des points d'extrémité d'un lien de communication bidirectionnel entre deux programmes exécutés sur le réseau.

Examples

Lire depuis socket

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try (
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out =
        new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream()));
    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in))
) {
    //Use the socket
}
```

Lire Douilles en ligne: https://riptutorial.com/fr/java/topic/9918/douilles

Chapitre 53: Editions Java, versions, versions et distributions

Examples

Différences entre les distributions Java SE JRE ou Java SE JDK

Les versions Sun / Oracle de Java SE se présentent sous deux formes: JRE et JDK. En termes simples, les JRE prennent en charge les applications Java et les JDK prennent également en charge le développement Java.

Java Runtime Environment

Java Runtime Environment ou les distributions JRE se composent de l'ensemble des bibliothèques et des outils nécessaires pour exécuter et gérer les applications Java. Les outils dans un JRE moderne typique incluent:

- La commande `java` pour exécuter un programme Java dans une machine virtuelle Java (Java Virtual Machine)
- La commande `jjs` pour exécuter le moteur Javascript Nashorn.
- La commande `keytool` pour manipuler les fichiers de clés Java.
- La commande `policytool` pour la modification des stratégies de sécurité du sandbox de sécurité.
- Les outils `pack200` et `unpack200` pour emballer et décompresser le fichier "pack200" pour le déploiement Web.
- Les commandes `orbd` , `rmid` , `rmiregistry` et `tnameserv` qui prennent en charge les applications Java CORBA et RMI.

Les installateurs "Desktop JRE" incluent un plug-in Java adapté à certains navigateurs Web. Ceci est délibérément exclu de "Server JRE" installers.linux syscall read benchmarku

À partir de Java 7 mise à jour 6, les installateurs JRE ont inclus JavaFX (version 2.2 ou ultérieure).

Kit de développement Java

Un kit de développement Java ou une distribution JDK inclut les outils JRE et des outils supplémentaires pour développer des logiciels Java. Les outils supplémentaires incluent généralement:

- La commande `javac` , qui compile le code source Java (".java") en fichiers bytecode (".class").
- Les outils pour créer des fichiers JAR tels que `jar` et `jarsigner`
- Des outils de développement tels que:
    - `appletviewer` pour exécuter des applets

    - `idlj` le `idlj` CORBA IDL vers Java
    - `javah` le générateur de `javah` JNI
    - `native2ascii` pour la conversion de jeu de caractères du code source Java
    - `schemagen` le générateur de schémas Java vers XML (partie de JAXB)
    - `serialver` génère la chaîne de version Java Object Serialization.
    - les outils de support `wsgen` et `wsimport` pour JAX-WS
- Outils de diagnostic tels que:
    - `jdb` le débogueur Java de base
    - `jmap` et `jhat` pour vider et analyser un tas Java.
    - `jstack` pour obtenir un vidage de pile de threads.
    - `javap` pour examiner les fichiers ".class".
- Outils de gestion et de surveillance des applications tels que:
    - `jconsole` une console de gestion,
    - `jstat` , `jstatd` , `jinfo` et `jps` pour la surveillance des applications

Une installation Sun / Oracle JDK typique comprend également un fichier ZIP contenant le code source des bibliothèques Java. Avant Java 6, c'était le seul code source Java accessible au public.

À partir de Java 6, le code source complet d'OpenJDK est disponible en téléchargement sur le site OpenJDK. Il n'est généralement pas inclus dans les packages JDK (Linux), mais est disponible sous forme de package séparé.

Quelle est la différence entre Oracle Hotspot et OpenJDK

Orthogonal à la dichotomie JRE-JDK, il existe deux types de versions Java largement disponibles:

- Les versions Oracle Hotspot sont celles que vous téléchargez à partir des sites de téléchargement Oracle.
- Les versions OpenJDK sont celles qui sont créées (généralement par des fournisseurs tiers) à partir des référentiels sources OpenJDK.

En termes fonctionnels, il y a peu de différence entre une version Hotspot et une version OpenJDK. Il existe des fonctionnalités "d'entreprise" supplémentaires dans Hotspot que les clients Java (payants) d'Oracle peuvent activer, mais à part cela, la même technologie est présente dans Hotspot et OpenJDK.

Un autre avantage de Hotspot sur OpenJDK est que les versions de correctifs pour Hotspot ont tendance à être disponibles un peu plus tôt. Cela dépend également de la flexibilité de votre fournisseur OpenJDK; Par exemple, combien de temps cela prend-il à une équipe de construction d'une distribution Linux pour préparer une nouvelle version d'OpenJDK, et à l'avoir dans ses référentiels publics.

Le revers de la médaille est que les versions Hotspot ne sont pas disponibles à partir des référentiels de paquets pour la plupart des distributions Linux. Cela signifie que le fait de garder votre logiciel Java à jour sur une machine Linux est généralement plus efficace si vous utilisez Hotspot.

Différences entre Java EE, Java SE, Java ME et JavaFX

La technologie Java est à la fois un langage de programmation et une plate-forme. Le langage de programmation Java est un langage orienté objet de haut niveau qui possède une syntaxe et un style particuliers. Une plate-forme Java est un environnement particulier dans lequel les applications de langage de programmation Java s'exécutent.

Il existe plusieurs plateformes Java. De nombreux développeurs, même les développeurs de langages de programmation Java de longue date, ne comprennent pas comment les différentes plates-formes sont liées les unes aux autres.

Les plates-formes de langage de programmation Java

Il existe quatre plates-formes du langage de programmation Java:

- Plate-forme Java, Standard Edition (Java SE)
- Plate-forme Java, Enterprise Edition (Java EE)
- Plate-forme Java, Micro Edition (Java ME)
- Java FX

Toutes les plates-formes Java se composent d'une machine virtuelle Java (VM) et d'une interface de programmation d'application (API). Java Virtual Machine est un programme, pour une plate-forme matérielle et logicielle particulière, qui exécute des applications de technologie Java. Une API est un ensemble de composants logiciels que vous pouvez utiliser pour créer d'autres composants ou applications logicielles. Chaque plate-forme Java fournit une machine virtuelle et une API, ce qui permet aux applications écrites pour cette plate-forme de s'exécuter sur tous les systèmes compatibles avec tous les avantages du langage de programmation Java: indépendance, puissance, stabilité, facilité de développement et Sécurité.

Java SE

Lorsque la plupart des gens pensent au langage de programmation Java, ils pensent à l'API Java SE. L'API de Java SE fournit les fonctionnalités essentielles du langage de programmation Java. Il définit tout, des types et objets de base du langage de programmation Java aux classes de haut niveau utilisées pour la mise en réseau, la sécurité, l'accès aux bases de données, le développement de l'interface graphique et l'analyse XML.

Outre l'API principale, la plate-forme Java SE se compose d'une machine virtuelle, d'outils de développement, de technologies de déploiement et d'autres bibliothèques de classes et kits d'outils couramment utilisés dans les applications de technologie Java.

Java EE

La plate-forme Java EE est construite sur la plate-forme Java SE. La plate-forme Java EE fournit un environnement d'API et d'exécution pour développer et exécuter des applications réseau à grande échelle, à plusieurs niveaux, évolutives, fiables et sécurisées.

Java ME

La plate-forme Java ME fournit une API et une machine virtuelle de faible encombrement pour exécuter des applications de langage de programmation Java sur de petits périphériques, tels que des téléphones portables. L'API est un sous-ensemble de l'API Java SE, avec des bibliothèques de classes spéciales utiles pour le développement d'applications pour petits périphériques. Les applications Java ME sont souvent des clients de services de plate-forme Java EE.

Java FX

La technologie Java FX est une plate-forme permettant de créer des applications Internet riches écrites en Java FX ScriptTM. Java FX Script est un langage déclaratif de type statique compilé en bytecode de la technologie Java, qui peut ensuite être exécuté sur une machine virtuelle Java. Les applications écrites pour la plate-forme Java FX peuvent inclure et lier des classes de langage de programmation Java et peuvent être des clients des services de plate-forme Java EE.

- Tiré de la documentation Oracle

Versions Java SE

Historique de la version Java SE

Le tableau suivant indique la chronologie des principales versions majeures de la plate-forme Java SE.

Java SE Version [1]	Nom de code	Fin de vie (gratuit [2])	Date de sortie
Java SE 9 (accès anticipé)	*Aucun*	avenir	2017-07-27 (estimé)
Java SE 8	*Aucun*	avenir	2014-03-18
Java SE 7	Dauphin	2015-04-14	2011-07-28
Java SE 6	Mustang	2013-04-16	2006-12-23

Java SE Version [1]	Nom de code	Fin de vie (gratuit [2])	Date de sortie
Java SE 5	tigre	2009-11-04	2004-10-04
Java SE 1.4.2	Mante	avant 2009-11-04	2003-06-26
Java SE 1.4.1	Trémie / Sauterelle	avant 2009-11-04	2002-09-16
Java SE 1.4	Merlin	avant 2009-11-04	2002-02-06
Java SE 1.3.1	Coccinelle	avant 2009-11-04	2001-05-17
Java SE 1.3	Crécerelle	avant 2009-11-04	2000-05-08
Java SE 1.2	Cour de récréation	avant 2009-11-04	1998-12-08
Java SE 1.1	Cierge magique	avant 2009-11-04	1997-02-19
Java SE 1.0	Chêne	avant 2009-11-04	1996-01-21

Notes de bas de page:

1. Les liens sont des copies en ligne de la documentation des versions respectives sur le site Web d'Oracle. La documentation de nombreuses versions plus anciennes n'est plus en ligne, bien qu'elle puisse généralement être téléchargée à partir des archives Oracle Java.
2. La plupart des versions historiques de Java SE ont dépassé leur date de fin de vie officielle. Lorsqu'une version Java passe cette étape, Oracle cesse de fournir des mises à jour gratuites. Les mises à jour sont toujours disponibles pour les clients avec des contrats de support.

La source:

- Dates de sortie du JDK par Roedy Green de Canadian Mind Products

Faits saillants de la version Java SE

Version Java SE	Points forts
Java SE 8	Expressions Lambda et flux inspirés de MapReduce. Le moteur Javascript Nashorn. Annotations sur les types et annotations répétitives. Extensions arithmétiques non signées. Nouvelles API de date et heure. Bibliothèques JNI liées statiquement. Lanceur JavaFX. Enlèvement de PermGen.
Java SE 7	Commutateurs de chaînes, *try-with-resource* , the diamond (`<>`), améliorations littérales numériques et améliorations / gestion des exceptions. Améliorations de

Version Java SE	Points forts
	la bibliothèque de accès simultanés. Prise en charge améliorée des systèmes de fichiers natifs. Timsort. Algorithmes de cryptage ECC. Amélioration du support graphique 2D (GPU). Annotations enfichables.
Java SE 6	Amélioration significative des performances de la plate-forme JVM et du Swing. API de langage de script et moteur Javascript Mozilla Rhino. JDBC 4.0. API du compilateur. JAXB 2.0. Prise en charge des services Web (JAX-WS)
Java SE 5	Génériques, annotations, auto-boxing, classes `enum` , varargs, améliorés `for` boucles et les importations statiques. Spécification du modèle de mémoire Java. Swing et améliorations RMI. Ajout du package `java.util.concurrent.*` Et du `Scanner` .
Java SE 1.4	Le mot-clé `assert` . Classes d'expressions régulières Chaîne d'exception. API NIO - E / S, `Buffer` et `Channel` non bloquants. `java.util.logging.*` API. Image I / O API. XML intégré et XSLT (JAXP). Sécurité intégrée et cryptographie (JCE, JSSE, JAAS). Java Web Start intégré. API de préférences.
Java SE 1.3	HotSpot JVM inclus Intégration CORBA / RMI. Interface JNDI (Java Naming and Directory Interface). Framework du débogueur (JPDA). API JavaSound. API proxy.
Java SE 1.2	Le mot clé `strictfp` . API de swing. Le plugin Java (pour les navigateurs Web). Interopérabilité CORBA. Cadre des collections.
Java SE 1.1	Classes internes Réflexion. JDBC. RMI. Unicode / flux de caractères. Prise en charge de l'internationalisation. Refonte du modèle d'événement AWT. JavaBeans.

La source:

- Wikipedia: Historique des versions de Java

Lire Editions Java, versions, versions et distributions en ligne:
https://riptutorial.com/fr/java/topic/8973/editions-java--versions--versions-et-distributions

Chapitre 54: Encapsulation

Introduction

Imaginez que vous ayez une classe avec des variables assez importantes et que celles-ci étaient définies (par d'autres programmeurs à partir de leur code) sur des valeurs inacceptables. Leur code entraînait des erreurs dans votre code. En tant que solution, dans OOP, vous permettez à l'état d'un objet (stocké dans ses variables) d'être modifié uniquement par des méthodes. Le masquage de l'état d'un objet et la fourniture de toutes les interactions via des méthodes d'objets sont connus sous le nom d'encapsulation de données.

Remarques

Il est beaucoup plus facile de commencer par marquer une variable `private` et de l'exposer si nécessaire que de masquer une variable déjà `public` .

Il existe une exception où l'encapsulation peut ne pas être bénéfique: structures de données "stupides" (classes dont le seul but est de contenir des variables).

```
public class DumbData {
    public String name;
    public int timeStamp;
    public int value;
}
```

Dans ce cas, l'interface de la classe *est* la donnée qu'elle contient.

Notez que les variables marquées comme `final` peuvent être marquées comme `public` sans violer l'encapsulation, car elles ne peuvent pas être modifiées après avoir été définies.

Examples

Encapsulation pour maintenir les invariants

Il y a deux parties d'une classe: l'interface et l'implémentation.

L'interface est la fonctionnalité exposée de la classe. Ses méthodes publiques et ses variables font partie de l'interface.

L'implémentation est le fonctionnement interne d'une classe. Les autres classes ne devraient pas avoir besoin de connaître l'implémentation d'une classe.

L'encapsulation fait référence à la pratique consistant à cacher l'implémentation d'une classe à tous les utilisateurs de cette classe. Cela permet à la classe de formuler des hypothèses sur son état interne.

Par exemple, prenez cette classe représentant un angle:

```
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(){}
}
```

Cette classe repose sur une hypothèse de base (ou *invariant*): **angleInDegrees et angleInRadians sont toujours synchronisés** . Si les membres de la classe étaient publics, il n'y aurait aucune garantie que les deux représentations des angles soient corrélées.

Encapsulation pour réduire le couplage

L'encapsulation vous permet d'apporter des modifications internes à une classe sans affecter le code qui appelle la classe. Cela réduit le *couplage* ou combien une classe donnée dépend de l'implémentation d'une autre classe.

Par exemple, modifions l'implémentation de la classe Angle à partir de l'exemple précédent:

```
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'implémentation de cette classe a été modifiée pour ne stocker qu'une représentation de l'angle et calculer l'autre angle si nécessaire.

Cependant, **l'implémentation a changé, mais l'interface ne l'a pas fait** . Si une classe appelante comptait sur l'accès à la méthode angleInRadians, elle devrait être modifiée pour utiliser la nouvelle version d' `Angle` . Les classes appelantes ne devraient pas se soucier de la représentation interne d'une classe.

Lire Encapsulation en ligne: https://riptutorial.com/fr/java/topic/1295/encapsulation

Chapitre 55: Encodage de caractère

Examples

Lecture de texte à partir d'un fichier encodé en UTF-8

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ReadingUTF8TextFile {

    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {
            wr.write("Strange cyrillic symbol Ы");
        }
        /* First Way. For big files */
        try (BufferedReader reader = Files.newBufferedReader(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {

            String line;
            while ((line = reader.readLine()) != null) {
                System.out.print(line);
            }
        }

        System.out.println(); //just separating output

        /* Second way. For small files */
        String s = new String(Files.readAllBytes(Paths.get("test.txt")),
StandardCharsets.UTF_8);
        System.out.print(s);
    }
}
```

Écrire du texte dans un fichier en UTF-8

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class WritingUTF8TextFile {
    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test2.txt"),
StandardCharsets.UTF_8)) {
```

```
            wr.write("Cyrillic symbol Ы");
        }
    }
}
```

Obtenir la représentation en octets d'une chaîne dans 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));
    }
}
```

Lire Encodage de caractère en ligne: https://riptutorial.com/fr/java/topic/2735/encodage-de-caractere

Chapitre 56: Ensembles

Examples

Déclarer un HashSet avec des valeurs

Vous pouvez créer une nouvelle classe qui hérite de HashSet:

```
Set<String> h = new HashSet<String>() {{
    add("a");
    add("b");
}};
```

Une solution en ligne:

```
Set<String> h = new HashSet<String>(Arrays.asList("a", "b"));
```

En utilisant goyave:

```
Sets.newHashSet("a", "b", "c")
```

Utiliser des flux:

```
Set<String> set3 = Stream.of("a", "b", "c").collect(toSet());
```

Types et utilisation des ensembles

Généralement, les ensembles sont un type de collection qui stocke des valeurs uniques. L'unicité est déterminée par les méthodes `equals()` et `hashCode()` .

Le tri est déterminé par le type de set.

`HashSet` - Tri aléatoire

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` - Ordre d'insertion

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` - Par `compareTo()` ou `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"]
```

Initialisation

Un ensemble est une collection qui ne peut pas contenir d'éléments en double. Il modélise l'abstraction d'ensemble mathématique.

`Set` a son implémentation dans différentes classes comme `HashSet` , `TreeSet` , `LinkedHashSet` .

Par exemple:

HashSet:

```
Set<T> set = new HashSet<T>();
```

Ici, `T` peut être `String` , `Integer` ou tout autre **objet** . **HashSet** permet une recherche rapide de O (1) mais ne trie pas les données ajoutées et perd l'ordre d'insertion des éléments.

TreeSet:

Il stocke les données de manière triée en sacrifiant une certaine vitesse pour les opérations de base qui prennent O (lg (n)). Il ne conserve pas l'ordre d'insertion des éléments.

```
TreeSet<T> sortedSet = new TreeSet<T>();
```

LinkedHashSet:

C'est une implémentation de liste `HashSet` de `HashSet` Once qui peut parcourir les éléments dans l'ordre dans `HashSet` ils ont été ajoutés. Le tri n'est pas prévu pour son contenu. O (1) opérations de base sont fournies, mais il y a un coût plus élevé que `HashSet` dans la `HashSet` de la liste des liens de support.

```
LinkedHashSet<T> linkedhashset = new LinkedHashSet<T>();
```

Bases de Set

Qu'est-ce qu'un ensemble?

Un ensemble est une structure de données qui contient un ensemble d'éléments ayant une propriété importante indiquant que deux éléments de l'ensemble ne sont pas égaux.

Types de set:

1. **HashSet:** Un jeu sauvegardé par une table de hachage (en fait une instance de HashMap)
2. **HashSet lié:** un jeu sauvegardé par une table de hachage et une liste chaînée, avec un ordre d'itération prévisible
3. **TreeSet:** Une implémentation NavigableSet basée sur un TreeMap.

Créer un ensemble

```
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
```

Ajout d'éléments à un ensemble

Des éléments peuvent être ajoutés à un ensemble en utilisant la méthode `add()`

```
 set.add(12); //  - Adds element 12 to the set
 set.add(13); //  - Adds element 13 to the set
```

Notre set après avoir exécuté cette méthode:

```
set = [12,13]
```

Supprimer tous les éléments d'un ensemble

```
set.clear();  //Removes all objects from the collection.
```

Après cet ensemble sera:

```
set = []
```

Vérifier si un élément fait partie de l'ensemble

L'existence d'un élément dans l'ensemble peut être vérifiée à l'aide de la méthode `contains()`

```
set.contains(0);  //Returns true if a specified object is an element within the set.
```

Sortie: `False`

Vérifiez si un ensemble est vide

`isEmpty()` méthode `isEmpty()` peut être utilisée pour vérifier si un ensemble est vide.

```
set.isEmpty();  //Returns true if the set has no elements
```

Sortie: True

Retirer un élément de l'ensemble

```
 set.remove(0); // Removes first occurrence of a specified object from the collection
```

Vérifiez la taille de l'ensemble

```
set.size(); //Returns the number of elements in the collection
```

Sortie: 0

Créer une liste à partir d'un ensemble existant

Utiliser une nouvelle liste

```
List<String> list = new ArrayList<String>(listOfElements);
```

Utiliser la méthode List.addAll ()

```
    Set<String> set = new HashSet<String>();
    set.add("foo");
    set.add("boo");

    List<String> list = new ArrayList<String>();
    list.addAll(set);
```

Utilisation de l'API Java 8

```
List<String> list = set.stream().collect(Collectors.toList());
```

Éliminer les doublons en utilisant Set

Supposons que vous ayez des `elements` collection et que vous souhaitiez créer une autre collection contenant les mêmes éléments mais avec tous les **doublons éliminés** :

```
Collection<Type> noDuplicates = new HashSet<Type>(elements);
```

Exemple :

```
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);
```

Sortie :

```
noDuplicates = [Marco, Emily, John, Jenny]
```

Lire Ensembles en ligne: https://riptutorial.com/fr/java/topic/3102/ensembles

Chapitre 57: Enum Carte

Introduction

La classe Java EnumMap est l'implémentation Map spécialisée pour les clés enum. Il hérite des classes Enum et AbstractMap.

les paramètres de la classe java.util.EnumMap.

K: C'est le type de clés maintenu par cette carte. V: C'est le type des valeurs mappées.

Examples

Enum Map Book Exemple

```
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);

        }
}
}
```

Lire Enum Carte en ligne: https://riptutorial.com/fr/java/topic/10158/enum-carte

Chapitre 58: Enum commençant par numéro

Introduction

Java ne permet pas que le nom de enum commence par un nombre comme 100A, 25K. Dans ce cas, nous pouvons ajouter le code avec _ (trait de soulignement) ou tout motif autorisé et le vérifier.

Examples

Enum avec le nom au début

```
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);
    }
}
```

Lire Enum commençant par numéro en ligne: https://riptutorial.com/fr/java/topic/10719/enum-commencant-par-numero

Chapitre 59: Enums

Introduction

Les `enum` Java (déclarées à l'aide du mot-clé `enum`) sont une syntaxe abrégée pour des quantités importantes de constantes d'une même classe.

Syntaxe

- [public / protected / private] enum Nom_enum {// Déclarez une nouvelle énumération.
- ENUM_CONSTANT_1 [, ENUM_CONSTANT_2 ...]; // Déclarez les constantes enum. Ce doit être la première ligne à l'intérieur de l'énumération et doit être séparé par des virgules, avec un point-virgule à la fin.
- ENUM_CONSTANT_1 (param) [, ENUM_CONSTANT_2 (param) ...]; // Déclarez les constantes enum avec des paramètres. Les types de paramètres doivent correspondre au constructeur.
- ENUM_CONSTANT_1 {...} [, ENUM_CONSTANT_2 {...} ...]; // Déclarez les constantes enum avec des méthodes surchargées. Cela doit être fait si l'énumération contient des méthodes abstraites; toutes ces méthodes doivent être mises en œuvre.
- ENUM_CONSTANT.name () // Retourne une chaîne avec le nom de la constante enum.
- ENUM_CONSTANT.ordinal () // Renvoie l'ordinal de cette constante d'énumération, sa position dans sa déclaration enum, où la constante initiale reçoit un ordinal de zéro.
- Enum_name.values () // Retourne un nouveau tableau (de type Enum_name []) contenant chaque constante de cette enum chaque fois qu'il est appelé.
- Enum_name.valueOf ("ENUM_CONSTANT") // Inverse de ENUM_CONSTANT.name () - renvoie la constante enum avec le nom donné.
- Enum.valueOf (Enum_name.class, "ENUM_CONSTANT") // Un synonyme du précédent: L'inverse de ENUM_CONSTANT.name () - renvoie la constante enum avec le nom donné.

Remarques

Restrictions

Les énumérations étendent toujours `java.lang.Enum` , il est donc impossible pour un enum d'étendre une classe. Cependant, ils peuvent implémenter de nombreuses interfaces.

Conseils & Astuces

En raison de leur représentation spécialisée, il existe des cartes et des ensembles plus efficaces qui peuvent être utilisés avec des énumérations comme clés. Celles-ci courent souvent plus vite que leurs homologues non spécialisés.

Examples

Déclarer et utiliser un enum de base

Enum peut être considéré comme du sucre syntaxique pour une classe scellée qui n'est instanciée que plusieurs fois au moment de la compilation pour définir un ensemble de constantes.

Un simple enum pour énumérer les différentes saisons serait déclaré comme suit:

```
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

Bien que les constantes enum ne doivent pas nécessairement être en majuscules, la convention Java stipule que les noms des constantes sont entièrement en majuscules, les mots étant séparés par des traits de soulignement.

Vous pouvez déclarer un Enum dans son propre fichier:

```
/**
 * This enum is declared in the Season.java file.
*/
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

Mais vous pouvez également le déclarer dans une autre 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
    }

}
```

Enfin, vous ne pouvez pas déclarer un Enum dans un corps de méthode ou un constructeur:

```
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;
    }

}
```

Les constantes énumérées en double ne sont pas autorisées:

```
public enum Season {
    WINTER,
    WINTER, //Compile Time Error : Duplicate Constants
    SPRING,
    SUMMER,
    FALL
}
```

Chaque constante d'énum est **`public`**, **`static`** et **`final`** par défaut. Comme chaque constante est `static`, il est possible d'y accéder directement en utilisant le nom enum.

Les constantes enum peuvent être transmises comme paramètres de méthode:

```
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"
```

Vous pouvez obtenir un tableau des constantes enum en utilisant la méthode `values()` . Les valeurs sont garanties dans l'ordre de déclaration dans le tableau renvoyé:

```
Season[] seasons = Season.values();
```

Remarque: cette méthode alloue un nouveau tableau de valeurs à chaque appel.

Pour parcourir les constantes énumérées:

```
public static void enumIterate() {
    for (Season s : Season.values()) {
        System.out.println(s.name());
    }
}
```

Vous pouvez utiliser des énumérations dans une déclaration de `switch` :

```
public static void enumSwitchExample(Season s) {
    switch(s) {
        case WINTER:
            System.out.println("It's pretty cold");
            break;
        case SPRING:
            System.out.println("It's warming up");
            break;
        case SUMMER:
            System.out.println("It's pretty hot");
            break;
        case FALL:
            System.out.println("It's cooling down");
            break;
    }
}
```

Vous pouvez également comparer les constantes énumérées en utilisant `==` :

```
Season.FALL == Season.WINTER    // false
Season.SPRING == Season.SPRING  // true
```

Une autre façon de comparer les constantes d'énumération consiste à utiliser les `equals()` comme ci-dessous, ce qui est considéré comme une mauvaise pratique car vous pouvez facilement tomber dans des pièges comme suit:

```
Season.FALL.equals(Season.FALL); // true
```

```
Season.FALL.equals(Season.WINTER); // false
Season.FALL.equals("FALL"); // false and no compiler error
```

De plus, bien que l'ensemble d'instances dans l' `enum` ne puisse pas être modifié au moment de l'exécution, les instances elles-mêmes ne sont pas intrinsèquement immuables car, comme toute autre classe, une `enum` peut contenir des champs mutables.

```
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
```

Cependant, une bonne pratique consiste à rendre les instances d' `enum` immuables, c'est-à-dire que lorsqu'elles n'ont pas de champs supplémentaires ou que tous ces champs sont marqués comme `final` et qu'ils sont immuables eux-mêmes. Cela garantira que pour une durée de vie de l'application, une `enum` ne `enum` pas de mémoire et qu'il est prudent d'utiliser ses instances sur tous les threads.

Enums implémentent implicitement `Serializable` et `Comparable` car la classe `Enum` fait:

```
public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable
```

Enums avec des constructeurs

Un `enum` ne peut pas avoir de constructeur public; cependant, les constructeurs privés sont acceptables (les constructeurs pour les énumérations sont package-private par défaut):

```
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
```

Il est recommandé de garder tous les champs privés et de fournir des méthodes de lecture, car il existe un nombre fini d'instances pour un enum.

Si vous deviez implémenter un `Enum` tant que `class` , cela ressemblerait à ceci:

```
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
```

Les constantes enum sont techniquement modifiables, donc un setter pourrait être ajouté pour modifier la structure interne d'une constante enum. Cependant, ceci est considéré comme une très mauvaise pratique et devrait être évité.

La meilleure pratique consiste à rendre les champs Enum immuables, avec en `final` :

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value){
        this.value = value;
    }

    ...

}
```

Vous pouvez définir plusieurs constructeurs dans le même enum. Lorsque vous le faites, les arguments que vous transmettez dans votre déclaration enum déterminent quel constructeur est

appelé:

```
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;
    }

    ...

}
```

Remarque: Tous les champs d'énumération non primitifs doivent implémenter `Serializable` car la classe `Enum` fait.

Utiliser des méthodes et des blocs statiques

Un enum peut contenir une méthode, comme toute classe. Pour voir comment cela fonctionne, nous allons déclarer un enum comme ceci:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;
}
```

Ayons une méthode qui renvoie l'enum dans la direction opposée:

```
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;
        }
    }
}
```

Cela peut être amélioré grâce à l'utilisation de champs et de blocs d'initialisation statiques:

```
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;
    }
}
```

Dans cet exemple, la direction opposée est stockée dans un champ d'instance privé `opposite` , qui est initialisé statiquement la première fois qu'une `Direction` est utilisée. Dans ce cas particulier (parce que `NORTH` référence `SOUTH` et inversement), nous ne pouvons pas utiliser des énumérations avec les `NORTH(SOUTH), SOUTH(NORTH), EAST(WEST), WEST(EAST) opposite` constructeurs ici (Constructors DU `NORTH(SOUTH), SOUTH(NORTH), EAST(WEST), WEST(EAST)` serait plus élégant et permettrait en `opposite` de être déclarée `final` , mais serait auto-référentielle et donc non autorisée).

Implémente l'interface

C'est une `enum` qui est également une fonction appelable qui teste les entrées de `String` contre des modèles d'expression régulière précompilés.

```
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"));
    }
}
```

Chaque membre du enum peut également implémenter la méthode:

```
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

Lorsqu'une méthode doit accepter un ensemble "extensible" de valeurs d' `enum` , le programmeur peut appliquer un polymorphisme comme sur une `class` normale en créant une interface qui sera utilisée n'importe où où l' `enum` sera utilisée:

```
public interface ExtensibleEnum {
    String name();
}
```

De cette façon, toute `enum` étiquetée par (implémentant) l'interface peut être utilisée comme paramètre, permettant au programmeur de créer une quantité variable d' `enum` qui sera acceptée par la méthode. Cela peut être utile, par exemple, dans les API où il existe une `enum` par défaut (non modifiable) et où l'utilisateur de ces API souhaite "étendre" l' `enum` avec plus de valeurs.

Un ensemble de valeurs enum par défaut peut être défini comme suit:

```
public enum DefaultValues implements ExtensibleEnum {
    VALUE_ONE, VALUE_TWO;
}
```

Des valeurs supplémentaires peuvent alors être définies comme ceci:

```
public enum ExtendedValues implements ExtensibleEnum {
    VALUE_THREE, VALUE_FOUR;
}
```

Exemple qui montre comment utiliser les énumérations - notez comment `printEnum()` accepte les valeurs des deux types `enum` :

```
private void printEnum(ExtensibleEnum val) {
    System.out.println(val.name());
}

printEnum(DefaultValues.VALUE_ONE);    // VALUE_ONE
printEnum(DefaultValues.VALUE_TWO);    // VALUE_TWO
printEnum(ExtendedValues.VALUE_THREE); // VALUE_THREE
printEnum(ExtendedValues.VALUE_FOUR);  // VALUE_FOUR
```

Remarque: Ce modèle ne vous empêche pas de redéfinir les valeurs enum, qui sont déjà définies dans une enum, dans une autre énumération. Ces valeurs enum seraient alors des instances différentes. De plus, il n'est pas possible d'utiliser switch-on-enum car nous n'avons que l'interface, pas le véritable `enum` .

Enums avec des méthodes abstraites

Enums peut définir des méthodes abstraites, que chaque membre `enum` doit implémenter.

```
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);
}
```

Cela permet à chaque membre enum de définir son propre comportement pour une opération donnée, sans avoir à activer les types dans une méthode de la définition de niveau supérieur.

Notez que ce modèle est une forme abrégée de ce qui est généralement obtenu en utilisant des interfaces de polymorphisme et / ou d'implémentation.

Documenter les enums

Le nom `enum` n'est pas toujours suffisamment clair pour être compris. Pour documenter un `enum` , utilisez 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;
    }
}
```

Obtenir les valeurs d'un enum

Chaque classe enum contient une méthode statique implicite nommée `values()` . Cette méthode retourne un tableau contenant toutes les valeurs de cette enum. Vous pouvez utiliser cette méthode pour parcourir les valeurs. Il est important de noter toutefois que cette méthode retourne un **nouveau** tableau à chaque appel.

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

    /**
    * Print out all the values in this enum.
    */
    public static void printAllDays() {
        for(Day day : Day.values()) {
            System.out.println(day.name());
        }
    }
```

```
}
```

Si vous avez besoin d'un `Set` vous pouvez également utiliser `EnumSet.allOf(Day.class)` .

Enum comme paramètre de type borné

Lors de l'écriture d'une classe avec des génériques dans Java, il est possible de s'assurer que le paramètre type est une énumération. Comme toutes les énumérations étendent la classe `Enum` , la syntaxe suivante peut être utilisée.

```
public class Holder<T extends Enum<T>> {
    public final T value;

    public Holder(T init) {
        this.value = init;
    }
}
```

Dans cet exemple, le type `T` *doit* être un enum.

Obtenir un enum constant par nom

Disons que nous avons une énumération `DayOfWeek` :

```
enum DayOfWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}
```

Un enum est compilé avec une méthode static `valueOf()` intégrée qui peut être utilisée pour rechercher une constante par son nom:

```
String dayName = DayOfWeek.SUNDAY.name();
assert dayName.equals("SUNDAY");

DayOfWeek day = DayOfWeek.valueOf(dayName);
assert day == DayOfWeek.SUNDAY;
```

Ceci est également possible en utilisant un type enum dynamique:

```
Class<DayOfWeek> enumType = DayOfWeek.class;
DayOfWeek day = Enum.valueOf(enumType, "SUNDAY");
assert day == DayOfWeek.SUNDAY;
```

Ces deux méthodes `valueOf()` lancent une `IllegalArgumentException` si l'énumération spécifiée n'a pas de constante avec un nom correspondant.

La bibliothèque Guava fournit une méthode d'assistance `Enums.getIfPresent()` qui retourne une `Optional` Guava pour éliminer la gestion des exceptions explicites:

```
DayOfWeek defaultDay = DayOfWeek.SUNDAY;
```

```
DayOfWeek day = Enums.valueOf(DayOfWeek.class, "INVALID").or(defaultDay);
assert day == DayOfWeek.SUNDAY;
```

Implémenter le modèle Singleton avec une énumération à un seul élément

Les constantes enum sont instanciées quand une énumération est référencée pour la première fois. Par conséquent, cela permet d'implémenter le modèle de conception du logiciel Singleton avec un enum à élément unique.

```
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
    }
}
```

Selon le livre "Effective Java" de Joshua Bloch, une énumération à un seul élément est le meilleur moyen d'implémenter un singleton. Cette approche présente les avantages suivants:

- sécurité du fil
- garantie d'instanciation unique
- sérialisation prête à l'emploi

Et comme le montre la section implémente l'interface, ce singleton pourrait également implémenter une ou plusieurs interfaces.

Enum avec des propriétés (champs)

Dans le cas où nous souhaitons utiliser `enum` avec plus d'informations et pas seulement comme des valeurs constantes, nous voulons pouvoir comparer deux énumérations.

Prenons l'exemple suivant:

```
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;
    }

}
```

Ici, nous avons défini un `Enum` appelé `Coin` qui représente sa valeur. Avec la méthode `isGreaterThan` on peut comparer deux `enum` s:

```
Coin penny = Coin.PENNY;
Coin dime = Coin.DIME;

System.out.println(penny.isGreaterThan(dime)); // prints: false
System.out.println(dime.isGreaterThan(penny)); // prints: true
```

Convertir enum en String

Parfois, vous souhaitez convertir votre enum en une chaîne, il y a deux façons de le faire.

Supposons que nous ayons:

```
public enum Fruit {
    APPLE, ORANGE, STRAWBERRY, BANANA, LEMON, GRAPE_FRUIT;
}
```

Alors, comment pouvons-nous convertir quelque chose comme `Fruit.APPLE` en `"APPLE"` ?

Convertir en utilisant le `name()`

`name()` est une méthode interne à `enum` qui renvoie la représentation `String` de l'énumération, la `String` retour représente ***exactement*** comment la valeur enum a été définie.

Par exemple:

```
System.out.println(Fruit.BANANA.name());       // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.name()); // "GRAPE_FRUIT"
```

Convertir en utilisant `toString()`

`toString()` est, *par défaut* , remplacé pour avoir le même comportement que `name()`

Cependant, `toString()` est probablement surchargée par les *développeurs* pour faire imprimer un

plus convivial utilisateur `String`

> N'utilisez pas `toString()` si vous voulez effectuer un archivage de votre code, `name()` est beaucoup plus stable pour cela. `toString()` lorsque vous allez afficher la valeur dans les journaux ou la sortie standard ou quelque chose

Par défaut:

```
System.out.println(Fruit.BANANA.toString());      // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "GRAPE_FRUIT"
```

Exemple de dépassement

```
System.out.println(Fruit.BANANA.toString());      // "Banana"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "Grape Fruit"
```

Enum corps spécifique constant

Dans un `enum` il est possible de définir un comportement spécifique pour une constante particulière de l' `enum` qui remplace le comportement par défaut de l' `enum` . Cette technique est appelée *corps spécifique constant* .

Supposons que trois élèves de piano - John, Ben et Luke - soient définis dans une `enum` appelée `PianoClass` , comme suit:

```
    enum PianoClass {
    JOHN, BEN, LUKE;
        public String getSex() {
            return "Male";
        }
        public String getLevel() {
            return "Beginner";
        }
    }
```

Et un jour, deux autres étudiants arrivent - Rita et Tom - avec un sexe (féminin) et un niveau (intermédiaire) qui ne correspondent pas aux précédents:

```
    enum PianoClass2 {
    JOHN, BEN, LUKE, RITA, TOM;
        public String getSex() {
            return "Male"; // issue, Rita is a female
        }
        public String getLevel() {
            return "Beginner"; // issue, Tom is an intermediate student
        }
    }
```

de sorte que le simple ajout des nouveaux étudiants à la déclaration constante, comme suit, n'est pas correct:

```
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
```

Il est possible de définir un comportement spécifique pour chaque constante, Rita et Tom, qui remplace le comportement par défaut de `PianoClass2` comme suit:

```
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";
    }
}
```

et maintenant le niveau de Tom et le sexe de Rita sont comme ils devraient être:

```
PianoClass3 tom = PianoClass3.TOM;
PianoClass3 rita = PianoClass3.RITA;
System.out.println(tom.getLevel()); // prints Intermediate
System.out.println(rita.getSex()); // prints Female
```

Une autre façon de définir un corps spécifique au contenu consiste à utiliser un constructeur, par exemple:

```
enum Friend {
    MAT("Male"),
    JOHN("Male"),
    JANE("Female");

    private String gender;

    Friend(String gender) {
        this.gender = gender;
    }

    public String getGender() {
        return this.gender;
    }
}
```

et utilisation:

```
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
```

Zéro instance 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...
}
```

Tout comme `enum` peut être utilisé pour les singletons (1 classe d'instance), il peut être utilisé pour les classes d'utilitaires (0 classes d'instance). Veillez simplement à terminer la liste (vide) des constantes enum avec un `;` .

Voir la question Énumération d'instance zéro vs constructeurs privés pour empêcher l'instanciation d'une discussion sur les avantages et les inconvénients par rapport aux constructeurs privés.

Enums avec des champs statiques

Si votre classe enum doit avoir des champs statiques, gardez à l'esprit qu'ils sont créés **après** les valeurs enum elles-mêmes. Cela signifie que le code suivant entraînera une `NullPointerException` :

```
enum Example {
    ONE(1), TWO(2);

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

    private Example(int value) {
        integers.put(this.name(), value);
    }
}
```

Un moyen possible de résoudre ce problème:

```
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);
    }
}
```

Ne pas initialiser le champ statique:

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

initialisation:

- créer les valeurs enum
  - comme effet secondaire appelé putValue () qui initialise les entiers
- les valeurs statiques sont définies
  - integers = null; // est exécuté après les énumérations pour que le contenu des entiers soit perdu

Comparer et contient les valeurs Enum

Enums ne contient que des constantes et peut être comparé directement avec `==` . Donc, seule la vérification de référence est nécessaire, pas besoin d'utiliser la méthode `.equals` . De plus, si les `.equals` utilisées de manière incorrecte, peuvent générer l' `NullPointerException` alors que ce n'est pas le cas avec la vérification `==` .

```
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!");
        }

    }
}
```

Pour grouper, compléter, `EnumSet` valeurs enum, nous avons la classe `EnumSet` qui contient différentes méthodes.

- `EnumSet#range` : Pour obtenir un sous-ensemble d'énum par plage définie par deux points de terminaison
- `EnumSet#of` : Ensemble de `EnumSet#of` spécifiques sans aucune plage. Plusieurs surcharge `of` méthodes sont là.
- `EnumSet#complementOf` : ensemble d'énum complément des valeurs enum fournies dans le paramètre 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");
      }
  }
}
```

Lire Enums en ligne: https://riptutorial.com/fr/java/topic/155/enums

Chapitre 60: Envoi de méthode dynamique

Introduction

Qu'est-ce que Dynamic Method Dispatch?

Dynamic Method Dispatch est un processus dans lequel l'appel à une méthode substituée est résolu à l'exécution plutôt qu'à la compilation. Lorsqu'une méthode substituée est appelée par une référence, Java détermine la version de cette méthode à exécuter en fonction du type d'objet auquel elle fait référence. Ceci est également connu sous le nom de polymorphisme d'exécution.

Nous verrons cela à travers un exemple.

Remarques

- Liaison dynamique = liaison tardive
- Les classes abstraites ne peuvent pas être instanciées, mais elles peuvent être sous-classées (Base pour une classe enfant)
- Une méthode abstraite est une méthode déclarée sans implémentation
- La classe abstraite peut contenir un mélange de méthodes déclarées avec ou sans implémentation
- Lorsqu'une classe abstraite est sous-classée, la sous-classe fournit généralement des implémentations pour toutes les méthodes abstraites de sa classe parente. Toutefois, si ce n'est pas le cas, la sous-classe doit également être déclarée abstraite.
- La distribution de méthode dynamique est un mécanisme par lequel un appel à une méthode remplacée est résolu au moment de l'exécution. C'est ainsi que Java implémente le polymorphisme d'exécution.
- Mise à jour: Transférer un sous-type sur un sur-type vers le haut de l'arbre d'héritage.
- Polymorphisme d'exécution = Polymorphisme dynamique

Examples

Dynamic Method Dispatch - Exemple de code

Classe abstraite:

```
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");
    }
}
```

Remplacement de la méthode abstraite dans la classe enfant:

```
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
```

La liaison dynamique / tardive conduit à l'envoi dynamique de la méthode:

```
//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();

    }
}
```

Résultat :

```
Scan for Internet Security
Detect Virus & Malware
```

Reprise:

```
objVS = new ClsFreeVersion();
objVS = new ClsPaidVersion();
objVS = new ClsTrialVersion()
```

Lire Envoi de méthode dynamique en ligne: https://riptutorial.com/fr/java/topic/9204/envoi-de-methode-dynamique

Chapitre 61: Évaluation XPath XML

Remarques

Les expressions XPath permettent de naviguer et de sélectionner un ou plusieurs nœuds dans un document d'arborescence XML, par exemple en sélectionnant un certain élément ou nœud d'attribut.

Voir cette recommandation du W3C pour une référence sur cette langue.

Examples

Évaluation d'un NodeList dans un document XML

Vu le document XML suivant:

```
<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>
```

Les éléments suivants récupèrent tous les noeuds d' `example` de la balise Java (utilisez cette méthode si vous n'évaluez qu'une seule fois XPath dans le fichier XML. Consultez d'autres exemples lorsque plusieurs appels XPath sont évalués dans le même fichier 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
...
```

Analyse de plusieurs expressions XPath en un seul XML

En utilisant le même exemple que **Evaluer un NodeList dans un document XML** , voici comment vous pouvez effectuer plusieurs appels XPath efficacement:

Vu le document XML suivant:

```
<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>
```

Voici comment utiliser XPath pour évaluer plusieurs expressions dans un document:

```
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

...
```

Analyse unique de XPath Expression plusieurs fois dans un XML

Dans ce cas, vous voulez que l'expression soit compilée avant les évaluations, afin que chaque appel à `evaluate` ne `compile` pas la même expression. La syntaxe simple serait:

```
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
```

Dans l'ensemble, deux appels à `XPathExpression.evaluate()` seront beaucoup plus efficaces que deux appels à `XPath.evaluate()` .

Lire Évaluation XPath XML en ligne: https://riptutorial.com/fr/java/topic/4148/evaluation-xpath-xml

Chapitre 62: Exceptions et gestion des exceptions

Introduction

Les objets de type `Throwable` et ses sous-types peuvent être envoyés à la pile avec le mot-clé `throw` et capturés avec `try…catch` instructions.

Syntaxe

- void someMethod () lève SomeException {} // déclaration de méthode, force les appelants de méthode à intercepter si SomeException est un type d'exception vérifié
- essayez {

```
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
```

  }
- enfin {

```
 //code that will always run, whether try block finishes or not
```

  }

Examples

Prendre une exception avec try-catch

Une exception peut être interceptée et gérée à l'aide de l'instruction `try...catch` . (En fait, les instructions `try` prennent d'autres formes, comme décrit dans d'autres exemples sur `try...catch...finally` et `try-with-resources` .)

Attrapez avec un bloc catch

La forme la plus simple ressemble à ceci:

```
try {
    doSomething();
} catch (SomeException e) {
    handle(e);
}
// next statement
```

Le comportement d'une simple `try...catch` est le suivant:

- Les instructions du bloc `try` sont exécutées.
- Si aucune exception n'est levée par les instructions du bloc `try` , le contrôle passe à l'instruction suivante après le `try...catch` .
- Si une exception est lancée dans le bloc `try` .
    - L'objet exception est testé pour voir s'il s'agit d'une instance de `SomeException` ou d'un sous-type.
    - S'il est, le `catch` bloc va *attraper* l'exception:
        - La variable `e` est liée à l'objet exception.
        - Le code dans le bloc `catch` est exécuté.
        - Si ce code renvoie une exception, l'exception nouvellement renvoyée est propagée à la place de celle d'origine.
        - Sinon, le contrôle passe à l'instruction suivante après le `try...catch` .
    - Si ce n'est pas le cas, l'exception d'origine continue à se propager.

Prise d'essai avec plusieurs prises

Un `try...catch` peut aussi avoir plusieurs blocs de `catch` . Par exemple:

```
try {
    doSomething();
} catch (SomeException e) {
    handleOneWay(e)
} catch (SomeOtherException e) {
    handleAnotherWay(e);
}
// next statement
```

S'il y a plusieurs blocs `catch` , ils sont essayés un à la fois en commençant par le premier, jusqu'à ce qu'une correspondance soit trouvée pour l'exception. Le gestionnaire correspondant est exécuté (comme ci-dessus), puis le contrôle est transmis à l'instruction suivante après l'instruction `try...catch` . Les blocs `catch` après celui qui correspond sont toujours ignorés, *même si le code du gestionnaire renvoie une exception* .

La stratégie d'appariement «descendante» a des conséquences dans les cas où les exceptions dans les blocs de `catch` ne sont pas disjointes. Par exemple:

```
try {
    throw new RuntimeException("test");
} catch (Exception e) {
    System.out.println("Exception");
} catch (RuntimeException e) {
```

```
    System.out.println("RuntimeException");
}
```

Cet extrait de code affichera "Exception" plutôt que "RuntimeException". Puisque `RuntimeException` est un sous-type d' `Exception` , le premier `catch` (plus général) sera mis en correspondance. La seconde `catch` (plus spécifique) ne sera jamais exécutée.

La leçon à en tirer est que les blocs de `catch` les plus spécifiques (en termes de types d'exception) devraient apparaître en premier et les plus généraux en dernier. (Certains compilateurs Java vous avertiront si un `catch` ne peut jamais être exécuté, mais ce n'est pas une erreur de compilation.)

Blocs d'interception multi-exception

Java SE 7

À partir de Java SE 7, un bloc `catch` unique peut gérer une liste d'exceptions non associées. Le type d'exception est répertorié, séparé par un symbole de barre verticale (`|`). Par exemple:

```
try {
    doSomething();
} catch (SomeException | SomeOtherException e) {
    handleSomeException(e);
}
```

Le comportement d'une capture multi-exception est une simple extension pour le cas à exception unique. Le `catch` correspond si l'exception levée correspond (au moins) à l'une des exceptions répertoriées.

Il y a une subtilité supplémentaire dans la spécification. Le type de `e` est une *union* synthétique des types d'exception de la liste. Lorsque la valeur de `e` est utilisée, son type statique est le supertype le moins commun de la union de type. Toutefois, si `e` est renvoyé dans le bloc `catch` , les types d'exception levés sont les types de l'union. Par exemple:

```
public void method() throws IOException, SQLException
    try {
        doSomething();
    } catch (IOException | SQLException e) {
        report(e);
        throw e;
    }
```

Dans ce qui précède, `IOException` et `SQLException` sont des exceptions vérifiées dont le supertype le moins commun est `Exception` . Cela signifie que la méthode de `report` doit correspondre au `report(Exception)` . Cependant, le compilateur sait que le `throw` ne peut lancer qu'une `SQLException` `IOException` ou une `SQLException` . Ainsi, la `method` peut être déclarée en tant que `throws IOException, SQLException` plutôt que `throws Exception` . (Ce qui est une bonne chose: voir Pitfall - Throwable, Exception, Error ou RuntimeException .)

Lancer une exception

L'exemple suivant montre les bases de la création d'une exception:

```
public void checkNumber(int number) throws IllegalArgumentException {
    if (number < 0) {
        throw new IllegalArgumentException("Number must be positive: " + number);
    }
}
```

L'exception est lancée sur la 3ème ligne. Cette déclaration peut être divisée en deux parties:

- `new IllegalArgumentException(...)` crée une instance de la classe `IllegalArgumentException` , avec un message décrivant l'erreur `IllegalArgumentException` par une exception.
- `throw ...` lance ensuite l'objet exception.

Lorsque l'exception est levée, les instructions englobantes se *terminent anormalement* jusqu'à ce que l'exception soit *gérée* . Ceci est décrit dans d'autres exemples.

Il est recommandé de créer et de lancer l'objet exception dans une seule instruction, comme indiqué ci-dessus. Il est également recommandé d'inclure un message d'erreur significatif dans l'exception pour aider le programmeur à comprendre la cause du problème. Cependant, ce n'est pas nécessairement le message que vous devez montrer à l'utilisateur final. (Pour commencer, Java ne prend pas en charge directement l'internationalisation des messages d'exception.)

Il y a encore quelques points à faire:

- Nous avons déclaré le `checkNumber` comme `throws IllegalArgumentException` . Ce n'était pas strictement nécessaire, car `IllegalArgumentException` est une exception vérifiée; voir Hiérarchie des exceptions Java - Exceptions non vérifiées et vérifiées . Toutefois, il est conseillé de le faire et d'inclure les exceptions renvoyant les commentaires javadoc d'une méthode.
- Code immédiatement après une instruction de `throw` est *inaccessible* . D'où si nous avons écrit ceci:

  ```
  throw new IllegalArgumentException("it is bad");
  return;
  ```

  le compilateur signalerait une erreur de compilation pour l'instruction `return` .

Chaîne d'exception

De nombreuses exceptions standard ont un constructeur avec un deuxième argument de `cause` en plus de l'argument de `message` conventionnel. La `cause` vous permet d'enchaîner les exceptions. Voici un exemple.

Tout d'abord, nous définissons une exception non vérifiée que notre application va lancer lorsqu'elle rencontre une erreur non récupérable. Notez que nous avons inclus un constructeur qui accepte un argument de `cause` .

```
public class AppErrorException extends RuntimeException {
    public AppErrorException() {
        super();
    }

    public AppErrorException(String message) {
        super(message);
    }

    public AppErrorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Ensuite, voici un code qui illustre le chaînage des exceptions.

```
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);
    }
}
```

Le `throw` dans le bloc `try` détecte un problème et le signale via une exception avec un message simple. En revanche, le `throw` dans le bloc `catch` gère l'exception `IOException` l' `IOException` dans une nouvelle exception (cochée). Cependant, il ne rejette pas l'exception d'origine. En faisant passer l' `IOException` tant que `cause` , nous l'enregistrons pour qu'elle puisse être imprimée dans la trace de la pile, comme expliqué dans la section Création et lecture de chaînes de caractères .

Exceptions personnalisées

Dans la plupart des cas, il est plus simple du point de vue de la conception de code d'utiliser des classes d' `Exception` génériques existantes lors du lancement d'exceptions. Cela est particulièrement vrai si vous avez uniquement besoin de l'exception pour transmettre un message d'erreur simple. Dans ce cas, RuntimeException est généralement préférable car il ne s'agit pas d'une exception vérifiée. D'autres classes d'exception existent pour les classes d'erreurs courantes:

- UnsupportedOperationException - une opération donnée n'est pas prise en charge
- IllegalArgumentException - une valeur de paramètre non valide a été transmise à une méthode
- IllegalStateException - votre API a atteint en interne une condition qui ne devrait jamais se produire ou qui résulte de l'utilisation de votre API de manière non valide

Les cas où vous **ne** souhaitez utiliser une classe d'exception personnalisée sont les suivantes:

- Vous écrivez une API ou une bibliothèque à utiliser par d'autres utilisateurs et vous

souhaitez autoriser les utilisateurs de votre API à intercepter et gérer spécifiquement les exceptions de votre API et *à les différencier des autres exceptions plus génériques* .
- Vous lancez des exceptions pour un **type d'erreur spécifique** dans une partie de votre programme, que vous souhaitez intercepter et gérer dans une autre partie de votre programme, et que vous souhaitez pouvoir différencier ces erreurs des autres erreurs plus génériques.

Vous pouvez créer vos propres exceptions personnalisées en étendant `RuntimeException` pour une exception non contrôlée ou en vérifiant l'exception en étendant toute `Exception` *qui n'est pas également la sous-classe de RuntimeException* , car:

> Les sous-classes d'exception qui ne sont pas également des sous-classes de RuntimeException sont des exceptions vérifiées

```
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;
    }
}
```

Ceux-ci peuvent être utilisés comme exceptions prédéfinies:

```
void validateString(String value){
    if (value.length() > 30){
        throw new StringTooLongException(value, 30);
    }
}
```

Et les champs peuvent être utilisés lorsque l'exception est interceptée et gérée:

```
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 );
    }
}
```

Gardez à l'esprit que, selon la documentation Java d' Oracle :

> [...] Si l'on peut raisonnablement s'attendre à ce qu'un client récupère d'une exception, faites-en une exception cochée. Si un client ne peut rien faire pour récupérer à partir de l'exception, faites-en une exception non contrôlée.

Plus:

- Pourquoi RuntimeException ne nécessite-t-il pas une gestion des exceptions explicite?

La déclaration d'essayer avec les ressources

Java SE 7

Comme l'illustre l'exemple de l' instruction try-catch-final , le nettoyage des ressources à l'aide d'une clause `finally` nécessite une quantité importante de code "chaud-plat" pour implémenter correctement les bordures. Java 7 fournit un moyen beaucoup plus simple de résoudre ce problème sous la forme de l'instruction *try-with-resources* .

Qu'est ce qu'une ressource?

Java 7 a introduit l'interface `java.lang.AutoCloseable` pour permettre la gestion des classes à l'aide de l'instruction *try-with-resources* . Les instances de classes qui implémentent `AutoCloseable` sont appelées *ressources* . Celles-ci doivent généralement être éliminées en temps opportun plutôt que de compter sur le ramasse-miettes pour en disposer.

L'interface `AutoCloseable` définit une méthode unique:

```
public void close() throws Exception
```

Une méthode `close()` doit éliminer la ressource de manière appropriée. La spécification indique qu'il est prudent d'appeler la méthode sur une ressource déjà supprimée. De plus, les classes qui implémentent `Autocloseable` sont *fortement encouragées* à déclarer la méthode `close()` pour générer une exception plus spécifique que `Exception` , voire aucune exception.

Un large éventail d'interfaces et de classes Java standard implémentent `AutoCloseable` . Ceux-ci inclus:

- `InputStream` , `OutputStream` et leurs sous-classes
- `Reader` , `Writer` et leurs sous-classes
- `Socket` et `ServerSocket` et leurs sous-classes
- `Channel` et ses sous-classes, et
- les interfaces JDBC `Connection` , `Statement` et `ResultSet` et leurs sous-classes.

Les classes d'application et tierces peuvent également le faire.

L'énoncé de base de try-with-resource

La syntaxe d'un *try-with-resources* est basée sur des formes classiques de *try-catch* , *try-finally* et *try-catch-finally* . Voici un exemple de formulaire "de base"; c'est à dire la forme sans `catch` ou `finally` .

```
try (PrintStream stream = new PrintStream("hello.txt")) {
```

```
    stream.println("Hello world!");
}
```

Les ressources à gérer sont déclarées comme variables dans la section `(...)` après la clause `try` . Dans l'exemple ci-dessus, nous déclarons un `stream` variable de ressource et l'initialisons à un nouveau `PrintStream` .

Une fois les variables de ressource initialisées, le bloc `try` est exécuté. Lorsque cela est terminé, `stream.close()` sera appelé automatiquement pour garantir que la ressource ne fuit pas. Notez que l'appel `close()` se produit peu importe la manière dont le bloc se termine.

Les instructions try-with-resource améliorées

L'instruction *try-with-resources* peut être améliorée avec `catch` blocs `catch` et `finally` , comme avec la syntaxe *try-catch-finally* pré-Java 7. L'extrait de code suivant ajoute un bloc `catch` à notre précédent pour gérer l' `FileNotFoundException` que le constructeur `PrintStream` peut lancer:

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
} catch (FileNotFoundException ex) {
    System.err.println("Cannot open the file");
} finally {
    System.err.println("All done");
}
```

Si l'initialisation de la ressource ou le bloc try lève l'exception, le bloc `catch` sera exécuté. Le bloc `finally` sera toujours exécuté, comme avec une instruction *try-catch-finally* classique.

Il y a quelques choses à noter cependant:

- La variable de ressource est *hors de portée* dans les blocs `catch` et `finally` .
- Le nettoyage des ressources se produira avant que l'instruction tente de correspondre au bloc `catch` .
- Si le nettoyage automatique des ressources a déclenché une exception, cela *peut se* produire dans l'un des blocs `catch` .

Gestion de plusieurs ressources

Les extraits de code ci-dessus montrent une seule ressource en cours de gestion. En fait, *try-with-resources* peut gérer plusieurs ressources dans une seule déclaration. Par exemple:

```
try (InputStream is = new FileInputStream(file1);
     OutputStream os = new FileOutputStream(file2)) {
    // Copy 'is' to 'os'
}
```

Cela se comporte comme prévu. Les deux `is` et `os` sont fermés automatiquement à la fin de l' `try` bloc. Il y a quelques points à noter:

- Les initialisations se produisent dans l'ordre du code, et les initialiseurs de variables de ressource ultérieurs peuvent utiliser les valeurs des précédentes.
- Toutes les variables de ressources initialisées avec succès seront nettoyées.
- Les variables de ressource sont nettoyées dans *l'ordre inverse* de leurs déclarations.

Ainsi, dans l'exemple ci - dessus, `is` est initialisé avant `os` et nettoyé après, et `is` sera nettoyé s'il y a une exception lors de l' initialisation `os` .

Équivalence d'essais avec ressources et d'essais classiques

La spécification de langage Java spécifie le comportement des formulaires *try-with-resource* en fonction de l'instruction *try-catch-finally* classique. (Veuillez vous référer au JLS pour plus de détails.)

Par exemple, ce *try-with-resource de base* :

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}
```

est défini pour être équivalent à ce *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);
        }
    }
}
```

(Le JLS spécifie que les variables `t` et `primaryException` seront invisibles pour le code Java normal.)

La forme améliorée de *try-with-resources* est spécifiée comme une équivalence avec la forme de base. Par exemple:

```
try (PrintStream stream = new PrintStream(fileName)) {
    stream.println("Hello world!");
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}
```

est équivalent à:

```
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");
}
```

Création et lecture de stacktraces

Lorsqu'un objet d'exception est créée (par exemple lorsque vous `new` il), le `Throwable` constructeur capture d' informations sur le contexte dans lequel l'exception a été créé. Par la suite, ces informations peuvent être générées sous la forme d'un stacktrace, qui peut être utilisé pour diagnostiquer le problème à l'origine de l'exception.

Impression d'un stacktrace

Imprimer une stacktrace consiste simplement à appeler la méthode `printStackTrace()` . Par exemple:

```
try {
    int a = 0;
    int b = 0;
    int c = a / b;
} catch (ArithmeticException ex) {
    // This prints the stacktrace to standard output
    ex.printStackTrace();
}
```

La méthode `printStackTrace()` sans arguments imprimera sur la sortie standard de l'application; c'est-à-dire le `System.out` actuel. Il existe également des `printStackTrace(PrintStream)` et des `printStackTrace(PrintWriter)` qui impriment sur un `Stream` ou un `Writer` spécifié.

Remarques:

1. Le stacktrace n'inclut pas les détails de l'exception elle-même. Vous pouvez utiliser la `toString()` pour obtenir ces détails. par exemple

```
    // Print exception and stacktrace
    System.out.println(ex);
    ex.printStackTrace();
```

2. L'impression Stacktrace doit être utilisée avec parcimonie. voir Piège - Piles superflues ou inappropriées . Il est souvent préférable d'utiliser une infrastructure de journalisation et de passer l'objet exception à journaliser.

Comprendre un stacktrace

Considérons le programme simple suivant composé de deux classes dans deux fichiers. (Nous avons montré les noms de fichiers et les numéros de ligne ajoutés à des fins d'illustration.)

```
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      }
```

Lorsque ces fichiers sont compilés et exécutés, nous obtiendrons la sortie suivante.

```
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)
```

Lisons cette ligne à la fois pour comprendre ce qu'elle nous dit.

La ligne n ° 1 nous indique que le thread appelé "main" est terminé en raison d'une exception non interceptée. Le nom complet de l'exception est `java.lang.ArithmeticException` et le message d'exception est "/ by zero".

Si nous recherchons les javadocs pour cette exception, cela dit:

Lancé lorsqu'une condition arithmétique exceptionnelle s'est produite. Par exemple, un entier "diviser par zéro" lève une instance de cette classe.

En effet, le message "/ by zero" est un indice fort que la cause de l'exception est qu'un code a tenté de diviser quelque chose par zéro. Mais quoi?

Les 3 lignes restantes sont la trace de la pile. Chaque ligne représente un appel de méthode (ou de constructeur) sur la pile d'appels, et chacune nous indique trois choses:

- le nom de la classe et de la méthode en cours d'exécution,
- le nom de fichier du code source,
- le numéro de ligne du code source de l'instruction en cours d'exécution

Ces lignes d'un stacktrace sont répertoriées avec le cadre de l'appel en cours. Le cadre supérieur de notre exemple ci-dessus se trouve dans la méthode `Test.bar` et à la ligne 9 du fichier Test.java. C'est la ligne suivante:

```
    return a / b;
```

Si nous examinons quelques lignes plus tôt dans le fichier où `b` est initialisé, il est évident que `b` aura la valeur zéro. Nous pouvons dire sans aucun doute que c'est la cause de l'exception.

Si nous devions aller plus loin, nous pouvons voir à partir du stacktrace que `bar()` été appelé depuis `foo()` à la ligne 3 de Test.java, et que `foo()` été à son tour appelé depuis `Main.main()` .

Remarque: Les noms de classe et de méthode dans les cadres de pile sont les noms internes des classes et des méthodes. Vous devrez reconnaître les cas inhabituels suivants:

- Une classe imbriquée ou interne ressemblera à "OuterClass $ InnerClass".
- Une classe interne anonyme ressemblera à "OuterClass $ 1", "OuterClass $ 2", etc.
- Lorsque du code dans un constructeur, un initialiseur de champ d'instance ou un bloc d'initialisation d'instance est en cours d'exécution, le nom de la méthode sera "".
- Lorsque le code d'un initialiseur de champ statique ou d'un bloc d'initialisation statique est en cours d'exécution, le nom de la méthode sera "".

(Dans certaines versions de Java, le code de formatage stacktrace détectera et éliminera les séquences répétées de pile, comme cela peut se produire lorsqu'une application échoue en raison d'une récursivité excessive.)

Chaînage des exceptions et empilements imbriqués

Java SE 1.4

Le chaînage des exceptions se produit lorsqu'un morceau de code intercepte une exception, puis en crée et en lance une nouvelle, en faisant passer la première exception comme cause. Voici un exemple:

```
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  }
```

Lorsque la classe ci-dessus est compilée et exécutée, nous obtenons le stacktrace suivant:

```
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
```

Le stacktrace commence par le nom de la classe, la méthode et la pile d'appels correspondant à l'exception qui (dans ce cas) a provoqué le blocage de l'application. Ceci est suivi par une ligne "Caused by:" qui signale l'exception de `cause` . Le nom de la classe et le message sont signalés, suivis des cadres de pile de l'exception cause. La trace se termine par un "... N more" qui indique que les N dernières images sont les mêmes que pour l'exception précédente.

Le "Causé par:" est uniquement inclus dans la sortie lorsque la `cause` l'exception principale n'est pas `null` . Les exceptions peuvent être chaînées indéfiniment et, dans ce cas, le stacktrace peut avoir plusieurs traces "Caused by:".

Remarque: le mécanisme de `cause` était uniquement exposé dans l'API `Throwable` de Java 1.4.0. Avant cela, le chaînage des exceptions devait être implémenté par l'application en utilisant un champ d'exception personnalisé pour représenter la cause et une méthode `printStackTrace` personnalisée.

Capturer une stacktrace en tant que chaîne

Parfois, une application doit être capable de capturer un stacktrace en tant que `String` Java, afin de pouvoir l'utiliser à d'autres fins. L'approche générale pour ce faire consiste à créer un `OutputStream` ou un `Writer` temporaire qui écrit dans un tampon en mémoire et le transmet à `printStackTrace(...)` .

Les bibliothèques Apache Commons et Guava fournissent des méthodes utilitaires pour capturer une stacktrace en tant que chaîne:

```
org.apache.commons.lang.exception.ExceptionUtils.getStackTrace(Throwable)

com.google.common.base.Throwables.getStackTraceAsString(Throwable)
```

Si vous ne pouvez pas utiliser de bibliothèques tierces dans votre base de code, la méthode suivante effectue la tâche:

```
   /**
     * 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();
    }
```

Notez que si vous avez l'intention d'analyser le stacktrace, il est plus simple d'utiliser `getStackTrace()` et `getCause()` que d'essayer d'analyser un stacktrace.

Traitement des exceptions interrompues

`InterruptedException` est une bête déroutante - elle apparaît dans des méthodes apparemment anodines comme `Thread.sleep()` , mais sa gestion incorrecte conduit à un code difficile à gérer qui se comporte mal dans des environnements concurrents.

À la base, si une `InterruptedException` est interceptée, cela signifie que quelqu'un, quelque part, appelé `Thread.interrupt()` sur le thread dans `Thread.interrupt()` votre code est actuellement exécuté. Vous pourriez être enclin à dire "C'est mon code! Je ne l'interromprai jamais!" " et donc faire quelque chose comme ça:

```
// Bad. Don't do this.
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  // disregard
}
```

Mais c'est exactement la mauvaise façon de gérer un événement "impossible". Si vous savez que votre application ne rencontrera jamais une `InterruptedException` vous devez traiter un tel événement comme une violation grave des hypothèses de votre programme et quitter le plus rapidement possible.

La manière correcte de traiter une interruption "impossible" est comme suit:

```
// When nothing will interrupt your code
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  Thread.currentThread().interrupt();
```

```
  throw new AssertionError(e);
}
```

Cela fait deux choses; il restaure d'abord le statut d'interruption du thread (comme si l'`InterruptedException` n'avait pas été lancée en premier lieu), puis lance une `AssertionError` indiquant que les invariants de base de votre application ont été violés. Si vous savez avec certitude que vous n'interromprez jamais le thread, ce code s'exécute en toute sécurité car le bloc `catch` ne doit jamais être atteint.

L'utilisation de la classe `Uninterruptibles` de Guava permet de simplifier ce modèle; L'appel à `Uninterruptibles.sleepUninterruptibly()` interrompt l'état interrompu d'un thread jusqu'à ce que la durée de veille expire (à quel moment il est restauré pour que les appels ultérieurs inspectent et lancent leur propre `InterruptedException`). Si vous savez que vous n'interromprez jamais ce code, cela évite d'avoir à envelopper vos appels de veille dans un bloc try-catch.

Plus souvent, cependant, vous ne pouvez pas garantir que votre thread ne sera jamais interrompu. En particulier, si vous écrivez du code qui sera exécuté par un `Executor` ou par un autre gestionnaire de threads, il est essentiel que votre code réponde rapidement aux interruptions, sinon votre application sera bloquée, voire bloquée.

Dans de tels cas, la meilleure chose à faire est généralement de permettre à `InterruptedException` de propager la pile d'appels, en ajoutant une `throws InterruptedException` à chaque méthode. Cela peut sembler compliqué mais c'est en fait une propriété souhaitable - les signatures de votre méthode indiquent maintenant aux appelants qu'ils répondront rapidement aux interruptions.

```
// Let the caller determine how to handle the interrupt if you're unsure
public void myLongRunningMethod() throws InterruptedException {
  ...
}
```

Dans des cas limités (par exemple, lors du remplacement d'une méthode ne `throw` aucune exception vérifiée), vous pouvez réinitialiser le statut interrompu sans déclencher une exception, en attendant que le code soit exécuté pour gérer l'interruption. Cela retarde le traitement de l'interruption, mais ne le supprime pas entièrement.

```
// 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 hiérarchie des exceptions Java - Exceptions non vérifiées et vérifiées

Toutes les exceptions Java sont des instances de classes dans la hiérarchie de classes Exception. Cela peut être représenté comme suit:

- `java.lang.Throwable` - Il s'agit de la classe de base pour toutes les classes d'exception. Ses

méthodes et ses constructeurs implémentent une gamme de fonctionnalités communes à toutes les exceptions.
    - `java.lang.Exception` - Ceci est la super-classe de toutes les exceptions normales.
        - diverses classes d'exception standard et personnalisées.
        - `java.lang.RuntimeException` - C'est la super-classe de toutes les exceptions normales qui sont des *exceptions non vérifiées* .
            - diverses classes d'exceptions d'exécution standard et personnalisées.
    - `java.lang.Error` - Ceci est la super-classe de toutes les exceptions "d'erreur fatale".

Remarques:

1. La distinction entre *les* exceptions *cochées* et *non vérifiées* est décrite ci-dessous.
2. La `Throwable` , `Exception` et `RuntimeException` doit être traitée comme `abstract` . voir Pitfall - Throwable, Exception, Error ou RuntimeException .
3. Les exceptions d' `Error` sont renvoyées par la machine virtuelle Java dans les situations où il serait dangereux ou imprudent qu'une application tente de récupérer.
4. Il serait imprudent de déclarer des sous-types personnalisés de `Throwable` . Les outils et bibliothèques Java peuvent supposer que les `Error` et `Exception` sont les seuls sous-types directs de `Throwable` , et se comporter de manière incorrecte si cette hypothèse est incorrecte.

Exceptions vérifiées et non vérifiées

L'une des critiques de la prise en charge des exceptions dans certains langages de programmation est qu'il est difficile de savoir quelles exceptions une méthode ou une procédure donnée peut générer. Étant donné qu'une exception non gérée risque de provoquer le blocage d'un programme, cela peut faire des exceptions une source de fragilité.

Le langage Java résout ce problème avec le mécanisme d'exception vérifié. Tout d'abord, Java classe les exceptions en deux catégories:

- Les exceptions vérifiées représentent généralement les événements anticipés qu'une application doit pouvoir gérer. Par exemple, `IOException` et ses sous-types représentent des conditions d'erreur pouvant se produire dans les opérations d'E / S. Les exemples incluent, l'ouverture du fichier échoue car un fichier ou un répertoire n'existe pas, les lectures et écritures réseau échouent car une connexion réseau a été interrompue, etc.

- Les exceptions non vérifiées représentent généralement des événements imprévus auxquels une application ne peut pas faire face. Ce sont généralement le résultat d'un bogue dans l'application.

(Dans ce qui suit, "renvoyé" fait référence à toute exception envoyée explicitement (par une instruction `throw`), ou implicitement (dans un déréférencement ayant échoué, tapez cast, etc.). De même, "propagated" fait référence à une exception lancée dans un appel imbriqué, et non pris en compte dans cet appel. L'exemple de code ci-dessous illustre cela.)

La deuxième partie du mécanisme d'exception vérifié est qu'il existe des restrictions sur les méthodes pour lesquelles une exception vérifiée peut se produire:

- Lorsqu'une exception vérifiée est lancée ou propagée dans une méthode, elle *doit* soit être interceptée par la méthode, soit répertoriée dans la clause `throws` la méthode. (L'importance de la `throws` clause est décrit dans cet exemple .)
- Lorsqu'une exception vérifiée est lancée ou propagée dans un bloc d'initialisation, elle doit être interceptée dans le bloc.
- Une exception vérifiée ne peut pas être propagée par un appel de méthode dans une expression d'initialisation de champ. (Il n'y a aucun moyen d'attraper une telle exception.)

En bref, une exception vérifiée doit être gérée ou déclarée.

Ces restrictions ne s'appliquent pas aux exceptions non vérifiées. Cela inclut tous les cas où une exception est lancée implicitement, car tous ces cas provoquent des exceptions non vérifiées.

Exemples d'exceptions vérifiés

Ces extraits de code sont destinés à illustrer les restrictions d'exception vérifiées. Dans chaque cas, nous affichons une version du code avec une erreur de compilation et une seconde version avec l'erreur corrigée.

```
// 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.
}
```

Le premier exemple montre comment les exceptions vérifiées peuvent être déclarées "lancées" si elles ne doivent pas être gérées dans la méthode.

```
// 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
    }
}
```

Le deuxième exemple montre comment traiter une exception vérifiée propagée.

```
// 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'exemple final montre comment gérer une exception vérifiée dans un initialiseur de champ statique.

```
// 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;
    }
}
```

Notez que dans ce dernier cas, nous devons aussi faire face aux problèmes qui `is` ne peuvent pas être attribués à plus d'une fois, mais doit également être attribué à, même dans le cas d'une exception.

introduction

Les exceptions sont des erreurs qui surviennent lors de l'exécution d'un programme. Considérons le programme Java ci-dessous qui divise deux entiers.

```
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);
    }
}
```

Maintenant, nous compilons et exécutons le code ci-dessus et voyons la sortie pour une tentative de division par zéro:

```
Input two integers
7 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Division.main(Disivion.java:14)
```

Division par zéro est une opération invalide qui produirait une valeur qui ne peut être représentée par un entier. Java gère cela en *lançant une* ***exception*** . Dans ce cas, l'exception est une instance de la classe *ArithmeticException* .

Remarque: L'exemple de création et de lecture de traces de pile explique la signification de la sortie après les deux nombres.

L'utilitaire d'une *exception* est le contrôle de flux qu'il autorise. Sans exceptions, une solution typique à ce problème peut être de vérifier si `b == 0` :

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

        int a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("Input two integers");

        a = input.nextInt();
        b = input.nextInt();

        if (b == 0) {
            System.out.println("You cannot divide by zero.");
            return;
        }

        result = a / b;

        System.out.println("Result = " + result);
    }
}
```

Ceci imprime le message `You cannot divide by zero.` à la console et quitte le programme de manière harmonieuse lorsque l'utilisateur essaie de diviser par zéro. Une manière équivalente de traiter ce problème via la *gestion des exceptions* serait de remplacer le contrôle de flux `if` par un bloc `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 bloc try est exécuté comme suit:

1. Commencez à exécuter le code dans le bloc `try` .
2. Si une *exception* se produit dans le bloc try, abandonnez immédiatement et vérifiez si cette exception est *interceptée* par le bloc `catch` (dans ce cas, lorsque l'exception est une instance de `ArithmeticException`).
3. Si l'exception est *interceptée* , elle est affectée à la variable `e` et le bloc `catch` est exécuté.
4. Si le bloc `try` ou `catch` est terminé (c'est-à-dire qu'aucune exception n'est interceptée lors de l'exécution du code), continuez à exécuter le code sous le bloc `try-catch` .

Il est généralement considéré comme une bonne pratique d'utiliser la *gestion des exceptions* dans le cadre du contrôle de flux normal d'une application où le comportement serait autrement indéfini ou inattendu. Par exemple, au lieu de retourner `null` lorsqu'une méthode échoue, il est généralement préférable de *lancer une exception* pour que l'application utilisant la méthode puisse définir son propre contrôle de flux pour la situation via la *gestion* des *exceptions* du type illustré ci-dessus. D'une certaine manière, cela résout le problème de devoir renvoyer un *type* particulier, étant donné que plusieurs types d' *exceptions* peuvent être *générés* pour indiquer le problème spécifique survenu.

Pour plus de conseils sur comment et comment ne pas utiliser les exceptions, reportez-vous à la page sur les pièges Java - Utilisation des exceptions

Retour d'instructions dans le bloc try catch

Bien que ce soit une mauvaise pratique, il est possible d'ajouter plusieurs instructions de retour dans un bloc de gestion des exceptions:

```
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;
    }
}
```

Cette méthode renverra toujours 7 car le bloc finally associé au bloc try / catch est exécuté avant que tout soit renvoyé. Maintenant, comme a finalement le `return 7;` , cette valeur remplace les valeurs de retour try / catch.

Si le bloc catch retourne une valeur primitive et que cette valeur primitive est modifiée ultérieurement dans le bloc finally, la valeur renvoyée dans le bloc catch sera renvoyée et les modifications du bloc finally seront ignorées.

L'exemple ci-dessous affichera "0" et 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;
        }
    }
}
```

Fonctionnalités avancées des exceptions

Cet exemple couvre certaines fonctionnalités avancées et cas d'utilisation des exceptions.

Examiner le callstack par programmation

Java SE 1.4

La principale utilisation des stacktraces d'exception est de fournir des informations sur une erreur

d'application et son contexte, de sorte que le programmeur puisse diagnostiquer et résoudre le problème. Parfois, il peut être utilisé pour d'autres choses. Par exemple, une classe `SecurityManager` peut avoir besoin d'examiner la pile d'appels pour décider si le code qui effectue un appel doit être approuvé.

Vous pouvez utiliser des exceptions pour examiner la pile d'appels par programme de la manière suivante:

```
    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());
```

Il y a quelques mises en garde importantes à ce sujet:

1. Les informations disponibles dans un `StackTraceElement` sont limitées. Il n'y a pas plus d'informations disponibles que celles affichées par `printStackTrace` . (Les valeurs des variables locales dans le cadre ne sont pas disponibles.)

2. Javadocs pour `getStackTrace()` indique qu'une machine virtuelle `getStackTrace()` est autorisée à laisser de côté les cadres:

   > Certaines machines virtuelles peuvent, dans certaines circonstances, omettre un ou plusieurs cadres de pile de la trace de pile. Dans le cas extrême, une machine virtuelle qui ne dispose d'aucune information de suivi de pile concernant ce produit jetable est autorisée à renvoyer un tableau de longueur nulle à partir de cette méthode.

Optimiser la construction des exceptions

Comme mentionné ailleurs, la construction d'une exception est assez coûteuse car elle implique la capture et l'enregistrement d'informations sur tous les cadres de pile du thread en cours. Parfois, nous savons que cette information ne sera jamais utilisée pour une exception donnée; par exemple, le stacktrace ne sera jamais imprimé. Dans ce cas, il existe un truc d'implémentation que nous pouvons utiliser dans une exception personnalisée pour que les informations ne soient pas capturées.

Les informations sur les trames de pile nécessaires pour les stacktraces sont capturées lorsque les constructeurs `Throwable` appellent la méthode `Throwable.fillInStackTrace()` . Cette méthode est `public` , ce qui signifie qu'une sous-classe peut la remplacer. L'astuce consiste à remplacer la méthode héritée de `Throwable` par une méthode qui ne fait rien; par exemple

```
public class MyException extends Exception {
    // constructors

    @Override
    public void fillInStackTrace() {
        // do nothing
    }
}
```

Le problème avec cette approche est qu'une exception qui remplace `fillInStackTrace()` ne peut jamais capturer la trace de la pile et est inutile dans les scénarios où vous en avez besoin.

Effacer ou remplacer le stacktrace

Java SE 1.4

Dans certains cas, le stacktrace d'une exception créée de manière normale contient des informations incorrectes ou des informations que le développeur ne souhaite pas révéler à l'utilisateur. Pour ces scénarios, `Throwable.setStackTrace` peut être utilisé pour remplacer le tableau d'objets `StackTraceElement` les informations.

Par exemple, les éléments suivants peuvent être utilisés pour ignorer les informations de pile d'une exception:

```
 exception.setStackTrace(new StackTraceElement[0]);
```

Suppression d'exceptions

Java SE 7

Java 7 a introduit la construction *try-with-resources* et le concept associé de suppression des exceptions. Considérez l'extrait suivant:

```
try (Writer w = new BufferedWriter(new FileWriter(someFilename))) {
    // do stuff
    int temp = 0 / 0;    // throws an ArithmeticException
}
```

Lorsque l'exception est levée, `try` appelle `close()` sur le `w` qui `FileWriter` toute sortie mise en mémoire tampon et fermera ensuite `FileWriter` . Mais que se passe-t-il si une `IOException` est lancée lors du vidage de la sortie?

Qu'est-ce qui se passe est que toute exception qui est levée lors du nettoyage d'une ressource est *supprimée* . L'exception est interceptée et ajoutée à la liste des exceptions supprimées de l'exception principale. Ensuite, le *try-with-resources* continuera avec le nettoyage des autres ressources. Enfin, l’exception primaire sera relancée.

Un modèle similaire se produit si une exception est générée lors de l'initialisation de la ressource ou si le bloc `try` termine normalement. La première exception lancée devient la principale exception, et celles qui découlent du nettoyage sont supprimées.

Les exceptions supprimées peuvent être extraites de l'objet exception primaire en appelant `getSuppressedExceptions` .

Les instructions try-finally et try-catch-finally

L'instruction `try...catch...finally` combine la gestion des exceptions avec le code de nettoyage.

Le bloc `finally` contient le code qui sera exécuté dans toutes les circonstances. Cela les rend appropriés pour la gestion des ressources et d'autres types de nettoyage.

Essayez-enfin

Voici un exemple de la forme plus simple (`try...finally`):

```
try {
    doSomething();
} finally {
    cleanUp();
}
```

Le comportement du `try...finally` se présente comme suit:

- Le code dans le bloc `try` est exécuté.
- Si aucune exception n'a été émise dans le bloc `try` :
    - Le code dans le bloc `finally` est exécuté.
    - Si le bloc `finally` lève une exception, cette exception est propagée.
    - Sinon, le contrôle passe à l'instruction suivante après l' `try...finally` .
- Si une exception a été émise dans le bloc try:
    - Le code dans le bloc `finally` est exécuté.
    - Si le bloc `finally` lève une exception, cette exception est propagée.
    - Sinon, l'exception d'origine continue à se propager.

Le code dans `finally` bloc sera toujours exécuté. (Les seules exceptions sont si `System.exit(int)` est appelé ou si la JVM panique.) Ainsi, un bloc `finally` est le code de lieu correct qui doit toujours être exécuté; fermeture de fichiers et autres ressources ou libération de verrous.

essayer-enfin

Notre deuxième exemple montre comment `catch` et `finally` peuvent être utilisés ensemble. Cela montre également que le nettoyage des ressources n'est pas simple.

```
// 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'ensemble complet des comportements (hypothétiques) de `try...catch...finally` dans cet exemple est trop compliqué à décrire ici. La version simple est que le code dans le bloc `finally` sera toujours exécuté.

En regardant cela du point de vue de la gestion des ressources:

- Nous déclarons la "ressource" (c.-à `reader` variable `reader`) avant le bloc `try` afin qu'elle soit possible pour le bloc `finally` .
- En mettant le `new FileReader(...)` , le `catch` est capable de gérer toute exception `IOError` lors de l'ouverture du fichier.
- Nous avons besoin d'un `reader.close()` dans le bloc `finally` car il existe des chemins d'exception que nous ne pouvons pas intercepter ni dans le bloc `try` ni dans le bloc `catch` .
- Cependant, une exception *pouvant* avoir été émise avant l'initialisation du `reader` , nous avons également besoin d'un test `null` explicite.
- Enfin, l'appel `reader.close()` pourrait (hypothétiquement) lancer une exception. Nous ne nous soucions pas de cela, mais si nous n'attrapons pas l'exception à la source, nous aurons besoin de la traiter plus haut dans la pile des appels.

Java SE 7

Java 7 et les versions ultérieures fournissent une syntaxe alternative à l'utilisation des ressources qui simplifie considérablement le nettoyage des ressources.

La clause 'throws' dans une déclaration de méthode

Le mécanisme d' *exception vérifié* de Java nécessite que le programmeur déclare que certaines méthodes *peuvent* générer des exceptions vérifiées spécifiées. Ceci est fait à l' aide de la `throws` clause. Par exemple:

```
public class OddNumberException extends Exception { // a checked exception
}

public void checkEven(int number) throws OddNumberException {
    if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

`throws OddNumberException` déclenche une `throws OddNumberException` indiquant qu'un appel à `checkEven` *peut* `checkEven` une exception de type `OddNumberException` .

Une `throws` clause peut déclarer une liste des types, et peut inclure des exceptions non vérifiées ainsi que les exceptions vérifiées.

```
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();
    }
```

```
}
```

Quel est l'intérêt de déclarer des exceptions non vérifiées comme levées?

La `throws` clause dans une déclaration de méthode a deux objectifs:

1. Il indique au compilateur quelles exceptions sont générées pour que le compilateur puisse signaler les exceptions non détectées (vérifiées) en tant qu'erreurs.
2. Il indique à un programmeur qui écrit du code qui appelle la méthode quelles exceptions attendre. À cette fin, il est souvent logique d'inclure des exceptions non vérifiées dans une liste de `throws` .

Remarque: la liste de `throws` est également utilisée par l'outil javadoc lors de la génération de la documentation de l'API, ainsi que par les astuces de la méthode typique de texte de survol de l'EDI.

Les jetons et la méthode

Les `throws` formes clause partie de la signature d'une méthode dans le but de passer outre la méthode. Une méthode de substitution peut être déclarée avec le même ensemble d'exceptions vérifiées que celles générées par la méthode substituée ou avec un sous-ensemble. Toutefois, la méthode de remplacement ne peut pas ajouter des exceptions vérifiées supplémentaires. Par exemple:

```
@Override
public void checkEven(int number) throws NullPointerException // OK—NullPointerException is an
unchecked exception
    ...

@Override
public void checkEven(Double number) throws OddNumberException // OK—identical to the
superclass
    ...

class PrimeNumberException extends OddNumberException {}
class NonEvenNumberException extends OddNumberException {}

@Override
public void checkEven(int number) throws PrimeNumberException, NonEvenNumberException //
OK—these are both subclasses

@Override
public void checkEven(Double number) throws IOExcepion         // ERROR
```

La raison de cette règle est que si une méthode de substitution peut lancer une exception vérifiée que la méthode surchargée ne peut pas lancer, cela pourrait casser la substituabilité du type.

Lire Exceptions et gestion des exceptions en ligne:

https://riptutorial.com/fr/java/topic/89/exceptions-et-gestion-des-exceptions

Chapitre 63: Expressions

Introduction

Les expressions en Java sont la construction principale pour effectuer des calculs.

Remarques

Pour obtenir une référence sur les opérateurs pouvant être utilisés dans les expressions, voir Opérateurs .

Examples

Priorité de l'opérateur

Lorsqu'une expression contient plusieurs opérateurs, elle peut potentiellement être lue de différentes manières. Par exemple, l'expression mathématique `1 + 2 x 3` peut être lue de deux manières:

1. Ajoutez `1` et `2` et multipliez le résultat par `3` . Cela donne la réponse `9` . Si nous ajoutions des parenthèses, cela ressemblerait à `( 1 + 2 ) x 3` .
2. Ajouter `1` au résultat de la multiplication par `2` et `3` . Cela donne la réponse `7` . Si nous ajoutions des parenthèses, cela ressemblerait à `1 + ( 2 x 3 )` .

En mathématiques, la convention est de lire l'expression de la seconde manière. La règle générale est que la multiplication et la division se font avant l'addition et la soustraction. Lorsqu'une notation mathématique plus avancée est utilisée, soit la signification est "évidente" (pour un mathématicien qualifié!), Soit des parenthèses sont ajoutées pour désambiguïser. Dans les deux cas, l'efficacité de la notation pour transmettre le sens dépend de l'intelligence et de la connaissance partagée des mathématiciens.

Java a les mêmes règles claires sur la façon de lire une expression, en fonction de la *priorité* des opérateurs utilisés.

En général, chaque opérateur reçoit une valeur de *priorité* ; voir le tableau ci-dessous.

Par exemple:

```
1 + 2 * 3
```

La priorité de `+` est inférieure à la priorité de `*` , donc le résultat de l'expression est 7, et non 9.

La description	Opérateurs / constructions (primaires)	Priorité	Associativité
Qualificatif Parenthèses Création d'instance Accès sur le terrain Accès aux tableaux Invocation de méthode Méthode de référence	nom . prénom (expr) new primaire . prénom primaire [expr] primary (expr, ...) primaire :: nom	15	De gauche à droite
Incrément de poste	expr ++ , expr --	14	-
Pré incrémentation Unary Cast [1]	++ expr, -- expr, + Expr, - expr, ~ expr, ! expr, (type) expr	13	- De droite à gauche De droite à gauche
Multiplicatif	* /%	12	De gauche à droite
Additif	+ -	11	De gauche à droite
Décalage	<< >> >>>	dix	De gauche à droite
Relationnel	<> <=> = instanceof	9	De gauche à droite
Égalité	==! =	8	De gauche à droite
Bitwise AND	Et	7	De gauche à droite
Bitwise exclusive OU	^	6	De gauche à droite
Bitwise inclus OU	|	5	De gauche à droite
Logique ET	&&	4	De gauche à droite
OU logique	||	3	De gauche à droite

La description	Opérateurs / constructions (primaires)	Priorité	Associativité
Conditionnel [1]	? :	2	De droite à gauche
Affectation Lambda [1]	= * = / =% = + = - = << = >> = >>> = & = ^ = | = ->	1	De droite à gauche

[1] La priorité de l'expression lambda est complexe, car elle peut également apparaître après un transtypage ou en tant que troisième partie de l'opérateur ternaire conditionnel.

Expressions constantes

Une expression constante est une expression qui donne un type primitif ou une chaîne et dont la valeur peut être évaluée au moment de la compilation en un littéral. L'expression doit être évaluée sans lancer une exception et doit être composée uniquement des éléments suivants:

- Littéraux primitifs et de chaînes.
- Tapez sur les types primitifs ou sur `String` .
- Les opérateurs unaires suivants: `+` , `-` , `~` et `!` .
- Les opérateurs binaires suivants: `*` , `/` , `%` , `+` , `-` , `<<` , `>>` , `>>>` , `<` , `<=` , `>` , `>=` , `== !=` , `&` , `^` , `|` , `&&` et `||` .
- L'opérateur conditionnel ternaire `? :` .
- Expressions constantes entre parenthèses.
- Noms simples faisant référence à des variables constantes. (Une variable constante est une variable déclarée comme `final` lorsque l'expression d'initialisation est elle-même une expression constante.)
- Noms qualifiés du formulaire `<TypeName> . <Identifier>` qui fait référence à des variables constantes.

Notez que la liste ci-dessus *exclut* `++` et `--` , les opérateurs d'affectation, `class` et `instanceof` , les appels de méthodes et les références aux variables générales ou aux champs.

Les expressions constantes de type `String` résultat dans un « interné » `String` et opérations en virgule flottante dans les expressions constantes sont évaluées avec la sémantique FP stricte.

Utilisations pour les expressions constantes

Des expressions constantes peuvent être utilisées (à peu près) partout où une expression normale peut être utilisée. Cependant, ils ont une signification particulière dans les contextes

suivants.

Les expressions constantes sont requises pour les étiquettes de cas dans les instructions switch. Par exemple:

```
switch (someValue) {
case 1 + 1:             // OK
case Math.min(2, 3):    // Error - not a constant expression
    doSomething();
}
```

Lorsque l'expression sur le côté droit d'une affectation est une expression constante, l'affectation peut effectuer une conversion restrictive primitive. Ceci est autorisé à condition que la valeur de l'expression constante se situe dans la plage du type sur le côté gauche. (Voir JLS 5.1.3 et 5.2) Par exemple:

```
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
```

Lorsqu'une expression constante est utilisée comme condition dans un `do` , `while` ou `for` , cela affecte l'analyse de lisibilité. Par exemple:

```
while (false) {
    doSomething();            // Error - statenent not reachable
}
boolean flag = false;
while (flag) {
    doSomething();            // OK
}
```

(Notez que ceci ne s'applique pas aux instructions `if` . Le compilateur Java permet que le bloc `then` ou `else` d'une instruction `if` soit inaccessible. Il s'agit de l'analogue Java de la compilation conditionnelle en C et C ++.)

Enfin, `static final` champs `static final` une classe ou d'une interface avec des initialiseurs d'expression constante sont initialisés avec impatience. Ainsi, il est garanti que ces constantes seront observées dans l'état initialisé, même s'il existe un cycle dans le graphe de dépendance d'initialisation de classe.

Pour plus d'informations, reportez-vous à JLS 15.28. Expressions constantes

Ordre d'évaluation des expressions

Les expressions Java sont évaluées selon les règles suivantes:

- Les opérandes sont évalués de gauche à droite.
- Les opérandes d'un opérateur sont évalués avant l'opérateur.
- Les opérateurs sont évalués en fonction de la priorité des opérateurs
- Les listes d'arguments sont évaluées de gauche à droite.

Exemple simple

Dans l'exemple suivant:

```
int i = method1() + method2();
```

l'ordre d'évaluation est:

1. L'opérande gauche `=` opérateur est évalué à l'adresse `i` .
2. L'opérande gauche de l'opérateur `+` (`method1()`) est évalué.
3. Le bon opérande de l'opérateur `+` (`method2()`) est évalué.
4. L'opération `+` est évaluée.
5. L'opération `=` est évaluée, affectant le résultat de l'addition à `i` .

Notez que si les effets des appels sont observables, vous pourrez observer que l'appel à la `method1` se produit avant l'appel à la `method2` .

Exemple avec un opérateur qui a un effet secondaire

Dans l'exemple suivant:

```
int i = 1;
intArray[i] = ++i + 1;
```

l'ordre d'évaluation est:

1. L'opérande gauche de l'opérateur `=` est évalué. Cela donne l'adresse de `intArray[1]` .
2. Le pré-incrément est évalué. Cela ajoute `1` à `i` et évalue à `2` .
3. L'opérande de droite du `+` est évalué.
4. L'opération `+` est évaluée à: `2 + 1` -> `3` .
5. L'opération `=` est évaluée, affectant `3` à `intArray[1]` .

Notez que puisque l'opérande gauche de `=` est évalué en premier, il n'est pas influencé par l'effet secondaire de la sous-expression `++i` .

Référence:

- JLS 15.7 - Ordre d'évaluation

Les bases de l'expression

Les expressions en Java sont la construction principale pour effectuer des calculs. Voici quelques exemples:

```
1                 // A simple literal is an expression
1 + 2             // A simple expression that adds two numbers
(i + j) / k       // An expression with multiple operations
(flag) ? c : d    // An expression using the "conditional" operator
```

```
(String) s        // A type-cast is an expression
obj.test()        // A method call is an expression
new Object()      // Creation of an object is an expression
new int[]         // Creation of an object is an expression
```

En général, une expression comprend les formes suivantes:

- Noms d'expression qui consistent en:
  - Identifiants simples; par exemple, un `someIdentifier`
  - Identificateurs qualifiés; par exemple `MyClass.someField`
- Primaires qui consistent en:
  - Littéraux; par exemple `1` , `1.0` , `'X'` , `"hello"` , `false` et `null`
  - Expressions littérales de classe; par exemple `MyClass.class`
  - `this` et `<TypeName> . this`
  - Expressions entre parenthèses; par exemple `( a + b )`
  - Expressions de création d'instance de classe; par exemple, `new MyClass(1, 2, 3)`
  - Expressions de création d'instance de tableau; p.ex. `new int[3]`
  - Expressions d'accès au champ; par exemple `obj.someField` ou `this.someField`
  - Expressions d'accès aux tableaux; par exemple, `vector[21]`
  - Invocations de méthodes; p.ex. `obj.doIt(1, 2, 3)`
  - Références de méthode (Java 8 et versions ultérieures) Par exemple, `MyClass::doIt`
- Les expressions d'opérateur unaire; par exemple `!a` ou `i++`
- Les expressions d'opérateur binaire; par exemple `a + b` ou `obj == null`
- Les expressions d'opérateurs ternaires; par exemple `(obj == null) ? 1 : obj.getCount()`
- Expressions lambda (Java 8 et versions ultérieures); `obj -> obj.getCount()`

Les détails des différentes formes d'expressions peuvent être trouvés dans d'autres sujets.

- La rubrique Opérateurs couvre les expressions opérateur unaires, binaires et ternaires.
- La rubrique Expressions Lambda couvre les expressions lambda et les expressions de référence de méthode.
- La rubrique Classes et objets couvre les expressions de création d'instance de classe.
- La rubrique Arrays couvre les expressions d'accès au tableau et les expressions de création d'instances de tableaux.
- La rubrique Literals couvre les différents types d'expressions littérales.

Le type d'une expression

Dans la plupart des cas, une expression a un type statique qui peut être déterminé au moment de la compilation en examinant et ses sous-expressions. Celles-ci sont appelées expressions *autonomes* .

Cependant, (en Java 8 et versions ultérieures) les types d’expressions suivantes peuvent être des *expressions poly* :

- Expressions parenthèses
- Expressions de création d'instance de classe
- Expressions d'appel de méthode

- Expressions de référence de méthode
- Expressions conditionnelles
- Expressions lambda

Lorsqu'une expression est une expression poly, son type peut être influencé par le *type de cible de* l'expression. c'est-à-dire à quoi sert-il?

La valeur d'une expression

La valeur d'une expression est une affectation compatible avec son type. La seule exception à cette règle est la *pollution par tas* ; Par exemple, les avertissements de "conversion non sécurisée" ont été (de manière inappropriée) supprimés ou ignorés.

Déclarations d'expression

Contrairement à beaucoup d'autres langages, Java ne permet généralement pas d'utiliser des expressions comme instructions. Par exemple:

```
public void compute(int i, int j) {
    i + j;   // ERROR
}
```

Étant donné que le résultat de l'évaluation d'une expression comme ne peut pas être utilisé et que cela ne peut pas affecter l'exécution du programme d'une autre manière, les concepteurs Java ont considéré qu'un tel usage était une erreur ou une erreur.

Cependant, cela ne s'applique pas à toutes les expressions. Un sous-ensemble d'expressions est (en fait) légal sous forme d'énoncés. L'ensemble comprend:

- Expression d'affectation, y compris les affectations d' *opération et de* conversion.
- Pré et post incrémenter et décrémenter les expressions.
- Appels de méthode (`void` ou non `void`).
- Expressions de création d'instance de classe.

Lire Expressions en ligne: https://riptutorial.com/fr/java/topic/8167/expressions

Chapitre 64: Expressions lambda

Introduction

Les expressions Lambda offrent un moyen clair et concis d'implémenter une interface à méthode unique utilisant une expression. Ils vous permettent de réduire la quantité de code à créer et à gérer. Bien que similaires aux classes anonymes, elles ne contiennent aucune information de type. La déduction de type doit avoir lieu.

Les références de méthode implémentent des interfaces fonctionnelles en utilisant des méthodes existantes plutôt que des expressions. Ils appartiennent également à la famille lambda.

Syntaxe

- () -> {expression de retour; } // Zéro-arité avec le corps de la fonction pour renvoyer une valeur.
- () -> expression // Abréviation de la déclaration ci-dessus; il n'y a pas de point-virgule pour les expressions.
- () -> {function-body} // Effet secondaire dans l'expression lambda pour effectuer des opérations.
- parameterName -> expression // expression lambda One-arity. Dans les expressions lambda avec un seul argument, la parenthèse peut être supprimée.
- (Tapez parameterName, tapez secondParameterName, ...) -> expression // lambda évaluant une expression avec les paramètres listés à gauche
- (parameterName, secondParameterName, ...) -> expression // Abréviation supprimant les types de paramètre pour les noms de paramètre. Ne peut être utilisé que dans des contextes pouvant être déduits par le compilateur où la taille de la liste de paramètres donnée correspond à un (et un seul) de la taille des interfaces fonctionnelles attendues.

Examples

Utilisation des expressions lambda pour trier une collection

Tri des listes

Avant Java 8, il était nécessaire d'implémenter l'interface `java.util.Comparator` avec une classe anonyme (ou nommée) lors du tri d'une liste [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());
        }
    }
);
```

À partir de Java 8, la classe anonyme peut être remplacée par une expression lambda. Notez que les types pour les paramètres `p1` et `p2` peuvent être omis, comme le compilateur les inférera automatiquement:

```
Collections.sort(
    people,
    (p1, p2) -> p1.getFirstName().compareTo(p2.getFirstName())
);
```

L'exemple peut être simplifié en utilisant `Comparator.comparing` et les références de méthode exprimées en utilisant le symbole `::` (deux points).

```
Collections.sort(
    people,
    Comparator.comparing(Person::getFirstName)
);
```

Une importation statique nous permet de l'exprimer de manière plus concise, mais on peut se demander si cela améliore la lisibilité globale:

```
import static java.util.Collections.sort;
import static java.util.Comparator.comparing;
//...
sort(people, comparing(Person::getFirstName));
```

Les comparateurs construits de cette manière peuvent également être enchaînés. Par exemple, après avoir comparé des personnes par leur prénom, s'il y a des personnes avec le même prénom, la méthode `thenComparing` avec également comparer par nom de famille:

```
sort(people, comparing(Person::getFirstName).thenComparing(Person::getLastName));
```

---

1 - Notez que Collections.sort (...) ne fonctionne que sur les collections qui sont des sous-types de `List` . Les API `Set` et `Collection` n'impliquent aucun classement des éléments.

Tri des cartes

Vous pouvez trier les entrées d'un `HashMap` par valeur de la même manière. (Notez qu'un `LinkedHashMap` doit être utilisé comme cible. Les clés d'un `HashMap` ordinaire `HashMap` sont pas ordonnées.)

```
Map<String, Integer> map = new HashMap();  // ... or any other Map class
// populate the map
```

```
map = map.entrySet()
    .stream()
    .sorted(Map.Entry.<String, Integer>comparingByValue())
    .collect(Collectors.toMap(k -> k.getKey(), v -> v.getValue(),
                              (k, v) -> k, LinkedHashMap::new));
```


Interfaces fonctionnelles

Lambdas ne peut fonctionner que sur une interface fonctionnelle, qui est une interface avec une seule méthode abstraite. Les interfaces fonctionnelles peuvent avoir un nombre quelconque de méthodes `default` ou `static` . (Pour cette raison, ils sont parfois appelés interfaces de méthode abstraite unique ou interfaces SAM).

```
interface Foo1 {
    void bar();
}

interface Foo2 {
    int bar(boolean baz);
}

interface Foo3 {
    String bar(Object baz, int mink);
}

interface Foo4 {
    default String bar() { // default so not counted
        return "baz";
    }
    void quux();
}
```

Lors de la déclaration d'une interface fonctionnelle, l'annotation `@FunctionalInterface` peut être ajoutée. Cela n'a pas d'effet particulier, mais une erreur de compilation sera générée si cette annotation est appliquée à une interface qui n'est pas fonctionnelle, rappelant ainsi que l'interface ne doit pas être modifiée.

```
@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
}
```

À l'inverse, il **ne** s'agit **pas d'** une interface fonctionnelle, car il existe plusieurs méthodes **abstraites** :

```
interface BadFoo {
    void bar();
    void quux(); // <-- Second method prevents lambda: which one should
                 // be considered as lambda?
}
```

Ce n'est **pas non plus** une interface fonctionnelle, car elle n'a pas de méthode:

```
interface BlankFoo2 { }
```

Prenez note de ce qui suit. Supposons que vous ayez

```
interface Parent { public int parentMethod(); }
```

et

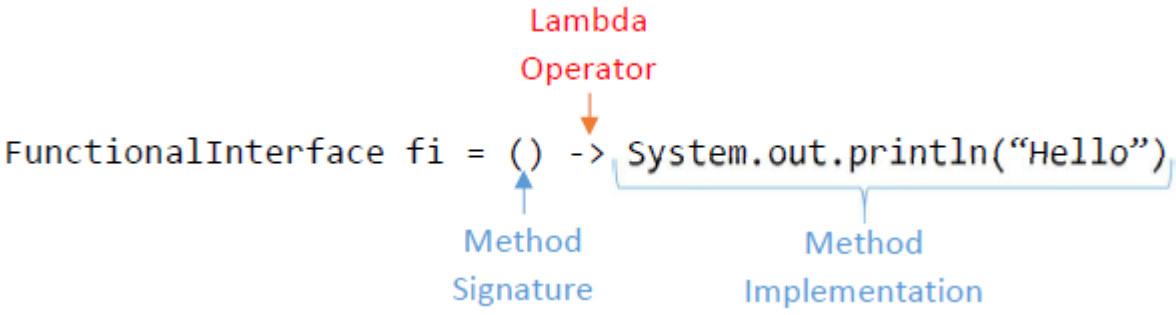
```
interface Child extends Parent { public int ChildMethod(); }
```

Ensuite, `Child` **ne peut pas** être une interface fonctionnelle car il dispose de deux méthodes spécifiées.

Java 8 fournit également un certain nombre d'interfaces fonctionnelles basées sur des modèles génériques dans le package `java.util.function` . Par exemple, l'interface intégrée `Predicate<T>` encapsule une méthode unique qui entre une valeur de type `T` et retourne un `boolean` .

Expressions lambda

La structure de base d'une expression Lambda est la suivante:



`fi` contiendra alors une instance singleton d'une classe, similaire à une classe anonyme, qui implémente `FunctionalInterface` et où la définition de la méthode est `{ System.out.println("Hello"); }` . En d'autres termes, ce qui précède équivaut principalement à:

```
FunctionalInterface fi = new FunctionalInterface() {
    @Override
    public void theOneMethod() {
        System.out.println("Hello");
```

```
    }
};
```

Le lambda est seulement "principalement équivalent" à la classe anonyme car dans un lambda, la signification des expressions comme `this` , `super` ou `toString()` référence à la classe dans laquelle l'assignation a lieu, pas à l'objet nouvellement créé.

Vous ne pouvez pas spécifier le nom de la méthode lorsque vous utilisez un lambda, mais vous ne devriez pas en avoir besoin, car une interface fonctionnelle ne doit avoir qu'une seule méthode abstraite, de sorte que Java remplace celle-ci.

Dans les cas où le type de lambda n'est pas certain (par exemple, des méthodes surchargées), vous pouvez ajouter une distribution au lambda pour indiquer au compilateur quel devrait être son type, comme ceci:

```
Object fooHolder = (Foo1) () -> System.out.println("Hello");
System.out.println(fooHolder instanceof Foo1); // returns true
```

Si la méthode unique de l'interface fonctionnelle prend des paramètres, les noms formels locaux de ceux-ci doivent apparaître entre les crochets du lambda. Il n'est pas nécessaire de déclarer le type du paramètre ou de le renvoyer car ceux-ci sont extraits de l'interface (bien que la déclaration des types de paramètres ne soit pas une erreur si vous le souhaitez). Ainsi, ces deux exemples sont équivalents:

```
Foo2 longFoo = new Foo2() {
    @Override
    public int bar(boolean baz) {
        return baz ? 1 : 0;
    }
};
Foo2 shortFoo = (x) -> { return x ? 1 : 0; };
```

Les parenthèses autour de l'argument peuvent être omises si la fonction n'a qu'un seul argument:

```
Foo2 np = x -> { return x ? 1 : 0; }; // okay
Foo3 np2 = x, y -> x.toString() + y // not okay
```

Retours implicites

Si le code placé dans un lambda est une *expression* Java plutôt qu'une *instruction* , il est traité comme une méthode qui renvoie la valeur de l'expression. Ainsi, les deux suivants sont équivalents:

```
IntUnaryOperator addOneShort = (x) -> (x + 1);
IntUnaryOperator addOneLong = (x) -> { return (x + 1); };
```

Accès aux variables locales (fermetures de valeur)

Comme les lambda sont des raccourcis syntaxiques pour les classes anonymes, elles suivent les mêmes règles pour accéder aux variables locales dans la portée englobante; les variables doivent être traitées comme `final` et non modifiées à l'intérieur du lambda.

```
IntUnaryOperator makeAdder(int amount) {
    return (x) -> (x + amount); // Legal even though amount will go out of scope
                                // because amount is not modified
}

IntUnaryOperator makeAccumulator(int value) {
    return (x) -> { value += x; return value; }; // Will not compile
}
```

Si vous devez envelopper une variable de cette manière, vous devez utiliser un objet régulier qui conserve une copie de la variable. En savoir plus dans Java Closures avec des expressions lambda.

Accepter les Lambdas

Parce qu'un lambda est une implémentation d'une interface, il n'y a rien à faire pour qu'une méthode accepte un lambda: toute fonction qui prend une interface fonctionnelle peut aussi accepter un lambda.

```
public void passMeALambda(Foo1 f) {
    f.bar();
}
passMeALambda(() -> System.out.println("Lambda called"));
```

Le type d'une expression lambda

Une expression lambda, en elle-même, n'a pas de type spécifique. S'il est vrai que les types et le nombre de paramètres, ainsi que le type d'une valeur de retour, peuvent véhiculer des informations de type, ces informations limiteront uniquement les types auxquels elles peuvent être affectées. Le lambda reçoit un type lorsqu'il est affecté à un type d'interface fonctionnelle de l'une des manières suivantes:

- Affectation directe à un type fonctionnel, par exemple `myPredicate = s -> s.isEmpty()`
- En le passant comme paramètre ayant un type fonctionnel, par exemple `stream.filter(s -> s.isEmpty())`
- Le `return s -> s.isEmpty()` depuis une fonction qui retourne un type fonctionnel, par exemple `return s -> s.isEmpty()`

- Le convertir en un type fonctionnel, par exemple `(Predicate<String>) s -> s.isEmpty()`

Jusqu'à ce qu'une telle affectation à un type fonctionnel soit faite, le lambda n'a pas de type défini. Pour illustrer, considérons l'expression lambda `o -> o.isEmpty()` . La même expression lambda peut être affectée à de nombreux types fonctionnels différents:

```
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();
```

Maintenant qu'ils sont attribués, les exemples présentés sont de types complètement différents, même si les expressions lambda se ressemblent et qu'ils ne peuvent pas être affectés les uns aux autres.

Références de méthode

Les références de méthode permettent aux méthodes statiques ou d'instance prédéfinies qui adhèrent à une interface fonctionnelle compatible d'être transmises en tant qu'argument au lieu d'une expression lambda anonyme.

Supposons que nous ayons un modèle:

```
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();
```

Référence de la méthode d'instance (à une instance arbitraire)

```
people.stream().map(Person::getName)
```

Le lambda équivalent:

```
people.stream().map(person -> person.getName())
```

Dans cet exemple, une référence de méthode à la méthode d'instance `getName()` de type `Person` est

en cours de transmission. Comme il est connu pour être du type collection, la méthode de l'instance (connue plus tard) sera appelée.

Référence de la méthode d'instance (à une instance spécifique)

```
people.forEach(System.out::println);
```

Puisque `System.out` est une instance de `PrintStream` , une référence de méthode à cette instance spécifique est transmise en tant qu'argument.

Le lambda équivalent:

```
people.forEach(person -> System.out.println(person));
```

Référence de méthode statique

Aussi, pour transformer des flux, nous pouvons appliquer des références à des méthodes statiques:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
numbers.stream().map(String::valueOf)
```

Cet exemple transmet une référence à la méthode static `valueOf()` sur le type `String` . Par conséquent, l'objet instance dans la collection est transmis en tant qu'argument à `valueOf()` .

Le lambda équivalent:

```
 numbers.stream().map(num -> String.valueOf(num))
```

Référence à un constructeur

```
List<String> strings = Arrays.asList("1", "2", "3");
strings.stream().map(Integer::new)
```

Lire Collecter les éléments d'un flux dans une collection pour voir comment collecter les éléments à collecter.

Le seul constructeur d'argument String du type `Integer` est utilisé ici pour construire un entier à partir de la chaîne fournie comme argument. Dans ce cas, tant que la chaîne représente un nombre, le flux sera mappé aux nombres entiers. Le lambda équivalent:

```
strings.stream().map(s -> new Integer(s));
```

Cheat-Sheet

Format de référence de la méthode	Code	Lambda équivalent
Méthode statique	`TypeName::method`	`(args) -> TypeName.method(args)`
Méthode non statique (sur instance *)	`instance::method`	`(args) -> instance.method(args)`
Méthode non statique (pas d'instance)	`TypeName::method`	`(instance, args) -> instance.method(args)`
Constructeur **	`TypeName::new`	`(args) -> new TypeName(args)`
Constructeur tableau	`TypeName[]::new`	`(int size) -> new TypeName[size]`

---

* `instance` peut être n'importe quelle expression qui donne une référence à une instance, par exemple `getInstance()::method` , `this::method`

** Si `TypeName` est une classe interne non statique, la référence du constructeur est uniquement valide dans le cadre d'une instance de classe externe

Implémentation de plusieurs interfaces

Parfois, vous souhaiterez peut-être avoir une expression lambda implémentant plusieurs interfaces. Ceci est surtout utile avec les interfaces de marqueurs (telles que java.io.Serializable) car elles n'ajoutent pas de méthodes abstraites.

Par exemple, vous voulez créer un `TreeSet` avec un `Comparator` personnalisé, puis le sérialiser et l'envoyer sur le réseau. L'approche triviale:

```
TreeSet<Long> ts = new TreeSet<>((x, y) -> Long.compare(y, x));
```

ne fonctionne pas car le lambda pour le comparateur n'implémente pas `Serializable` . Vous pouvez résoudre ce problème en utilisant des types d'intersection et en spécifiant explicitement que ce lambda doit être sérialisable:

```
TreeSet<Long> ts = new TreeSet<>(
    (Comparator<Long> & Serializable) (x, y) -> Long.compare(y, x));
```

Si vous utilisez fréquemment des types d'intersection (par exemple, si vous utilisez un framework tel qu'Apache Spark où presque tout doit être sérialisable), vous pouvez créer des interfaces vides et les utiliser à la place dans votre code:

```
public interface SerializableComparator extends Comparator<Long>, Serializable {}
```

```
public class CustomTreeSet {
  public CustomTreeSet(SerializableComparator comparator) {}
}
```

De cette façon, vous êtes assuré que le comparateur passé sera sérialisable.

Lambdas et motif d'exécutions

Il existe plusieurs bons exemples d'utilisation de lambdas en tant qu'interface fonctionnelle dans des scénarios simples. Un cas d'utilisation assez courant qui peut être amélioré par lambdas est ce qu'on appelle le modèle Execute-Around. Dans ce modèle, vous disposez d'un ensemble de codes d'installation / de suppression standard nécessaires pour plusieurs scénarios entourant un code spécifique à un cas d'utilisation. Un exemple typique de ceci est le fichier io, la base de données io, les blocs 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();
    }
}
```

Alors, appeler cette méthode avec un lambda pourrait ressembler à ceci:

```
public static void updateMyDAO(MyVO vo) throws DatabaseException {
    doProcessing((Connection conn) -> MyDAO.update(conn, ObjectMapper.map(vo)));
}
```

Ceci n'est pas limité aux opérations d'E / S. Il peut s’appliquer à tout scénario dans lequel des tâches similaires de configuration / élimination sont applicables avec des variations mineures. Le principal avantage de ce modèle est la réutilisation du code et l’application de DRY (ne vous répète pas).

Utiliser l'expression lambda avec votre propre interface fonctionnelle

Les Lambdas sont censés fournir un code d'implémentation en ligne pour les interfaces de méthode unique et la possibilité de les faire circuler comme nous l'avons fait avec les variables normales. Nous les appelons Interface fonctionnelle.

Par exemple, écrire un objet Runnable dans une classe anonyme et démarrer un thread ressemble à ceci:

```
//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();
```

Maintenant, comme ci-dessus, disons que vous avez une interface personnalisée:

```
interface TwoArgInterface {
    int operate(int a, int b);
}
```

Comment utilisez-vous lambda pour implémenter cette interface dans votre code? Identique à l'exemple Runnable ci-dessus. Voir le programme de pilote ci-dessous:

```
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` ne renvoie que de la méthode lambda, pas de la méthode externe

La méthode `return` ne renvoie que de la méthode lambda et non de la méthode externe.

Attention, c'est *différent* de Scala et 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!
  });
}
```

Cela peut conduire à un comportement inattendu lors de la tentative d'écriture de constructions de langage propres, car dans les constructions intégrées telles que `for` boucles, le `return` se

comporte différemment:

```
void demo2() {
  for (int i = 0; i < 3; i++) {
    System.out.println(i);
    return; // Return from 'demo2' entirely
  }
}
```

Dans Scala et Kotlin, la `demo` et la `demo2` que `0` . Mais ce n'est *pas* plus cohérent. L'approche Java est compatible avec le refactoring et l'utilisation de classes - le `return` dans le code en haut et le code ci-dessous se comporte de la même façon:

```
void demo3() {
  threeTimes(new MyIntConsumer());
}

class MyIntConsumer implements IntConsumer {
  public void accept(int i) {
    System.out.println(i);
    return;
  }
}
```

Par conséquent, le Java `return` est plus compatible avec les méthodes de classe et refactoring, mais moins avec le `for` et `while` builtins, ceux - ci restent particulièrement.

De ce fait, les deux suivants sont équivalents en Java:

```
IntStream.range(1, 4)
    .map(x -> x * x)
    .forEach(System.out::println);
IntStream.range(1, 4)
    .map(x -> { return x * x; })
    .forEach(System.out::println);
```

De plus, l'utilisation de try-with-resources est sûre en Java:

```
class Resource implements AutoCloseable {
  public void close() { System.out.println("close()"); }
}

void executeAround(Consumer<Resource> f) {
  try (Resource r = new Resource()) {
    System.out.print("before ");
    f.accept(r);
    System.out.print("after ");
  }
}

void demo4() {
  executeAround(r -> {
    System.out.print("accept() ");
    return; // Does not return from demo4, but frees the resource.
  });
```

```
}
```

imprimera `before accept() after close()` . Dans la sémantique de Scala et Kotlin, les essais avec ressources ne seraient pas fermés, mais ils imprimeraient `before accept()` seulement.

Fermetures Java avec des expressions lambda.

Une fermeture lambda est créée lorsqu'une expression lambda référence les variables d'une étendue englobante (globale ou locale). Les règles pour cela sont les mêmes que pour les méthodes en ligne et les classes anonymes.

Les variables locales d'une étendue englobante utilisées dans un lambda doivent être `final` . Avec Java 8 (la première version prenant en charge lambdas), il n'est pas nécessaire de les *déclarer* `final` dans le contexte extérieur, mais elles doivent être traitées de cette manière. Par exemple:

```
int n = 0; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
```

Ceci est légal tant que la valeur de la variable `n` n'est pas modifiée. Si vous essayez de changer la variable, à l'intérieur ou à l'extérieur du lambda, vous obtiendrez l'erreur de compilation suivante:

> "Les variables locales référencées à partir d'une expression lambda doivent être *finales* ou *effectivement finales* ".

Par exemple:

```
int n = 0;
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
n++; // Will generate an error.
```

S'il est nécessaire d'utiliser une variable variable dans un lambda, l'approche normale consiste à déclarer une copie `final` de la variable et à utiliser la copie. Par exemple

```
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
```

Naturellement, le corps du lambda ne voit pas les modifications apportées à la variable d'origine.

Notez que Java ne prend pas en charge les fermetures réelles. Un lambda Java ne peut pas être

créé d'une manière qui lui permet de voir les changements dans l'environnement dans lequel il a été instancié. Si vous souhaitez implémenter une fermeture qui observe ou modifie son environnement, vous devez la simuler en utilisant une classe régulière. Par exemple:

```
// Does not compile ...
public IntUnaryOperator createAccumulator() {
    int value = 0;
    IntUnaryOperator accumulate = (x) -> { value += x; return value; };
    return accumulate;
}
```

L'exemple ci-dessus ne sera pas compilé pour les raisons évoquées précédemment. Nous pouvons contourner l'erreur de compilation comme suit:

```
// Compiles, but is incorrect ...
public class AccumulatorGenerator {
    private int value = 0;

    public IntUnaryOperator createAccumulator() {
        IntUnaryOperator accumulate = (x) -> { value += x; return value; };
        return accumulate;
    }
}
```

Le problème est que cela rompt le contrat de conception de l'interface `IntUnaryOperator` qui stipule que les instances doivent être fonctionnelles et sans état. Si une telle fermeture est transmise à des fonctions intégrées acceptant des objets fonctionnels, elle est susceptible de provoquer des pannes ou un comportement erroné. Les fermetures qui encapsulent un état mutable doivent être implémentées comme des classes régulières. Par exemple.

```
// Correct ...
public class Accumulator {
   private int value = 0;

   public int accumulate(int x) {
      value += x;
      return value;
   }
}
```

Lambda - Exemple d'écoute

Auditeur de classe anonyme

Avant Java 8, il est très courant qu'une classe anonyme soit utilisée pour gérer l'événement click d'un JButton, comme indiqué dans le code suivant. Cet exemple montre comment implémenter un écouteur anonyme dans la portée de `btn.addActionListener` .

```
JButton btn = new JButton("My Button");
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
```

```
        System.out.println("Button was pressed");
    }
});
```

Auditeur lambda

Étant donné que l'interface `ActionListener` ne définit qu'une méthode `actionPerformed()` , il s'agit d'une interface fonctionnelle, ce qui signifie qu'il est possible d'utiliser des expressions Lambda pour remplacer le code standard. L'exemple ci-dessus peut être réécrit en utilisant les expressions Lambda comme suit:

```
JButton btn = new JButton("My Button");
btn.addActionListener(e -> {
    System.out.println("Button was pressed");
});
```

Style traditionnel au style Lambda

Façon traditionnelle

```
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));
    }
}
```

Style Lambda

1. Supprimez le nom de la classe et le corps de l'interface fonctionnelle.

```
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. Déclaration de type facultative

```
MathOperation isEven = (num) -> {
    return num%2 == 0;
};
```

3. Parenthèse optionnelle autour du paramètre, s'il s'agit d'un paramètre unique

```
MathOperation isEven = num -> {
    return num%2 == 0;
};
```

4. Accolades optionnelles, s'il n'y a qu'une seule ligne dans le corps de la fonction
5. Mot clé de retour facultatif, s'il n'y a qu'une seule ligne dans le corps de la fonction

```
MathOperation isEven = num -> num%2 == 0;
```

Lambdas et utilisation de la mémoire

Comme les lambdas Java sont des fermetures, ils peuvent "capturer" les valeurs des variables dans la portée lexicale. Bien que tous les lambda ne capturent rien - de simples lambda comme `s -> s.length()` ne capturent rien et s'appellent *sans état* - la capture de lambdas nécessite un objet temporaire pour contenir les variables capturées. Dans cet extrait de code, le lambda `() -> j` est un lambda de capture et peut provoquer l'allocation d'un objet lorsqu'il est évalué:

```
public static void main(String[] args) throws Exception {
    for (int i = 0; i < 1000000000; i++) {
        int j = i;
        doSomethingWithLambda(() -> j);
    }
}
```

Bien que cela ne soit pas immédiatement évident puisque le `new` mot-clé n'apparaît nulle part dans le fragment, ce code est susceptible de créer 1 000 000 000 d'objets séparés pour représenter les instances de l'expression `() -> j` lambda. Cependant, il convient également de noter que les futures versions de Java [1] pourraient être en mesure de l'optimiser, de sorte que *lors de l'exécution,* les instances lambda soient réutilisées ou représentées d'une autre manière.

---

1 - Par exemple, Java 9 introduit une phase facultative de "lien" à la séquence de compilation Java, ce qui permettra de réaliser de telles optimisations globales.

Utilisation d'expressions lambda et de prédicats pour obtenir une ou plusieurs valeurs dans une liste

À partir de Java 8, vous pouvez utiliser des expressions et des prédicats lambda.

Exemple: Utilisez des expressions lambda et un prédicat pour obtenir une certaine valeur dans une liste. Dans cet exemple, chaque personne sera imprimée avec le fait qu'elle ait 18 ans ou plus.

Classe de personne:

```
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'interface intégrée Predicate des packages java.util.function.Predicate est une interface fonctionnelle avec une méthode de `boolean test(T t)` .

Exemple d'utilisation:

```
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.");
            }
        }
    }
}
```

Le `print(personList, p -> p.getAge() >= 18);` méthode prend une expression lambda (parce que le prédicat est utilisé un paramètre) où vous pouvez définir l'expression nécessaire. La méthode de test du vérificateur vérifie si cette expression est correcte ou non: `checker.test(person)` .

Vous pouvez facilement changer cela pour quelque chose d'autre, par exemple pour `print(personList, p -> p.getName().startsWith("J"));` . Cela vérifiera si le nom de la personne commence par un "J".

Lire Expressions lambda en ligne: https://riptutorial.com/fr/java/topic/91/expressions-lambda

Chapitre 65: Expressions régulières

Introduction

Une expression régulière est une séquence spéciale de caractères qui aide à faire correspondre ou à trouver d'autres chaînes ou ensembles de chaînes, en utilisant une syntaxe spécialisée contenue dans un modèle. Java prend en charge l'utilisation des expressions régulières via le package `java.util.regex` . Cette rubrique a pour but d'introduire et d'aider les développeurs à mieux comprendre les exemples d'utilisation des expressions régulières en Java.

Syntaxe

- Pattern patternName = Pattern.compile (regex);
- MatcherName matcherName = patternName.matcher (textToSearch);
- matcherName.matches () // Renvoie true si textToSearch correspond exactement à l'expression régulière
- matcherName.find () // Recherche dans textToSearch pour la première instance d'une sous-chaîne correspondant à l'expression régulière. Les appels suivants rechercheront le reste de la chaîne.
- matcherName.group (groupNum) // Retourne la sous-chaîne à l'intérieur d'un groupe de capture
- matcherName.group (groupName) // Retourne la sous-chaîne à l'intérieur d'un groupe de capture nommé (Java 7+)

Remarques

Importations

Vous devrez ajouter les importations suivantes avant de pouvoir utiliser Regex:

```
import java.util.regex.Matcher
import java.util.regex.Pattern
```

Pièges

Dans Java, une barre oblique inverse est échappée avec une double barre oblique inverse. Par conséquent, une barre oblique inverse dans la chaîne de regex doit être entrée en tant que double barre oblique inverse. Si vous avez besoin d'échapper à une double barre oblique inverse (pour faire correspondre une seule barre oblique inverse avec l'expression régulière, vous devez la saisir en tant que double barre oblique inverse).

Symboles importants expliqués

Personnage	La description
`*`	Correspondre au caractère précédent ou à la sous-expression 0 fois ou plus
`+`	Faire correspondre le caractère ou la sous-expression précédent 1 fois ou plus
`?`	Faire correspondre le caractère ou la sous-expression précédent 0 ou 1 fois

Lectures complémentaires

La rubrique regex contient plus d'informations sur les expressions régulières.

Examples

Utilisation de groupes de capture

Si vous devez extraire une partie de la chaîne de la chaîne d'entrée, vous pouvez utiliser des **groupes** de **capture** de regex.

Pour cet exemple, nous commencerons par une simple expression de numéro de téléphone:

```
\d{3}-\d{3}-\d{4}
```

Si des parenthèses sont ajoutées à l'expression rationnelle, chaque ensemble de parenthèses est considéré comme un *groupe de capture* . Dans ce cas, nous utilisons ce que l'on appelle des groupes de capture numérotés:

```
(\d{3})-(\d{3})-(\d{4})
^_____^ ^_____^ ^_____^
Group 1 Group 2 Group 3
```

Avant de pouvoir l'utiliser en Java, nous ne devons pas oublier de suivre les règles de Strings, en évitant les barres obliques inverses, ce qui entraîne le modèle suivant:

```
"(\\d{3})-(\\d{3})-(\\d{4})"
```

Nous devons d'abord compiler le modèle regex pour créer un `Pattern` , puis nous avons besoin d'un `Matcher` correspondant à notre chaîne d'entrée avec le pattern:

```
Pattern phonePattern = Pattern.compile("(\\d{3})-(\\d{3})-(\\d{4})");
Matcher phoneMatcher = phonePattern.matcher("abcd800-555-1234wxyz");
```

Ensuite, le Matcher doit trouver la première sous-séquence correspondant à l'expression régulière:

```
phoneMatcher.find();
```

Maintenant, en utilisant la méthode de groupe, nous pouvons extraire les données de la chaîne:

```
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"
```

Remarque: `Matcher.group()` peut être utilisé à la place de `Matcher.group(0)` .

Java SE 7

Java 7 a introduit les groupes de capture nommés. Les groupes de capture nommés fonctionnent de la même manière que les groupes de capture numérotés (mais avec un nom au lieu d'un nombre), bien qu'il y ait de légères modifications de syntaxe. L'utilisation de groupes de capture nommés améliore la lisibilité.

Nous pouvons modifier le code ci-dessus pour utiliser des groupes nommés:

```
(?<AreaCode>\d{3})-(\d{3})-(\d{4})
^-----------------^ ^-----^ ^-----^
AreaCode            Group 2 Group 3
```

Pour obtenir le contenu de "AreaCode", nous pouvons utiliser à la place:

```
String aCode = phoneMatcher.group("AreaCode"); //"800"
```

Utilisation de regex avec un comportement personnalisé en compilant le modèle avec des indicateurs

Un `Pattern` peut être compilé avec des flags, si le regex est utilisé en tant que `String` , utilisez des modificateurs inline:

```
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".
```

Caractères d'échappement

Généralement

Pour utiliser des caractères spécifiques à une expression régulière (`?+|` Etc.) dans leur sens littéral, ils doivent être échappés. Dans les expressions rationnelles courantes, cela se fait par une barre oblique inverse `\` . Cependant, comme il a une signification particulière dans les chaînes Java, vous devez utiliser une double barre oblique inverse `\\` .

Ces deux exemples ne fonctionneront pas:

```
"???".replaceAll ("?", "!"); //java.util.regex.PatternSyntaxException
"???".replaceAll ("\?", "!"); //Invalid escape sequence
```

Cet exemple fonctionne

```
"???".replaceAll ("\\?", "!"); //"!!!"
```

Fractionnement d'une chaîne délimitée par un tuyau

Cela ne renvoie pas le résultat attendu:

```
"a|b".split ("|"); // [a, |, b]
```

Cela renvoie le résultat attendu:

```
"a|b".split ("\\|"); // [a, b]
```

Échapper à la barre oblique inverse `\`

Cela donnera une erreur:

```
"\\".matches("\\"); // PatternSyntaxException
"\\".matches("\\\"); // Syntax Error
```

Cela marche:

```
"\\".matches("\\\\"); // true
```

Correspondant à un littéral regex.

Si vous devez faire correspondre des caractères faisant partie de la syntaxe des expressions régulières, vous pouvez marquer tout ou partie du motif comme un littéral d'expression régulière.

`\Q` marque le début du littéral regex. `\E` marque la fin du littéral regex.

```
// 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 moyen plus simple de le faire sans avoir à mémoriser les séquences d'échappement `\Q` et `\E` consiste à utiliser `Pattern.quote()`

```
"[123".matches(Pattern.quote("[") + "123"); // returns true
```

Ne correspondant pas à une chaîne donnée

Pour faire correspondre quelque chose qui ne contient *pas* une chaîne donnée, on peut utiliser la lecture négative:

Syntaxe de regex: `(?!string-to-not-match)`

Exemple:

```
//not matching "popcorn"
String regexString = "^(?!popcorn).*$";
System.out.println("[popcorn] " + ("popcorn".matches(regexString) ? "matched!" : "nope!"));
System.out.println("[unicorn] " + ("unicorn".matches(regexString) ? "matched!" : "nope!"));
```

Sortie:

```
[popcorn] nope!
[unicorn] matched!
```

Correspondant à une barre oblique inverse

Si vous souhaitez faire correspondre une barre oblique inverse dans votre expression régulière, vous devrez y échapper.

La barre oblique inverse est un caractère d'échappement dans les expressions régulières. Vous pouvez utiliser '\\' pour faire référence à une seule barre oblique inverse dans une expression régulière.

Cependant, la barre oblique inverse est *également* un caractère d'échappement dans les chaînes de caractères Java. Pour créer une expression régulière à partir d'un littéral de chaîne, vous devez échapper à chacune de *ses* barres obliques inverses. Dans une chaîne, le littéral '\\\\' peut être utilisé pour créer une expression régulière avec '\\', qui à son tour peut correspondre à '\'.

Par exemple, considérez la correspondance de chaînes comme "C: \ dir \ myfile.txt". Une expression régulière `([A-Za-z]):\\(.*)` Correspondra et fournira la lettre de lecteur en tant que groupe de capture. Notez le double backslash.

Pour exprimer ce modèle dans un littéral de chaîne Java, chacune des barres obliques inverses de l'expression régulière doit être échappée.

```
String path = "C:\\dir\\myfile.txt";
System.out.println( "Local path: " + path ); // "C:\dir\myfile.txt"

String regex = "([A-Za-z]):\\\\.*"; // Four to match one
System.out.println("Regex:      " + regex ); // "([A-Za-z]):\\(.*)"

Pattern pattern = Pattern.compile( regex );
Matcher matcher = pattern.matcher( path );
if ( matcher.matches()) {
    System.out.println( "This path is on drive " + matcher.group( 1 ) + ":.");
    // This path is on drive C:.
}
```

Si vous voulez faire correspondre *deux* barres obliques inverses, vous vous retrouverez à utiliser huit dans une chaîne littérale, pour représenter quatre dans l'expression régulière, pour correspondre à deux.

```
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'.
}
```

Lire Expressions régulières en ligne: https://riptutorial.com/fr/java/topic/135/expressions-regulieres

Chapitre 66: Fichier I / O

Introduction

Java I / O (Input and Output) est utilisé pour traiter l'entrée et produire la sortie. Java utilise le concept de flux pour accélérer le fonctionnement des E / S. Le package java.io contient toutes les classes requises pour les opérations d'entrée et de sortie. La gestion des fichiers est également effectuée dans Java par Java I / O API.

Examples

Lecture de tous les octets dans un octet []

Java 7 a introduit la classe Files très 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();
}
```

Lecture d'une image à partir d'un fichier

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

...

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

Ecrire un octet [] dans un fichier

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 plupart des API de fichiers java.io acceptent à la fois les `String` et les `File` comme arguments, de sorte que vous pouvez aussi bien utiliser

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

API Stream vs Writer / Reader

Cours d' eau fournissent l'accès le plus direct au contenu binaire, de sorte que toute `InputStream` / `OutputStream` implémentations fonctionnent toujours sur `int` s et `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);
}
```

Il y a quelques exceptions, notamment le `PrintStream` qui ajoute la "possibilité d'imprimer facilement des représentations de différentes valeurs de données". Cela permet d'utiliser `System.out` fois en tant que `InputStream` binaire et en tant que sortie textuelle en utilisant des méthodes telles que `System.out.println()` .

En outre, certaines implémentations de flux fonctionnent comme une interface avec des contenus de niveau supérieur tels que les objets Java (voir Sérialisation) ou les types natifs, par exemple `DataOutputStream` / `DataInputStream` .

Avec les classes `Writer` et `Reader` , Java fournit également une API pour les flux de caractères explicites. Bien que la plupart des applications basent ces implémentations sur des flux, l'API de flux de caractères n'expose aucune méthode pour le contenu binaire.

```
// 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();
```

Chaque fois qu'il est nécessaire d'encoder des caractères en données binaires (par exemple, lors de l'utilisation des classes `InputStreamWriter` / `OutputStreamWriter`), vous devez spécifier un jeu de caractères si vous ne souhaitez pas dépendre du jeu de caractères par défaut de la plate-forme. En cas de doute, utilisez un encodage compatible Unicode, par exemple UTF-8, pris en charge sur toutes les plates-formes Java. Par conséquent, vous devriez probablement rester à l'écart des classes telles que `FileWriter` et `FileReader` car celles-ci utilisent toujours le charset de plate-forme par défaut. Une meilleure façon d'accéder aux fichiers en utilisant des flux de caractères est la suivante:

```
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();
```

L'un des `Reader` les plus couramment utilisés est `BufferedReader` qui fournit une méthode permettant de lire des lignes entières de texte à partir d'un autre lecteur. C'est probablement le moyen le plus simple de lire un flux de caractères ligne par ligne:

```
// 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);
}
```

Lire un fichier entier à la fois

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

\ Z est le symbole EOF (End of File). Lorsqu'il est défini comme délimiteur, le scanner lit le remplissage jusqu'à ce que l'indicateur EOF soit atteint.

Lecture d'un fichier avec un scanner

Lecture d'un fichier ligne par ligne

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

mot par mot

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

et vous pouvez également modifier le séparateur à l'aide de la méthode scanner.useDelimeter ()

Itérer sur un répertoire et filtrer par extension de fichier

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

Migration de java.io.File vers Java 7 NIO (java.nio.file.Path)

Ces exemples supposent que vous connaissez déjà le NIO de Java 7 en général et que vous avez l'habitude d'écrire du code en utilisant `java.io.File` . Utilisez ces exemples pour trouver rapidement une documentation plus centrée sur NIO pour la migration.

Il y a beaucoup plus à NIO de Java 7, comme les fichiers mappés en mémoire ou l' ouverture d'un fichier ZIP ou JAR à l'aide de FileSystem . Ces exemples ne couvriront qu'un nombre limité de cas d'utilisation de base.

En règle générale, si vous avez l'habitude d'effectuer une opération de lecture / écriture du système de fichiers à l'aide d'une méthode d'instance `java.io.File` , vous la trouverez en tant que méthode statique dans `java.nio.file.Files` .

Pointez sur un chemin

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

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

Chemins relatifs à un autre chemin

```
// 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");
```

Conversion de fichier depuis / vers un chemin pour une utilisation avec des bibliothèques

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

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

Vérifiez si le fichier existe et supprimez-le s'il le fait

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

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

Ecrire dans un fichier via un OutputStream

Il existe plusieurs manières d'écrire et de lire un fichier à l'aide de NIO pour différentes contraintes de performances et de mémoire, de lisibilité et de cas d'utilisation, tels que `FileChannel` , `Files.write(Path path, byte\[\] bytes, OpenOption... options)` . .. Dans cet exemple, seul `OutputStream` est couvert, mais vous êtes fortement encouragé à en apprendre davantage sur les fichiers mappés en mémoire et les différentes méthodes statiques disponibles dans `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));
    }
}
```

Itération sur chaque fichier dans un dossier

```
// -> 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;
    }
});
```

Itération du dossier récursif

```
// -> 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);
        }
    }
}
```

Lecture / écriture de fichier à l'aide de FileInputStream / FileOutputStream

Ecrivez dans un fichier 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();
}
```

Lire depuis le fichier 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();
}
```

Notez que depuis Java 1.7, l'instruction try-with-resources a été introduite, ce qui a simplifié la mise en œuvre des opérations de lecture / écriture:

Ecrivez dans un fichier 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();
}
```

Lire depuis le fichier 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();
}
```

Lecture d'un fichier binaire

Vous pouvez lire un fichier binaire en utilisant ce morceau de code dans toutes les versions récentes de Java:

Java SE 1.4

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

Si vous utilisez Java 7 ou une version ultérieure, il existe un moyen plus simple d'utiliser l' `nio API` :

Java SE 7

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

Verrouillage

Un fichier peut être verrouillé à l'aide de l'API `FileChannel` pouvant être `FileChannel` partir de `streams` et de `readers` d'entrée / sortie

Exemple avec des `streams`

// Ouvrir un flux de fichiers 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
```

Copier un fichier en utilisant InputStream et OutputStream

Nous pouvons directement copier les données d'une source vers un récepteur de données en utilisant une boucle. Dans cet exemple, nous lisons des données depuis un InputStream et en même temps, nous écrivons dans un OutputStream. Une fois que nous avons fini de lire et d'écrire, nous devons fermer la ressource.

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

Lecture d'un fichier à l'aide du canal et du tampon

`Channel` utilise un `Buffer` pour lire / écrire des données. Un tampon est un conteneur de taille fixe où nous pouvons écrire un bloc de données à la fois. `Channel` est plus rapide que les E / S basées

sur le flux.

Pour lire les données d'un fichier en utilisant `Channel` nous devons suivre les étapes suivantes:

1. Nous avons besoin d'une instance de `FileInputStream` . `FileInputStream` a une méthode nommée `getChannel()` qui renvoie un canal.
2. Appelez la méthode `getChannel()` de FileInputStream et acquérez Channel.
3. Créez un ByteBuffer. ByteBuffer est un conteneur de taille fixe d'octets.
4. Channel a une méthode de lecture et nous devons fournir un ByteBuffer comme argument à cette méthode de lecture. ByteBuffer a deux modes - humeur en lecture seule et humeur en écriture seule. Nous pouvons changer le mode en utilisant l'appel de la méthode `flip()` . La mémoire tampon a une position, une limite et une capacité. Une fois qu'un tampon est créé avec une taille fixe, sa limite et sa capacité sont identiques à la taille et la position commence à zéro. Alors qu'un tampon est écrit avec des données, sa position augmente progressivement. Changer de mode signifie changer la position. Pour lire des données depuis le début d'un tampon, il faut mettre la position à zéro. La méthode flip () modifie la position
5. Lorsque nous appelons la méthode de lecture du `Channel` , elle remplit le tampon en utilisant des données.
6. Si nous avons besoin de lire les données du `ByteBuffer` , nous devons retourner le tampon pour changer son mode en écriture seule en mode lecture seule, puis continuer à lire les données du tampon.
7. Lorsqu'il n'y a plus de données à lire, la méthode `read()` du canal renvoie 0 ou -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();
  }
 }
}
```

Copier un fichier avec Channel

Nous pouvons utiliser `Channel` pour copier le contenu du fichier plus rapidement. Pour ce faire, nous pouvons utiliser la méthode `transferTo()` de `FileChannel` .

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

public class FileCopier {

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

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


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

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

Lecture d'un fichier à l'aide de BufferedInputStream

Lire un fichier en utilisant un `BufferedInputStream` généralement plus rapide que `FileInputStream` car il maintient un tampon interne pour stocker les octets lus à partir du flux d'entrée sous-jacent.

```
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();
        }

    }
}
```

Ecrire un fichier en utilisant Channel and Buffer

Pour écrire des données dans un fichier à l'aide de `Channel` vous devez suivre les étapes suivantes:

1. Tout d'abord, nous devons obtenir un objet de `FileOutputStream`
2. Acquérir `FileChannel` appelant la méthode `getChannel()` depuis `FileOutputStream`
3. Créez un `ByteBuffer` puis remplissez-le avec des données
4. Ensuite, nous devons appeler la méthode `flip()` du `ByteBuffer` et la passer en argument de la méthode `write()` du `FileChannel`
5. Une fois que nous avons fini d'écrire, nous devons fermer la ressource

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

Ecrire un fichier en utilisant PrintStream

Nous pouvons utiliser la classe `PrintStream` pour écrire un fichier. Il existe plusieurs méthodes qui vous permettent d'imprimer des valeurs de type de données. `println()` méthode `println()` ajoute une nouvelle ligne. Une fois l'impression terminée, nous devons vider le `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();
        }

    }
}
```

Itérer sur un sous répertoire répertoires d'impression

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

Ajouter des répertoires

Pour créer un nouveau répertoire à partir d'une instance de `File` , vous devez utiliser l'une des deux méthodes suivantes: `mkdirs()` ou `mkdir()` .

- `mkdir()` - Crée le répertoire nommé par ce nom de chemin abstrait. (source)
- `mkdirs()` - Crée le répertoire nommé par ce nom de chemin abstrait, y compris les répertoires parents nécessaires mais inexistants. Notez que si cette opération échoue, elle a peut-être réussi à créer certains des répertoires parents nécessaires. (source)

Remarque: `createNewFile()` ne créera pas de nouveau répertoire uniquement un fichier.

```
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
```

Blocage ou redirection de la sortie / erreur standard

Parfois, une bibliothèque tierce mal conçue écrit des diagnostics indésirables dans les flux `System.out` ou `System.err` . Les solutions recommandées seraient soit de trouver une meilleure bibliothèque, soit (dans le cas de l'Open Source) de résoudre le problème et de fournir un correctif aux développeurs.

Si les solutions ci-dessus ne sont pas réalisables, vous devriez alors envisager de rediriger les flux.

Redirection sur la ligne de commande

Sur un UNIX, Linux ou MacOSX peut être fait à partir du shell en utilisant `>` redirection. Par exemple:

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

Le premier redirige la sortie standard et l'erreur standard vers "/ dev / null", ce qui rejette tout ce qui est écrit dans ces flux. La seconde redirige la sortie standard vers "out.log" et l'erreur standard vers "error.log".

(Pour plus d'informations sur la redirection, reportez-vous à la documentation du shell de commandes que vous utilisez. Des conseils similaires s'appliquent à Windows.)

Vous pouvez également implémenter la redirection dans un script d'encapsuleur ou un fichier de commandes qui lance l'application Java.

Redirection dans une application Java

Il est également possible de rediriger les flux *dans* une application Java en utilisant `System.setOut()` et `System.setErr()` . Par exemple, l'extrait de code suivant redirige la sortie standard et l'erreur standard vers 2 fichiers journaux:

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

Si vous voulez supprimer complètement la sortie, vous pouvez créer un flux de sortie qui "écrit" dans un descripteur de fichier non valide. C'est fonctionnellement équivalent à écrire dans "/ dev / null" sous UNIX.

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

Attention: soyez prudent avec `setOut` et `setErr` :

1. La redirection affectera l'ensemble de la machine virtuelle Java.
2. Ce faisant, vous supprimez la capacité de l'utilisateur à rediriger les flux à partir de la ligne de commande.

Accéder au contenu d'un fichier ZIP

L'API FileSystem de Java 7 permet de lire et d'ajouter des entrées depuis ou vers un fichier Zip à l'aide de l'API de fichier Java NIO, de la même manière que pour tout autre système de fichiers.

Le FileSystem est une ressource qui doit être correctement fermée après utilisation. Par conséquent, le bloc try-with-resources doit être utilisé.

Lecture d'un fichier existant

```
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();
}
```

Créer un nouveau fichier

```
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();
}
```

Lire Fichier I / O en ligne: https://riptutorial.com/fr/java/topic/93/fichier-i---o

Chapitre 67: Fichiers JAR multi-versions

Introduction

L'une des fonctionnalités introduites dans Java 9 est le Jar multi-version (MRJAR) qui permet de regrouper du code ciblant plusieurs versions de Java dans le même fichier Jar. La fonctionnalité est spécifiée dans JEP 238 .

Examples

Exemple de contenu de fichier Jar multi-version

En définissant `Multi-Release: true` dans le fichier MANIFEST.MF, le fichier Jar devient un Jar multi-versions et le runtime Java (à condition qu'il prenne en charge le format MRJAR) sélectionne les versions appropriées des classes en fonction de la version majeure actuelle. .

La structure d'un tel pot est la suivante:

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

- Sur les JDK <9, seules les classes de l'entrée racine sont visibles pour le runtime Java.
- Sur un JDK 9, les classes A et B seront chargées à partir du répertoire `root/META-INF/versions/9` , tandis que C et D seront chargées à partir de l'entrée de base.
- Sur un JDK 10, la classe A serait chargée à partir du répertoire `root/META-INF/versions/10` .

Créer un Jar multi-release à l'aide de l'outil JAR

La commande `jar` peut être utilisée pour créer un Jar multi-versions contenant deux versions de la même classe compilées pour Java 8 et Java 9, mais avec un avertissement indiquant que les classes sont identiques:

```
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'option `--release 9` indique à `jar` d'inclure tout ce qui suit (le package `demo` dans le `sampleproject-9`

) dans une entrée versionnée du MRJAR, à savoir sous `root/META-INF/versions/9` . Le résultat est le contenu suivant:

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

Créons maintenant une classe appelée Main qui `SampleClass` l'URL de `SampleClass` et l'ajoute à la version Java 9:

```
package demo;

import java.net.URL;

public class Main {

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

Si nous compilons cette classe et réexécutons la commande jar, nous obtenons une erreur:

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

La raison en est que l'outil `jar` empêche l'ajout de classes publiques aux entrées versionnées si elles ne sont pas également ajoutées aux entrées de base. Ceci est fait pour que MRJAR expose la même API publique pour les différentes versions de Java. Notez qu'au moment de l'exécution, cette règle n'est pas requise. Il ne peut être appliqué que par des outils tels que le `jar` . Dans ce cas particulier, le but de `Main` est d'exécuter un exemple de code, nous pouvons donc simplement ajouter une copie dans l'entrée de base. Si la classe faisait partie d'une nouvelle implémentation dont nous avons seulement besoin pour Java 9, elle pourrait être rendue non publique.

Pour ajouter `Main` à l'entrée root, nous devons d'abord le compiler pour cibler une version antérieure à Java 9. Cela peut être fait en utilisant la nouvelle option `--release` de `javac` :

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

L'exécution de la classe Main montre que SampleClass est chargé depuis le répertoire versionné:

```
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 d'une classe chargée dans un Jar multi-release

Compte tenu du Jar multi-versions suivant:

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

La classe suivante imprime l'URL de `SampleClass` :

```
package demo;

import java.net.URL;

public class Main {

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

Si la classe est compilée et ajoutée à l'entrée versionnée pour Java 9 dans MRJAR, son exécution entraînerait:

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

Lire Fichiers JAR multi-versions en ligne: https://riptutorial.com/fr/java/topic/9866/fichiers-jar-multi-versions

Chapitre 68: Files d'attente et deques

Examples

L'utilisation de PriorityQueue

`PriorityQueue` est une structure de données. Comme `SortedSet` , `PriorityQueue` trie également ses éléments en fonction de leurs priorités. Les éléments, qui ont une priorité plus élevée, viennent en premier. Le type de `PriorityQueue` doit implémenter `comparable` interface `comparable` ou de `comparator` , dont les méthodes déterminent les priorités des éléments de la structure de données.

```
//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 en tant que file FIFO

La classe `java.util.LinkedList` , lors de l'implémentation de `java.util.List` est une implémentation polyvalente de l'interface `java.util.Queue` fonctionnant également selon le principe FIFO (First In, First Out) .

Dans l'exemple ci-dessous, avec la méthode `offer()` , les éléments sont insérés dans la `LinkedList` . Cette opération d'insertion s'appelle la mise en `enqueue` . Dans le `while` en boucle ci - dessous, les éléments sont retirés de la `Queue` d' `Queue` sur la base FIFO. Cette opération s'appelle `dequeue` .

```
Queue<String> queue = new LinkedList<String>();

queue.offer( "first element" );
queue.offer( "second element" );
queue.offer( "third element" );
queue.offer( "fourth. element" );
queue.offer( "fifth. element" );
```

```
while ( !queue.isEmpty() ) {
  System.out.println( queue.poll() );
}
```

La sortie de ce code est

```
first element
second element
third element
fourth element
fifth element
```

Comme on le voit dans la sortie, le premier élément inséré "premier élément" est supprimé en premier lieu, "deuxième élément" est supprimé à la deuxième place, etc.

Piles

Qu'est-ce qu'un Stack?

En Java, Stacks est une structure de données LIFO (Last In, First Out) pour les objets.

API de pile

Java contient une API Stack avec les méthodes suivantes

```
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
```

Exemple

```
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");
      }
   }
}
```

Ce retourne:

```
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

Une BlockingQueue est une interface, qui est une file d'attente qui se bloque lorsque vous essayez de retirer la file d'attente de celle-ci et que la file d'attente est vide ou si vous essayez de mettre les éléments en file d'attente et que la file d'attente est déjà pleine. Un thread essayant de se libérer de la file d'attente vide est bloqué jusqu'à ce qu'un autre thread insère un élément dans la file d'attente. Un thread qui tente de mettre en file d'attente un élément dans une file d'attente complète est bloqué jusqu'à ce qu'un autre thread crée de l'espace dans la file d'attente, soit en retirant un ou plusieurs éléments ou en effaçant complètement la file d'attente.

Les méthodes de BlockingQueue se présentent sous quatre formes, avec différentes manières de gérer les opérations qui ne peuvent pas être satisfaites immédiatement, mais qui peuvent être satisfaites à l’avenir: on jette une exception, la seconde renvoie une valeur spéciale (nulle ou fausse selon opération), le troisième bloque indéfiniment le thread en cours jusqu’à ce que l’opération réussisse, et le quatrième ne bloque qu’une limite de temps maximale donnée avant

d’abandonner.

Opération	Jette une exception	Valeur Spéciale	Blocs	Le temps est écoulé
Insérer	ajouter()	offre (e)	mettre (e)	offre (e, temps, unité)
Retirer	retirer()	sondage()	prendre()	sondage (heure, unité)
Examiner	élément()	peek ()	N / A	N / A

Un `BlockingQueue` peut être **limité** ou **non** . Un BlockingQueue borné est un qui est initialisé avec une capacité initiale.

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);
```

Tout appel à une méthode put () sera bloqué si la taille de la file d'attente est égale à la capacité initiale définie.

Une file d'attente illimitée est une file qui est initialisée sans capacité, en fait, par défaut, elle est initialisée avec Integer.MAX_VALUE.

Les implémentations courantes de `BlockingQueue` sont les suivantes:

1. ArrayBlockingQueue
2. LinkedBlockingQueue
3. PriorityBlockingQueue

Regardons maintenant un exemple de `ArrayBlockingQueue` :

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<>(2);
bQueue.put("This is entry 1");
System.out.println("Entry one done");
bQueue.put("This is entry 2");
System.out.println("Entry two done");
bQueue.put("This is entry 3");
System.out.println("Entry three done");
```

Cela va imprimer:

```
Entry one done
Entry two done
```

Et le thread sera bloqué après la deuxième sortie.

Interface de file d'attente

Les bases

Une `Queue` est une collection pour contenir des éléments avant le traitement. Les files d'attente ordonnent généralement, mais pas nécessairement, les éléments d'une manière FIFO (premier entré, premier sorti).

Head of the queue est l'élément qui serait supprimé par un appel à supprimer ou à interroger. Dans une file d'attente FIFO, tous les nouveaux éléments sont insérés en queue de file.

L'interface de file d'attente

```
public interface Queue<E> extends Collection<E> {
    boolean add(E e);

    boolean offer(E e);

    E remove();

    E poll();

    E element();

    E peek();
}
```

Chaque méthode de `Queue` existe sous deux formes:

- on lève une exception si l'opération échoue;
- other renvoie une valeur spéciale si l'opération échoue (`null` ou `false` selon l'opération).

Type d'opération	Lance une exception	Renvoie une valeur spéciale
Insérer	`add(e)`	`offer(e)`
Retirer	`remove()`	`poll()`
Examiner	`element()`	`peek()`

Deque

Un `Deque` est une "file d'attente double", ce qui signifie que des éléments peuvent être ajoutés au début ou à la fin de la file. La file d'attente seulement peut ajouter des éléments à la queue d'une file d'attente.

Le `Deque` hérite de l'interface `Queue` , ce qui signifie que les méthodes habituelles restent, mais l'interface Deque offre des méthodes supplémentaires pour être plus flexible avec une file d'attente. Les méthodes supplémentaires parlent d'elles-mêmes si vous savez comment fonctionne une file d'attente, puisque ces méthodes sont destinées à ajouter plus de flexibilité:

Méthode	Brève description
`getFirst()`	Obtient le premier élément de la **tête** de la file d'attente sans le supprimer.

Méthode	Brève description
`getLast()`	Obtient le premier élément de la **queue** de la file sans le supprimer.
`addFirst(E e)`	Ajoute un élément à la **tête** de la file d'attente
`addLast(E e)`	Ajoute un élément à la **queue** de la file d'attente
`removeFirst()`	Supprime le premier élément en **tête** de la file d'attente
`removeLast()`	Supprime le premier élément à la **queue** de la file d' attente

Bien entendu, les mêmes options pour les `offer` , les `poll` et les `peek` sont disponibles, mais elles ne fonctionnent pas avec des exceptions, mais plutôt avec des valeurs spéciales. Il ne sert à rien de montrer ce qu'ils font ici.

Ajouter et accéder aux éléments

Pour ajouter des éléments à la queue d'un Deque, vous appelez sa méthode `add()` . Vous pouvez également utiliser les `addFirst()` et `addLast()` , qui ajoutent des éléments à la tête et à la queue de 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
```

Vous pouvez jeter un coup d'œil à l'élément en tête de la file sans retirer l'élément de la file d'attente. Cela se fait via la méthode `element()` . Vous pouvez également utiliser les `getFirst()` et `getLast()` , qui renvoient le premier et le dernier élément de `Deque` . Voici à quoi cela ressemble:

```
String firstElement0 = dequeA.element();
String firstElement1 = dequeA.getFirst();
String lastElement = dequeA.getLast();
```

Suppression d'éléments

Pour supprimer des éléments d'un duque, appelez les méthodes `remove()` , `removeFirst()` et `removeLast()` . Voici quelques exemples:

```
String firstElement = dequeA.remove();
String firstElement = dequeA.removeFirst();
String lastElement  = dequeA.removeLast();
```

Lire Files d'attente et deques en ligne: https://riptutorial.com/fr/java/topic/7196/files-d-attente-et-deques

Chapitre 69: FileUpload vers AWS

Introduction

Télécharger le fichier dans le compartiment AWS s3 à l'aide de l'API Spring Rest.

Examples

Télécharger le fichier dans le compartiment s3

> Ici, nous allons créer un APi de repos qui prendra l'objet fichier en tant que paramètre multipart de l'avant et le chargera dans le compartiment S3 en utilisant l'API de repos Java.

Exigence : - clé secrète et clé d'accès pour le compartiment s3 où vous souhaitez télécharger votre fichier.

code: - 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;

    }


}
```

Fonction frontale

```
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);
});
```

Lire FileUpload vers AWS en ligne: https://riptutorial.com/fr/java/topic/10589/fileupload-vers-aws

Chapitre 70: FilLocal

Remarques

Utilisé de préférence pour les objets qui dépendent des composants internes lors de l'appel d'un appel, mais qui sont sans état, comme `SimpleDateFormat` , `Marshaller`

Pour l'utilisation `Random` ThreadLocal, envisagez d'utiliser `ThreadLocalRandom`

Examples

Initialisation fonctionnelle de ThreadLocal Java 8

```
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);
    }
}
```

Utilisation de base de ThreadLocal

Java `ThreadLocal` est utilisé pour créer des variables locales de thread. Il est connu que les threads d'un objet partagent ses variables, la variable n'est donc pas thread-safe. Nous pouvons utiliser la synchronisation pour la sécurité des threads mais si nous voulons éviter la synchronisation, ThreadLocal nous permet de créer des variables locales au thread, c'est-à-dire que seul ce thread peut lire ou écrire sur ces variables. ne seront pas en mesure d'accéder aux variables ThreadLocal les uns des autres.

Cela peut être utilisé, nous pouvons utiliser des variables `ThreadLocal` . dans les situations où vous avez un pool de threads comme par exemple dans un service Web. Par exemple, Création d' un `SimpleDateFormat` objet chaque fois pour chaque demande prend du temps et statique ne peut pas être créé `SimpleDateFormat` est pas thread - safe, afin que nous puissions créer un ThreadLocal afin que nous puissions effectuer fil des opérations de sécurité sans les frais généraux de la création `SimpleDateFormat` chaque temps.

Le morceau de code ci-dessous montre comment il peut être utilisé:

Chaque thread a sa propre variable `ThreadLocal` et ils peuvent utiliser ses méthodes `get()` et `set()` pour obtenir la valeur par défaut ou changer sa valeur en Thread.

`ThreadLocal` instances `ThreadLocal` sont généralement des champs statiques privés dans les classes qui souhaitent associer l'état à un thread.

Voici un petit exemple montrant l'utilisation de ThreadLocal dans le programme Java et prouvant que chaque thread a sa propre copie de la variable `ThreadLocal` .

```
package com.examples.threads;

import java.text.SimpleDateFormat;
import java.util.Random;

public class ThreadLocalExample implements Runnable{

    // SimpleDateFormat is not thread-safe, so give one to each thread
 // SimpleDateFormat is not thread-safe, so give one to each thread
    private static final ThreadLocal<SimpleDateFormat> formatter = new
ThreadLocal<SimpleDateFormat>(){
        @Override
        protected SimpleDateFormat initialValue()
        {
            return new SimpleDateFormat("yyyyMMdd HHmm");
        }
    };

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalExample obj = new ThreadLocalExample();
        for(int i=0 ; i<10; i++){
            Thread t = new Thread(obj, ""+i);
            Thread.sleep(new Random().nextInt(1000));
            t.start();
        }
    }

    @Override
    public void run() {
        System.out.println("Thread Name= "+Thread.currentThread().getName()+" default
Formatter = "+formatter.get().toPattern());
        try {
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        formatter.set(new SimpleDateFormat());

        System.out.println("Thread Name= "+Thread.currentThread().getName()+" formatter =
"+formatter.get().toPattern());
    }

}
```

Sortie:

```
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
```

Comme nous pouvons le voir à la sortie, Thread-0 a changé la valeur du formateur, mais le formateur par défaut du thread-2 est toujours identique à la valeur initialisée.

Plusieurs threads avec un objet partagé

Dans cet exemple, nous avons un seul objet mais il est partagé entre / exécuté sur différents threads. L'utilisation ordinaire des champs pour enregistrer l'état ne serait pas possible car l'autre thread verrait cela aussi (ou probablement pas voir).

```
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();
    }
}
```

Dans Foo nous comptons à partir de zéro. Au lieu de sauvegarder l'état dans un champ, nous stockons notre numéro actuel dans l'objet ThreadLocal qui est accessible statiquement. Notez que la synchronisation dans cet exemple n'est pas liée à l'utilisation de ThreadLocal mais garantit plutôt une meilleure sortie de la console.

```
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) {
                }
            }
        }
    }
}
```

De la sortie, nous pouvons voir que chaque thread compte pour lui-même et n'utilise pas la valeur de l'autre:

```
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
```

Lire FilLocal en ligne: https://riptutorial.com/fr/java/topic/2001/fillocal

Chapitre 71: Fonctionnalités de Java SE 8

Introduction

Dans cette rubrique, vous trouverez un résumé des nouvelles fonctionnalités ajoutées au langage de programmation Java dans Java SE 8. Il existe de nombreuses autres nouvelles fonctionnalités dans d'autres domaines, tels que JDBC et Java Virtual Machine (JVM), qui ne seront pas couvertes. dans ce sujet.

Remarques

Référence: améliorations de Java SE 8

Examples

Nouvelles fonctionnalités du langage de programmation Java SE 8

- Expressions Lambda , une nouvelle fonctionnalité de langue, a été introduite dans cette version. Ils vous permettent de traiter les fonctionnalités comme un argument de méthode ou un code en tant que données. Les expressions Lambda vous permettent d'exprimer de manière plus compacte des instances d'interfaces à méthode unique (appelées interfaces fonctionnelles).
    - Les références de méthode fournissent des expressions lambda faciles à lire pour les méthodes qui ont déjà un nom.
    - Les méthodes par défaut permettent d'ajouter de nouvelles fonctionnalités aux interfaces des bibliothèques et de garantir la compatibilité binaire avec le code écrit pour les anciennes versions de ces interfaces.
    - Les API nouvelles et améliorées qui tirent parti des expressions et flux Lambda dans Java SE 8 décrivent les classes nouvelles et améliorées qui tirent parti des expressions et des flux lambda.
- Type Inference amélioré - Le compilateur Java tire parti du typage de la cible pour déduire les paramètres de type d'une invocation de méthode générique. Le type cible d'une expression est le type de données attendu par le compilateur Java, selon l'emplacement de l'expression. Par exemple, vous pouvez utiliser le type de cible d'une instruction d'affectation pour l'inférence de type dans Java SE 7. Cependant, dans Java SE 8, vous pouvez utiliser le type de cible pour l'inférence de type dans d'autres contextes.
    - Saisie de cible dans les expressions lambda
    - Type Inférence
- Les annotations répétées permettent d'appliquer le même type d'annotation plus d'une fois à la même déclaration ou à la même utilisation de type.
- Les annotations de type permettent d'appliquer une annotation partout où un type est utilisé, pas uniquement dans une déclaration. Utilisée avec un système de type enfichable, cette fonctionnalité permet une vérification de type améliorée de votre code.
- Méthode de réflexion des paramètres - Vous pouvez obtenir les noms des paramètres

formels de toute méthode ou constructeur avec la méthode `java.lang.reflect.Executable.getParameters` . (Les classes Method et Constructor étendent la classe Executable et héritent donc de la méthode `Executable.getParameters`). Cependant, les fichiers `.class` ne stockent pas les noms de paramètres formels par défaut. Pour stocker les noms de paramètres formels dans un fichier `.class` particulier et permettre ainsi à l'API Reflection d'extraire les noms de paramètres formels, compilez le fichier source avec l'option `-parameters` du compilateur javac.

- Date-heure-api - Ajout d'une nouvelle heure api dans `java.time` . Si vous l'utilisez, vous n'avez pas besoin de désigner un fuseau horaire.

Lire Fonctionnalités de Java SE 8 en ligne: https://riptutorial.com/fr/java/topic/8267/fonctionnalites-de-java-se-8

Chapitre 72: Fonctionnalités Java SE 7

Introduction

Dans cette rubrique, vous trouverez un récapitulatif des nouvelles fonctionnalités ajoutées au langage de programmation Java dans Java SE 7. Il existe de nombreuses autres nouvelles fonctionnalités dans d'autres domaines tels que JDBC et Java Virtual Machine (JVM) qui ne seront pas couvertes. dans ce sujet.

Remarques

Améliorations de Java SE 7

Examples

Nouvelles fonctionnalités du langage de programmation Java SE 7

- Littéraux binaires : Les types intégraux (octet, court, entier et long) peuvent également être exprimés à l'aide du système de nombres binaires. Pour spécifier un littéral binaire, ajoutez le préfixe 0b ou 0B au numéro.
- Strings in switch Instructions : Vous pouvez utiliser un objet String dans l'expression d'une instruction switch
- L'instruction try-with-resources : l' instruction try-with-resources est une instruction try qui déclare une ou plusieurs ressources. Une ressource est un objet qui doit être fermé une fois le programme terminé. L'instruction try-with-resources garantit que chaque ressource est fermée à la fin de l'instruction. Tout objet qui implémente java.lang.AutoCloseable, qui inclut tous les objets qui implémentent java.io.Closeable, peut être utilisé en tant que ressource.
- Attraper plusieurs types d'exceptions et rediriger des exceptions avec une vérification de type améliorée : un seul bloc catch peut gérer plusieurs types d'exception. Cette fonctionnalité peut réduire la duplication de code et réduire la tentation d'attraper une exception trop large.
- Underscores dans les littéraux numériques : n'importe quel nombre de caractères de soulignement (_) peut apparaître n'importe où entre des chiffres dans un littéral numérique. Cette fonctionnalité vous permet, par exemple, de séparer des groupes de chiffres dans des littéraux numériques, ce qui peut améliorer la lisibilité de votre code.
- Inférence de type pour la création d'instance générique : Vous pouvez remplacer les arguments de type requis pour appeler le constructeur d'une classe générique par un ensemble vide de paramètres de type (<>) tant que le compilateur peut déduire les arguments de type du contexte. Cette paire de crochets est appelée de manière informelle le diamant.
- Amélioration des avertissements et erreurs du compilateur lors de l'utilisation de paramètres formels non-réifiérents avec les méthodes Varargs

Littéraux binaires

```
// 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 déclaration d'essayer avec les ressources

L'exemple lit la première ligne d'un fichier. Il utilise une instance de `BufferedReader` pour lire les données du fichier. `BufferedReader` est une ressource qui doit être fermée après la fin du programme:

```
static String readFirstLineFromFile(String path) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

Dans cet exemple, la ressource déclarée dans l'instruction try-with-resources est un `BufferedReader` . L'instruction de déclaration apparaît entre parenthèses immédiatement après le mot clé try. La classe `BufferedReader` , dans Java SE 7 et `java.lang.AutoCloseable` ultérieures, implémente l'interface `java.lang.AutoCloseable` . Étant donné que l'instance `BufferedReader` est déclarée dans une instruction try-with-resource, elle sera fermée, que l'instruction try se termine normalement ou brusquement (à la suite de la méthode `BufferedReader.readLine` `IOException` une `IOException`).

Underscores dans les littéraux numériques

L'exemple suivant montre d'autres manières d'utiliser le trait de soulignement dans les littéraux numériques:

```
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;
```

Vous pouvez placer des traits de soulignement uniquement entre les chiffres; vous ne pouvez pas placer des traits de soulignement dans les endroits suivants:

- Au début ou à la fin d'un nombre

- Adjacent à un point décimal dans un littéral à virgule flottante
- Avant un suffixe F ou L
- Dans les positions où une chaîne de chiffres est attendue

Type Inference pour la création d'instance générique

Vous pouvez utiliser

```
Map<String, List<String>> myMap = new HashMap<>();
```

au lieu de

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

Cependant, vous ne pouvez pas utiliser

```
List<String> list = new ArrayList<>();
list.add("A");

  // The following statement should fail since addAll expects
  // Collection<? extends String>

list.addAll(new ArrayList<>());
```

parce qu'il ne peut pas compiler. Notez que le diamant fonctionne souvent dans les appels de méthode; Cependant, il est conseillé d'utiliser le diamant principalement pour les déclarations de variables.

Strings in switch Déclarations

```
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;
}
```

Lire Fonctionnalités Java SE 7 en ligne: https://riptutorial.com/fr/java/topic/8272/fonctionnalites-java-se-7

Chapitre 73: Format de nombre

Examples

Format de nombre

Différents pays ont des formats de nombres différents et en considérant cela, nous pouvons avoir différents formats en utilisant Locale of java. L'utilisation de paramètres régionaux peut aider à formater

```
Locale locale = new Locale("en", "IN");
NumberFormat numberFormat = NumberFormat.getInstance(locale);
```

en utilisant le format ci-dessus, vous pouvez effectuer diverses tâches

1. Numéro de format

   ```
   numberFormat.format(10000000.99);
   ```

2. Format de devise

   ```
   NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
   currencyFormat.format(10340.999);
   ```

3. Pourcentage de format

   ```
   NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
   percentageFormat.format(10929.999);
   ```

4. Contrôle du nombre de chiffres

```
numberFormat.setMinimumIntegerDigits(int digits)
numberFormat.setMaximumIntegerDigits(int digits)
numberFormat.setMinimumFractionDigits(int digits)
numberFormat.setMaximumFractionDigits(int digits)
```

Lire Format de nombre en ligne: https://riptutorial.com/fr/java/topic/7399/format-de-nombre

Chapitre 74: Fractionner une chaîne en parties de longueur fixe

Remarques

Le but ici est de ne pas perdre de contenu, donc la regex ne doit consommer aucune correspondance. Au contraire , il doit correspondre *entre* le dernier caractère de l'entrée cible précédente et le premier caractère de la prochaine entrée cible. Par exemple, pour les sous-chaînes de 8 caractères, nous devons décomposer l'entrée (c'est-à-dire la correspondance) aux endroits indiqués ci-dessous:

```
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
              ^               ^               ^
```

Ignorer les espaces dans l'entrée qui étaient nécessaires pour afficher *entre les* positions de caractère.

Examples

Briser une chaîne en sous-chaînes de longueur connue

L'astuce consiste à utiliser un look-behind avec le regex `\G` , qui signifie "fin du match précédent":

```
String[] parts = str.split("(?<=\\G.{8})");
```

L'expression rationnelle correspond à 8 caractères à la fin du dernier match. Puisque dans ce cas la correspondance est de largeur nulle, on pourrait simplement dire "8 caractères après la dernière correspondance".

De manière pratique, `\G` est initialisé pour démarrer l'entrée, donc cela fonctionne aussi pour la première partie de l'entrée.

Briser une chaîne en sous-chaînes de longueur variable

Identique à l'exemple de longueur connue, mais insérez la longueur dans l'expression régulière:

```
int length = 5;
String[] parts = str.split("(?<=\\G.{" + length + "})");
```

Lire Fractionner une chaîne en parties de longueur fixe en ligne:
https://riptutorial.com/fr/java/topic/5613/fractionner-une-chaine-en-parties-de-longueur-fixe

Chapitre 75: FTP (protocole de transfert de fichiers)

Syntaxe

- FTPClient connect (hôte InetAddress, port int)
- Connexion FTPClient (nom d'utilisateur String, mot de passe String)
- FTPClient déconnecter ()
- FTPRéponse getReplyStrings ()
- boolean storeFile (Chaîne distante, InputStream local)
- OutputStream storeFileStream (Chaîne distante)
- boolean setFileType (int fileType)
- boolean completePendingCommand ()

Paramètres

Paramètres	Détails
hôte	Soit le nom d'hôte ou l'adresse IP du serveur FTP
Port	Le port du serveur FTP
Nom d'utilisateur	Le nom d'utilisateur du serveur FTP
mot de passe	Le mot de passe du serveur FTP

Examples

Connexion et connexion à un serveur FTP

Pour commencer à utiliser FTP avec Java, vous devez créer un nouveau `FTPClient` , puis vous connecter et vous connecter au serveur en utilisant `.connect(String server, int port)` et `.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);
    }
}
```

Maintenant, nous avons les bases faites. Mais que faire si nous avons une erreur de connexion au serveur? Nous voulons savoir quand quelque chose ne va pas et obtenir le message d'erreur. Ajoutons du code pour intercepter les erreurs lors de la connexion.

```
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 {

}
```

Décomposons ce que nous venons de faire, pas à pas.

```
showServerReply(ftp);
```

Cela fait référence à une fonction que nous allons effectuer ultérieurement.

```
int replyCode = ftp.getReplyCode();
```

Cela récupère le code de réponse / erreur du serveur et le stocke sous forme d'entier.

```
if (!FTPReply.isPositiveCompletion(replyCode)) {
    System.out.printIn("Operation failed. Server reply code: " + replyCode)
    return;
}
```

Cela vérifie le code de réponse pour voir s'il y avait une erreur. S'il y avait une erreur, il afficherait simplement "Échec de l'opération. Code de réponse du serveur:" suivi du code d'erreur. Nous avons également ajouté un bloc try / catch que nous ajouterons à l'étape suivante. Ensuite, créons également une fonction qui vérifie les `ftp.login()` .

```
boolean success = ftp.login(user, pass);
showServerReply(ftp);
if (!success) {
    System.out.println("Failed to log into the server");
        return;
    } else {
        System.out.println("LOGGED IN SERVER");
    }
```

Brisons ce bloc aussi.

```
boolean success = ftp.login(user, pass);
```

Cela ne va pas simplement tenter de se connecter au serveur FTP, il va également stocker le résultat sous forme de booléen.

```
showServerReply(ftp);
```

Cela permettra de vérifier si le serveur nous a envoyé des messages, mais nous devrons d'abord créer la fonction à l'étape suivante.

```
if (!success) {
System.out.println("Failed to log into the server");
    return;
} else {
    System.out.println("LOGGED IN SERVER");
}
```

Cette déclaration vérifiera si nous nous sommes connectés avec succès; Si c'est le cas, il affichera "LOGGED IN SERVER", sinon il affichera "Impossible de se connecter au serveur". Ceci est notre script jusqu'à présent:

```
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 {

        }
```

```
    }
}
```

Maintenant, nous allons créer le bloc Catch au cas où nous aurions des erreurs avec le processus entier.

```
} catch (IOException ex) {
    System.out.println("Oops! Something went wrong.");
    ex.printStackTrace();
}
```

Le bloc catch terminé affichera maintenant "Oops! Quelque chose ne va pas." et le stacktrace s'il y a une erreur. Maintenant, notre dernière étape consiste à créer le `showServerReply()` nous utilisons depuis un certain temps.

```
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);
        }
    }
}
```

Cette fonction prend un `FTPClient` tant que variable et l'appelle "ftp". Après cela, il stocke toutes les réponses du serveur du serveur dans un tableau de chaînes. Ensuite, il vérifie si des messages ont été stockés. S'il y en a, il imprime chacun d'eux comme "SERVER: [reply]". Maintenant que cette fonction est terminée, voici le script terminé:

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

Nous devons d'abord créer un nouveau `FTPClient` et essayer de nous connecter au serveur et nous connecter en utilisant `.connect(String server, int port)` et `.login(String username, String password)` . Il est important de se connecter et de se connecter en utilisant un bloc try / catch au cas où notre code ne parviendrait pas à se connecter au serveur. Nous devrons également créer une fonction qui vérifie et affiche tous les messages que nous pouvons recevoir du serveur lorsque nous essayons de nous connecter et de nous connecter. Nous appellerons cette fonction " `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();
        }
    }
}
```

Après cela, vous devriez maintenant avoir votre serveur FTP connecté à votre script Java.

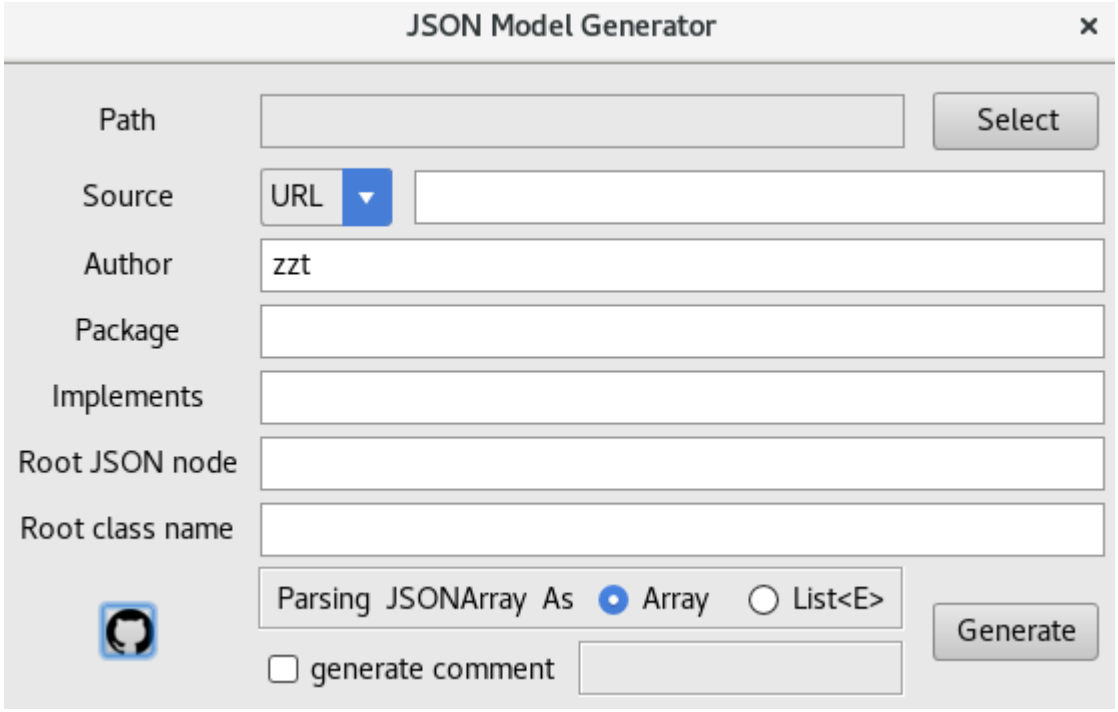
Lire FTP (protocole de transfert de fichiers) en ligne: https://riptutorial.com/fr/java/topic/5228/ftp--protocole-de-transfert-de-fichiers-

Chapitre 76: Génération de code Java

Examples

Générer un POJO à partir de JSON

- Installez le plug-in JSON Model Genrator d'Intellij en effectuant une recherche dans le paramètre Intellij.
- Démarrer le plugin à partir de 'Tools'
- Entrez le champ de l'interface utilisateur comme suit ('Path' Path 'Source' 、 'Package' est requis):



- Cliquez sur le bouton "Générer" et vous avez terminé.

Lire Génération de code Java en ligne: https://riptutorial.com/fr/java/topic/9400/generation-de-code-java

Chapitre 77: Génération de nombres aléatoires

Remarques

Rien n'est vraiment aléatoire et le javadoc appelle donc ces nombres de manière pseudo-aléatoire. Ces nombres sont créés avec un générateur de nombres pseudo-aléatoires .

Examples

Nombres aléatoires

Java fournit, dans le cadre du package `utils` , un générateur de nombres pseudo-aléatoires de base, nommé de manière appropriée `Random` . Cet objet peut être utilisé pour générer une valeur pseudo-aléatoire comme n'importe quel type de données numérique intégré (`int` , `float` , etc.). Vous pouvez également l'utiliser pour générer une valeur booléenne aléatoire ou un tableau aléatoire d'octets. Un exemple d'utilisation est le suivant:

```
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.
```

REMARQUE: cette classe ne produit que des nombres pseudo-aléatoires de qualité assez faible et ne devrait jamais être utilisée pour générer des nombres aléatoires pour des opérations cryptographiques ou d'autres situations où un caractère aléatoire de qualité supérieure est critique (pour cela, vous souhaitez utiliser la classe `SecureRandom` , comme indiqué ci-dessous). Une explication de la distinction entre le caractère aléatoire "sécurisé" et "aléatoire" dépasse le cadre de cet exemple.

Nombres aléatoires dans une plage spécifique

La méthode `nextInt(int bound)` de `Random` accepte une limite exclusive supérieure, c'est-à-dire un nombre que la valeur aléatoire renvoyée doit être inférieure à. Cependant, seule la méthode `nextInt` accepte une borne; `nextLong` , `nextDouble` etc. ne le font pas.

```
Random random = new Random();
random.nextInt(1000); // 0 - 999

int number = 10 + random.nextInt(100); // number is in the range of 10 to 109
```

À partir de Java 1.7, vous pouvez également utiliser `ThreadLocalRandom` (source). Cette classe fournit un générateur de nombres pseudo-aléatoires (PRNG). Notez que la méthode `nextInt` de cette classe accepte les limites supérieure et inférieure.

```
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);
```

Notez que la documentation officielle indique que `nextInt(int bound)` peut faire des choses étranges lorsque la `bound` est proche de $2^{30}+1$ (emphase ajoutée):

> L'algorithme est un peu délicat. **Il rejette les valeurs qui entraîneraient une distribution inégale** (due au fait que 2 ^ 31 n'est pas divisible par n). La probabilité qu'une valeur soit rejetée dépend de n. **Le pire des cas est n = 2 ^ 30 + 1, pour lequel la probabilité d'un rejet est 1/2, et le nombre attendu d'itérations avant que la boucle ne se termine est 2.**

En d'autres termes, spécifier une limite diminuera (légèrement) les performances de la méthode `nextInt` , et cette diminution de la performance deviendra plus prononcée à mesure que la `bound` approche la moitié de la valeur max int.

Génération de nombres pseudo-aléatoires sécurisés par cryptographie

`Random` et `ThreadLocalRandom` sont assez bons pour un usage quotidien, mais ils ont un gros problème: ils sont basés sur un générateur de congruence linéaire , un algorithme dont la sortie peut être prédite assez facilement. Ainsi, ces deux classes **ne** conviennent **pas** aux utilisations cryptographiques (telles que la génération de clés).

On peut utiliser `java.security.SecureRandom` dans les situations où un PRNG avec une sortie très difficile à prévoir est requis. Prédire les nombres aléatoires créés par les instances de cette classe est suffisamment difficile pour étiqueter la classe comme étant **sécurisée sur le plan cryptographique** .

```
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));
    }
}
```

En plus d'être sécurisé sur le plan cryptographique, `SecureRandom` dispose d'une période gigantesque de 2 ^{160} , contre 2 ^{48} pour la période `Random` . Il a cependant l'inconvénient d'être beaucoup plus lent que `Random` et d'autres PRNG linéaires tels que Mersenne Twister et Xorshift .

Notez que la mise en œuvre de SecureRandom dépend à la fois de la plate-forme et du fournisseur. Le `SecureRandom` par défaut (fourni par le fournisseur `SUN` dans `sun.security.provider.SecureRandom`):

- sur des systèmes de type Unix, dotés de données provenant de `/dev/random` et / ou `/dev/urandom` .
- sur Windows, doté d'appels à `CryptGenRandom()` dans CryptoAPI .

Sélectionner des nombres aléatoires sans doublons

```
/**
 * 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;
}
```

La méthode fonctionne en boucle à travers un tableau qui a la taille de la longueur demandée et trouve la longueur restante des nombres possibles. Il définit un nombre aléatoire de ces nombres possibles `newRandSpot` et trouve ce nombre dans le nombre restant non pris. Cela se fait en parcourant la plage et en vérifiant si ce nombre a déjà été pris.

Par exemple, si la plage est 5 et que la longueur est 3 et que nous avons déjà choisi le nombre 2. Nous avons alors 4 nombres restants, nous obtenons donc un nombre aléatoire compris entre 1 et 4 et nous parcourons la plage (5). que nous avons déjà utilisé (2).

Maintenant, disons que le nombre suivant choisi entre 1 et 4 est 3. Sur la première boucle, nous obtenons 1 qui n'a pas encore été pris, nous pouvons donc en retirer 1 de 3, ce qui fait 2. Maintenant, sur la deuxième boucle, nous obtenons 2 donc nous ne faisons rien. Nous suivons ce schéma jusqu'à ce que nous arrivions à 4 où une fois que nous supprimons 1, il devient 0 et nous définissons donc le nouveau randomNumber sur 4.

Génération de nombres aléatoires avec une graine spécifiée

```
//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);
```

L'utilisation de la même graine pour générer des nombres aléatoires renverra les mêmes nombres à chaque fois, donc définir une graine différente pour chaque instance `Random` est une bonne idée si vous ne voulez pas vous retrouver avec des numéros en double.

`System.currentTimeMillis()` est une bonne méthode pour obtenir un `Long` différent pour chaque appel:

```
Random random = new Random(System.currentTimeMillis());
ThreadLocalRandom.current().setSeed(System.currentTimeMillis());
```

Générer un nombre aléatoire en utilisant apache-common lang3

Nous pouvons utiliser `org.apache.commons.lang3.RandomUtils` pour générer des nombres aléatoires

en utilisant une seule ligne.

```
int x = RandomUtils.nextInt(1, 1000);
```

La méthode `nextInt(int startInclusive, int endExclusive)` prend une plage.

En dehors de int, nous pouvons générer des nombres `long` , `double` , `float` et `bytes` aléatoires en utilisant cette classe.

`RandomUtils` classe `RandomUtils` contient les méthodes suivantes:

```
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.
```

Lire Génération de nombres aléatoires en ligne: https://riptutorial.com/fr/java/topic/890/generation-de-nombres-aleatoires

Chapitre 78: Génériques

Introduction

Les génériques sont une fonctionnalité de programmation générique qui étend le système de type Java pour permettre à un type ou à une méthode de fonctionner sur des objets de différents types tout en offrant une sécurité de type à la compilation. En particulier, la structure de collections Java prend en charge les génériques pour spécifier le type d'objets stockés dans une instance de collection.

Syntaxe

- class ArrayList <E> {} // une classe générique avec le paramètre de type E
- class HashMap <K, V> {} // une classe générique avec deux paramètres de type K et V
- <E> void print (élément E) {} // une méthode générique avec le paramètre de type E
- ArrayList <String> noms; // déclaration d'une classe générique
- ArrayList <?> Objets; // déclaration d'une classe générique avec un paramètre de type inconnu
- new ArrayList <String> () // instanciation d'une classe générique
- new ArrayList <> () // instanciation avec inférence de type "diamond" (Java 7 ou ultérieur)

Remarques

Les génériques sont implémentés dans Java through Type Erasure, ce qui signifie que pendant l'exécution, les informations de type spécifiées dans l'instanciation d'une classe générique ne sont pas disponibles. Par exemple, l'instruction `List<String> names = new ArrayList<>();` produit un objet liste à partir duquel le type d'élément `String` ne peut pas être récupéré à l'exécution. Toutefois, si la liste est stockée dans un champ de type `List<String>` , ou transmise à un paramètre de méthode / constructeur du même type ou renvoyée par une méthode de ce type de retour, les informations de type complètes *peuvent* être récupérées à l'exécution via l'API Java Reflection.

Cela signifie également que lors de la conversion vers un type générique (par exemple: `(List<String>) list`), la distribution est une *distribution non contrôlée* . Étant donné que le paramètre `<String>` est effacé, la machine virtuelle Java ne peut pas vérifier si une conversion d'une `List<?>` Vers une `List<String>` est correcte. La JVM ne voit un cast que pour `List` to `List` à l'exécution.

Examples

Créer une classe générique

Les génériques permettent aux classes, interfaces et méthodes de prendre d'autres classes et interfaces en tant que paramètres de type.

Cet exemple utilise la classe générique `Param` pour prendre un seul **paramètre de type** `T` , délimité par des crochets (`<>`):

```
public class Param<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

Pour instancier cette classe, fournissez un **argument de type** à la place de `T` Par exemple, `Integer` :

```
Param<Integer> integerParam = new Param<Integer>();
```

L'argument type peut être n'importe quel type de référence, y compris les tableaux et autres types génériques:

```
Param<String[]> stringArrayParam;
Param<int[][]> int2dArrayParam;
Param<Param<Object>> objectNestedParam;
```

Dans Java SE 7 et versions ultérieures, l'argument de type peut être remplacé par un ensemble vide d'arguments de type (`<>`) appelé *diamant* :

Java SE 7

```
Param<Integer> integerParam = new Param<>();
```

Contrairement aux autres identificateurs, les paramètres de type ne comportent aucune contrainte de dénomination. Cependant, leurs noms sont généralement la première lettre de leur but en majuscule. (Cela est vrai même dans les JavaDocs officiels.)
Les exemples incluent `T` pour "type" , `E` pour "élément" et `K` / `V` pour "clé" / "valeur" .

Extension d'une classe générique

```
public abstract class AbstractParam<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
```

```
        this.value = value;
    }
}
```

`AbstractParam` est une classe abstraite déclarée avec un paramètre de type `T` Lors de l'extension de cette classe, ce paramètre de type peut être remplacé par un argument de type écrit à l'intérieur de `<>` ou le paramètre de type peut rester inchangé. Dans les premier et deuxième exemples ci-dessous, `String` et `Integer` remplacent le paramètre type. Dans le troisième exemple, le paramètre de type reste inchangé. Le quatrième exemple n'utilise pas de génériques du tout, il est donc similaire à si la classe avait un paramètre `Object` . Le compilateur avertira que `AbstractParam` est un type brut, mais il compilera la classe `ObjectParam` . Le cinquième exemple a 2 paramètres de type (voir «paramètres de type multiples» ci-dessous), en choisissant le deuxième paramètre comme paramètre de type transmis à la super-classe.

```
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> {
    // ...
}
```

Ce qui suit est l'utilisation:

```
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);
```

Notez que dans la classe `Email` , la `T getValue()` agit comme si elle avait une signature de `String`

`getValue()` et que la `void setValue(T)` agit comme si elle avait été déclarée `void setValue(String)` .

Il est également possible d'instancier avec une classe interne anonyme avec des accolades vides (`{}`):

```
AbstractParam<Double> height = new AbstractParam<Double>(){};
height.setValue(198.6);
```

Notez que l' utilisation du diamant avec des classes internes anonymes n'est pas autorisée.

Paramètres de type multiple

Java permet d'utiliser plusieurs paramètres de type dans une classe ou une interface générique. Plusieurs paramètres de type peuvent être utilisés dans une classe ou une interface en plaçant une **liste** de types **séparés** par des **virgules** entre les crochets. Exemple:

```
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'utilisation peut se faire comme ci-dessous:

```
MultiGenericParam<String, String> aParam = new MultiGenericParam<String, String>("value1",
"value2");
MultiGenericParam<Integer, Double> dayOfWeekDegrees = new MultiGenericParam<Integer,
Double>(1, 2.6);
```

Déclaration d'une méthode générique

Les méthodes peuvent aussi avoir des paramètres de type génériques .

```
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);
    }
}
```

Notez que nous n'avons pas à transmettre un argument de type réel à une méthode générique. Le compilateur déduit l'argument de type pour nous, en fonction du type de cible (par exemple, la variable à laquelle nous attribuons le résultat) ou des types d'arguments réels. Il infère généralement l'argument de type le plus spécifique qui rendra le type d'appel correct.

Parfois, bien que rarement, il peut être nécessaire de remplacer cette inférence de type par des arguments de type explicites:

```
void usage() {
    consumeObjects(this.<Object>makeList("Jeff", "Atwood").stream());
}

void consumeObjects(Stream<Object> stream) { ... }
```

Il est nécessaire dans cet exemple parce que le compilateur ne peut pas "regarder en avant" pour voir que `Object` est désiré pour `T` après avoir appelé `stream()` et que cela impliquerait autrement `String` basé sur les arguments `makeList` . Notez que le langage Java ne supporte pas omettre la classe ou de l' objet sur lequel la méthode est appelée (`this` dans l'exemple ci - dessus) lorsque des arguments de type sont explicitement prévus.

Le diamant

Java SE 7

Java 7 a introduit le *Diamond* [1] pour supprimer certaines plaques chauffantes autour de l'instanciation de classe générique. Avec Java 7+, vous pouvez écrire:

```
List<String> list = new LinkedList<>();
```

Où vous deviez écrire dans les versions précédentes, ceci:

```
List<String> list = new LinkedList<String>();
```

L'une des limitations concerne les classes anonymes , dans lesquelles vous devez toujours fournir le paramètre type dans l'instanciation:

```
// 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

Bien que l'utilisation du diamant avec Anonymous Inner Classes ne soit pas prise en charge dans Java 7 et 8, **il sera inclus en tant que nouvelle fonctionnalité dans Java 9** .

Note de bas de page:

1 - Certaines personnes appellent l'utilisation `<>` l' *opérateur* diamantaire. Ceci est une erreur. Le diamant ne se comporte pas comme un opérateur et n'est ni décrit ni répertorié en tant qu'opérateur dans les tutoriels Java (officiels) ou JLS. En effet, `<>` n'est même pas un jeton Java distinct. C'est plutôt un jeton `<` suivi d'un jeton `>` , et il est légal (bien que de mauvais style) d'avoir des espaces ou des commentaires entre les deux. Le JLS et les tutoriels se réfèrent systématiquement à `<>` comme "le diamant", ce qui en fait le terme correct.

Nécessitant plusieurs bornes supérieures ("étend A & B")

Vous pouvez exiger un type générique pour étendre plusieurs limites supérieures.

Exemple: nous voulons trier une liste de numéros, mais le `Number` n'implémente pas `Comparable` .

```
public <T extends Number & Comparable<T>> void sortNumbers( List<T> n ) {
  Collections.sort( n );
}
```

Dans cet exemple, `T` doit étendre `Number` *et* implémenter `Comparable<T>` qui devrait correspondre à toutes les implémentations de nombres intégrées "normales" comme `Integer` ou `BigDecimal` mais ne correspond pas aux plus exotiques comme `Striped64` .

Étant donné que l'héritage multiple n'est pas autorisé, vous pouvez utiliser au plus une classe en tant que limite et ce doit être la première liste. Par exemple, `<T extends Comparable<T> & Number>` n'est pas autorisé car Comparable est une interface et non une classe.

Créer une classe générique limitée

Vous pouvez restreindre les types valides utilisés dans une **classe générique** en limitant ce type dans la définition de classe. Étant donné la hiérarchie de type simple suivante:

```
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";
    }
}
```

Sans **génériques limités** , nous ne pouvons pas créer une classe de conteneur à la fois générique et sachant que chaque élément est un animal:

```
public class AnimalContainer<T> {

    private Collection<T> col;

    public AnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // Illegal, type T doesn't have makeSound()
            // it is used as an java.lang.Object here
            System.out.println(t.makeSound());
        }
    }
}
```

Avec la liaison générique dans la définition de classe, c'est désormais possible.

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

Cela limite également les instanciations valides du type générique:

```
// 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>();
```

Décider entre `T`,`? super T`, et `? étend T`

La syntaxe des génériques Java limite les caractères génériques, représentant le type inconnu par `?` est:

- `? extends T` représente un caractère générique supérieur. Le type inconnu représente un type qui doit être un sous-type de T ou un type T lui-même.
- `? super T` représente un caractère générique limité inférieur. Le type inconnu représente un type qui doit être un supertype de T ou un type T lui-même.

En règle générale, vous devez utiliser

- `? extends T` si vous avez seulement besoin d'un accès "read" ("input")
- `? super T` si vous avez besoin d'un accès "écriture" ("sortie")
- `T` si vous avez besoin des deux ("modifier")

Utiliser `extends` ou `super` est généralement *préférable* car cela rend votre code plus flexible (comme dans: autoriser l'utilisation de sous-types et de supertypes), comme vous le verrez ci-dessous.

```
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) {}
}
```

Le compilateur sera maintenant capable de détecter certaines mauvaises utilisations:

```
 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!
}
```

Choisir le bon `T` , `? super T` ou `? extends T` est *nécessaire* pour permettre l'utilisation avec des sous-types. Le compilateur peut alors assurer la sécurité du type; vous ne devriez pas avoir besoin de lancer (ce qui n'est pas sûr et peut causer des erreurs de programmation) si vous les utilisez correctement.

Si ce n'est pas facile à comprendre, n'oubliez pas la règle **PECS** :

> **P** roducer utilise des **"E"** et **C** xtends utilise onsumer **"S** uper".

(Le producteur n'a qu'un accès en écriture et le consommateur n'a qu'un accès en lecture)

Avantages de la classe et de l'interface génériques

Le code qui utilise des génériques présente de nombreux avantages par rapport au code non générique. Voici les principaux avantages

Contrôles de type plus forts au moment de la compilation

Un compilateur Java applique une vérification de type forte au code générique et génère des erreurs si le code viole la sécurité de type. Corriger les erreurs de compilation est plus facile que de corriger les erreurs d'exécution, ce qui peut être difficile à trouver.

Elimination des moulages

L'extrait de code suivant sans génériques nécessite la conversion en:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Lors de la réécriture pour *utiliser des génériques* , le code ne nécessite pas de conversion:

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0);   // no cast
```

Permettre aux programmeurs d'implémenter des algorithmes génériques

En utilisant des génériques, les programmeurs peuvent implémenter des algorithmes génériques qui fonctionnent sur des collections de différents types, peuvent être personnalisés et sont de type sûr et plus faciles à lire.

Liaison du paramètre générique à plus de 1 type

Les paramètres génériques peuvent également être liés à plusieurs types à l'aide de la syntaxe `T extends Type1 & Type2 & ...`.

Supposons que vous souhaitiez créer une classe dont le type générique devrait implémenter `Flushable` et `Closeable` , vous pouvez écrire

```
class ExampleClass<T extends Flushable & Closeable> {
}
```

Maintenant, `ExampleClass` accepte uniquement comme paramètres génériques, les types qui implémentent à la fois `Flushable` **et** `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.
```

Les méthodes de classe peuvent choisir d'inférer des arguments de type générique comme `Closeable` ou `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.
    }
}
```

Remarque:

Vous ne pouvez pas lier le paramètre générique à l'un des types utilisant la clause *OR* (`|`). Seule la clause *AND* (`&`) est prise en charge. Le type générique ne peut étendre qu'une classe et de nombreuses interfaces. La classe doit être placée au début de la liste.

Instancier un type générique

En raison de l'effacement de type, les éléments suivants ne fonctionneront pas:

```
public <T> void genericMethod() {
    T t = new T(); // Can not instantiate the type T.
}
```

Le type `T` est effacé. Puisque, à l'exécution, la JVM ne sait pas ce que `T` était à l'origine, elle ne sait pas quel constructeur appeler.

Solutions de contournement

1. Passer la classe de `T` lors de l'appel de `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);
```

Ce qui génère des exceptions, car il n'y a aucun moyen de savoir si la classe passée a un constructeur par défaut accessible.

Java SE 8

2. Passer une référence au constructeur de `T` :

```
public <T> void genericMethod(Supplier<T> cons) {
    T t = cons.get();
}
```

```
genericMethod(String::new);
```

Se référant au type générique déclaré dans sa propre déclaration

Comment allez-vous utiliser une instance d'un type générique (éventuellement supplémentaire) hérité dans une déclaration de méthode dans le type générique lui-même déclaré? C'est l'un des problèmes que vous rencontrerez lorsque vous approfondirez un peu les génériques, mais que vous rencontrerez un problème assez commun.

Supposons que nous ayons un type `DataSeries<T>` (interface ici), qui définit une série de données générique contenant des valeurs de type `T` Il est fastidieux de travailler directement avec ce type lorsque nous souhaitons effectuer de nombreuses opérations avec, par exemple, des valeurs doubles. Nous définissons donc `DoubleSeries extends DataSeries<Double>` . Supposons maintenant que le type `DataSeries<T>` ait une méthode `add(values)` qui ajoute une autre série de la même longueur et en renvoie une nouvelle. Comment pouvons-nous appliquer le type de `values` et le type de retour à `DoubleSeries` plutôt qu'à `DataSeries<Double>` dans notre classe dérivée?

Le problème peut être résolu en ajoutant un paramètre de type générique renvoyant à et étendant le type en cours de déclaration (appliqué à une interface ici, mais il en va de même pour les classes):

```
public interface DataSeries<T, DS extends DataSeries<T, DS>> {
    DS add(DS values);
    List<T> data();
}
```

Ici, `T` représente le type de données que la série contient, par exemple `Double` et `DS` la série elle-même. Un type hérité (ou des types) peut maintenant être facilement implémenté en substituant le

paramètre mentionné ci-dessus par un type dérivé correspondant, produisant ainsi une définition concrète à `Double` base du formulaire:

```
public interface DoubleSeries extends DataSeries<Double, DoubleSeries> {
    static DoubleSeries instance(Collection<Double> data) {
        return new DoubleSeriesImpl(data);
    }
}
```

À ce moment, même un IDE implémentera l'interface ci-dessus avec les types corrects en place, ce qui, après un peu de remplissage de contenu, pourrait ressembler à ceci:

```
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);
    }
}
```

Comme vous pouvez le voir, la méthode `add` est déclarée en tant que `DoubleSeries add(DoubleSeries values)` et le compilateur est satisfait.

Le modèle peut être imbriqué si nécessaire.

Utilisation de instanceof avec Generics

Utiliser des génériques pour définir le type dans instanceof

Considérons la classe générique suivante `Example` déclaré avec le paramètre formel `<T>` :

```
class Example<T> {
    public boolean isTypeAString(String s) {
        return s instanceof T; // Compilation error, cannot use T as class type here
    }
}
```

Cela provoquera toujours une erreur de compilation car dès que le compilateur compile le *source Java en bytecode Java,* il applique un processus appelé *effacement de type* , qui convertit tout le code générique en code non générique, rendant impossible la distinction entre les types T au moment de l'exécution. Le type utilisé avec `instanceof` doit être *reifiable* , ce qui signifie que toutes les informations sur le type doivent être disponibles au moment de l'exécution, ce qui n'est généralement pas le cas pour les types génériques.

La classe suivante représente à quoi ressemblent deux classes différentes d' `Example` , `Example<String>` et `Example<Number>` après que les génériques ont été supprimés par *type d'effacement* :

```
class Example { // formal parameter is gone
    public boolean isTypeAString(String s) {
        return s instanceof Object; // Both <String> and <Number> are now Object
    }
}
```

Comme les types ont disparu, la JVM ne peut pas savoir quel type est `T`

Exception à la règle précédente

Vous pouvez toujours utiliser *des caractères génériques sans limite* (?) Pour spécifier un type dans l' `instanceof` comme suit:

```
    public boolean isAList(Object obj) {
        return obj instanceof List<?>;
    }
```

Cela peut être utile pour évaluer si une instance `obj` est une `List` ou non:

```
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
```

En fait, les caractères génériques non liés sont considérés comme un type reifiable.

Utiliser une instance générique avec instanceof

Le revers de la médaille est que l'utilisation d'une instance `t` de `T` avec `instanceof` est légale, comme le montre l'exemple suivant:

```
class Example<T> {
    public boolean isTypeAString(T t) {
        return t instanceof String; // No compilation error this time
    }
}
```

car après le type d'effacement, la classe ressemblera à ceci:

```
class Example { // formal parameter is gone
    public boolean isTypeAString(Object t) {
        return t instanceof String; // No compilation error this time
    }
}
```

Étant donné que, même si l'effacement de type se produit de toute façon, la machine virtuelle Java peut maintenant distinguer différents types de mémoire, même s'ils utilisent le même type de référence (`Object`), comme le montre l'extrait suivant:

```
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
```

Différentes manières d'implémenter une interface générique (ou d'étendre une classe générique)

Supposons que l'interface générique suivante ait été déclarée:

```
public interface MyGenericInterface<T> {
    public void foo(T t);
}
```

Vous trouverez ci-dessous la liste des moyens possibles pour la mettre en œuvre.

Implémentation de classe non générique avec un type spécifique

Choisissez un type spécifique pour remplacer le paramètre de type formel `<T>` de `MyGenericClass` et implémentez-le, comme le montre l'exemple suivant:

```
public class NonGenericClass implements MyGenericInterface<String> {
    public void foo(String t) { } // type T has been replaced by String
}
```

Cette classe ne traite que de `String` , ce qui signifie que l'utilisation de `MyGenericInterface` avec des paramètres différents (par exemple, `Integer` , `Object` etc.) ne sera pas compilée, comme le montre l'extrait de code suivant:

```
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
```

Implémentation de classe générique

Déclarez une autre interface générique avec le paramètre de type formel `<T>` qui implémente `MyGenericInterface` , comme suit:

```
public class MyGenericSubclass<T> implements MyGenericInterface<T> {
    public void foo(T t) { } // type T is still the same
    // other methods...
}
```

Notez qu'un paramètre de type formel différent peut avoir été utilisé, comme suit:

```
public class MyGenericSubclass<U> implements MyGenericInterface<U> { // equivalent to the
previous declaration
    public void foo(U t) { }
    // other methods...
}
```

Implémentation de la classe de type brute

Déclarez une classe non générique qui implémente `MyGenericInteface` tant que *type brut* (sans utiliser de générique du tout), comme suit:

```
public class MyGenericSubclass implements MyGenericInterface {
    public void foo(Object t) { } // type T has been replaced by Object
    // other possible methods
}
```

Cette méthode n'est **pas** recommandée, car elle n'est pas sécurisée à 100% à l'exécution, car elle mélange *le type brut* (de la sous-classe) avec les *génériques* (de l'interface), ce qui est également source de confusion. Les compilateurs Java modernes émettront un avertissement avec ce type d'implémentation, néanmoins le code - pour des raisons de compatibilité avec les anciennes JVM (1.4 ou antérieures) - sera compilé.

Toutes les méthodes énumérées ci-dessus sont également autorisées lorsque vous utilisez une classe générique en tant que super-type au lieu d'une interface générique.

Utilisation de génériques pour la diffusion automatique

Avec les génériques, il est possible de renvoyer tout ce que l'appelant attend:

```
private Map<String, Object> data;
public <T> T get(String key) {
    return (T) data.get(key);
}
```

La méthode compilera avec un avertissement. Le code est en fait plus sûr qu'il n'y paraît, car le runtime Java effectuera un transtypage lorsque vous l'utiliserez:

```
Bar bar = foo.get("bar");
```

C'est moins sûr quand vous utilisez des types génériques:

```
List<Bar> bars = foo.get("bars");
```

Ici, la distribution fonctionnera lorsque le type renvoyé est un type quelconque de `List` (c'est-à-dire que le retour de `List<String>` ne déclenche pas une `ClassCastException` ; vous l'obtiendrez éventuellement lors de la sortie d'éléments de la liste).

Pour contourner ce problème, vous pouvez créer une API qui utilise des clés tapées:

```
public final static Key<List<Bar>> BARS = new Key<>("BARS");
```

avec cette méthode `put()` :

```
public <T> T put(Key<T> key, T value);
```

Avec cette approche, vous ne pouvez pas insérer le mauvais type dans la carte, le résultat sera donc toujours correct (à moins que vous ne créiez accidentellement deux clés portant le même nom mais des types différents).

En relation:

- Carte de type sécurisé

Obtenir une classe qui vérifie le paramètre générique à l'exécution

De nombreux paramètres génériques non liés, tels que ceux utilisés dans une méthode statique, ne peuvent pas être récupérés à l'exécution (voir *Autres threads* sur l' *effacement*). Cependant, une stratégie commune est utilisée pour accéder au type satisfaisant un paramètre générique sur une classe à l'exécution. Cela permet un code générique qui dépend de l'accès au type *sans* avoir à envoyer des informations de type à chaque appel.

Contexte

Le paramétrage générique sur une classe peut être inspecté en créant une classe interne anonyme. Cette classe capturera les informations de type. En général, ce mécanisme est désigné sous le nom de **jeton de type super** , qui est détaillé dans l' article du blog de Neal Gafter .

Les implémentations

Trois implémentations courantes en Java sont:

- Type de goyaveToken
- Référence du type de printemps
- Type de JacksonRéférence

Exemple d'utilisation

```
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();
    }
}
```

Lire Génériques en ligne: https://riptutorial.com/fr/java/topic/92/generiques

Chapitre 79: Gestion de la mémoire Java

Remarques

En Java, les objets sont alloués dans le tas et la mémoire de tas est récupérée par la récupération de place automatique. Un programme d'application ne peut pas supprimer explicitement un objet Java.

Les principes de base de la récupération de la mémoire Java sont décrits dans l'exemple de la récupération de la mémoire . D'autres exemples décrivent la finalisation, comment déclencher manuellement le ramasse-miettes et le problème des fuites de stockage.

Examples

Finalisation

Un objet Java peut déclarer une méthode de `finalize` . Cette méthode est appelée juste avant que Java ne libère la mémoire pour l'objet. Il ressemblera généralement à ceci:

```
public class MyClass {

    //Methods for the class

    @Override
    protected void finalize() throws Throwable {
        // Cleanup code
    }
}
```

Cependant, il existe des mises en garde importantes sur le comportement de la finalisation Java.

- Java ne garantit pas qu'une méthode `finalize()` sera appelée.
- Java ne garantit même pas qu'une méthode `finalize()` sera appelée quelque temps pendant la durée de vie de l'application en cours d'exécution.
- La seule chose garantie est que la méthode sera appelée avant que l'objet soit supprimé ... si l'objet est supprimé.

Les mises en garde ci-dessus signifient que c'est une mauvaise idée de compter sur la méthode `finalize` pour effectuer des actions de nettoyage (ou autres) qui doivent être effectuées en temps opportun. Dépendre de la finalisation peut entraîner des fuites de stockage, des fuites de mémoire et d'autres problèmes.

En bref, il existe très peu de situations où la finalisation est en fait une bonne solution.

Les finaliseurs ne s'exécutent qu'une seule fois

Normalement, un objet est supprimé après sa finalisation. Cependant, cela n'arrive pas tout le

temps. Prenons l'exemple suivant [1] :

```
public class CaptainJack {
    public static CaptainJack notDeadYet = null;

    protected void finalize() {
        // Resurrection!
        notDeadYet = this;
    }
}
```

Lorsqu'une instance de `CaptainJack` devient inaccessible et que le ramasse-miette tente de le récupérer, la méthode `notDeadYet finalize()` attribue une référence à l'instance à la variable `notDeadYet` . Cela rendra l'instance accessible une fois de plus, et le ramasse-miettes ne le supprimera pas.

Question: Le capitaine Jack est-il immortel?

Réponse: non

Le catch est que la JVM exécutera uniquement un finaliseur sur un objet une fois dans sa vie. Si vous attribuez la valeur `null` à `notDeadYet` provoquant une inaccessibilité de l'instance `notDeadYet` , le ramasse-miettes n'appelle pas `finalize()` sur l'objet.

1 - Voir https://en.wikipedia.org/wiki/Jack_Harkness .

Déclenchement manuel du GC

Vous pouvez déclencher manuellement le garbage collector en appelant

```
System.gc();
```

Toutefois, Java ne garantit pas l'exécution du Garbage Collector au retour de l'appel. Cette méthode "suggère simplement" à la machine virtuelle Java (Java Virtual Machine) que vous voulez qu'elle exécute le ramasse-miettes, mais ne le force pas à le faire.

Il est généralement considéré comme une mauvaise pratique de tenter de déclencher manuellement la récupération de place. La JVM peut être exécutée avec l'option `-XX:+DisableExplicitGC` pour désactiver les appels à `System.gc()` . Le déclenchement de la récupération de place en appelant `System.gc()` peut perturber les activités normales de gestion des ordures / de promotion d'objet de l'implémentation spécifique du ramasse-miettes utilisée par la JVM.

Collecte des ordures

L'approche C ++ - new and delete

Dans un langage comme le C ++, le programme d'application est responsable de la gestion de la mémoire utilisée par la mémoire allouée dynamiquement. Lorsqu'un objet est créé dans le tas C

++ à l'aide de l'opérateur `new` , il doit y avoir une utilisation correspondante de l'opérateur `delete` pour éliminer l'objet:

- Si le programme oublie de `delete` un objet et ne fait que l'oublier, la mémoire associée est perdue pour l'application. Le terme pour cette situation est une *fuite de mémoire* , et trop de mémoire fuit, une application est susceptible d'utiliser de plus en plus de mémoire et finit par tomber en panne.
- D'un autre côté, si une application tente de `delete` le même objet deux fois ou d'utiliser un objet après sa suppression, l'application risque de se bloquer en raison de problèmes de corruption de mémoire.

Dans un programme C ++ compliqué, l'implémentation de la gestion de la mémoire en utilisant `new` et `delete` peut prendre beaucoup de temps. En effet, la gestion de la mémoire est une source commune de bogues.

L'approche Java - garbage collection

Java adopte une approche différente. Au lieu d'un opérateur de `delete` explicite, Java fournit un mécanisme automatique appelé récupération de place pour récupérer la mémoire utilisée par les objets devenus inutiles. Le système d'exécution Java prend la responsabilité de trouver les objets à éliminer. Cette tâche est effectuée par un composant appelé « *garbage collector*» ou «GC».

A tout moment pendant l'exécution d'un programme Java, nous pouvons diviser l'ensemble de tous les objets existants en deux sous-ensembles distincts [1] :

- Les objets accessibles sont définis par le JLS comme suit:

  > Un objet accessible est tout objet auquel on peut accéder dans tout calcul continu potentiel à partir d'un thread en direct.

  En pratique, cela signifie qu'il existe une chaîne de références à partir d'une variable locale dans la portée ou d'une variable `static` permettant à un code d'atteindre l'objet.

- Les objets inaccessibles sont des objets qui *ne peuvent pas* être atteints comme ci-dessus.

Tout objet inaccessible est *éligible* pour la récupération de la mémoire. Cela ne signifie pas qu'ils *seront* collectés. En réalité:

- Un objet inaccessible *n'est pas* collecté immédiatement après être devenu inaccessible [1] .
- Un objet inaccessible *peut ne jamais* être récupéré.

La spécification du langage Java donne beaucoup de latitude à une implémentation JVM pour décider quand collecter des objets inaccessibles. Il donne également (en pratique) l'autorisation à une implémentation JVM d'être prudente dans la manière dont elle détecte les objets inaccessibles.

La seule chose que JLS garantit, c'est qu'aucun objet *accessible* ne sera jamais récupéré.

Que se passe-t-il lorsqu'un objet devient inaccessible

Tout d'abord, rien ne se produit spécifiquement lorsqu'un objet *devient* inaccessible. Les choses ne se produisent que lorsque le ramasse-miettes s'exécute *et* qu'il détecte que l'objet est inaccessible. De plus, il est fréquent qu'un cycle de CPG ne détecte pas tous les objets inaccessibles.

Lorsque le CPG détecte un objet inaccessible, les événements suivants peuvent se produire.

1. S'il existe des objets `Reference` faisant référence à l'objet, ces références seront effacées avant la suppression de l'objet.
2. Si l'objet est *définissable* , il sera finalisé. Cela se produit avant que l'objet soit supprimé.
3. L'objet peut être supprimé et la mémoire qu'il occupe peut être récupérée.

Notez qu'il existe une séquence claire dans laquelle les événements ci-dessus *peuvent* se produire, mais rien n'oblige le ramasse-miettes à effectuer la suppression finale d'un objet spécifique dans un délai spécifique.

Exemples d'objets accessibles et inaccessibles

Prenons les exemples de classes suivants:

```
// 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
    }
}
```

Examinons ce qui se passe quand on appelle `test()` . Les instructions T1, T2 et T3 créent des objets `Node` , et tous les objets sont accessibles via les variables `n1` , `n2` et `n3` , respectivement. L'instruction T4 assigne la référence à l'objet 2nd `Node` au champ `next` du premier. Lorsque cela est fait, le 2ème `Node` est accessible via deux chemins:

```
n2 -> Node2
n1 -> Node1, Node1.next -> Node2
```

Dans l'instruction T5, nous affectons `null` à `n2` . Cela brise la première des chaînes d'accessibilité pour `Node2` , mais la seconde reste intacte, donc `Node2` est toujours accessible.

Dans l'instruction T6, nous affectons `null` à `n3` . Cela casse la seule chaîne d'accessibilité pour `Node3` , ce qui rend inaccessible `Node3` . Cependant, `Node1` et `Node2` sont tous deux encore accessibles via la `n1` variable.

Enfin, lorsque la méthode `test()` retourne, ses variables locales `n1` , `n2` et `n3` sont hors de portée et ne peuvent donc pas être consultées. Cela brise les chaînes restantes pour joignabilité `Node1` et `Node2` , et tous les `Node` objets sont ni inaccessibles et *admissibles* à la collecte des ordures.

---

1 - Ceci est une simplification qui ignore la finalisation et les classes de `Reference` . 2 - Hypothétiquement, une implémentation Java pourrait le faire, mais le coût de la performance le rend peu pratique.

Réglage de la taille du tas, du permGen et de la pile

Lorsqu'une machine virtuelle Java démarre, elle doit connaître la taille du tas et la taille par défaut des piles de threads. Celles-ci peuvent être spécifiées en utilisant des options de ligne de commande sur la commande `java` . Pour les versions de Java antérieures à Java 8, vous pouvez également spécifier la taille de la région PermGen du tas.

Notez que PermGen a été supprimé dans Java 8 et que si vous tentez de définir la taille de PermGen, l'option sera ignorée (avec un message d'avertissement).

Si vous ne spécifiez pas explicitement les tailles de tas et de piles, la machine virtuelle Java utilisera les valeurs par défaut calculées de manière spécifique à la version et à la plate-forme. Cela peut entraîner votre application utilisant trop peu ou trop de mémoire. Ceci est généralement OK pour les piles de threads, mais cela peut être problématique pour un programme qui utilise beaucoup de mémoire.

Définition des tailles de tas, PermGen et par défaut:

Les options JVM suivantes définissent la taille de segment de mémoire:

- `-Xms<size>` - définit la taille initiale du tas
- `-Xmx<size>` - définit la taille maximale du tas
- `-XX:PermSize<size>` - définit la taille initiale du PermGen
- `-XX:MaxPermSize<size>` : définit la taille maximale de PermGen
- `-Xss<size>` - définit la taille de la pile de threads par défaut

Le paramètre `<size>` peut être un nombre d'octets ou peut avoir un suffixe de `k` , `m` ou `g` . Ces derniers spécifient la taille en kilo-octets, mégaoctets et gigaoctets respectivement.

Exemples:

```
$ java -Xms512m -Xmx1024m JavaApp
$ java -XX:PermSize=64m -XX:MaxPermSize=128m JavaApp
$ java -Xss512k JavaApp
```

Trouver les tailles par défaut:

L'option `-XX:+printFlagsFinal` peut être utilisée pour imprimer les valeurs de tous les indicateurs avant de démarrer la machine virtuelle Java. Cela peut être utilisé pour imprimer les valeurs par défaut pour le tas et les paramètres de taille de pile comme suit:

- Pour Linux, Unix, Solaris et Mac OSX

  $ java -XX: + PrintFlagsFinal -version | grep -iE 'HeapSize | PermSize | ThreadStackSize'

- Pour les fenêtres:

  java -XX: + PrintFlagsFinal -version | findstr / i "HeapSize PermSize ThreadStackSize"

La sortie des commandes ci-dessus ressemblera à ceci:

```
uintx InitialHeapSize                          := 20655360        {product}
uintx MaxHeapSize                              := 331350016       {product}
uintx PermSize                                  = 21757952        {pd product}
uintx MaxPermSize                               = 85983232        {pd product}
 intx ThreadStackSize                           = 1024            {pd product}
```

Les tailles sont données en octets.

Fuites de mémoire en Java

Dans l'exemple de la récupération de données , nous avons sous-entendu que Java résout le problème des fuites de mémoire. Ce n'est pas vraiment vrai. Un programme Java peut perdre de la mémoire, même si les causes des fuites sont assez différentes.

Les objets accessibles peuvent fuir

Considérez l'implémentation de pile naïve suivante.

```
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;
    }
}
```

Lorsque vous `push` un objet et que vous `push` faites `pop` immédiatement, il y aura toujours une référence à l'objet dans le tableau de la `stack` .

La logique de l'implémentation de la pile signifie que cette référence ne peut pas être renvoyée à un client de l'API. Si un objet a été sauté, alors nous pouvons prouver qu'il ne peut *être «accessible dans aucun calcul continu potentiel à partir d'un thread en direct»* . Le problème est qu'une JVM de génération actuelle ne peut pas le prouver. Les JVM de génération actuelle ne prennent pas en compte la logique du programme pour déterminer si les références sont accessibles. (Pour commencer, ce n'est pas pratique.)

Mais mettre de côté la question de savoir ce que signifie réellement l' *accessibilité* , nous avons clairement une situation où l'implémentation de `NaiveStack` est "accrochée" à des objets qui doivent être récupérés. C'est une fuite de mémoire.

Dans ce cas, la solution est simple:

```
    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;
    }
```

Les caches peuvent être des fuites de mémoire

Une stratégie commune pour améliorer les performances du service consiste à mettre en cache les résultats. L'idée est de conserver un enregistrement des requêtes courantes et de leurs résultats dans une structure de données en mémoire appelée cache. Ensuite, chaque fois qu'une demande est faite, vous recherchez la requête dans le cache. Si la recherche réussit, vous renvoyez les résultats enregistrés correspondants.

Cette stratégie peut être très efficace si elle est correctement mise en œuvre. Cependant, si elle est implémentée de manière incorrecte, un cache peut être une fuite de mémoire. Prenons l'exemple suivant:

```
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;
    }
}
```

Le problème avec ce code est que, bien que tout appel à `doRequest` puisse ajouter une nouvelle entrée au cache, rien ne les supprime. Si le service reçoit continuellement des tâches différentes, le cache finira par consommer toute la mémoire disponible. Ceci est une forme de fuite de mémoire.

Une solution à ce problème consiste à utiliser un cache avec une taille maximale et à supprimer les anciennes entrées lorsque le cache dépasse le maximum. (Jeter la dernière entrée récemment utilisée est une bonne stratégie.) Une autre approche consiste à créer le cache à l'aide de `WeakHashMap` afin que la JVM puisse expulser les entrées de cache si le tas commence à être trop plein.

Lire Gestion de la mémoire Java en ligne: https://riptutorial.com/fr/java/topic/2804/gestion-de-la-memoire-java

Chapitre 80: Getters et Setters

Introduction

Cet article traite des getters et des setters; le moyen standard de fournir un accès aux données dans les classes Java.

Examples

Ajout de getters et de setters

L'encapsulation est un concept de base en POO. Il s'agit d'encapsuler les données et le code en une seule unité. Dans ce cas, il est recommandé de déclarer les variables comme `private` et d'y accéder via `Getters` et `Setters` pour les afficher et / ou les modifier.

```
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;
  }
}
```

Ces variables privées ne sont pas accessibles directement depuis l'extérieur de la classe. Ils sont donc protégés contre les accès non autorisés. Mais si vous voulez les voir ou les modifier, vous pouvez utiliser Getters et Setters.

`getXxx()` méthode `getXxx()` renvoie la valeur actuelle de la variable `xxx` , tandis que vous pouvez définir la valeur de la variable `xxx` aide de `setXxx()` .

La convention de dénomination des méthodes est (dans l'exemple la variable s'appelle `variableName`):

- Toutes les variables non `boolean`

```
getVariableName()   //Getter, The variable name should start with uppercase
setVariableName(..) //Setter, The variable name should start with uppercase
```

- variables `boolean`

```
  isVariableName()     //Getter, The variable name should start with uppercase
  setVariableName(...) //Setter, The variable name should start with uppercase
```

Les getters et les setters publics font partie de la définition de propriété d'un bean Java.

Utiliser un setter ou un getter pour implémenter une contrainte

Setters et Getters permettent à un objet de contenir des variables privées accessibles et modifiées avec des restrictions. Par exemple,

```
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;
    }
}
```

Dans cette classe `Person` , il existe une seule variable: `name` . Cette variable est accessible à l'aide de la méthode `getName()` et modifiée à l'aide de la `setName(String)` . Toutefois, si vous définissez un nom, le nouveau nom doit avoir une longueur supérieure à 2 caractères et ne pas être nul. L'utilisation d'une méthode setter plutôt que de rendre le `name` variable public permet aux autres utilisateurs de définir la valeur de `name` avec certaines restrictions. La même chose peut être appliquée à la méthode getter:

```
public String getName(){
   if(name.length()>16)
      return "Name is too large!";
   else
      return name;
}
```

Dans la méthode `getName()` modifiée ci-dessus, le `name` est renvoyé uniquement si sa longueur est inférieure ou égale à 16. Sinon, `"Name is too large"` est renvoyé. Cela permet au programmeur de créer des variables accessibles et modifiables à leur guise, empêchant les classes clientes d'éditer les variables de manière indésirable.

Pourquoi utiliser des getters et des setters?

Considérons une classe de base contenant un objet avec des getters et des setters dans Java:

```
public class CountHolder {
  private int count = 0;
```

```
  public int getCount() { return count; }
  public void setCount(int c) { count = c; }
}
```

Nous ne pouvons pas accéder à la variable `count` car elle est privée. Mais nous pouvons accéder aux `getCount()` et `setCount(int)` car elles sont publiques. Pour certains, cela pourrait soulever la question; pourquoi présenter l'intermédiaire? Pourquoi ne pas simplement les rendre publiques?

```
public class CountHolder {
  public int count = 0;
}
```

À toutes fins utiles, ces deux sont exactement les mêmes, du point de vue des fonctionnalités. La différence entre eux est l'extensibilité. Considérez ce que chaque classe dit:

- **Premièrement** : "J'ai une méthode qui vous donnera une valeur `int` et une méthode qui définira cette valeur sur un autre `int` ".
- **Deuxièmement** : "J'ai un `int` que vous pouvez définir et obtenir à votre guise."

Celles-ci peuvent sembler similaires, mais la première est en réalité beaucoup plus protégée dans sa nature; il vous permet seulement d'interagir avec sa nature interne comme **il le** dicte. Cela laisse la balle dans son camp; il arrive à choisir comment les interactions internes se produisent. Le second a exposé son implémentation interne en externe, et est désormais non seulement sujet aux utilisateurs externes, mais, dans le cas d'une API, **engagé** à maintenir cette implémentation (ou à publier une API non compatible).

Considérons si nous voulons synchroniser l'accès à la modification et à l'accès au compte. Dans le premier, c'est simple:

```
public class CountHolder {
  private int count = 0;

  public synchronized int getCount() { return count; }
  public synchronized void setCount(int c) { count = c; }
}
```

mais dans le deuxième exemple, cela est maintenant presque impossible sans passer par et en modifiant chaque endroit où la variable `count` est référencée. Pire encore, s'il s'agit d'un élément que vous fournissez dans une bibliothèque pour être consommé par d'autres, vous n'avez **aucun** moyen d'effectuer cette modification et vous devez faire le choix difficile mentionné ci-dessus.

Donc, cela pose la question; les variables publiques sont-elles une bonne chose (ou du moins pas mal)?

Je ne suis pas sûr D'une part, vous pouvez voir des exemples de variables publiques qui ont résisté à l'épreuve du temps (IE: la variable `out` référencée dans `System.out`). D'autre part, fournir une variable publique n'apporte aucun avantage en dehors des frais généraux extrêmement minimes et de la réduction potentielle de la verbosité. Ma ligne directrice serait ici que, si vous avez l' intention de faire d' un public variable, vous devez juger de ces critères avec préjudice

extrême:

1. La variable ne devrait avoir aucune raison concevable de changer **jamais** dans sa mise en œuvre. C'est quelque chose qui est extrêmement facile à bousiller (et, même si vous faites les choses correctement, les exigences peuvent changer), ce qui explique pourquoi les getters / setters constituent l'approche commune. Si vous avez une variable publique, il faut vraiment y réfléchir, surtout si elle est publiée dans une bibliothèque / framework / API.
2. La variable doit être référencée assez souvent pour que les gains minimaux résultant de la réduction de la verbosité le justifient. Je ne pense même pas que les frais généraux liés à l'utilisation d'une méthode par rapport au référencement direct devraient être pris en compte ici. C'est beaucoup trop négligeable pour 99,9% des demandes.

Il y a probablement plus que ce que je ne pensais pas. Si vous avez un doute, utilisez toujours les getters / setters.

Lire Getters et Setters en ligne: https://riptutorial.com/fr/java/topic/3560/getters-et-setters

Chapitre 81: Graphiques 2D en Java

Introduction

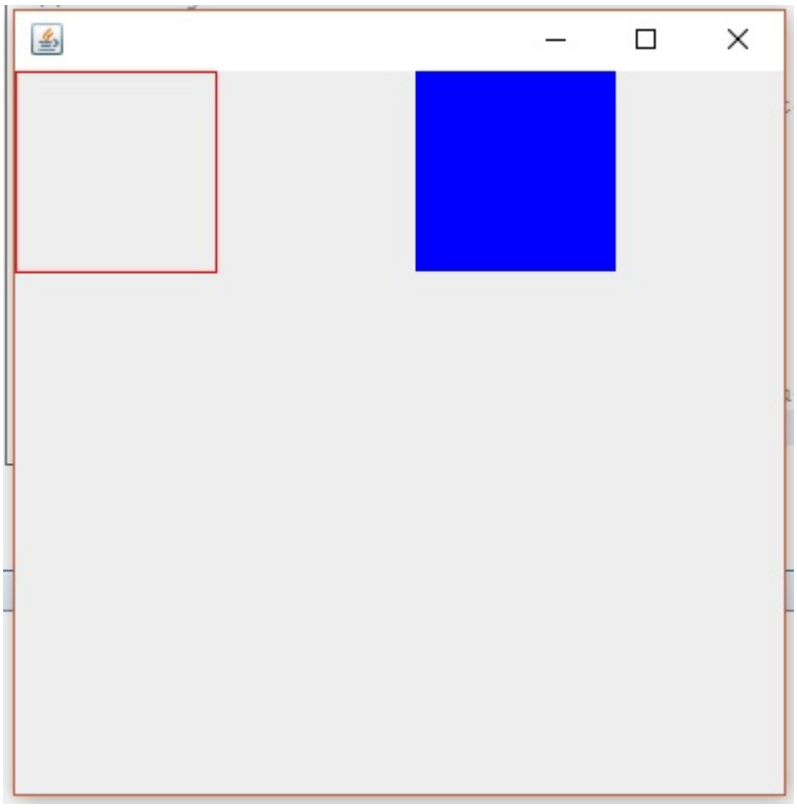
Les graphiques sont des images ou des dessins visuels sur certaines surfaces, comme un mur, une toile, un écran, un papier ou une pierre pour informer, illustrer ou divertir. Cela comprend: la représentation graphique des données, comme dans la conception et la fabrication assistées par ordinateur, la composition et les arts graphiques, ainsi que dans les logiciels éducatifs et récréatifs. Les images générées par un ordinateur sont appelées graphiques informatiques.

L'API Java 2D est puissante et complexe. Il existe plusieurs façons de faire des graphiques 2D en Java.

Examples

Exemple 1: Dessiner et remplir un rectangle à l'aide de Java

C'est un exemple qui imprime un rectangle et remplit une couleur dans le rectangle.



https://i.stack.imgur.com/dlC5v.jpg

La plupart des méthodes de la classe Graphics peuvent être divisées en deux groupes de base:

1. Méthodes de dessin et de remplissage vous permettant de rendre des formes, du texte et des images de base
2. Méthodes de paramétrage des attributs, qui affectent la manière dont le dessin et le remplissage apparaissent

Exemple de code: Commençons par un petit exemple de dessin d'un rectangle et d'une couleur de remplissage. Là, nous déclarons deux classes, une classe est MyPanel et une autre classe est Test. Dans la classe MyPanel, nous utilisons les méthodes drawRect () & fillRect () pour dessiner un rectangle et y insérer la couleur. Nous définissons la couleur par la méthode setColor (Color.blue). Dans Second Class, nous testons notre graphique, qui est la classe de test, nous créons un Frame et y ajoutons MyPanel avec p = new objet MyPanel (). En exécutant Test Class, nous voyons un rectangle et un rectangle rempli de couleur bleue.

Première 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
    }
}
```

Deuxième 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();
    }
}
```

Pour plus d'explications sur la disposition des bordures:
https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html

paintComponent ()

- C'est une méthode principale pour peindre
- Par défaut, il peint d'abord l'arrière-plan
- Après cela, il effectue une peinture personnalisée (cercle de dessin, rectangles, etc.)

Graphic2D fait référence à la classe Graphic2D

Remarque: L'API Java 2D vous permet d'effectuer facilement les tâches suivantes:

- Tracez des lignes, des rectangles et toute autre forme géométrique.
- Remplissez ces formes avec des couleurs solides ou des dégradés et des textures.
- Dessinez du texte avec des options pour un contrôle précis de la police et du processus de rendu.
- Dessinez des images, en appliquant éventuellement des opérations de filtrage.
- Appliquez des opérations telles que la composition et la transformation au cours des opérations de rendu ci-dessus.

Exemple 2: Dessin et remplissage 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);
Cette méthode dessinera un ovale à la position x et y spécifiée avec une hauteur et une largeur données.

g2.fillOval (int x, int y, int height, int width); Cette méthode remplira un ovale à la position x et y spécifiée avec une hauteur et une largeur données.

Lire Graphiques 2D en Java en ligne: https://riptutorial.com/fr/java/topic/10127/graphiques-2d-en-java

Chapitre 82: Hashtable

Introduction

Hashtable est une classe dans les collections Java qui implémente l'interface Map et étend la classe Dictionary

Ne contient que des éléments uniques et son synchronisé

Examples

Hashtable

```
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);
   }
}
```

Lire Hashtable en ligne: https://riptutorial.com/fr/java/topic/10709/hashtable

Chapitre 83: Héritage

Introduction

L'héritage est une caractéristique orientée objet de base dans laquelle une classe acquiert et s'étend sur les propriétés d'une autre classe, à l'aide du mot `extends` clé `extends` . Pour les interfaces et les `implements` mots-clés, voir interfaces .

Syntaxe

- classe ClassB étend ClassA {...}
- class ClassB implémente InterfaceA {...}
- interface InterfaceB étend InterfaceA {...}
- classe ClassB étend ClassA implémente InterfaceC, InterfaceD {...}
- classe abstraite AbstractClassB étend ClassA {...}
- classe abstraite AbstractClassB étend AbstractClassA {...}
- Classe abstraite AbstractClassB extend ClassA implémente InterfaceC, InterfaceD {...}

Remarques

L'héritage est souvent combiné avec des génériques afin que la classe de base ait un ou plusieurs paramètres de type. Voir Création d'une classe générique .

Examples

Classes abstraites

Une classe abstraite est une classe marquée avec le mot-clé `abstract` . Contrairement aux classes non abstraites, il peut contenir des méthodes abstraites sans implémentation. Il est cependant valable de créer une classe abstraite sans méthodes abstraites.

Une classe abstraite ne peut pas être instanciée. Il peut être sous-classé (étendu) tant que la sous-classe est soit abstraite, soit implémenter toutes les méthodes marquées comme abstraites par les super-classes.

Un exemple de classe abstraite:

```
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 doit être marquée abstraite lorsqu'elle a au moins une méthode abstraite. Une méthode abstraite est une méthode sans implémentation. D'autres méthodes peuvent être déclarées dans une classe abstraite ayant une implémentation afin de fournir un code commun à toutes les sous-classes.

Tenter d'instancier cette classe fournira une erreur de compilation:

```
//error: Component is abstract; cannot be instantiated
Component myComponent = new Component();
```

Cependant, une classe qui étend `Component` et fournit une implémentation pour toutes ses méthodes abstraites et peut être instanciée.

```
public class Button extends Component {

    @Override
    public void render() {
        //render a button
    }
}

public class TextBox extends Component {

    @Override
    public void render() {
        //render a textbox
    }
}
```

Les instances d'héritage de classes peuvent également être converties en tant que classe parente (héritage normal) et elles fournissent un effet polymorphe lorsque la méthode abstraite est appelée.

```
Component myButton = new Button();
Component myTextBox = new TextBox();

myButton.render(); //renders a button
myTextBox.render(); //renders a text box
```

Classes abstraites vs interfaces

Les classes abstraites et les interfaces fournissent toutes deux un moyen de définir des signatures de méthode tout en exigeant que la classe d'extension / d'implémentation fournisse l'implémentation.

Il existe deux différences clés entre les classes abstraites et les interfaces:

- Une classe ne peut étendre qu'une seule classe, mais peut implémenter de nombreuses interfaces.
- Une classe abstraite peut contenir `static` champs d'instance (non `static`), mais les

interfaces ne peuvent contenir que `static` champs `static` .

Java SE 8

Les méthodes déclarées dans les interfaces ne pouvant contenir des implémentations, des classes abstraites ont été utilisées lorsqu'il était utile de fournir des méthodes supplémentaires que les implémentations appelaient les méthodes abstraites.

Java SE 8

Java 8 permet aux interfaces de contenir des méthodes par défaut , généralement implémentées à l'aide des autres méthodes de l'interface , ce qui rend les interfaces et les classes abstraites aussi puissantes à cet égard.

Sous-classes anonymes de classes abstraites

Comme commodité, Java permet l'instanciation d'instances anonymes de sous-classes de classes abstraites, qui fournissent des implémentations pour les méthodes abstraites lors de la création du nouvel objet. En utilisant l'exemple ci-dessus, cela pourrait ressembler à ceci:

```
Component myAnonymousComponent = new Component() {
    @Override
    public void render() {
        // render a quick 1-time use component
    }
}
```

Héritage statique

La méthode statique peut être héritée de manière similaire aux méthodes normales, mais contrairement aux méthodes normales, il est impossible de créer des méthodes " abstraites " pour forcer la substitution de la méthode statique. Ecrire une méthode avec la même signature qu'une méthode statique dans une super classe semble être une forme de substitution, mais cela crée une nouvelle fonction qui cache l'autre.

```
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'exécution de l'une de ces classes produit la sortie:

```
Hello
BaseClass's num: 5
Hello
Hey
Static says Hi
StaticOverride's num: test
```

Notez que contrairement à l'héritage normal, les méthodes d'héritage statiques ne sont pas masquées. Vous pouvez toujours appeler la méthode `sayHello` base en utilisant `BaseClass.sayHello()` . Mais les classes héritent des méthodes statiques *si* aucune méthode avec la même signature n'est trouvée dans la sous-classe. Si les signatures de deux méthodes varient, les deux méthodes peuvent être exécutées à partir de la sous-classe, même si le nom est identique.

Les champs statiques se cachent de la même manière.

Utiliser 'final' pour restreindre l'héritage et remplacer

Classes finales

Lorsqu'il est utilisé dans une déclaration de `class` , le modificateur `final` empêche les autres classes d'être déclarées qui `extend` la classe. Une classe `final` est une classe "feuille" dans la hiérarchie des classes d'héritage.

```
// This declares a final class
```

```
final class MyFinalClass {
    /* some code */
}

// Compilation error: cannot inherit from final MyFinalClass
class MySubClass extends MyFinalClass {
    /* more code */
}
```

Cas d'utilisation pour les classes finales

Les classes finales peuvent être combinées à un constructeur `private` pour contrôler ou empêcher l'instanciation d'une classe. Cela peut être utilisé pour créer une soi-disant "classe utilitaire" qui définit uniquement les membres statiques; c'est-à-dire des constantes et des méthodes statiques.

```
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;
    }

}
```

Les classes immuables doivent également être déclarées comme `final` . (Une classe immuable est une classe dont les instances ne peuvent pas être modifiées après leur création; reportez-vous à la rubrique Objets indisponibles .) En faisant cela, vous ne pouvez pas créer une sous-classe mutable d'une classe immuable. Cela violerait le principe de substitution de Liskov qui exige qu'un sous-type obéisse au «contrat comportemental» de ses supertypes.

D'un point de vue pratique, déclarer une classe immuable comme étant `final` permet de raisonner plus facilement sur le comportement du programme. Il résout également les problèmes de sécurité dans le scénario où du code non fiable est exécuté dans un sandbox de sécurité. (Par exemple, puisque `String` est déclaré comme `final` , une classe de confiance n'a pas à craindre d'être amenée à accepter une sous-classe mutable, que l'appelant non fiable pourrait alors modifier subrepticement.)

Un inconvénient des classes `final` est qu'elles ne fonctionnent pas avec certains cadres moqueurs tels que Mockito. Mise à jour: Mockito version 2 supporte maintenant les moqueries des classes finales.

Méthodes finales

Le `final` modificateur peut également être appliqué aux méthodes pour éviter qu'elles soient remplacées dans les sous-classes:

```
public class MyClassWithFinalMethod {
```

```
    public final void someMethod() {
    }
}

public class MySubClass extends MyClassWithFinalMethod {

    @Override
    public void someMethod() { // Compiler error (overridden method is final)
    }
}
```

Les méthodes finales sont généralement utilisées lorsque vous souhaitez restreindre ce qu'une sous-classe peut changer dans une classe sans interdire complètement les sous-classes.

Le `final` modificateur peut également être appliqué aux variables, mais la signification de `final` pour les variables n'est pas liée à l'héritage.

Le principe de substitution de Liskov

Substituabilité est un principe dans la programmation orientée objet présenté par Barbara Liskov dans une keynote de la conférence 1987 indiquant que, si la classe `B` est une sous - classe de la classe `A` , alors où `A` est prévu, `B` peut être utilisé à la place:

```
class A {...}
class B extends A {...}

public void method(A obj) {...}

A a = new B(); // Assignment OK
method(new B()); // Passing as parameter OK
```

Cela s'applique également lorsque le type est une interface, où il n'y a pas besoin de relation hiérarchique entre les objets:

```
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
```

Maintenant, la liste contient des objets qui ne proviennent pas de la même hiérarchie de classes.

Héritage

Avec l'utilisation du mot `extends` clé extend parmi les classes, toutes les propriétés de la super-classe (également appelée *classe parent* ou *classe de base*) sont présentes dans la sous-classe (également appelée *classe enfant* ou *classe dérivée*).

```
public class BaseClass {

    public void baseMethod(){
        System.out.println("Doing base class stuff");
    }
}

public class SubClass extends BaseClass {

}
```

Les instances de `SubClass` ont *hérité de* la méthode `baseMethod()` :

```
SubClass s = new SubClass();
s.baseMethod();  //Valid, prints "Doing base class stuff"
```

Un contenu supplémentaire peut être ajouté à une sous-classe. Cela permet des fonctionnalités supplémentaires dans la sous-classe sans aucune modification de la classe de base ou de toute autre sous-classe de cette même classe de 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"
```

Les champs sont également hérités:

```
public class BaseClassWithField {

    public int x;

}

public class SubClassWithField extends BaseClassWithField {

    public SubClassWithField(int x) {
        this.x = x; //Can access fields
    }
}
```

`private` champs et méthodes `private` existent toujours dans la sous-classe, mais ne sont pas accessibles:

```
public class BaseClassWithPrivateField {

    private int x = 5;

    public int getX() {
        return x;
    }
}

public class SubClassInheritsPrivateField extends BaseClassWithPrivateField {

    public void printX() {
        System.out.println(x); //Illegal, can't access private field x
        System.out.println(getX()); //Legal, prints 5
    }
}

SubClassInheritsPrivateField s = new SubClassInheritsPrivateField();
int x = s.getX(); //x will have a value of 5.
```

En Java, chaque classe peut s'étendre au maximum sur une autre classe.

```
public class A{}
public class B{}
public class ExtendsTwoClasses extends A, B {} //Illegal
```

Ceci est connu sous le nom d'héritage multiple, et bien qu'il soit légal dans certaines langues, Java ne le permet pas avec les classes.

En conséquence, chaque classe possède une chaîne ancestrale non ramifiée de classes menant à `Object` , à partir de laquelle toutes les classes descendent.

Méthodes d'héritage et de statique

En Java, les classes parent et enfant peuvent avoir des méthodes statiques avec le même nom. Mais dans de tels cas, l'implémentation de la méthode statique dans child cache l'implémentation de la classe parente, ce n'est pas une substitution de méthode. Par exemple:

```
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");
```

```
    }
  }
}
```

Les méthodes statiques sont liées à une classe et non à une instance et cette liaison de méthode se produit au moment de la compilation. Etant donné que dans le premier appel à `staticMethod()` , référence de classe parent `p` a été utilisé, `Parent` la version de de `staticMethod()` est invoquée. En second cas, nous avons jetés `p` dans l' `Child` classe, `Child` de `staticMethod()` exécuté.

Ombrage variable

Les variables sont SHADOWED et les méthodes sont OVERRIDDEN. La variable à utiliser dépend de la classe dont la variable est déclarée. La méthode à utiliser dépend de la classe réelle de l'objet référencé par la variable.

```
class Car {
    public int gearRatio = 8;

    public String accelerate() {
        return "Accelerate : Car";
    }
}

class SportsCar extends Car {
    public int gearRatio = 9;

    public String accelerate() {
        return "Accelerate : SportsCar";
    }

    public void test() {

    }


    public static void main(String[] args) {

        Car car = new SportsCar();
        System.out.println(car.gearRatio + "  " + car.accelerate());
        // will print out 8  Accelerate : SportsCar
    }
}
```

Rétrécissement et élargissement des références d'objet

Lancer une instance d'une classe de base dans une sous-classe comme dans: `b = (B) a;` est appelé *rétrécissement* (car vous essayez de restreindre l'objet de classe de base à un objet de classe plus spécifique) et nécessite un transtypage explicite.

Convertir une instance d'une sous-classe en une classe de base comme dans: `A a = b;` est appelé l' *élargissement* et n'a pas besoin d'une conversion de type.

Pour illustrer, considérons les déclarations de classe suivantes et le code de test:

```
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 déclaration `Vehicle vehicle = new Car();` est une instruction Java valide. Chaque instance de `Car` est également un `Vehicle` . Par conséquent, l'affectation est légale sans nécessiter un transtypage explicite.

Par contre, `Car c = vehicle;` n'est pas valide. Le type statique de la variable `vehicle` est `Vehicle` ce qui signifie qu'il peut se référer à une instance de `Car` , Camion , MotorCycle , `or any other current or future subclass of` Véhicule . `(Or indeed, an instance of` Vehicle `itself, since we did not declare it as an class.) The assignment cannot be allowed, since that might lead to` abstraite `class.) The assignment cannot be allowed, since that might lead to` voiture à se `referring to a` instance de Truck`.

Pour éviter cette situation, nous devons ajouter un type-cast explicite:

```
Car c = (Car) vehicle;
```

Le type-cast indique au compilateur que nous nous *attendons* à la valeur du `vehicle` soit une `Car` ou une sous - classe de `Car` . Si nécessaire, le compilateur insérera du code pour effectuer une vérification du type à l'exécution. Si la vérification échoue, une `ClassCastException` sera lancée lorsque le code est exécuté.

Notez que tous les types ne sont pas valides. Par exemple:

```
String s = (String) vehicle;  // not valid
```

Le compilateur Java sait que une instance qui est de type compatible avec `Vehicle` *ne peut jamais être* de type compatible avec `String` . Le cast de type n'a jamais pu réussir, et le JLS impose une erreur de compilation.

Programmation à une interface

L'idée derrière la programmation d'une interface est de baser le code principalement sur des interfaces et d'utiliser uniquement des classes concrètes au moment de l'instanciation. Dans ce contexte, un bon code traitant par exemple des collections Java ressemblera à ceci (non pas que la méthode elle-même soit d'aucune utilité, mais seulement une illustration):

```
public <T> Set<T> toSet(Collection<T> collection) {
  return Sets.newHashSet(collection);
}
```

alors que le code incorrect peut ressembler à ceci:

```
public <T> HashSet<T> toSet(ArrayList<T> collection) {
  return Sets.newHashSet(collection);
}
```

Non seulement le premier peut être appliqué à un plus grand choix d'arguments, mais ses résultats seront plus compatibles avec le code fourni par d'autres développeurs qui adhèrent généralement au concept de programmation à une interface. Cependant, les raisons les plus importantes pour utiliser le premier sont:

- la plupart du temps, le contexte dans lequel le résultat est utilisé ne nécessite pas et ne devrait pas nécessiter autant de détails que la mise en œuvre concrète le prévoit;
- adhérer à une interface force le code plus propre et moins de piratage, mais une autre méthode publique est ajoutée à une classe desservant un scénario spécifique;
- le code est plus testable car les interfaces sont facilement modifiables;
- enfin, le concept aide même si une seule implémentation est attendue (au moins pour la testabilité).

Alors, comment peut-on facilement appliquer le concept de programmation à une interface lors de l'écriture d'un nouveau code en ayant à l'esprit une implémentation particulière? Une option couramment utilisée est la combinaison des modèles suivants:

- programmation à une interface
- usine
- constructeur

L'exemple suivant basé sur ces principes est une version simplifiée et tronquée d'une implémentation RPC écrite pour différents protocoles:

```
public interface RemoteInvoker {
  <RQ, RS> CompletableFuture<RS> invoke(RQ request, Class<RS> responseClass);
}
```

L'interface ci-dessus n'est pas supposée être instanciée directement via une fabrique, au lieu de cela nous dérivons d'autres interfaces plus concrètes, une pour l'invocation HTTP et une pour AMQP, chacune ayant une fabrique et un générateur pour construire des instances. l'interface ci-dessus:

```
public interface AmqpInvoker extends RemoteInvoker {
  static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
    return new AmqpInvokerBuilder(instanceId, factory);
  }
}
```

Les instances de `RemoteInvoker` pour l'utilisation avec AMQP peuvent maintenant être construites aussi facilement (ou plus impliquées selon le constructeur):

```
RemoteInvoker invoker = AmqpInvoker.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

Et l'invocation d'une demande est aussi simple que:

```
Response res = invoker.invoke(new Request(data), Response.class).get();
```

Java 8 permettant de placer des méthodes statiques directement dans les interfaces, la fabrique intermédiaire est devenue implicite dans le code ci-dessus remplacé par `AmqpInvoker.with()` . En Java avant la version 8, le même effet peut être obtenu avec une classe `Factory` interne:

```
public interface AmqpInvoker extends RemoteInvoker {
  class Factory {
    public static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
      return new AmqpInvokerBuilder(instanceId, factory);
    }
  }
}
```

L'instanciation correspondante deviendrait alors:

```
RemoteInvoker invoker = AmqpInvoker.Factory.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

Le générateur utilisé ci-dessus pourrait ressembler à ceci (bien qu'il s'agisse d'une simplification car celle-ci permet de définir jusqu'à 15 paramètres différents des valeurs par défaut). Notez que la construction n'est pas publique, elle n'est donc utilisable que depuis l'interface `AmqpInvoker` ci-dessus:

```
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);
  }
}
```

Généralement, un générateur peut également être généré à l'aide d'un outil tel que FreeBuilder.

Enfin, l'implémentation standard (et la seule attendue) de cette interface est définie comme une classe locale de package pour appliquer l'utilisation de l'interface, de la fabrique et du générateur:

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

Pendant ce temps, ce modèle s'est avéré très efficace pour développer tout notre nouveau code, quelle que soit la simplicité ou la complexité de la fonctionnalité.

Classe abstraite et utilisation de l'interface: relation "Is-a" vs capacité "Has-a"

Quand utiliser des classes abstraites: Pour implémenter le même comportement ou un comportement différent parmi plusieurs objets associés

Quand utiliser les interfaces: pour implémenter un contrat par plusieurs objets non liés

Les classes abstraites create "is a" relations alors que les interfaces fournissent "a une" capacité.

Cela peut être vu dans le code ci-dessous:

```
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;
    }
}
```

sortie:

```
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
```

Notes clés:

1. `Animal` est une classe abstraite avec des attributs partagés: `name` et `lifeExpectancy` et des méthodes abstraites: `remember()` et `protectOwner()` . `Dog` and `Cat` sont des `Animals` qui ont implémenté les méthodes `remember()` et `protectOwner()` .

2. `Cat` peut `climb()` mais `Dog` ne peut pas. `Dog` peut `think()` mais `Cat` ne peut pas. Ces fonctionnalités spécifiques sont ajoutées à `Cat` et `Dog` par implémentation.

3. `Man` n'est pas un `Animal` mais il peut `Think` , `Learn` , `Apply` et `Climb` .
4. `Cat` n'est pas un `Man` mais il peut `Climb` .
5. `Dog` n'est pas un `Man` mais il peut `Learn`
6. `Man` n'est ni un `Cat` ni un `Dog` mais peut avoir certaines des capacités de ces deux derniers sans étendre `Animal` , `Cat` ou `Dog` . Ceci est fait avec des interfaces.
7. Même si `Animal` est une classe abstraite, il a un constructeur, contrairement à une interface.

TL; DR:

Les classes non liées peuvent avoir des capacités via des interfaces, mais les classes associées modifient le comportement via l'extension des classes de base.

Reportez-vous à la page de documentation Java pour savoir laquelle utiliser dans un cas d'utilisation spécifique.

Envisagez d'utiliser des classes abstraites si ...

1. Vous voulez partager du code entre plusieurs classes étroitement liées.
2. Vous vous attendez à ce que les classes qui étendent votre classe abstraite disposent de nombreuses méthodes ou champs communs, ou requièrent des modificateurs d'accès autres que publics (tels que protégés et privés).
3. Vous souhaitez déclarer des champs non statiques ou non finaux.

Envisagez d'utiliser des interfaces si ...

1. Vous vous attendez à ce que des classes non liées implémentent votre interface. Par exemple, de nombreux objets non liés peuvent implémenter l'interface `Serializable` .
2. Vous voulez spécifier le comportement d'un type de données particulier mais ne vous préoccupez pas de savoir qui implémente son comportement.
3. Vous voulez profiter de l'héritage multiple de type.

Passer outre dans l'héritage

La substitution dans Héritage est utilisée lorsque vous utilisez une méthode déjà définie à partir d'une super-classe dans une sous-classe, mais d'une manière différente de la façon dont la méthode a été conçue à l'origine dans la super-classe. La dérogation permet à l'utilisateur de réutiliser le code en utilisant le matériel existant et en le modifiant pour mieux répondre aux besoins de l'utilisateur.

L'exemple suivant montre comment `ClassB` remplace les fonctionnalités de `ClassA` en modifiant ce qui est envoyé via la méthode d'impression:

Exemple:

```
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");
    }
}
```

Sortie:

UNE

B

Lire Héritage en ligne: https://riptutorial.com/fr/java/topic/87/heritage

Chapitre 84: Heure locale

Syntaxe

- LocalTime time = LocalTime.now (); // Initialise avec l'horloge système actuelle
- LocalTime time = LocalTime.MIDNIGHT; // 00:00
- LocalTime time = LocalTime.NOON; // 12:00
- LocalTime time = LocalTime.of (12, 12, 45); // 12:12:45

Paramètres

Méthode	Sortie
`LocalTime.of(13, 12, 11)`	13:12:11
`LocalTime.MIDNIGHT`	00:00
`LocalTime.NOON`	12:00
`LocalTime.now()`	Heure actuelle de l'horloge système
`LocalTime.MAX`	Heure locale maximale prise en charge 23: 59: 59.999999999
`LocalTime.MIN`	Heure locale minimale prise en charge 00:00
`LocalTime.ofSecondOfDay(84399)`	23:59:59, obtient le temps de la valeur du deuxième jour
`LocalTime.ofNanoOfDay(2000000000)`	00:00:02, Obtient le temps de la valeur de nanos de jour

Remarques

Comme le nom de classe indique, `LocalTime` représente une heure sans fuseau horaire. Cela ne représente pas une date. C'est une étiquette simple pour un temps donné.

La classe est basée sur les valeurs et la méthode `equals` doit être utilisée lors des comparaisons.

Cette classe provient du package java.time.

Examples

Modification du temps

Vous pouvez ajouter des heures, des minutes, des secondes et des nanosecondes:

```
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)
```

Les fuseaux horaires et leur décalage horaire

```
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");
    }
}
```

Durée entre deux heures locales

Il y a deux façons équivalentes de calculer la quantité d'unité de temps entre deux `LocalTime` : (1) `until(Temporal, TemporalUnit)` méthode `until(Temporal, TemporalUnit)` unité `until(Temporal, TemporalUnit)` et (2) unité `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);
    }
}
```

Introduction

LocalTime est une classe immuable et sécurisée pour les threads, utilisée pour représenter l'heure, souvent vue sous forme d'heure-seconde. Le temps est représenté avec une précision à la nanoseconde. Par exemple, la valeur "13: 45.30.123456789" peut être stockée dans un LocalTime.

Cette classe ne stocke ni ne représente une date ou un fuseau horaire. Au lieu de cela, il s'agit d'une description de l'heure locale telle qu'elle apparaît sur une horloge murale. Il ne peut pas représenter un instant sur la ligne de temps sans informations supplémentaires telles qu'un décalage ou un fuseau horaire. Ceci est une classe basée sur la valeur, la méthode égale doit être utilisée pour les comparaisons.

Des champs

MAX - Le maximum de LocalTime pris en charge, '23: 59: 59.999999999 '. MINUIT, MIN, MIDI

Méthodes statiques importantes

now (), maintenant (horloge), maintenant (zone ZoneId), analyse (texte CharSequence)

Méthodes d'instance importantes

isAfter (LocalTime autre), isBefore (LocalTime autre), moins (TemporalAmount amountToSubtract), moins (long amountToSubtract, unité TemporalUnit), plus (TemporalAmount amountToAdd), plus (montant longAjout, unité TemporalUnit)

```
ZoneId zone = ZoneId.of("Asia/Kolkata");
LocalTime now = LocalTime.now();
LocalTime now1 = LocalTime.now(zone);
LocalTime then = LocalTime.parse("04:16:40");
```

La différence de temps peut être calculée de l'une des manières suivantes

```
long timeDiff = Duration.between(now, now1).toMinutes();
long timeDiff1 = java.time.temporal.ChronoUnit.MINUTES.between(now2, now1);
```

Vous pouvez également ajouter / soustraire des heures, des minutes ou des secondes de tout objet de LocalTime.

minusHours (long hoursToSubtract), minusMinutes (long hoursToSubtract), minusNanos (long hoursToSubtract), plusHours (long hoursToSubtract), plusMinutes (long hoursToMinutes), plusNanos (long nanosToSubtract), plusSeconds (long secondsToSubtract)

```
now.plusHours(1L);
now1.minusMinutes(20L);
```

Lire Heure locale en ligne: https://riptutorial.com/fr/java/topic/3065/heure-locale

Chapitre 85: HttpURLConnection

Remarques

- L'utilisation de HttpUrlConnection sur Android nécessite que vous ajoutiez l'autorisation Internet à votre application (dans le `AndroidManifest.xml`).
- Il existe également d'autres bibliothèques et clients Java HTTP, tels que OkHttp de Square, qui sont plus faciles à utiliser et peuvent offrir de meilleures performances ou davantage de fonctionnalités.

Examples

Obtenir le corps de la réponse à partir d'une URL en tant que chaîne

```
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();
}
```

Cela téléchargera les données textuelles de l'URL spécifiée et les renverra sous forme de chaîne.

Comment ça marche:

- Tout d'abord, nous créons un `HttpUrlConnection` partir de notre URL, avec une `new URL(url).openConnection()` . Nous `UrlConnection` le `UrlConnection` retourne en `HttpUrlConnection` , nous avons donc accès à des choses comme l'ajout d'en-têtes (tels que User Agent) ou la vérification du code de réponse. (Cet exemple ne le fait pas, mais c'est facile à ajouter.)

- Ensuite, créez `InputStream` en `InputStream` basant sur le code de réponse (pour la gestion des erreurs)
- Ensuite, créez un `BufferedReader` qui nous permet de lire le texte de `InputStream` nous recevons de la connexion.
- Maintenant, nous ajoutons le texte à `StringBuilder` , ligne par ligne.
- Fermez le `InputStream` et retournez la chaîne que nous avons maintenant.

Remarques:

- Cette méthode lancera une `IoException` en cas d'échec (telle qu'une erreur réseau ou aucune connexion Internet) et `IoException` une `IoException` `MalformedUrlException` *non* `MalformedUrlException` si l'URL indiquée n'est pas valide.
- Il peut être utilisé pour lire n'importe quelle URL qui renvoie du texte, comme les pages Web (HTML), les API REST qui renvoient JSON ou XML, etc.
- Voir aussi: Lire l'URL de la chaîne dans quelques lignes de code Java .

Usage:

Est très simple:

```
String text = getText("http://example.com");
//Do something with the text from example.com, in this case the HTML.
```

Données 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();
    }
}
```

Cela affichera les données dans l'URL spécifiée, puis lira la réponse ligne par ligne.

Comment ça marche

- Comme d'habitude, nous obtenons la `HttpURLConnection` partir d'une `URL` .
- Définissez le type de contenu à l'aide de `setRequestProperty` , par défaut `application/x-www-form-urlencoded`
- `setDoOutput(true)` indique à la connexion que nous allons envoyer des données.
- Ensuite, nous obtenons le `OutputStream` en appelant `getOutputStream()` et en y écrivant des données. N'oubliez pas de le fermer après avoir terminé.
- Nous lisons enfin la réponse du serveur.

Supprimer une ressource

```
   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();
        }
    }
```

Cela supprimera la ressource dans l'URL spécifiée, puis imprimera l'en-tête de réponse.

Comment ça marche

- nous obtenons le `HttpURLConnection` partir d'une `URL` .
- Définissez le type de contenu à l'aide de `setRequestProperty` , par défaut `application/x-www-form-urlencoded`
- `setDoInput(true)` indique à la connexion que nous avons l'intention d'utiliser la connexion URL pour la saisie.
- `setRequestMethod("DELETE")` pour exécuter HTTP DELETE

Enfin, nous imprimons l'en-tête de réponse du serveur.

Vérifier si la ressource existe

```
/**
 * Checks if a resource exists by sending a HEAD-Request.
 * @param url The url of a resource which has to be checked.
 * @return true if the response code is 200 OK.
 */
public static final boolean checkIfResourceExists(URL url) throws IOException {
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("HEAD");
    int code = conn.getResponseCode();
    conn.disconnect();
    return code == 200;
}
```

Explication:

Si vous vérifiez simplement si une ressource existe, il vaut mieux utiliser une requête HEAD qu'une requête GET. Cela évite le transfert de la ressource.

Notez que la méthode ne renvoie que `true` si le code de réponse est `200` . Si vous prévoyez des réponses de redirection (c.-à-d. 3XX), vous devrez peut-être améliorer la méthode pour les respecter.

Exemple:

```
checkIfResourceExists(new URL("http://images.google.com/")); // true
checkIfResourceExists(new URL("http://pictures.google.com/")); // false
```

Lire HttpURLConnection en ligne: https://riptutorial.com/fr/java/topic/156/httpurlconnection

Chapitre 86: Implémentations du système de plug-in Java

Remarques

Si vous utilisez un IDE et / ou un système de construction, il est beaucoup plus facile de configurer ce type de projet. Vous créez un module d'application principal, puis un module d'API, puis créez un module de plug-in et faites-le dépendre du module de l'API ou des deux. Ensuite, vous configurez l'emplacement des artefacts du projet - dans notre cas, les plug-ins compilés peuvent être envoyés directement dans le répertoire "plugins", évitant ainsi tout mouvement manuel.

Examples

Utiliser URLClassLoader

Il existe plusieurs manières d'implémenter un système de plug-in pour une application Java. L'un des plus simples est d'utiliser *URLClassLoader* . L'exemple suivant impliquera un peu de code JavaFX.

Supposons que nous ayons un module d'une application principale. Ce module est supposé charger des plugins sous forme de Jars depuis le dossier 'plugins'. Code initial:

```
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);
    }
}
```

Ensuite, nous créons une interface qui représentera un plug-in.

```
package main;

public interface Plugin
```

```
{
    default void initialize()
    {
        System.out.println("Initialized "+this.getClass().getName());
    }
    default String name(){return getClass().getSimpleName();}
}
```

Nous voulons charger les classes qui implémentent cette interface, donc nous devons d'abord filtrer les fichiers qui ont une extension '.jar':

```
File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
```

S'il y a des fichiers, nous devons créer des collections d'URL et de noms de classe:

```
    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);
        }

    }
```

Ajoutons un HashSet statique à *MainApplication* qui contiendra les plugins chargés:

```
static HashSet<Plugin> plugins=new HashSet<>();
```

Ensuite, nous instancions un *URLClassLoader* et *effectuons* une itération sur les noms de classes, en instanciant des classes qui implémentent une interface *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();}
});
```

Ensuite, nous pouvons appeler les méthodes du plugin, par exemple, pour les initialiser:

```
if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded plugins:"));
    plugins.forEach(plugin -> {
        plugin.initialize();
        loadedPlugins.getChildren().add(new Label(plugin.name()));
});
```

Le code final de *MainApplication* :

```
package main;
public class MainApplication extends Application
{
    static HashSet<Plugin> plugins=new HashSet<>();
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins");
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
        File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
        VBox loadedPlugins=new VBox(6);
        loadedPlugins.setAlignment(Pos.CENTER);
        if(files!=null && files.length>0)
        {
            ArrayList<String> classes=new ArrayList<>();
            ArrayList<URL> urls=new ArrayList<>(files.length);
            for(File file:files)
            {
                JarFile jar=new JarFile(file);
                jar.stream().forEach(jarEntry -> {
                    if(jarEntry.getName().endsWith(".class"))
                    {
                        classes.add(jarEntry.getName());
                    }
                });
                URL url=file.toURI().toURL();
                urls.add(url);
            }
            URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new
URL[urls.size()]));
            classes.forEach(className->{
                try
                {
                    Class
cls=urlClassLoader.loadClass(className.replaceAll("/",".").replace(".class",""));
                    Class[] interfaces=cls.getInterfaces();
                    for(Class intface:interfaces)
                    {
                        if(intface.equals(Plugin.class))
                        {
                            Plugin plugin=(Plugin) cls.newInstance();
                            plugins.add(plugin);
```

```
                            break;
                        }
                    }
                }
                catch (Exception e){e.printStackTrace();}
            });
            if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded
plugins:"));
            plugins.forEach(plugin -> {
                plugin.initialize();
                loadedPlugins.getChildren().add(new Label(plugin.name()));
            });
        }
        Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
        Scene scene=new
Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] a)
    {
            launch(a);
    }
}
```

Créons deux plugins. De toute évidence, la source du plugin devrait être dans un module séparé.

```
package plugins;

import main.Plugin;

public class FirstPlugin implements Plugin
{
    //this plugin has default behaviour
}
```

Deuxième 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"));
    }
}
```

Ces plugins doivent être intégrés dans des fichiers Jars standard. Ce processus dépend de votre IDE ou d’autres outils.

Lorsque les Jars seront placés directement dans les «plugins», *MainApplication* les détectera et instanciera les classes appropriées.

Lire Implémentations du système de plug-in Java en ligne:
https://riptutorial.com/fr/java/topic/7160/implementations-du-systeme-de-plug-in-java

Chapitre 87: InputStreams et OutputStreams

Syntaxe

- int read (octet [] b) lève une exception IOException

Remarques

Notez que la plupart du temps, vous n'utilisez PAS directement `InputStream` mais utilisez `BufferedStream` , ou similaire. En effet, `InputStream` lit à partir de la source chaque fois que la méthode de lecture est appelée. Cela peut entraîner une utilisation importante du processeur dans les changements de contexte du noyau.

Examples

Lecture de InputStream dans une chaîne

Parfois, vous souhaiterez peut-être lire des octets dans une chaîne. Pour ce faire , vous aurez besoin de trouver quelque chose qui convertit entre `byte` et UTF-16 Codepoints « Java natif » utilisé comme `char` . Cela se fait avec un `InputStreamReader` .

Pour accélérer le processus un peu, il est "habituel" d'allouer un tampon, afin de ne pas avoir trop de temps lors de la lecture de 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();
}
```

Transformer cet exemple en code Java SE 6 (et inférieur) est omis comme exercice pour le lecteur.

Écrire des octets dans un OutputStream

Écrire des octets dans un `OutputStream` un octet à la fois

```
OutputStream stream = object.getOutputStream();

byte b = 0x00;
stream.write( b );
```

Ecrire un tableau d'octets

```
byte[] bytes = new byte[] { 0x00, 0x00 };

stream.write( bytes );
```

Écrire une section d'un tableau d'octets

```
int offset = 1;
int length = 2;
byte[] bytes = new byte[] { 0xFF, 0x00, 0x00, 0xFF };

stream.write( bytes, offset, length );
```

Fermeture des cours d'eau

La plupart des flux doivent être fermés lorsque vous en avez terminé, sinon vous pourriez introduire une fuite de mémoire ou laisser un fichier ouvert. Il est important que les flux soient fermés même si une exception est levée.

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
}
```

N'oubliez pas: try-with-resources garantit que les ressources ont été fermées à la sortie du bloc, que cela se produise avec le flux de contrôle habituel ou à cause d'une exception.

Java SE 6

Parfois, try-with-resources n'est pas une option, ou vous supportez peut-être une version antérieure de Java 6 ou antérieure. Dans ce cas, la manipulation est d'utiliser un `finally` bloc:

```
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...
    }
}
```

Notez que la fermeture d'un flux d'encapsuleur fermera également son flux sous-jacent. Cela signifie que vous ne pouvez pas envelopper un flux, fermer le wrapper, puis continuer à utiliser le flux d'origine.

Copier le flux d'entrée vers le flux de sortie

Cette fonction copie les données entre deux flux -

```
void copy(InputStream in, OutputStream out) throws IOException {
    byte[] buffer = new byte[8192];
    while ((bytesRead = in.read(buffer)) > 0) {
        out.write(buffer, 0, bytesRead);
    }
}
```

Exemple -

```
 // reading from System.in and writing to System.out
copy(System.in, System.out);
```

Emballage des flux d'entrée / sortie

`OutputStream` et `InputStream` ont de nombreuses classes différentes, chacune avec une fonctionnalité unique. En encapsulant un flux autour d'un autre, vous obtenez la fonctionnalité des deux flux.

Vous pouvez envelopper un flux autant de fois que vous le souhaitez, prenez simplement en note la commande.

Combinaisons utiles

Écrire des caractères dans un fichier en utilisant un tampon

```
File myFile = new File("targetFile.txt");
PrintWriter writer = new PrintWriter(new BufferedOutputStream(new FileOutputStream(myFile)));
```

Compression et cryptage des données avant l'écriture dans un fichier lors de l'utilisation d'un tampon

```
Cipher cipher = ... // Initialize cipher
File myFile = new File("targetFile.enc");
BufferedOutputStream outputStream = new BufferedOutputStream(new DeflaterOutputStream(new
CipherOutputStream(new FileOutputStream(myFile), cipher)));
```

Liste des wrappers de flux d'entrée / sortie

Emballage	La description
BufferedOutputStream / BufferedInputStream	Alors que `OutputStream` écrit des données un octet à la fois, `BufferedOutputStream` écrit les données en blocs. Cela réduit le nombre d'appels système, améliorant ainsi les performances.
DeflaterOutputStream / DeflaterInputStream	Effectue la compression des données.
InflaterOutputStream / InflaterInputStream	Effectue la décompression des données.
CipherOutputStream / CipherInputStream	Crypte / Décrypte les données.
DigestOutputStream / DigestInputStream	Génère Message Digest pour vérifier l'intégrité des données.
CheckedOutputStream / CheckedInputStream	Génère un CheckSum. CheckSum est une version plus triviale de Message Digest.
DataOutputStream / DataInputStream	Permet d'écrire des types de données primitifs et des chaînes. Conçu pour écrire des octets. Plate-forme indépendante.
PrintStream	Permet d'écrire des types de données primitifs et des chaînes. Conçu pour écrire des octets. Plate-forme dépendante.
OutputStreamWriter	Convertit un OutputStream en un écrivain. Un OutputStream traite les octets pendant que Writers traite les caractères
PrintWriter	Appelle automatiquement OutputStreamWriter. Permet d'écrire des types de données primitifs et des chaînes. Strictement pour écrire des caractères et mieux pour écrire des caractères

Exemple 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+"-");
    }
  }
}
```

Lire InputStreams et OutputStreams en ligne: https://riptutorial.com/fr/java/topic/110/inputstreams-et-outputstreams

Chapitre 88: Installation de Java (Standard Edition)

Introduction

Cette page de documentation donne accès aux instructions d'installation de `java standard edition` sur les `Windows` , `Linux` et `macOS` .

Examples

Définition de% PATH% et% JAVA_HOME% après l'installation sous Windows

Hypothèses:

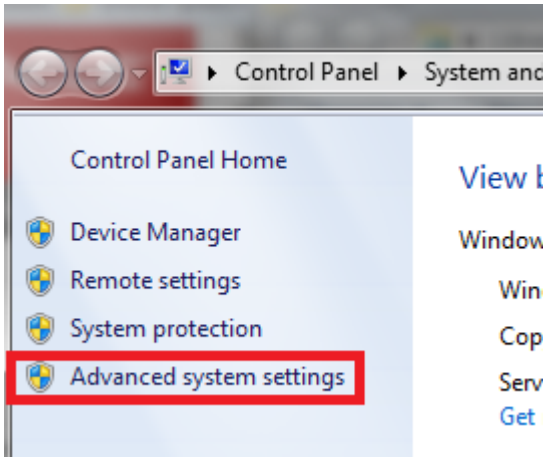
- Un JDK Oracle a été installé.
- Le JDK a été installé dans le répertoire par défaut.

Étapes de configuration

1. Ouvrez l'Explorateur Windows.

2. Dans le volet de navigation sur la gauche, cliquez avec le bouton droit sur *Ce PC* (ou *Ordinateur* pour les anciennes versions de Windows). Il y a un moyen plus court sans utiliser l'explorateur dans les versions Windows actuelles: appuyez simplement sur `Win` + `Pause`

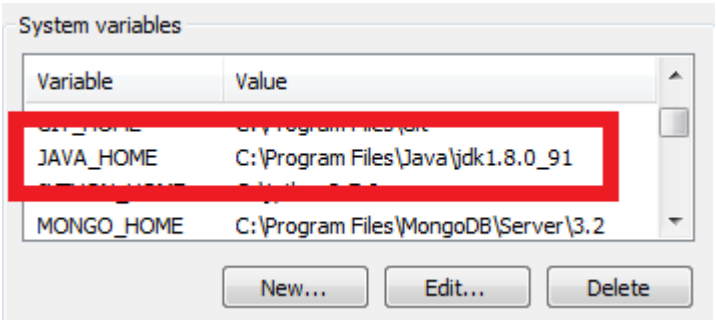
3. Dans la nouvelle fenêtre du Panneau de configuration, cliquez *sur Paramètres système avancés,* qui doivent se trouver dans le coin supérieur gauche. Cela ouvrira la fenêtre *Propriétés* du *système* .

   Vous pouvez également taper `SystemPropertiesAdvanced` (insensible à la casse) dans *Run* (`Win` + `R`) et `SystemPropertiesAdvanced` sur `Entrée` .

4. Dans l'onglet *Avancé* des *propriétés système,* sélectionnez le bouton `Variables d'environnement ...` dans le coin inférieur droit de la fenêtre.

5. Ajoutez une **nouvelle variable système** en cliquant sur le bouton `Nouveau ...` dans *Variables système* portant le nom `JAVA_HOME` et dont la valeur correspond au chemin du répertoire dans lequel le JDK a été installé. Après avoir entré ces valeurs, appuyez sur `OK` .



6. Faites défiler la liste des *variables système* et sélectionnez la variable `Path` .

7. **ATTENTION:** Windows s'appuie sur `Path` pour rechercher des programmes importants. Si tout ou partie est supprimé, Windows peut ne pas fonctionner correctement. Il doit être modifié pour permettre à Windows d'exécuter le JDK. Dans cette optique, cliquez sur le bouton "Modifier ..." avec la variable `Path` sélectionnée. Ajoutez `%JAVA_HOME%\bin;` au début de la variable `Path` .

Il est préférable d'ajouter au début de la ligne, car le logiciel Oracle utilisé pour enregistrer leur propre version de Java dans `Path` - Cela entraînera votre version à ignorer si elle se produit après la déclaration d'Oracle.

Vérifie ton travail

1. Ouvrez l'invite de commande en cliquant sur Démarrer, puis en tapant `cmd` et en appuyant sur `Enter` .
2. Entrez `javac -version` dans l'invite. Si cela a réussi, la version du JDK sera imprimée à l'écran.

Remarque: Si vous devez réessayer, fermez l'invite avant de vérifier votre travail. Cela forcera Windows à obtenir la nouvelle version de `Path` .

Sélection d'une version Java SE appropriée

Il y a eu de nombreuses versions de Java depuis la version d'origine de Java 1.0 en 1995. (Reportez-vous à l'historique des versions de Java pour obtenir un résumé.) Cependant, la plupart des versions ont dépassé leurs dates de fin de vie officielles. Cela signifie que le fournisseur (généralement Oracle maintenant) a cessé de développer la version et ne fournit plus de correctifs publics / gratuits pour les bogues ou les problèmes de sécurité. (Les correctifs privés sont généralement disponibles pour les personnes / organisations ayant un contrat de support,

contactez le bureau de vente de votre fournisseur.)

En règle générale, la version Java SE recommandée sera la dernière mise à jour de la dernière version publique. Actuellement, cela signifie la dernière version disponible de Java 8. Java 9 devrait être publié en 2017. (Java 7 a dépassé sa fin de vie et la dernière version publique date d'avril 2015. Java 7 et les versions antérieures ne sont pas recommandées.)

Cette recommandation s'applique à tous les nouveaux développements Java et à tous ceux qui apprennent Java. Cela s'applique également aux personnes qui souhaitent simplement exécuter un logiciel Java fourni par un tiers. En règle générale, le code Java bien écrit fonctionnera sur une nouvelle version de Java. (Mais vérifiez les notes de version du logiciel et contactez l'auteur / fournisseur / fournisseur si vous avez des doutes.)

Si vous travaillez sur une base de code Java plus ancienne, nous vous conseillons de vous assurer que votre code s'exécute sur la dernière version de Java. Il est plus difficile de décider quand commencer à utiliser les fonctionnalités des nouvelles versions de Java, car cela affectera votre capacité à prendre en charge les clients qui ne peuvent pas ou ne veulent pas leur installation Java.

Nom de la version Java et de la version

La publication de Java est un peu déroutante. Il existe en fait deux systèmes de dénomination et de numérotation, comme indiqué dans ce tableau:

Version JDK	Nom 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 rebaptisé 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 (pas encore publié)

1 - Il semble qu'Oracle a l'intention de rompre avec sa pratique antérieure consistant à utiliser un schéma de "numéro de version sémantique" dans les chaînes de la version Java. Reste à savoir s'ils vont y arriver.

Le "SE" dans les noms commerciaux fait référence à Standard Edition. Ceci est la version de base pour exécuter Java sur la plupart des ordinateurs portables, des PC et des serveurs (sauf

Android).

Il existe deux autres éditions officielles de Java: "Java ME" est la Micro Edition et "Java EE" est la version Enterprise. La saveur Android de Java est également très différente de Java SE. Java ME, Java EE et Android Java n'entrent pas dans le cadre de cette rubrique.

Le numéro de version complet d'une version Java ressemble à ceci:

```
1.8.0_101-b13
```

Ceci dit JDK 1.8.0, Update 101, Build # 13. Oracle fait référence à cela dans les notes de publication en tant que:

```
Java™ SE Development Kit 8, Update 101 (JDK 8u101)
```

Le numéro de mise à jour est important - Oracle publie régulièrement des mises à jour pour une version majeure avec des correctifs de sécurité, des correctifs de bogues et (dans certains cas) de nouvelles fonctionnalités. Le numéro de build est généralement sans importance. Notez que Java 8 et Java 1.8 se *réfèrent à la même chose* ; Java 8 n'est que le nom "marketing" de Java 1.8.

De quoi ai-je besoin pour le développement Java

Une installation JDK et un éditeur de texte constituent le strict minimum pour le développement Java. (Il est intéressant d’avoir un éditeur de texte capable de mettre en évidence la syntaxe Java, mais vous pouvez vous en passer.)

Cependant, pour un travail de développement sérieux, il est recommandé d’utiliser également les éléments suivants:

- Un IDE Java tel que Eclipse, Intellij IDEA ou NetBeans
- Un outil de compilation Java tel que Ant, Gradle ou Maven
- Un système de contrôle de version pour la gestion de votre base de code (avec des sauvegardes appropriées et une réplication hors site)
- Outils de test et outils CI (intégration continue)

Installation d'un Java JDK sous Linux

Utilisation du gestionnaire de packages

Les versions JDK et / ou JRE pour OpenJDK ou Oracle peuvent être installées à l'aide du gestionnaire de paquets sur la plupart des distributions Linux classiques. (Les choix disponibles dépendent de la distribution.)

En règle générale, la procédure consiste à ouvrir la fenêtre du terminal et à exécuter les commandes indiquées ci-dessous. (Il est supposé que vous avez un accès suffisant pour exécuter les commandes en tant qu'utilisateur "root" ... ce que fait la commande `sudo` . Si vous ne le faites pas, veuillez alors contacter les administrateurs de votre système.)

L'utilisation du gestionnaire de paquets est recommandée car elle facilite (généralement) la mise à jour de votre installation Java.

`apt-get` , distributions Linux basées sur Debian (Ubuntu, etc.)

Les instructions suivantes installeront Oracle Java 8:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Remarque: Pour configurer automatiquement les variables d'environnement Java 8, vous pouvez installer le package suivant:

```
$ sudo apt-get install oracle-java8-set-default
```

Créer un fichier `.deb`

Si vous préférez créer le fichier `.deb` vous-même à partir du fichier `.tar.gz` téléchargé depuis Oracle, procédez comme suit (en supposant que vous avez téléchargé le `./<jdk>.tar.gz .tar.gz` dans `./<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
```

Remarque : Cela suppose que l'entrée soit fournie sous la forme d'un fichier ".tar.gz".

`slackpkg` , distributions Linux basées sur Slackware

```
sudo slapt-get install default-jdk
```

`yum` , RedHat, CentOS, etc.

```
sudo yum install java-1.8.0-openjdk-devel.x86_64
```

`dnf` , Fedora

Sur les dernières éditions de Fedora, `yum` a remplacé `dnf` .

```
sudo dnf install java-1.8.0-openjdk-devel.x86_64
```

Dans les versions récentes de Fedora, il n'y a pas de paquet pour installer Java 7 et les versions antérieures.

`pacman` , distributions Linux basées sur Arch

```
sudo pacman -S jdk8-openjdk
```

L'utilisation de `sudo` n'est pas requise si vous utilisez l'utilisateur root.

Gentoo Linux

Le guide Java Gentoo est géré par l'équipe Java de Gentoo et conserve une page de wiki mise à jour contenant les packages de portage et les indicateurs USE nécessaires.

Installation des JDK Oracle sur Redhat, CentOS, Fedora

Installation de JDK à partir d'un fichier Oracle JDK ou JRE `tar.gz`

1. Téléchargez le fichier d'archive Oracle ("tar.gz") approprié pour la version souhaitée à partir du site de téléchargement Oracle Java .
2. Changer de répertoire à l'endroit où vous voulez placer l'installation;
3. Décompresser le fichier d'archive; par exemple

```
tar xzvf jdk-8u67-linux-x64.tar.gz
```

Installation à partir d'un fichier Oracle Java RPM.

1. Récupérez le fichier RPM requis pour la version souhaitée sur le site de téléchargement Oracle Java .
2. Installez en utilisant la commande `rpm` . Par exemple:

```
$ sudo rpm -ivh jdk-8u67-linux-x644.rpm
```

Installation d'un Java JDK ou JRE sous Windows

Seuls les JDK et JRE Oracle sont disponibles pour les plates-formes Windows. La procédure d'installation est simple:

1. Visitez la page de téléchargement Oracle Java:
2. Cliquez sur le bouton JDK, le bouton JRE ou le bouton JRE du serveur. Notez que pour développer en Java, vous avez besoin du JDK. Pour connaître la différence entre JDK et JRE, voir ici
3. Faites défiler jusqu'à la version que vous souhaitez télécharger. (En règle générale, le plus récent est recommandé.)
4. Sélectionnez le bouton radio "Accepter le contrat de licence".
5. Téléchargez le programme d'installation Windows x86 (32 bits) ou Windows x64 (64 bits).
6. Exécutez le programme d'installation ... normalement pour votre version de Windows.

Une autre manière d'installer Java sous Windows à l'aide de l'invite de commandes consiste à utiliser Chocolately:

1. Installez Chocolately depuis https://chocolatey.org/

2. Ouvrez une instance de cmd, par exemple appuyez sur `Win` + `R` , puis tapez "cmd" dans la fenêtre "Exécuter" suivie d'une entrée.

3. Dans votre instance de cmd, exécutez la commande suivante pour télécharger et installer un JDK Java 8:

```
C:\> choco install jdk8
```

Mise en place avec des versions portables

Dans certains cas, vous pouvez installer JDK / JRE sur un système avec des privilèges limités, comme une VM, ou installer et utiliser plusieurs versions ou architectures (x64 / x86) de JDK / JRE. Les étapes restent les mêmes jusqu'au moment où vous téléchargez le programme d'installation (.EXE). Les étapes suivantes sont les suivantes (les étapes sont applicables à JDK / JRE 7 et versions ultérieures, pour les versions plus anciennes, elles sont légèrement différentes dans les noms de dossiers et de fichiers):

1. Déplacez le fichier vers un emplacement approprié où vous souhaitez que vos fichiers binaires Java résident de manière permanente.

2. Installez 7-Zip ou sa version portable si vous disposez de privilèges limités.

3. Avec 7-Zip, extrayez les fichiers du fichier EXE d'installation Java vers l'emplacement.

4. Ouvrez l'invite de commande en maintenant les `Shift` et `Right-Click` dans le dossier de l'explorateur ou naviguez jusqu'à cet emplacement depuis n'importe où.

5. Accédez au dossier nouvellement créé. Disons que le nom du dossier est `jdk-7u25-windows-x64` . Alors tapez `cd jdk-7u25-windows-x64` . Ensuite, tapez les commandes suivantes dans l'ordre:

   ```
   cd .rsrc\JAVA_CAB10

   extrac32 111
   ```

6. Cela créera un fichier `tools.zip` à cet emplacement. Extrayez le `tools.zip` avec 7-Zip pour que les fichiers qu'il `tools.zip` soient désormais créés sous les `tools` du même répertoire.

7. Maintenant, exécutez ces commandes sur l'invite de commande déjà ouverte:

   ```
   cd tools

   for /r %x in (*.pack) do .\bin\unpack200 -r "%x" "%~dx%~px%~nx.jar"
   ```

8. Attendez que la commande soit terminée. Copiez le contenu des `tools` à l'emplacement où vous souhaitez placer vos fichiers binaires.

De cette façon, vous pouvez installer toutes les versions de JDK / JRE dont vous avez besoin pour être installé simultanément.

Message original: http://stackoverflow.com/a/6571736/1448252

Installation d'un Java JDK sur macOS

Oracle Java 7 et Java 8

Java 7 et Java 8 pour macOS sont disponibles auprès d'Oracle. Cette page Oracle répond à beaucoup de questions sur Java pour Mac. Notez que Java 7 antérieur à 7u25 a été désactivé par Apple pour des raisons de sécurité.

En général, Oracle Java (version 7 et ultérieure) nécessite un Mac Intel fonctionnant sous macOS 10.7.3 ou ultérieur.

Installation de Oracle Java

Les installateurs JDK et JRE Java 7 & 8 pour macOS peuvent être téléchargés sur le site Web d'Oracle:

- Java 8 - Téléchargements Java SE
- Java 7 - Oracle Java Archive.

Après avoir téléchargé le package approprié, double-cliquez sur le package et suivez le processus d'installation normal. Un JDK devrait être installé ici:

```
/Library/Java/JavaVirtualMachines/<version>.jdk/Contents/Home
```

où correspond à la version installée.

Commutation en ligne de commande

Lorsque Java est installé, la version installée est automatiquement définie par défaut. Pour basculer entre différents, utilisez:

```
export JAVA_HOME=/usr/libexec/java_home -v 1.6 #Or 1.7 or 1.8
```

Les fonctions suivantes peuvent être ajoutées au `~/.bash_profile` (si vous utilisez le shell Bash par défaut) pour en faciliter l'utilisation:

```
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 sur macOS

Sur les anciennes versions de macOS (10.11 El Capitan et versions antérieures), la version d'Apple de Java 6 est préinstallée. Si installé, il peut être trouvé à cet endroit:

```
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

Notez que Java 6 a dépassé sa fin de vie, il est donc recommandé de mettre à niveau vers une version plus récente. Vous trouverez plus d'informations sur la réinstallation d'Apple Java 6 sur le site Web d'Oracle.

Configurer et changer les versions Java sur Linux en utilisant des alternatives

Utiliser des alternatives

De nombreuses distributions Linux utilisent la commande `alternatives` pour basculer entre les différentes versions d'une commande. Vous pouvez l'utiliser pour basculer entre les différentes versions de Java installées sur une machine.

1. Dans un shell de commande, définissez $ JDK sur le chemin d'accès d'un JDK nouvellement installé; par exemple

   ```
   $ JDK=/Data/jdk1.8.0_67
   ```

2. Utilisez des `alternatives --install` pour ajouter les commandes du Java SDK à des alternatives:

   ```
   $ 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
   ```

   Etc.

Vous pouvez maintenant basculer entre les différentes versions d'une commande Java comme suit:

```
$ 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
$
```

Pour plus d'informations sur l'utilisation d' `alternatives` , reportez-vous à l'entrée manuelle alternatives (8) .

Installations basées sur Arch

Les installations basées sur Arch Linux sont `archlinux-java` avec la commande `archlinux-java` pour changer de version Java.

Liste des environnements installés

```
$ archlinux-java status
Available Java environments:
  java-7-openjdk (default)
  java-8-openjdk/jre
```

Changer l'environnement actuel

```
# archlinux-java set <JAVA_ENV_NAME>
```

Par exemple:

```
# archlinux-java set java-8-openjdk/jre
```

Plus d'informations peuvent être trouvées sur le Wiki Arch Linux

Vérification et configuration post-installation sous Linux

Après avoir installé un SDK Java, il est conseillé de vérifier qu'il est prêt à être utilisé. Vous pouvez le faire en exécutant ces deux commandes, en utilisant votre compte utilisateur normal:

```
$ java -version
$ javac -version
```

Ces commandes impriment les informations de version du JRE et du JDK (respectivement) figurant sur le chemin de recherche de commandes de votre shell. Recherchez la chaîne de version JDK / JRE.

- Si l'une des commandes ci-dessus échoue en indiquant "commande introuvable", le JRE ou

le JDK ne se trouve pas du tout dans le chemin de recherche; allez dans **Configuration de PATH directement** ci-dessous.
- Si l'une des commandes ci-dessus affiche une chaîne de version différente de celle à laquelle vous vous attendiez, vous devez ajuster votre chemin de recherche ou le système "alternatives"; aller à la **vérification des alternatives**
- Si les chaînes de version correctes sont affichées, vous avez presque terminé; passer à la **vérification de JAVA_HOME**

Configuration de PATH directement

S'il n'y a pas de `java` ou de `javac` sur le chemin de recherche pour le moment, la solution simple consiste à l'ajouter à votre chemin de recherche.

Tout d'abord, trouvez où vous avez installé Java; voir **où Java a été installé?** ci-dessous si vous avez des doutes.

Ensuite, en supposant que `bash` est votre shell de commandes, utilisez un éditeur de texte pour ajouter les lignes suivantes à la fin de `~/.bash_profile` ou `~/.bashrc` (si vous utilisez Bash comme shell).

```
JAVA_HOME=<installation directory>
PATH=$JAVA_HOME/bin:$PATH

export JAVA_HOME
export PATH
```

... en remplaçant `<installation directory>` par le chemin d'accès de votre répertoire d'installation Java. Notez que ce qui précède suppose que le répertoire d'installation contient un répertoire `bin` et que le répertoire `bin` contient les commandes `java` et `javac` que vous essayez d'utiliser.

Ensuite, générez le fichier que vous venez de modifier afin que les variables d'environnement de votre shell actuel soient mises à jour.

```
$ source ~/.bash_profile
```

Ensuite, répétez les vérifications de la version `java` et `javac` . S'il y a toujours des problèmes, utilisez `which java` et `which javac` pour vérifier que vous avez correctement mis à jour les variables d'environnement.

Enfin, déconnectez-vous et reconnectez-vous afin que les variables d'environnement mises à jour soient appliquées à tous vos shells. Vous devriez maintenant être fait.

Vérification des alternatives

Si `java -version` ou `javac -version` fonctionné mais a donné un numéro de version inattendu, vous devez vérifier d'où proviennent les commandes. Utilisez `which` et `ls -l` pour trouver ceci comme suit:

```
$ ls -l `which java`
```

Si le résultat ressemble à ceci:

```
lrwxrwxrwx. 1 root root 22 Jul 30 22:18 /usr/bin/java -> /etc/alternatives/java
```

alors la `alternatives` versions `alternatives` est utilisée. Vous devez décider de continuer à l'utiliser ou simplement le remplacer en définissant directement `PATH` .

- Configurer et changer les versions Java sous Linux en utilisant des alternatives
- Voir "Configurer PATH directement" ci-dessus.

Où Java était-il installé?

Java peut être installé à différents endroits, en fonction de la méthode d'installation.

- Les RPM Oracle placent l'installation Java dans "/ usr / java".
- Sur Fedora, l'emplacement par défaut est "/ usr / lib / jvm".
- Si Java était installé à la main à partir de fichiers ZIP ou JAR, l'installation pourrait être n'importe où.

Si vous avez du mal à trouver le répertoire d'installation, nous vous suggérons d'utiliser `find` (ou `slocate`) pour trouver la commande. Par exemple:

```
$ find / -name java -type f 2> /dev/null
```

Cela vous donne les noms de chemin pour tous les fichiers appelés `java` sur votre système. (La redirection de l'erreur standard vers "/ dev / null" supprime les messages concernant les fichiers et les répertoires auxquels vous ne pouvez pas accéder.)

Installer oracle java sur Linux avec le dernier fichier tar

Suivez les étapes ci-dessous pour installer Oracle JDK à partir du dernier fichier tar:

1. Téléchargez le dernier fichier tar ici - Le dernier en date est Java SE Development Kit 8u112.

2. Vous avez besoin de privilèges sudo:

   ```
   sudo su
   ```

3. Créez un répertoire pour jdk install:

   ```
   mkdir /opt/jdk
   ```

4. Extrait du tar téléchargé dans celui-ci:

```
tar -zxf jdk-8u5-linux-x64.tar.gz -C /opt/jdk
```

5. Vérifiez si les fichiers sont extraits:

```
ls /opt/jdk
```

6. Définir Oracle JDK comme JVM par défaut:

```
update-alternatives --install /usr/bin/java java /opt/jdk/jdk1.8.0_05/bin/java 100
```

et

```
update-alternatives --install /usr/bin/javac javac /opt/jdk/jdk1.8.0_05/bin/javac 100
```

7. Vérifiez la version de Java:

java -version

Production attendue:

```
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)
```

Lire Installation de Java (Standard Edition) en ligne:
https://riptutorial.com/fr/java/topic/4754/installation-de-java--standard-edition-

Chapitre 89: Interface de l'outil JVM

Remarques

Interface de l'outil JVM TM

Version 1.2

http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

Examples

Itérer sur les objets accessibles depuis l'objet (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);
            }
        }
    }
}
```

Obtenez l'environnement JVMTI

Méthode Inside Agent_OnLoad:

```
jvmtiEnv* jvmti;
/* Get JVMTI environment */
vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
```

Exemple d'initialisation à l'intérieur de la méthode 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;
}
```

Lire Interface de l'outil JVM en ligne: https://riptutorial.com/fr/java/topic/3316/interface-de-l-outil-jvm

Chapitre 90: Interface Dequeue

Introduction

Un Deque est une collection linéaire qui supporte l'insertion et le retrait d'éléments aux deux extrémités.

Le nom deque est l'abréviation de "file d'attente double" et se prononce généralement "deck".

La plupart des implémentations Deque n'imposent pas de limites au nombre d'éléments qu'elles peuvent contenir, mais cette interface prend en charge les déques à capacité limitée ainsi que celles sans limite de taille fixe.

L'interface Deque est un type de données abstrait plus riche que Stack et Queue car elle implémente à la fois des piles et des files d'attente

Remarques

Les génériques peuvent être utilisés avec Deque.

```
Deque<Object> deque = new LinkedList<Object>();
```

Lorsqu'un deque est utilisé comme file d'attente, le comportement FIFO (First-In-First-Out) est obtenu.

Deques peut également être utilisé comme piles LIFO (Last-In-First-Out).

Pour plus d'informations sur les méthodes, consultez cette documentation.

Examples

Ajout d'éléments à 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");
```

Enlever des éléments de Deque

```
//Retrieves and removes the head of the queue represented by this deque
Object headItem = deque.remove();

//Retrieves and removes the first element of this deque.
Object firstItem = deque.removeFirst();

//Retrieves and removes the last element of this deque.
Object lastItem = deque.removeLast();
```

Récupérer un élément sans le supprimer

```
//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();
```

Itérer à travers 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;
}
```

Lire Interface Dequeue en ligne: https://riptutorial.com/fr/java/topic/10156/interface-dequeue

Chapitre 91: Interface Fluent

Remarques

Buts

L'objectif principal d'une interface Fluent est d'améliorer la lisibilité.

Lorsqu'il est utilisé pour construire des objets, les choix disponibles pour l'appelant peuvent être clairement définis et appliqués via des contrôles à la compilation. Par exemple, considérez l'arbre suivant des options représentant des étapes le long du chemin pour construire un objet complexe:

```
A -> B
  -> C -> D -> Done
       -> E -> Done
       -> F -> Done.
       -> G -> H -> I -> Done.
```

Un constructeur utilisant une interface fluide permettrait à l'appelant de voir facilement quelles options sont disponibles à chaque étape. Par exemple, **A -> B** est possible, mais **A -> C** n'est pas et entraînerait une erreur de compilation.

Examples

Vérité - Cadre de test courant

De "Comment utiliser la vérité" 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();
```

Style de programmation fluide

Dans un style de programmation courant, vous retournez `this` partir de méthodes courantes (setter) qui ne renverraient rien dans un style de programmation non fluide.

Cela vous permet d'enchaîner les différents appels de méthode, ce qui rend votre code plus court et plus facile à gérer pour les développeurs.

Considérez ce code non fluide:

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

Comme les méthodes setter ne renvoient rien, nous avons besoin de 4 instructions dans la méthode `main` pour instancier une `Person` avec certaines données et l'imprimer. Avec un style courant, ce code peut être modifié pour:

```
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'idée est de toujours renvoyer un objet pour permettre la construction d'une chaîne d'appels de méthode et pour utiliser des noms de méthodes qui reflètent une expression naturelle. Ce style courant rend le code plus lisible.

Lire Interface Fluent en ligne: https://riptutorial.com/fr/java/topic/5090/interface-fluent

Chapitre 92: Interface native Java

Paramètres

Paramètre	Détails
JNIEnv	Pointeur vers l'environnement JNI
jobject	L'objet qui a appelé la méthode `native` non `static`
jclass	La classe qui a invoqué la méthode `native static`

Remarques

La configuration de JNI nécessite à la fois un compilateur Java et un compilateur natif. Selon l'IDE et le système d'exploitation, une configuration est requise. Un guide pour Eclipse peut être trouvé ici . Un tutoriel complet peut être trouvé ici .

Voici les étapes à suivre pour configurer le lien Java-C ++ sur Windows:

- Compilez les fichiers source Java (`.java`) dans les classes (`.class`) en utilisant `javac` .
- Créez des fichiers d'en-tête (`.h`) à partir des classes Java contenant `native` méthodes `native` utilisant `javah` . Ces fichiers "instruisent" le code natif dont il est responsable de l'implémentation.
- Incluez les fichiers d'en-tête (`#include`) dans les fichiers source C ++ (`.cpp`) implémentant les méthodes `native` .
- Compilez les fichiers source C ++ et créez une bibliothèque (`.dll`). Cette bibliothèque contient l'implémentation du code natif.
- Spécifiez le chemin de la bibliothèque (`-Djava.library.path`) et chargez-le dans le fichier source Java (`System.loadLibrary(...)`).

Les rappels (appel des méthodes Java à partir du code natif) nécessitent de spécifier un descripteur de méthode. Si le descripteur est incorrect, une erreur d'exécution se produit. Pour cette raison, il est utile de faire les descripteurs pour nous, cela peut être fait avec `javap -s` .

Examples

Appel des méthodes C ++ à partir de Java

Les méthodes statiques et membres de Java peuvent être marquées comme *natives* pour indiquer que leur implémentation se trouve dans un fichier de bibliothèque partagé. Lors de l'exécution d'une méthode native, la JVM recherche une fonction correspondante dans les bibliothèques chargées (voir Chargement de bibliothèques natives) en utilisant un simple schéma de gestion des noms, effectue la conversion des arguments et la pile.

Code 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);
    }
}
```

Code C ++

Les fichiers d'en-tête contenant des déclarations de fonctions natives doivent être générés à l'aide de l'outil `javah` sur les classes cibles. Exécuter la commande suivante dans le répertoire de construction:

```
javah -o com_example_jni_JNIJava.hpp com.example.jni.JNIJava
```

... produit le fichier d'en-tête suivant (*commentaires supprimés pour des raisons de concision*):

```
// 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
```

Voici un exemple d'implémentation:

```
// 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;
}
```

Sortie

L'exécution de la classe d'exemple ci-dessus génère la sortie suivante:

> C ++ appelé "printString" depuis Java
> Vous avez obtenu le résultat de la moyenne C ++: 3.4

Appeler les méthodes Java à partir de C ++ (rappel)

L'appel d'une méthode Java à partir de code natif est un processus en deux étapes:

1. obtenir un pointeur de méthode avec la fonction JNI `GetMethodID` , en utilisant le nom de la méthode et le descripteur;
2. appelez l'une des fonctions de la `Call*Method` répertoriées ici .

Code 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);
    }
}
```

Code 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);
}
```

Sortie

Got float de C ++: 5.221
Got int de C ++: 17

Obtenir le descripteur

Les descripteurs (ou *signatures de type internes*) sont obtenus à l'aide du programme **javap** sur le fichier `.class` compilé. Voici la sortie de `javap -p -s com.example.jni.JNIJavaCallback` :

```
Compiled from "JNIJavaCallback.java"
public class com.example.jni.JNIJavaCallback {
  static {};
    descriptor: ()V

  public com.example.jni.JNIJavaCallback();
    descriptor: ()V

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V

  public native void callback();
    descriptor: ()V

  public static void printNum(int);
    descriptor: (I)V  // <---- Needed

  public void printFloat(float);
    descriptor: (F)V  // <---- Needed
}
```

Chargement de bibliothèques natives

L'idiome commun pour charger des fichiers de bibliothèque partagée en Java est le suivant:

```
public class ClassWithNativeMethods {
    static {
        System.loadLibrary("Example");
    }

    public native void someNativeMethod(String arg);
    ...
```

Les appels à `System.loadLibrary` sont presque toujours statiques pour se produire lors du chargement de la classe, garantissant qu'aucune méthode native ne peut s'exécuter avant le chargement de la bibliothèque partagée. Cependant, ce qui suit est possible:

```
public class ClassWithNativeMethods {
    // Call this before using any native method
    public static void prepareNativeMethods() {
        System.loadLibrary("Example");
    }

    ...
```

Cela permet de différer le chargement de la bibliothèque partagée jusqu'à ce que cela soit nécessaire, mais nécessite un soin particulier pour éviter `java.lang.UnsatisfiedLinkError` s.

Recherche de fichier cible

Les fichiers de bibliothèque partagés sont recherchés dans les chemins définis par la propriété système `java.library.path` , qui peuvent être `-Djava.library.path=` l'aide de l'argument `-Djava.library.path=` JVM au moment de l'exécution:

```
java -Djava.library.path=path/to/lib/:path/to/other/lib MainClassWithNativeMethods
```

Attention aux séparateurs de chemin système: par exemple, Windows utilise ; au lieu de :

Notez que `System.loadLibrary` résout les noms de fichiers de bibliothèque en fonction de la plate-forme: l'extrait de code ci-dessus attend un fichier nommé `libExample.so` sous Linux et `Example.dll` sous Windows.

Une alternative à `System.loadLibrary` est `System.load(String)` , qui prend le chemin d'accès complet à un fichier de bibliothèque partagée, contournant la recherche `java.library.path` :

```
public class ClassWithNativeMethods {
    static {
        System.load("/path/to/lib/libExample.so");
    }

    ...
```

Lire Interface native Java en ligne: https://riptutorial.com/fr/java/topic/168/interface-native-java

Chapitre 93: Interfaces

Introduction

Une *interface* est un type de référence, similaire à une classe, qui peut être déclaré à l'aide du mot-clé d' `interface` . Les interfaces ne peuvent contenir que des constantes, des signatures de méthode, des méthodes par défaut, des méthodes statiques et des types imbriqués. Les corps de méthodes n'existent que pour les méthodes par défaut et les méthodes statiques. Comme les classes abstraites, les interfaces ne peuvent pas être instanciées: elles ne peuvent être implémentées que par des classes ou étendues par d'autres interfaces. L'interface est un moyen courant d'obtenir une abstraction complète en Java.

Syntaxe

- interface publique Foo {void foo (); / * toute autre méthode * /}
- interface publique Foo1 étend Foo {void bar (); / * toute autre méthode * /}
- la classe publique Foo2 implémente Foo, Foo1 {/ * implémentation de Foo et Foo1 * /}

Examples

Déclaration et implémentation d'une interface

Déclaration d'une interface utilisant le mot-clé `interface` :

```
public interface Animal {
    String getSound(); // Interface methods are public by default
}
```

Annulation de l'annotation

```
@Override
public String getSound() {
    // Code goes here...
}
```

Cela oblige le compilateur à vérifier que nous remplaçons et empêche le programme de définir une nouvelle méthode ou de fausser la signature de la méthode.

Les interfaces sont implémentées à l'aide du mot clé `implements` .

```
public class Cat implements Animal {

    @Override
    public String getSound() {
        return "meow";
    }
}
```

```
public class Dog implements Animal {

    @Override
    public String getSound() {
        return "woof";
    }
}
```

Dans l'exemple, les classes `Cat` et `Dog` **doivent** définir la méthode `getSound()` car les méthodes d'une interface sont intrinsèquement abstraites (à l'exception des méthodes par défaut).

Utiliser les interfaces

```
Animal cat = new Cat();
Animal dog = new Dog();

System.out.println(cat.getSound()); // prints "meow"
System.out.println(dog.getSound()); // prints "woof"
```

Implémentation de plusieurs interfaces

Une classe Java peut implémenter plusieurs interfaces.

```
public interface NoiseMaker {
    String noise = "Making Noise"; // interface variables are public static final by default

    String makeNoise(); //interface methods are public abstract by default
}

public interface FoodEater {
    void eat(Food food);
}

public class Cat implements NoiseMaker, FoodEater {
    @Override
    public String makeNoise() {
        return "meow";
    }

    @Override
    public void eat(Food food) {
        System.out.println("meows appreciatively");
    }
}
```

Notez que la classe `Cat` **doit** implémenter les méthodes `abstract` héritées dans les deux interfaces. En outre, notez qu'une classe peut pratiquement implémenter autant d'interfaces que nécessaire (la limite de la JVM est limitée à **65 535**).

```
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
```

Remarque:

1. Toutes les variables déclarées dans une interface sont `public static final`
2. Toutes les méthodes déclarées dans une interface sont `public abstract` (cette instruction est valide uniquement via Java 7. A partir de Java 8, vous êtes autorisé à avoir des méthodes dans une interface, qui ne doivent pas nécessairement être abstraites; ces méthodes sont appelées méthodes par défaut)
3. Les interfaces ne peuvent être déclarées comme `final`
4. Si plusieurs interfaces déclarent une méthode ayant une signature identique, elle est effectivement traitée comme une seule méthode et vous ne pouvez pas distinguer la méthode d'interface implémentée
5. Un fichier **InterfaceName.class** correspondant serait généré pour chaque interface, lors de la compilation.

Extension d'une interface

Une interface peut étendre une autre interface via le mot `extends` clé extend.

```
public interface BasicResourceService {
    Resource getResource();
}

public interface ExtendedResourceService extends BasicResourceService {
    void updateResource(Resource resource);
}
```

Désormais, une classe implémentant `ExtendedResourceService` devra implémenter à la fois `getResource()` et `updateResource()` .

Extension de plusieurs interfaces

Contrairement aux classes, le mot `extends` clé extend peut être utilisé pour étendre plusieurs interfaces (séparées par des virgules), ce qui permet de combiner des interfaces dans une nouvelle interface.

```
public interface BasicResourceService {
    Resource getResource();
}

public interface AlternateResourceService {
    Resource getAlternateResource();
}

public interface ExtendedResourceService extends BasicResourceService,
AlternateResourceService {
    Resource updateResource(Resource resource);
}
```

Dans ce cas, une classe implémentant `ExtendedResourceService` devra implémenter `getResource()` , `getAlternateResource()`

et `updateResource()` .

Utiliser des interfaces avec des génériques

Supposons que vous souhaitiez définir une interface permettant de publier / consommer des données vers et depuis différents types de canaux (par exemple, AMQP, JMS, etc.), mais que vous souhaitiez pouvoir changer les détails d'implémentation ...

Définissons une interface IO de base pouvant être réutilisée sur plusieurs implémentations:

```
public interface IO<IncomingType, OutgoingType> {

    void publish(OutgoingType data);
    IncomingType consume();
    IncomingType RPCSubmit(OutgoingType data);

}
```

Maintenant, je peux instancier cette interface, mais comme nous n'avons pas d'implémentations par défaut pour ces méthodes, il faudra une implémentation lorsque nous l'instancierons:

```
    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
```

Nous pouvons aussi faire quelque chose de plus utile avec cette interface, disons que nous voulons l'utiliser pour envelopper certaines fonctions de base de RabbitMQ:

```
public class RabbitMQ implements IO<String, String> {

    private String exchange;
    private String queue;
```

```
    public RabbitMQ(String exchange, String queue){
        this.exchange = exchange;
        this.queue = queue;
    }

    @Override
    public void publish(String data) {
        rabbit.basicPublish(exchange, queue, data.getBytes());
    }

    @Override
    public String consume() {
        return rabbit.basicConsume(exchange, queue);
    }

    @Override
    public String RPCSubmit(String data) {
        return rabbit.rpcPublish(exchange, queue, data);
    }

}
```

Disons que je veux utiliser cette interface IO maintenant pour compter les visites sur mon site depuis le dernier redémarrage du système et afficher ensuite le nombre total de visites - vous pouvez faire quelque chose comme ceci:

```
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);
    }

}
```

Maintenant, utilisons le 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
```

Lorsque vous implémentez plusieurs interfaces, vous ne pouvez pas implémenter la même interface deux fois. Cela s'applique également aux interfaces génériques. Ainsi, le code suivant n'est pas valide et entraînera une erreur de compilation:

```
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); }
}
```

Utilité des interfaces

Les interfaces peuvent être extrêmement utiles dans de nombreux cas. Par exemple, disons que vous avez une liste d’animaux et que vous souhaitez parcourir la liste en imprimant chacun le son qu’ils font.

```
{cat, dog, bird}
```

Une façon d'y parvenir serait d'utiliser des interfaces. Cela permettrait d'appeler la même méthode sur toutes les classes

```
public interface Animal {
    public String getSound();
}
```

Toute classe qui `implements Animal` doit également avoir une méthode `getSound()` , mais toutes peuvent avoir des implémentations différentes

```
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";
    }
}
```

Nous avons maintenant trois classes différentes, chacune ayant une méthode `getSound()` . Comme toutes ces classes `implement` l'interface `Animal` , qui déclare la méthode `getSound()` , toute instance d'un `Animal` peut être appelée sur `getSound()`

```
Animal dog = new Dog();
Animal cat = new Cat();
Animal bird = new Bird();

dog.getSound(); // "Woof"
cat.getSound(); // "Meow"
bird.getSound(); // "Chirp"
```

Parce que chacun de ces `Animal` est un `Animal` , nous pourrions même mettre les animaux dans une liste, les parcourir en boucle et imprimer leurs sons.

```
Animal[] animals = { new Dog(), new Cat(), new Bird() };
for (Animal animal : animals) {
    System.out.println(animal.getSound());
}
```

Comme l'ordre du tableau est `Dog` , `Cat` , puis `Bird` , *"Woof Meow Chirp"* sera imprimé sur la console.

Les interfaces peuvent également être utilisées comme valeur de retour pour les fonctions. Par exemple, retourner un `Dog` si l'entrée est *"chien"* , `Cat` si l'entrée est *"chat"* , et `Bird` si c'est *"oiseau"* , et imprimer le son de cet animal pourrait être fait en utilisant

```
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
```

Les interfaces sont également utiles pour l'extensibilité, car si vous souhaitez ajouter un nouveau type d' `Animal` , vous n'avez rien à changer avec les opérations que vous effectuez.

Implémentation d'interfaces dans une classe abstraite

Une méthode définie dans une `interface` est par défaut `public abstract` . Lorsqu'une `abstract class` implémente une `interface` , toutes les méthodes définies dans l' `interface` ne doivent pas être implémentées par la `abstract class` . En effet, une `class` déclarée `abstract` peut contenir des déclarations de méthode abstraites. Il incombe donc à la première sous-classe concrète d'implémenter toutes `abstract` méthodes `abstract` héritées des interfaces et / ou de la `abstract class` .

```
public interface NoiseMaker {
    void makeNoise();
}

public abstract class Animal implements NoiseMaker {
    //Does not need to declare or implement makeNoise()
    public abstract void eat();
}

//Because Dog is concrete, it must define both makeNoise() and eat()
public class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Borf borf");
    }

    @Override
    public void eat() {
        System.out.println("Dog eats some kibble.");
    }
}
```

A partir de Java 8 en avant , il est possible pour une `interface` de déclarer par `default` les implémentations de méthodes qui signifie que la méthode ne sera pas `abstract` , donc des sous-classes concrètes ne seront pas contraints de mettre en œuvre la méthode , mais héritera de la `default` la mise en œuvre , sauf contrordre.

Méthodes par défaut

Introduites dans Java 8, les méthodes par défaut permettent de spécifier une implémentation dans une interface. Cela pourrait être utilisé pour éviter la classe typique "Base" ou "Abstract" en fournissant une implémentation partielle d'une interface et en limitant la hiérarchie des sous-classes.

Implémentation du modèle d'observateur

Par exemple, il est possible d'implémenter le modèle Observer-Listener directement dans l'interface, offrant plus de flexibilité aux classes d'implémentation.

```
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);
        }
    }
}
```

Maintenant, n'importe quelle classe peut être rendue "Observable" simplement en implémentant l'interface Observable, tout en étant libre de faire partie d'une hiérarchie de classes différente.

```
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();
    }
}
```

Problème de diamant

Le compilateur Java 8 est conscient du problème de diamant qui se produit lorsqu'une classe implémente des interfaces contenant une méthode avec la même signature.

Pour le résoudre, une classe d'implémentation doit remplacer la méthode partagée et fournir sa propre implémentation.

```
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"
    }
}
```

Il y a toujours le problème d'avoir des méthodes avec le même nom et les mêmes paramètres avec des types de retour différents, qui ne compileront pas.

Utiliser les méthodes par défaut pour résoudre les problèmes de compatibilité

Les implémentations de méthodes par défaut sont très pratiques si une méthode est ajoutée à une interface dans un système existant où les interfaces sont utilisées par plusieurs classes.

Pour éviter de casser le système entier, vous pouvez fournir une implémentation de méthode par défaut lorsque vous ajoutez une méthode à une interface. De cette façon, le système continuera à compiler et les implémentations réelles pourront être effectuées étape par étape.

Pour plus d'informations, voir la rubrique Méthodes par défaut .

Modificateurs dans les interfaces

Le Oracle Java Style Guide indique:

> Les modificateurs ne doivent pas être écrits lorsqu'ils sont implicites.

(Voir Modifiers dans Oracle Official Code Standard pour le contexte et un lien vers le document Oracle réel.)

Ce guide de style s'applique particulièrement aux interfaces. Considérons l'extrait de code suivant:

```
interface I {
    public static final int VARIABLE = 0;

    public abstract void method();

    public static void staticMethod() { ... }
    public default void defaultMethod() { ... }
}
```

Les variables

Toutes les variables d'interface sont des *constantes* implicites avec des modificateurs implicites `public` (accessibles pour tous), `static` (accessibles par nom d'interface) et `final` (doivent être initialisées pendant la déclaration):

```
~~public static final~~ int VARIABLE = 0;
```

Les méthodes

1. Toutes les méthodes qui *ne fournissent pas d'implémentation* sont implicitement `public` et `abstract` .

```
~~public abstract~~ void method();
```

Java SE 8

2. Toutes les méthodes avec `static` modificateur `static` ou `default` *doivent fournir une implémentation* et sont implicitement `public` .

```
~~public~~ static void staticMethod() { ... }
```

Une fois que toutes les modifications ci-dessus auront été appliquées, nous obtiendrons ce qui suit:

```
interface I {
    int VARIABLE = 0;

    void method();

    static void staticMethod() { ... }
    default void defaultMethod() { ... }
}
```

Renforcer les paramètres de type borné

Les paramètres de type lié vous permettent de définir des restrictions sur les arguments de type générique:

```
class SomeClass {

}

class Demo<T extends SomeClass> {

}
```

Mais un paramètre de type ne peut être lié qu'à un seul type de classe.

Un type d'interface peut être lié à un type qui a déjà une liaison. Ceci est réalisé en utilisant le symbole `&` :

```
interface SomeInterface {

}

class GenericClass<T extends SomeClass & SomeInterface> {

}
```

Cela renforce la liaison, nécessitant potentiellement des arguments de type à dériver de plusieurs types.

Plusieurs types d'interface peuvent être liés à un paramètre de type:

```
class Demo<T extends SomeClass & FirstInterface & SecondInterface> {

}
```

Mais devrait être utilisé avec prudence. Les liaisons d'interface multiples sont généralement un signe d' odeur de code , ce qui suggère qu'un nouveau type doit être créé, qui agit comme un adaptateur pour les autres types:

```
interface NewInterface extends FirstInterface, SecondInterface {

}

class Demo<T extends SomeClass & NewInterface> {
```

```
}
```

Lire Interfaces en ligne: https://riptutorial.com/fr/java/topic/102/interfaces

Chapitre 94: Interfaces fonctionnelles

Introduction

Dans Java 8+, une *interface fonctionnelle* est une interface qui ne comporte qu'une seule méthode abstraite (hormis les méthodes de Object). Voir JLS §9.8. Interfaces Fonctionnelles .

Examples

Liste des interfaces fonctionnelles standard de Java Runtime Library par signature

Types de paramètres	Type de retour	Interface
()	vide	Runnable
()	T	Fournisseur
()	booléen	BooleanSupplier
()	int	IntSupplier
()	longue	LongSupplier
()	double	DoubleSupplier
(T)	vide	Consommateur <T>
(T)	T	UnaryOperator <T>
(T)	R	Fonction <T, R>
(T)	booléen	Prédicat <T>
(T)	int	ToIntFunction <T>
(T)	longue	ToLongFunction <T>
(T)	double	ToDoubleFunction <T>
(T, T)	T	BinaryOperator <T>
(T, U)	vide	BiConsumer <T, U>
(T, U)	R	BiFunction <T, U, R>
(T, U)	booléen	BiPredicate <T, U>

Types de paramètres	Type de retour	Interface
(T, U)	int	ToIntBiFunction <T, U>
(T, U)	longue	ToLongBiFunction <T, U>
(T, U)	double	ToDoubleBiFunction <T, U>
(T, int)	vide	ObjIntConsumer <T>
(T, long)	vide	ObjLongConsumer <T>
(T, double)	vide	ObjDoubleConsumer <T>
(int)	vide	IntConsumer
(int)	R	IntFunction <R>
(int)	booléen	IntPredicate
(int)	int	IntUnaryOperator
(int)	longue	IntToLongFunction
(int)	double	IntToDoubleFunction
(int, int)	int	IntBinaryOperator
(longue)	vide	LongConsumer
(longue)	R	LongFunction <R>
(longue)	booléen	LongPredicate
(longue)	int	LongToIntFunction
(longue)	longue	LongUnaryOperator
(longue)	double	LongToDoubleFunction
(long, long)	longue	LongBinaryOperator
(double)	vide	DoubleConsumer
(double)	R	DoubleFunction <R>
(double)	booléen	DoublePredicate
(double)	int	DoubleToIntFunction
(double)	longue	DoubleToLongFunction

Types de paramètres	Type de retour	Interface
(double)	double	DoubleUnaryOperator
(double double)	double	DoubleBinaryOperator

Lire Interfaces fonctionnelles en ligne: https://riptutorial.com/fr/java/topic/10001/interfaces-fonctionnelles

Chapitre 95: Invocation de méthode distante (RMI)

Remarques

RMI nécessite 3 composants: client, serveur et une interface distante partagée. L'interface distante partagée définit le contrat client-serveur en spécifiant les méthodes qu'un serveur doit implémenter. L'interface doit être visible pour le serveur afin qu'il puisse implémenter les méthodes; l'interface doit être visible pour le client afin qu'il connaisse les méthodes ("services") fournies par le serveur.
Tout objet implémentant une interface distante est destiné à jouer le rôle de serveur. En tant que telle, une relation client-serveur dans laquelle le serveur peut également invoquer des méthodes dans le client est en fait une relation serveur-serveur. Ceci est appelé *callback* puisque le serveur peut rappeler le "client". Dans cette optique, il est acceptable d'utiliser la désignation *client* pour les serveurs qui fonctionnent comme tels.

L'interface distante partagée est toute interface d'extension `Remote` . Un objet qui fonctionne comme un serveur subit ce qui suit:

1. Implémente l'interface distante partagée, explicitement ou implicitement, en étendant `UnicastRemoteObject` qui implémente `Remote` .
2. Exporté, soit implicitement s'il étend `UnicastRemoteObject` , soit explicitement en étant transmis à `UnicastRemoteObject#exportObject` .
3. Lié à un registre, soit directement par le biais du `Registry` soit indirectement via `Naming` . Cela n'est nécessaire que pour établir la communication initiale, car d'autres tronçons peuvent être transmis directement via RMI.

Dans la configuration du projet, les projets client et serveur sont totalement indépendants, mais chacun spécifie un projet partagé dans son chemin de génération. Le projet partagé contient les interfaces distantes.

Examples

Client-Server: appel de méthodes dans une JVM à partir d'une autre

L'interface distante partagée:

```
package remote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteServer extends Remote {

    int stringToInt(String string) throws RemoteException;
}
```

Le serveur implémentant l'interface distante partagée:

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

Le client appelant une méthode sur le serveur (à distance):

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

Sortie:

> Serveur reçu: "120"
> Client reçu: 120

Callback: invoquer des méthodes sur un "client"

Vue d'ensemble

Dans cet exemple, 2 clients envoient des informations entre eux via un serveur. Un client envoie au serveur un numéro qui est transmis au second client. Le second client divise par deux le nombre et le renvoie au premier client via le serveur. Le premier client fait la même chose. Le serveur arrête la communication lorsque le nombre renvoyé par l'un des clients est inférieur à 10. La valeur de retour du serveur aux clients (le numéro converti en représentation sous forme de chaîne) fait ensuite un retour en arrière.

1. Un serveur de connexion se lie à un registre.
2. Un client recherche le serveur de connexion et appelle la méthode de `login` avec ses informations. Alors:
    - Le serveur de connexion stocke les informations client. Il inclut le stub du client avec les méthodes de rappel.
    - Le serveur de connexion crée et renvoie un stub de serveur ("connexion" ou "session") au client à stocker. Il inclut le stub du serveur avec ses méthodes incluant une méthode de `logout` (inutilisée dans cet exemple).
3. Un client appelle le `passInt` du serveur avec le nom du client destinataire et un `int` .
4. Le serveur appelle la `half` sur le client destinataire avec cet `int` . Cela déclenche une communication de va-et-vient (appels et rappels) jusqu'à son arrêt par le serveur.

Les interfaces distantes partagées

Le serveur de connexion:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteLogin extends Remote {
```

```
    RemoteConnection login(String name, RemoteClient client) throws RemoteException;
}
```

Le serveur:

```
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;
}
```

Le client:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteClient extends Remote {

    void half(int i) throws RemoteException;
}
```

Les implémentations

Le serveur de connexion:

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

Le serveur:

```
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);
    }
}
```

Le client:

```
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 + "\"");
    }
}
```

Exécuter l'exemple:

1. Exécutez le serveur de connexion.
2. Exécutez un client avec les arguments `Client2 Client1 1097` .
3. Exécutez un client avec les arguments `Client1 Client2 1098` .

Les sorties apparaîtront dans 3 consoles puisqu'il y a 3 JVM. ici ils sont regroupés:

Client2 connecté
Client1 connecté
Serveur reçu de Client1: 120
Serveur reçu de Client2: 60
Serveur reçu de Client1: 30
Serveur reçu de Client2: 15
Serveur reçu de Client1: 7
Client1 reçu: "7"
Client2 reçu: "15"
Client1 reçu: "30"
Client2 reçu: "60"

Exemple simple de RMI avec implémentation client et serveur

Ceci est un exemple simple de RMI avec cinq classes Java et deux packages, *serveur* et *client* .

Package de serveur

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();
    }
}
```

Package client

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);
        }
    }
}
```

Testez votre application

- Démarrez la méthode principale de Server.java. Sortie:

```
Server : Registry created.
Server : PersonList registered
```

- Commencez la méthode principale de PersonTest.java. Sortie:

```
Peter Pan
Pippi Langstrumpf
```

Lire Invocation de méthode distante (RMI) en ligne:
https://riptutorial.com/fr/java/topic/171/invocation-de-methode-distante--rmi-

Chapitre 96: Itérateur et Iterable

Introduction

`java.util.Iterator` est l'interface Java SE standard pour les objets qui implémentent le modèle de conception Iterator. L'interface `java.lang.Iterable` est destinée aux objets pouvant *fournir* un itérateur.

Remarques

Il est possible d'itérer sur un tableau en utilisant la boucle `for` -each, bien que les tableaux Java n'implémentent pas Iterable; l'itération est effectuée par JVM en utilisant un index non accessible en arrière-plan.

Examples

Utiliser Iterable en boucle

Les cours de mise en œuvre `Iterable<>` l' interface peuvent être utilisés dans `for` boucles. Ce n'est en fait que du sucre syntaxique pour obtenir un itérateur à partir de l'objet et l'utiliser pour obtenir tous les éléments de manière séquentielle; cela rend le code plus clair, plus rapide à écrire et moins sujet aux erreurs.

```
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);
        }
    }
}
```

Utiliser l'itérateur brut

Bien que l'utilisation de la boucle foreach (ou "extended for loop") soit simple, il est parfois utile d'utiliser directement l'itérateur. Par exemple, si vous souhaitez générer un ensemble de valeurs

séparées par des virgules, mais ne souhaitez pas que le dernier élément comporte une virgule:

```
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(",");
    }
}
```

C'est beaucoup plus facile et plus clair que d'avoir une variable `isLastEntry` ou de faire des calculs avec l'index de boucle.

Créer votre propre Iterable.

Pour créer votre propre Iterable comme avec n'importe quelle interface, il vous suffit d'implémenter les méthodes abstraites dans l'interface. Pour `Iterable` il n'y en a qu'un qui s'appelle `iterator()` . Mais son type de retour `Iterator` est lui-même une interface avec trois méthodes abstraites. Vous pouvez retourner un itérateur associé à une collection ou créer votre propre implémentation personnalisée:

```
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");
            }
        };
    }
}
```

Utiliser:

```
public static void main(String[] args) {
    for(char c : new Alphabet()) {
```

```
        System.out.println("c = " + c);
    }
}
```

Le nouvel `Iterator` doit être associé à un état indiquant le premier élément, chaque appel à la prochaine actualisant son état pour indiquer le suivant. Le `hasNext()` vérifie si l'itérateur est à la fin. Si l'itérateur était connecté à une collection modifiable, la méthode facultative `remove()` l'itérateur pourrait être implémentée pour supprimer l'élément actuellement pointé de la collection sous-jacente.

Suppression d'éléments à l'aide d'un itérateur

La méthode `Iterator.remove()` est une méthode facultative qui supprime l'élément renvoyé par l'appel précédent à `Iterator.next()` . Par exemple, le code suivant remplit une liste de chaînes puis supprime toutes les chaînes vides.

```
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());
```

Sortie:

```
Old Size : 5
New Size : 3
```

Notez que le code ci-dessus est un moyen sûr de supprimer des éléments lors de l'itération d'une collection type. Si, au contraire, vous tentez de supprimer des éléments d'une collection comme ceci:

```
for (String el: names) {
    if (el.equals("")) {
        names.remove(el); // WRONG!
    }
}
```

une collection typique (telle que `ArrayList`) qui fournit aux itérateurs une sémantique d'itérateur *rapide échoue* pour lancer une `ConcurrentModificationException` .

La méthode `remove()` ne peut être appelée qu'une seule fois après un appel `next()` . S'il est appelé avant d'appeler `next()` ou s'il est appelé deux fois après un appel `next()` appel `remove()` lancera

une `IllegalStateException` .

L'opération de `remove` est décrite comme une opération *facultative* ; Autrement dit, tous les itérateurs ne le permettront pas. Les exemples où il n'est pas pris en charge incluent les itérateurs pour les collections immuables, les vues en lecture seule des collections ou les collections de taille fixe. Si `remove()` est appelée lorsque l'itérateur ne prend pas en charge la suppression, une `UnsupportedOperationException` sera lancée.

Lire Itérateur et Iterable en ligne: https://riptutorial.com/fr/java/topic/172/iterateur-et-iterable

Chapitre 97: Java Native Access

Examples

Introduction à la JNA

Qu'est ce que la JNA?

Java Native Access (JNA) est une bibliothèque développée par la communauté fournissant aux programmes Java un accès facile aux bibliothèques partagées natives (fichiers `.dll` sous Windows, fichiers `.so` sous Unix ...)

Comment puis-je l'utiliser?

- Tout d'abord, téléchargez la dernière version de JNA et référencez son fichier jna.jar dans votre projet CLASSPATH.
- Deuxièmement, copiez, compilez et exécutez le code Java ci-dessous

  Aux fins de cette introduction, nous supposons que la plate-forme native utilisée est Windows. Si vous utilisez une autre plate-forme, remplacez simplement la chaîne `"msvcrt"` par la chaîne `"c"` dans le code ci-dessous.

Le petit programme Java ci-dessous imprimera un message sur la console en appelant la fonction 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 !");
    }
}
```

Où aller maintenant?

Sautez sur un autre sujet ici ou allez sur le site officiel .

Lire Java Native Access en ligne: https://riptutorial.com/fr/java/topic/5244/java-native-access

Chapitre 98: Java Virtual Machine (JVM)

Examples

Ce sont les bases.

JVM est une **machine informatique abstraite** ou une **machine virtuelle** qui réside dans votre RAM. Il dispose d'un environnement d'exécution indépendant de la plate-forme qui interprète le code d'octet Java en code machine natif. (Javac est Java Compiler qui compile votre code Java dans Bytecode)

Le programme Java s'exécutera dans la machine virtuelle Java, qui est ensuite mappée sur la machine physique sous-jacente. C'est l'un des outils de programmation du JDK.

(`Byte code` est un code indépendant de la plate-forme qui s'exécute sur chaque plate-forme et le `Machine code` est un code spécifique à la plate-forme exécuté uniquement sur une plate-forme spécifique telle que Windows ou Linux;

Quelques composants: -

- Class Loder - charge le fichier .class dans la RAM.
- Vérificateur de bytecode - vérifiez s'il existe des violations de restriction d'accès dans votre code.
- Moteur d'exécution: convertit le code d'octet en code machine exécutable.
- JIT (juste à temps) - JIT fait partie de JVM qui améliorait les performances de JVM. Il compilera ou traduira dynamiquement le bytecode Java en code machine natif pendant le temps d'exécution.

(Édité)

Lire Java Virtual Machine (JVM) en ligne: https://riptutorial.com/fr/java/topic/8110/java-virtual-machine--jvm-

Chapitre 99: JavaBean

Introduction

JavaBeans (TM) est un modèle de conception d'API de classe Java qui permet d'utiliser des instances (beans) dans divers contextes et d'utiliser divers outils *sans* écrire explicitement du code Java. Les modèles se composent de conventions permettant de définir des getters et des setters pour les *propriétés* , de définir des constructeurs et de définir des API d'écouteur d'événement.

Syntaxe

- **Règles de dénomination des propriétés JavaBean**
- Si la propriété n'est pas un booléen, le préfixe de la méthode getter doit être obtenu. Par exemple, getSize () est un nom getter JavaBeans valide pour une propriété nommée "size". N'oubliez pas qu'il n'est pas nécessaire d'avoir une variable nommée size. Le nom de la propriété est déduit des getters et des setters, et non des variables de votre classe. Ce que vous retournez de getSize () dépend de vous.
- Si la propriété est un booléen, le préfixe de la méthode getter est get ou is. Par exemple, getStopped () ou isStopped () sont des noms JavaBeans valides pour une propriété booléenne.
- Le préfixe de la méthode setter doit être défini. Par exemple, setSize () est le nom JavaBean valide d'une propriété nommée size.
- Pour compléter le nom d'une méthode getter ou setter, remplacez la première lettre du nom de la propriété par une majuscule, puis ajoutez-la au préfixe approprié (get, is ou set).
- Les signatures de méthode Setter doivent être marquées comme publiques, avec un type de retour vide et un argument représentant le type de propriété.
- Les signatures de la méthode Getter doivent être marquées comme publiques, ne prendre aucun argument et avoir un type de retour correspondant au type d'argument de la méthode setter pour cette propriété.
- **Règles de nommage JavaBean Listener**
- Les noms de méthode d'écoute utilisés pour "enregistrer" un écouteur avec une source d'événement doivent utiliser le préfixe add, suivi du type d'écouteur. Par exemple, addActionListener () est un nom valide pour une méthode qu'une source d'événement devra autoriser d'autres personnes à s'inscrire aux événements Action.
- Les noms de méthode d'écoute utilisés pour supprimer ("unregister") un écouteur doivent utiliser le préfixe remove, suivi du type d'écouteur (en utilisant les mêmes règles que la méthode d'inscription d'inscription).
- Le type d'écouteur à ajouter ou à supprimer doit être transmis en tant qu'argument de la méthode.
- Les noms de méthode d'écoute doivent se terminer par le mot "Listener".

Remarques

Pour qu'une classe soit un bean Java doit suivre cette norme - en résumé:

- Toutes ses propriétés doivent être privées et uniquement accessibles par des getters et des setters.
- Il doit avoir un constructeur public sans argument.
- Doit implémenter l'interface `java.io.Serializable` .

Examples

Bean Java Basique

```
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;
    }
}
```

Lire JavaBean en ligne: https://riptutorial.com/fr/java/topic/8157/javabean

Chapitre 100: JAXB

Introduction

JAXB ou Java Architecture for XML Binding (JAXB) est une infrastructure logicielle permettant aux développeurs Java de mapper des classes Java avec des représentations XML. Cette page présentera les lecteurs à JAXB en utilisant des exemples détaillés de ses fonctions fournies principalement pour le marshaling et le non marshaling des objets Java au format xml et vice-versa.

Syntaxe

- JAXB.marshall (objet, fileObjOfXML);
- Objet obj = JAXB.unmarshall (fileObjOfXML, className);

Paramètres

Paramètre	Détails
fileObjOfXML	Objet `File` d'un fichier XML
nom du cours	Nom d'une classe avec l'extension `.class`

Remarques

À l'aide de l'outil XJC disponible dans le JDK, le code Java d'une structure xml décrit dans un schéma xml (fichier `.xsd`) peut être généré automatiquement, voir la rubrique XJC .

Examples

Ecriture d'un fichier XML (regroupement d'un objet)

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

En utilisant l'annotation `XMLRootElement` , nous pouvons marquer une classe comme élément racine

d'un fichier 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()` est utilisé pour écrire le contenu de l'objet dans un fichier XML. Ici, `user` objet `user` et un nouvel objet `File` sont transmis en tant qu'arguments au `marshal()` .

En cas d'exécution réussie, cela crée un fichier XML nommé `UserDetails.xml` dans le chemin de classe avec le contenu ci-dessous.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

Lecture d'un fichier XML (désarchivage)

Pour lire un fichier XML nommé `UserDetails.xml` avec le contenu ci-dessous

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

Nous avons besoin d'une classe POJO nommée `User.java` comme ci-dessous

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

Ici, nous avons créé les variables et le nom de la classe en fonction des nœuds XML. Pour les mapper, nous utilisons l'annotation `XmlRootElement` sur la 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!");
        }
    }
}
```

Ici, la méthode `unmarshal()` est utilisée pour analyser le fichier XML. Il prend le nom de fichier XML et le type de classe comme deux arguments. Ensuite, nous pouvons utiliser les méthodes getter de l'objet pour imprimer les données.

Utilisation de XmlAdapter pour générer le format xml souhaité

Lorsque le format XML souhaité diffère du modèle d'objet Java, une implémentation XmlAdapter peut être utilisée pour transformer un objet de modèle en objet au format xml et inversement. Cet exemple montre comment mettre la valeur d'un champ dans un attribut d'un élément avec le nom du champ.

```
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);
    }
}
```

Configuration automatique du mappage de champs / propriétés (@XmlAccessorType)

Annotation `@XmlAccessorType` détermine si les champs / propriétés seront automatiquement sérialisés en XML. Notez que les annotations de champ et de méthode `@XmlElement` , `@XmlAttribute` ou `@XmlTransient` ont priorité sur les paramètres par défaut.

```
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
```

Sortie

```
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>
```

Configuration manuelle du champ / propriété XML

Les annotations `@XmlElement` , `@XmlAttribute` ou `@XmlTransient` et autres dans le package `javax.xml.bind.annotation` permettent au programmeur de spécifier comment et comment les champs ou propriétés marqués doivent être sérialisés.

```
@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!");
        }
    }
}
```

Spécification d'une instance XmlAdapter pour (ré) utiliser des données existantes

Parfois, des instances spécifiques de données doivent être utilisées. La récréation n'est pas

souhaitée et le référencement `static` données `static` aurait une odeur de code.

Il est possible de spécifier un `XmlAdapter` exemple , le `Unmarshaller` doit utiliser, ce qui permet à l'utilisateur d'utiliser `XmlAdapter` s sans constructeur zéro arg et / ou transmettre des données à l'adaptateur.

Exemple

Classe d'utilisateurs

La classe suivante contient un nom et une image d'utilisateur.

```
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);
    }

}
```

Adaptateur

Pour éviter de créer la même image en mémoire deux fois (ainsi que de télécharger à nouveau les

données), l'adaptateur stocke les images dans une carte.

Java SE 7

Pour un code Java 7 valide, remplacez la méthode `getImage` par

```
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();
```

```
    }

}
```

Exemple XML

Les 2 xml suivants sont pour *Jon Skeet* et son homologue earth 2, qui sont tous deux identiques et utilisent donc le même 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"
```

Utiliser l'adaptateur

```
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());
```

Liaison d'un espace de noms XML à une classe Java sérialisable.

Ceci est un exemple de fichier `package-info.java` qui lie un espace de noms XML à une classe Java sérialisable. Cela doit être placé dans le même paquet que les classes Java qui doivent être sérialisées à l'aide de l'espace de noms.

```
/**
```

```
 * 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;
```

Utiliser XmlAdapter pour rogner la chaîne.

```
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();
    }
}
```

Et dans package-info.java, ajoutez la déclaration suivante.

```
@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;
```

Lire JAXB en ligne: https://riptutorial.com/fr/java/topic/147/jaxb

Chapitre 101: JAX-WS

Examples

Authentification de base

La façon de faire un appel JAX-WS avec une authentification de base est un peu évidente.

Voici un exemple où `Service` est la représentation de classe de service et `Port` est le port de service auquel vous souhaitez accéder.

```
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();
```

Lire JAX-WS en ligne: https://riptutorial.com/fr/java/topic/4105/jax-ws

Chapitre 102: JMX

Introduction

La technologie JMX fournit les outils nécessaires à la création de solutions distribuées, basées sur le Web, modulaires et dynamiques pour la gestion et la surveillance des périphériques, des applications et des réseaux axés sur les services. De par sa conception, cette norme est adaptée à l'adaptation des systèmes existants, à l'implémentation de nouvelles solutions de gestion et de surveillance et au branchement des futures.

Examples

Exemple simple avec Platform MBean Server

Disons que nous avons un serveur qui enregistre les nouveaux utilisateurs et les salue avec un message. Et nous voulons surveiller ce serveur et modifier certains de ses paramètres.

Tout d'abord, nous avons besoin d'une interface avec nos méthodes de surveillance et de contrôle

```
public interface UserCounterMBean {
    long getSleepTime();

    void setSleepTime(long sleepTime);

    int getUserCount();

    void setUserCount(int userCount);

    String getGreetingString();

    void setGreetingString(String greetingString);

    void stop();
}
```

Et une implémentation simple qui nous permettra de voir comment cela fonctionne et comment nous l'affectons

```
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) {
```

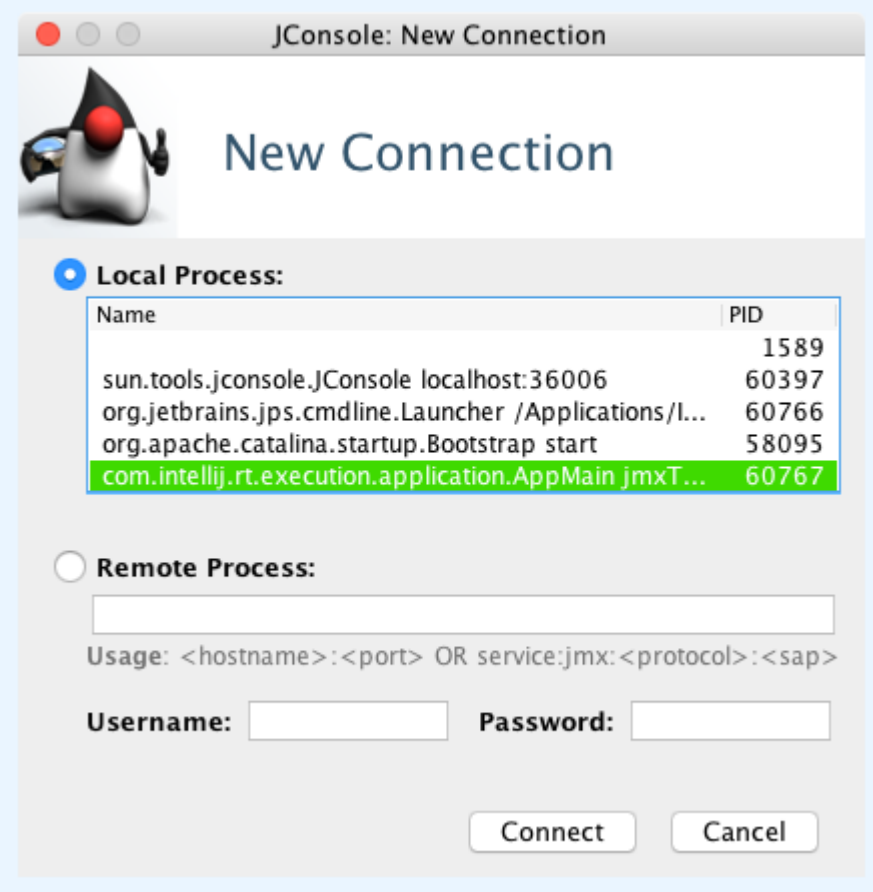
```
        this.sleepTime.set(sleepTime);
    }

    @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) {
            }
        }
    }
}
```

Pour un exemple simple avec une gestion locale ou distante, nous devons enregistrer notre MBean:

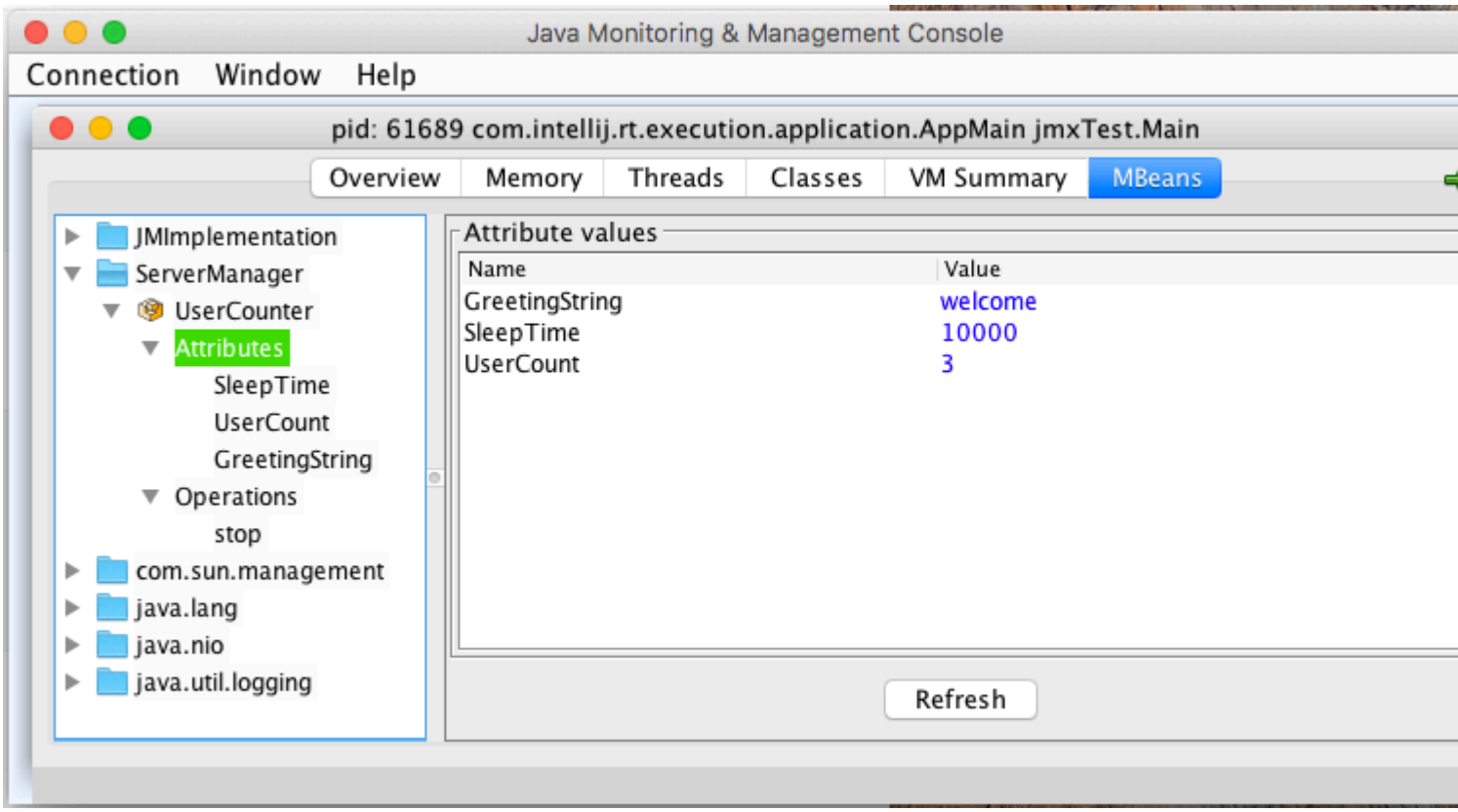
```
import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;

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();
    }
}
```

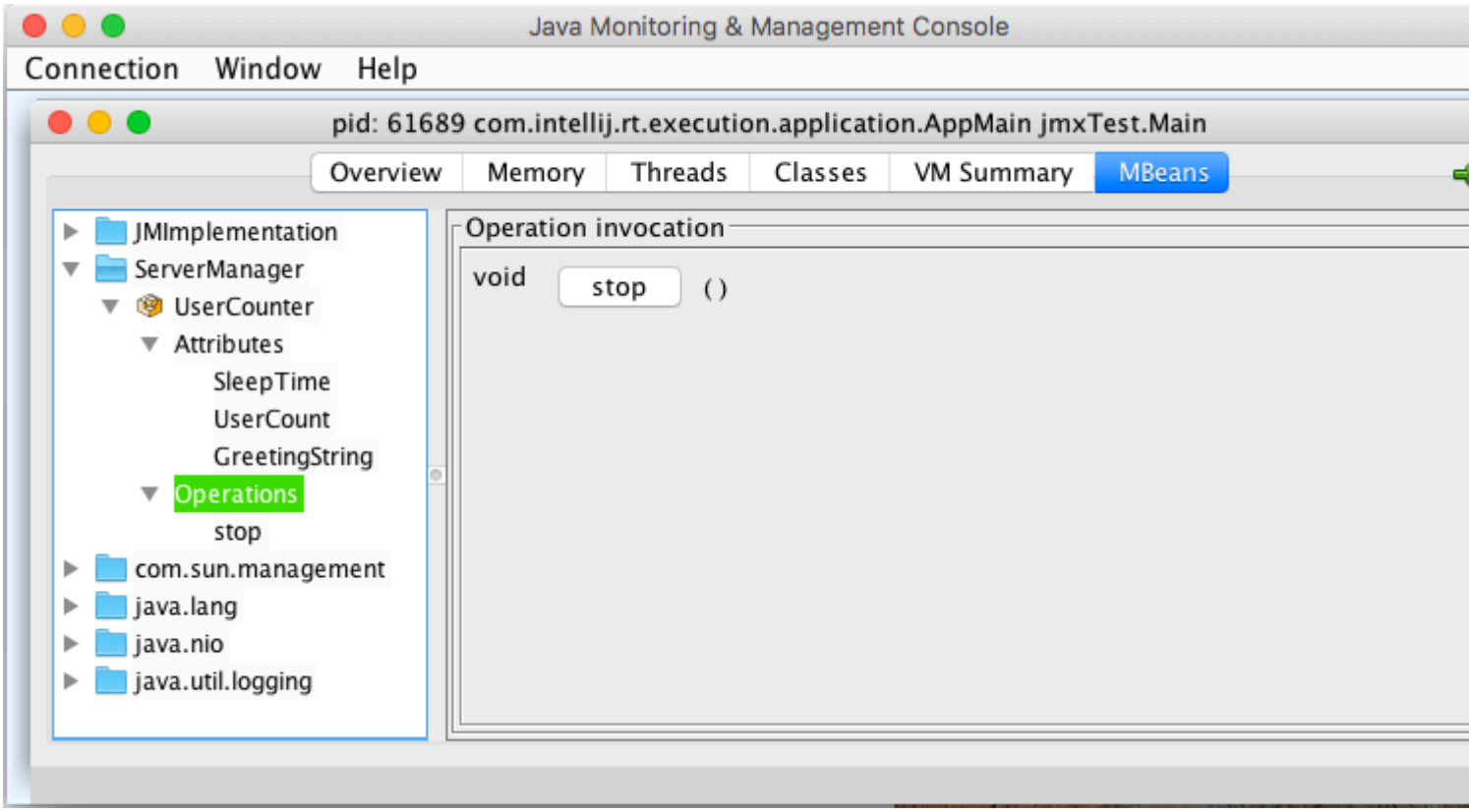
Après cela, nous pouvons exécuter notre application et nous y connecter via jConsole, qui se trouve dans votre répertoire `$JAVA_HOME/bin` . Premièrement, nous devons trouver notre processus java local avec notre application



puis passez à l'onglet MBeans et trouvez le MBean que nous avons utilisé dans notre classe Main en tant que `ObjectName` (dans l'exemple ci-dessus, il s'agit de `ServerManager`). Dans la section `Attributes` , nous pouvons voir les attributs. Si vous avez spécifié la méthode get uniquement, l'attribut sera lisible mais pas accessible en écriture. Si vous avez spécifié les méthodes get et set, l'attribut serait lisible et accessible en écriture.



Les méthodes spécifiées peuvent être appelées dans la section `Operations` .



Si vous voulez pouvoir utiliser la gestion à distance, vous aurez besoin de paramètres JVM supplémentaires, tels que:

```
-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
```

Ces paramètres se trouvent au chapitre 2 des guides JMX . Après cela, vous pourrez vous connecter à votre application via jConsole à distance avec l' `jconsole host:port` ou en spécifiant l' `host:port` ou `service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi` dans l'interface graphique de jConsole.

Liens utiles:

- Guides JMX
- Meilleures pratiques JMX

Lire JMX en ligne: https://riptutorial.com/fr/java/topic/9278/jmx

Chapitre 103: JNDI

Examples

RMI via JNDI

Cet exemple montre comment JNDI fonctionne dans RMI. Il a deux rôles:

- fournir au serveur une API bind / unbind / rebind au registre RMI
- fournir au client une API de recherche / liste au registre RMI.

Le registre RMI fait partie de RMI, pas de JNDI.

Pour simplifier, nous utiliserons `java.rmi.registry.CreateRegistry()` pour créer le registre RMI.

1. Server.java (le serveur 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 (le 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 de serveur RMI)

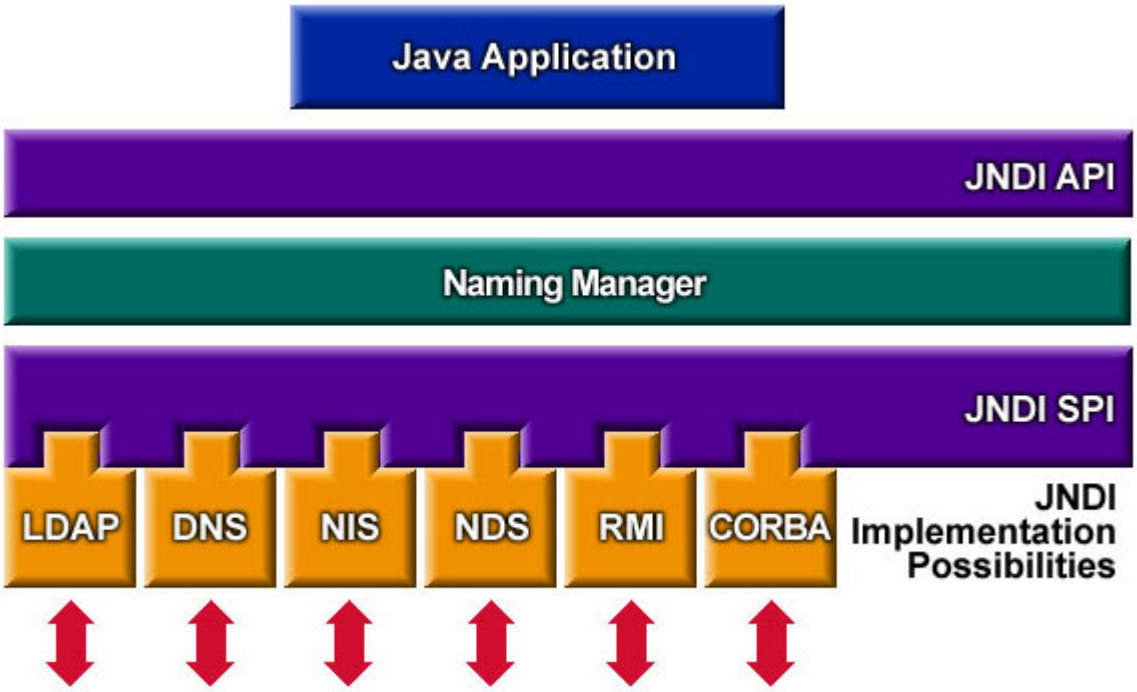
```
package com.neohope.jndi.test;

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;
    }
}
```

Comment exécuter le programme:

1. construire et démarrer le serveur
2. construire et démarrer le client

Présenter



L' **interface JNDI (Java Naming and Directory Interface)** est une API Java pour un service d'annuaire qui permet aux clients de logiciels Java de découvrir et de rechercher des données et des objets via un nom. Il est conçu pour être indépendant de toute implémentation de service d'annuaire ou de nom spécifique.

L'architecture JNDI se compose d'une **API** (interface de programmation d'application) et d'une interface **SPI** (Service Provider Interface). Les applications Java utilisent cette API pour accéder à divers services de noms et de répertoires. Le SPI permet de brancher de manière transparente divers services de noms et de répertoires, ce qui permet à l'application Java utilisant l'API de la technologie JNDI d'accéder à leurs services.

Comme vous pouvez le voir ci-dessus, JNDI prend en charge LDAP, DNS, NIS, NDS, RMI et

CORBA. Bien sûr, vous pouvez l'étendre.

Comment ça marche

Dans cet exemple, l'interface RMI Java utilise l'API JNDI pour rechercher des objets dans un réseau. Si vous voulez rechercher un objet, vous avez besoin d'au moins deux informations:

- Où trouver l'objet

Le registre RMI gère les liaisons de noms, il vous indique où trouver l'objet.

- Le nom de l'objet

Quel est le nom d'un objet? C'est généralement une chaîne, il peut également s'agir d'un objet qui implémente l'interface Name.

Pas à pas

1. Tout d'abord, vous avez besoin d'un registre qui gère la liaison de noms. Dans cet exemple, nous utilisons `java.rmi.registry.LocateRegistry` .

```
//This will start a registry on localhost, port 1234
registry = LocateRegistry.createRegistry(1234);
```

2. Le client et le serveur ont tous deux besoin d'un contexte. Le serveur utilise le contexte pour lier le nom et l'objet. Le client utilise le contexte pour rechercher le nom et obtenir l'objet.

```
//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. Le serveur lie le nom et l'objet

```
//The jndi name is "/neohope/jndi/test01"
bindJNDI("/neohope/jndi/test01", msg);
```

4. Le client recherche l'objet par le nom "/ neohope / jndi / test01"

```
//look up the object by name "java:com/neohope/jndi/test01"
NeoMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
```

5. Maintenant, le client peut utiliser l'objet

6. Lorsque le serveur se termine, vous devez nettoyer.

```
ctx.unbind("/neohope/jndi/test01");
ctx.close();
```

Lire JNDI en ligne: https://riptutorial.com/fr/java/topic/5720/jndi

Chapitre 104: Journalisation (java.util.logging)

Examples

Utilisation du consignateur par défaut

Cet exemple montre comment utiliser l'API de journalisation par défaut.

```
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()});
    }

}
```

Niveaux de journalisation

Java Logging Api a 7 niveaux . Les niveaux en ordre décroissant sont:

- `SEVERE` (valeur la plus élevée)
- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`

- `FINER`
- `FINEST` (valeur la plus basse)

Le niveau par défaut est `INFO` (mais cela dépend du système et de la machine virtuelle utilisée).

Remarque : Il existe également des niveaux `OFF` (peuvent être utilisés pour désactiver la déconnexion) et `ALL` (l'opportunité de `OFF`).

Exemple de code pour ceci:

```
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");
    }
}
```

Par défaut, l'exécution de cette classe ne produira que des messages de niveau supérieur à `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
```

Enregistrement de messages complexes (efficacement)

Regardons un exemple de journalisation que vous pouvez voir dans de nombreux programmes:

```
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'exemple ci-dessus semble très bien, mais de nombreux programmeurs oublient que Java VM est une machine à pile. Cela signifie que tous les paramètres de la méthode sont calculés **avant d'** exécuter la méthode.

Ce fait est crucial pour la connexion à Java, en particulier pour la journalisation de `FINER` à des niveaux bas tels que `FINE` , `FINER` , `FINEST` qui sont désactivés par défaut. Regardons le bytecode Java pour la méthode `takeOrder()` .

Le résultat pour `javap -c LoggingComplex.class` est quelque chose comme ceci:

```
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 ligne 39 exécute la journalisation proprement dite. Tous les travaux précédents (chargement de variables, création de nouveaux objets, concaténation de chaînes dans la méthode de `format`) peuvent être inutiles si le niveau de consignation est défini plus haut que `FINE` (et par défaut c'est le cas). Une telle journalisation peut être très inefficace, consommant des ressources inutiles de mémoire et de processeur.

C'est pourquoi vous devriez demander si le niveau que vous souhaitez utiliser est activé.

La bonne manière devrait être:

```
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
}
```

Depuis Java 8:

La classe Logger a des méthodes supplémentaires qui prennent un `Supplier<String>` comme paramètre, qui peut simplement être fourni par 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
}
```

La méthode Fournisseurs `get()` - dans ce cas, le lambda - est uniquement appelée lorsque le niveau correspondant est activé et donc `if` construction n'est plus nécessaire.

Lire Journalisation (java.util.logging) en ligne:
https://riptutorial.com/fr/java/topic/2010/journalisation--java-util-logging-

Chapitre 105: JShell

Introduction

JShell est un REPL interactif pour Java ajouté dans JDK 9. Il permet aux développeurs d'évaluer instantanément les expressions, les classes de test et d'expérimenter le langage Java. Un accès anticipé pour jdk 9 peut être obtenu à l'adresse suivante: http://jdk.java.net/9/

Syntaxe

- $ jshell - Démarrez le JShell REPL
- jshell> / <commande> - Exécuter une commande JShell donnée
- jshell> / exit - Quittez JShell
- jshell> / help - Voir une liste de commandes JShell
- jshell> <expression_java> - Evalue l'expression Java donnée (point-virgule facultatif)
- jshell> / vars OU / méthodes OR / types - Voir une liste de variables, méthodes ou classes, respectivement.
- jshell> / open <fichier> - lit un fichier en entrée du shell
- jshell> / edit <identifiant> - éditer un extrait dans l'éditeur de jeux
- jshell> / set editor <command> - définit la commande à utiliser pour éditer les extraits à l'aide de / edit
- jshell> / drop <identifier> - supprime un extrait
- jshell> / reset - Réinitialise la JVM et supprime tous les extraits

Remarques

JShell nécessite le JDK Java 9, qui peut actuellement (mars 2017) être téléchargé en tant que snapshots à accès anticipé à partir de jdk9.java.net . Si, lorsque vous essayez d'exécuter la commande `jshell` , vous obtenez une erreur commençant par `Unable to locate an executable` , assurez-vous que `JAVA_HOME` est défini correctement.

Importations par défaut

Les packages suivants sont importés automatiquement lorsque JShell démarre:

```
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

Entrer et quitter JShell

Démarrer JShell

Avant d'essayer de démarrer JShell, assurez-vous que votre variable d'environnement `JAVA_HOME` pointe vers une installation JDK 9. Pour démarrer JShell, exécutez la commande suivante:

```
$ jshell
```

Si tout se passe bien, vous devriez voir une invite `jshell>` .

Quitter JShell

Pour quitter JShell, exécutez la commande suivante à partir de l'invite JShell:

```
jshell> /exit
```

Expressions

Dans JShell, vous pouvez évaluer les expressions Java, avec ou sans points-virgules. Celles-ci peuvent aller d'expressions de base et d'énoncés à des expressions plus complexes:

```
jshell> 4+2
jshell> System.out.printf("I am %d years old.\n", 421)
```

Les boucles et les conditionnels vont bien aussi:

```
jshell> for (int i = 0; i<3; i++) {
   ...> System.out.println(i);
   ...> }
```

Il est important de noter que les **expressions dans les blocs doivent avoir des points-virgules!**

Les variables

Vous pouvez déclarer des variables locales dans JShell:

```
jshell> String s = "hi"
jshell> int i = s.length
```

Gardez à l'esprit que les variables peuvent être redéclarées avec différents types; Ceci est parfaitement valable dans JShell:

```
jshell> String var = "hi"
```

```
jshell> int var = 3
```

Pour voir une liste de variables, entrez `/vars` à l'invite JShell.

Méthodes et Classes

Vous pouvez définir des méthodes et des classes dans JShell:

```
jshell> void speak() {
   ...> System.out.println("hello");
   ...> }

jshell> class MyClass {
   ...> void doNothing() {}
   ...> }
```

Aucun modificateur d'accès n'est nécessaire. Comme pour les autres blocs, les points-virgules sont obligatoires dans les corps de méthode. Gardez à l'esprit que, comme pour les variables, il est possible de redéfinir les méthodes et les classes. Pour afficher une liste de méthodes ou de classes, entrez `/methods` ou `/types` à l'invite JShell, respectivement.

Editer les extraits

L'unité de code de base utilisée par JShell est l' **extrait** de code ou l' **entrée source** . Chaque fois que vous déclarez une variable locale ou définissez une méthode ou une classe locale, vous créez un fragment dont le nom est l'identificateur de la variable / méthode / classe. À tout moment, vous pouvez modifier un extrait que vous avez créé avec la commande `/edit` . Par exemple, disons que j'ai créé la classe `Foo` avec une seule méthode, `bar` :

```
jshell> class Foo {
   ...> void bar() {
   ...> }
   ...> }
```

Maintenant, je veux remplir le corps de ma méthode. Plutôt que de réécrire la classe entière, je peux l'éditer:

```
jshell> /edit Foo
```

Par défaut, un éditeur de swing apparaîtra avec les fonctionnalités les plus élémentaires possibles. Cependant, vous pouvez changer l'éditeur que JShell utilise:

```
jshell> /set editor emacs
jshell> /set editor vi
jshell> /set editor nano
jshell> /set editor -default
```

Notez que si **la nouvelle version de l'extrait de code contient des erreurs de syntaxe, il est possible qu'il ne soit pas enregistré.** De même, un extrait de code est créé uniquement si la

déclaration / définition d'origine est syntaxiquement correcte; ce qui suit ne fonctionne pas:

```
jshell> String st = String 3
//error omitted
jshell> /edit st
|  No such snippet: st
```

Toutefois, les extraits peuvent être compilés et donc modifiables malgré certaines erreurs de compilation, telles que les types incompatibles - les travaux suivants:

```
jshell> int i = "hello"
//error omitted
jshell> /edit i
```

Enfin, les extraits peuvent être supprimés à l'aide de la commande `/drop` :

```
jshell> int i = 13
jshell> /drop i
jshell> System.out.println(i)
|  Error:
|  cannot find symbol
|    symbol:   variable i
|  System.out.println(i)
|
```

Pour supprimer tous les extraits de code, réinitialisant ainsi l'état de la machine virtuelle Java, utilisez `\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
|  ^
```

Lire JShell en ligne: https://riptutorial.com/fr/java/topic/9511/jshell

Chapitre 106: JSON en Java

Introduction

JSON (JavaScript Object Notation) est un format d'échange de données léger, indépendant du langage et basé sur le langage, facile à lire et à écrire pour les utilisateurs et les ordinateurs. JSON peut représenter deux types structurés: les objets et les tableaux. JSON est souvent utilisé dans les applications Ajax, les configurations, les bases de données et les services Web RESTful. L'API Java pour le traitement JSON fournit des API portables pour analyser, générer, transformer et interroger JSON.

Remarques

Cet exemple se concentre sur l'analyse et la création de JSON en Java en utilisant diverses bibliothèques telles que la bibliothèque **Google Gson** , Jackson Object Mapper, etc.

Des exemples utilisant d'autres bibliothèques peuvent être trouvés ici: Comment analyser JSON en Java

Examples

Codage des données en tant que JSON

Si vous devez créer un objet `JSONObject` et y insérer des données, considérez l'exemple suivant:

```
// 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"]
   }
*/
```

Décodage des données JSON

Si vous avez besoin d'obtenir des données à partir d'un objet `JSONObject` , `JSONObject` l'exemple suivant:

```
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);
```

méthodes optXXX vs getXXX

`JSONObject` et `JSONArray` ont quelques méthodes très utiles pour gérer la possibilité qu'une valeur que vous essayez d'obtenir n'existe pas ou soit d'un autre type.

```
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"
```

Les mêmes règles s'appliquent aux méthodes `getXXX` / `optXXX` de `JSONArray` .

Objet à JSON (bibliothèque Gson)

Supposons que vous avez une classe appelée `Person` avec juste le `name`

```
private class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Code:

```
Gson g = new Gson();

Person person = new Person("John");
System.out.println(g.toJson(person)); // {"name":"John"}
```

Bien sûr, le pot de Gson doit être sur le chemin de classe .

JSON à objet (bibliothèque Gson)

Supposons que vous avez une classe appelée `Person` avec juste le `name`

```
private class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Code:

```
Gson gson = new Gson();
String json = "{\"name\": \"John\"}";

Person person = gson.fromJson(json, Person.class);
System.out.println(person.name); //John
```

Vous devez avoir la bibliothèque gson dans votre classpath.

Extraire un seul élément de JSON

```
String json = "{\"name\": \"John\", \"age\":21}";
```

```
JsonObject jsonObject = new JsonParser().parse(json).getAsJsonObject();

System.out.println(jsonObject.get("name").getAsString()); //John
System.out.println(jsonObject.get("age").getAsInt()); //21
```

Utilisation de Jackson Object Mapper

Modèle Pojo

```
public class Model {
    private String firstName;
    private String lastName;
    private int age;
    /* Getters and setters not shown for brevity */
}
```

Exemple: chaîne à objet

```
Model outputObject = objectMapper.readValue(
     "{\"firstName\":\"John\",\"lastName\":\"Doe\",\"age\":23}",
     Model.class);
System.out.println(outputObject.getFirstName());
//result: John
```

Exemple: objet à chaîne

```
String jsonString = objectMapper.writeValueAsString(inputObject));
//result: {"firstName":"John","lastName":"Doe","age":23}
```

Détails

Déclaration d'importation nécessaire:

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

Maven dépendance: jackson-databind

`ObjectMapper`

```
//creating one
ObjectMapper objectMapper = new ObjectMapper();
```

- `ObjectMapper` est threadsafe
- recommandé: avoir une instance partagée et statique

Désérialisation:

```
<T> T readValue(String content, Class<T> valueType)
```

- `valueType` doit être spécifié - le retour sera de ce type
- Jette
    - `IOException` - en cas de problème d'E / S de bas niveau
    - `JsonParseException` - si l'entrée sous-jacente contient un contenu non valide
    - `JsonMappingException` - si la structure JSON en entrée ne correspond pas à la structure de l'objet

Exemple d'utilisation (jsonString est la chaîne d'entrée):

```
Model fromJson = objectMapper.readValue(jsonString, Model.class);
```

Méthode de sérialisation:

String writeValueAsString (valeur d'objet)

- Jette
    - `JsonProcessingException` en cas d'erreur
    - Note: avant la version 2.1, la clause throws incluait IOException; 2.1 l'a enlevé.

Itération JSON

`JSONObject` sur les propriétés `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` sur les valeurs `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 - méthodes de chaînage

Vous pouvez utiliser le chaînage de méthode tout en travaillant avec `JSONObject` et `JSONArray` .

Exemple JSONObject

```
JSONObject obj = new JSONObject();//Initialize an empty JSON object
//Before: {}
obj.put("name","Nikita").put("age","30").put("isMarried","true");
//After: {"name":"Nikita","age":30,"isMarried":true}
```

JSONArray

```
JSONArray arr = new JSONArray();//Initialize an empty array
//Before: []
arr.put("Stack").put("Over").put("Flow");
//After: ["Stack","Over","Flow"]
```

JSONObject.NULL

Si vous devez ajouter une propriété avec une valeur `null` , vous devez utiliser le `JSONObject.NULL` final statique prédéfini et non la référence `null` Java standard.

`JSONObject.NULL` est une valeur sentinelle utilisée pour définir explicitement une propriété avec une valeur vide.

```
JSONObject obj = new JSONObject();
obj.put("some", JSONObject.NULL);   //Creates: {"some":null}
System.out.println(obj.get("some"));//prints: null
```

Remarque

```
JSONObject.NULL.equals(null); //returns true
```

Ce qui constitue une **violation manifeste** du contrat `Java.equals()` :

> Pour toute valeur de référence non nulle x, x.equals (null) doit renvoyer false

JsonArray to Java List (Bibliothèque Gson)

Voici un simple JsonArray que vous souhaitez convertir en Java `ArrayList` :

```
{
    "list": [
                "Test_String_1",
                "Test_String_2"
            ]
}
```

`JsonArray` maintenant la liste ' `JsonArray` ' à la méthode suivante qui renvoie une liste de `ArrayList` Java correspondante:

```
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;
}
```

Vous devez ajouter la dépendance `POM.xml` suivante à votre fichier `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>
```

Ou vous devriez avoir le `com.google.code.gson:gson:jar:<version>` jar `com.google.code.gson:gson:jar:<version>` dans votre classpath.

Désérialiser la collection JSON à la collection d'objets à l'aide de Jackson

Supposons que vous ayez une classe de pojo `Person`

```
public class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Et vous voulez l'analyser dans un tableau JSON ou une carte d'objets Person. En raison de l'effacement du type, vous ne pouvez pas construire directement les classes de `List<Person>` et `Map<String, Person>` *(et donc les utiliser pour désérialiser JSON)* . Pour surmonter cette limitation, jackson propose deux approches: `TypeFactory` et `TypeReference` .

TypeFactory

L'approche adoptée ici consiste à utiliser une fabrique (et sa fonction d'utilitaire statique) pour créer votre type pour vous. Les paramètres pris en compte sont la collection que vous souhaitez utiliser (liste, ensemble, etc.) et la classe que vous souhaitez stocker dans cette collection.

TypeReference

L'approche par référence de type semble plus simple car elle vous permet d'économiser un peu de saisie et est plus propre. TypeReference accepte un paramètre de type, où vous passez le type souhaité `List<Person>` . Vous instanciez simplement cet objet TypeReference et l'utilisez comme conteneur de type.

Voyons maintenant comment désérialiser réellement votre JSON dans un objet Java. Si votre JSON est formaté en tant que tableau, vous pouvez le désérialiser en tant que liste. S'il existe une structure imbriquée plus complexe, vous souhaiterez désérialiser une carte. Nous examinerons des exemples des deux.

Désérialisation du tableau JSON

```
String jsonString = "[{\"name\": \"Alice\"}, {\"name\": \"Bob\"}]"
```

Approche TypeFactory

```
CollectionType listType =
    factory.constructCollectionType(List.class, Person.class);
List<Preson> list = mapper.readValue(jsonString, listType);
```

Approche TypeReference

```
TypeReference<Person> listType = new TypeReference<List<Person>>() {};
List<Person> list = mapper.readValue(jsonString, listType);
```

Désérialisation de la carte JSON

```
String jsonString = "{\"0\": {\"name\": \"Alice\"}, \"1\": {\"name\": \"Bob\"}}"
```

Approche TypeFactory

```
CollectionType mapType =
    factory.constructMapLikeType(Map.class, String.class, Person.class);
List<Person> list = mapper.readValue(jsonString, mapType);
```

Approche TypeReference

```
TypeReference<Person> mapType = new TypeReference<Map<String, Person>>() {};
Map<String, Person> list = mapper.readValue(jsonString, mapType);
```

Détails

Déclaration d'importation utilisée:

```
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.type.CollectionType;
```

Instances utilisées:

```
ObjectMapper mapper = new ObjectMapper();
TypeFactory factory = mapper.getTypeFactory();
```

Remarque

Bien `TypeReference` approche `TypeReference` semble meilleure, elle présente plusieurs inconvénients:

1. `TypeReference` doit être instancié à l'aide d'une classe anonyme
2. Vous devez fournir une explication générique

Si vous ne le faites pas, vous risquez de perdre l'argument de type générique qui entraînera une défaillance de la désérialisation.

Lire JSON en Java en ligne: https://riptutorial.com/fr/java/topic/840/json-en-java

Chapitre 107: JVM Flags

Remarques

Il est fortement recommandé d'utiliser uniquement ces options:

- Si vous avez une compréhension approfondie de votre système.
- Sachez que, si elles ne sont pas utilisées correctement, ces options peuvent avoir un effet négatif sur la stabilité ou les performances de votre système.

Informations collectées à partir de la documentation Java officielle .

Examples

-XXaggressive

`-XXaggressive` est un ensemble de configurations qui permettent à la JVM de fonctionner à grande vitesse et d'atteindre un état stable dès que possible. Pour atteindre cet objectif, la JVM utilise davantage de ressources internes au démarrage; Cependant, une optimisation adaptative est nécessaire une fois l'objectif atteint. Nous vous recommandons d'utiliser cette option pour les applications de longue durée qui consomment beaucoup de mémoire et qui fonctionnent seules.

Usage:

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

<param>	La description
`opt`	Planifie des optimisations adaptatives plus tôt et permet de nouvelles optimisations, qui devraient être la valeur par défaut dans les futures versions.
`memory`	Configure le système de mémoire pour les charges de travail gourmandes en mémoire et espère activer de grandes quantités de ressources mémoire pour assurer un débit élevé. JRockit JVM utilisera également de grandes pages, si disponibles.

-XXallocClearChunks

Cette option vous permet d'effacer un TLA pour les références et les valeurs au moment de l'allocation TLA et de pré-récupérer le prochain segment. Lorsqu'un entier, une référence ou tout autre élément est déclaré, sa valeur par défaut est 0 ou null (selon le type). Au moment opportun, vous devrez effacer ces références et ces valeurs pour libérer la mémoire du tas afin que Java puisse l'utiliser ou la réutiliser. Vous pouvez le faire lorsque l'objet est alloué ou, en utilisant cette option, lorsque vous demandez un nouveau TLA.

Usage:

```
-XXallocClearChunks
```

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

Ce qui précède est une option booléenne et est généralement recommandé sur les systèmes IA64; en fin de compte, son utilisation dépend de l'application. Si vous souhaitez définir la taille des blocs effacés, combinez cette option avec `-XXallocClearChunkSize` . Si vous utilisez cet indicateur sans spécifier de valeur booléenne, la valeur par défaut est `true` .

-XXallocClearChunkSize

Lorsqu'elle est utilisée avec `-XXallocClearChunkSize` , cette option définit la taille des blocs à effacer. Si cet indicateur est utilisé mais qu'aucune valeur n'est spécifiée, la valeur par défaut est 512 octets.

Usage:

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

-XXcallProfiling

Cette option active l'utilisation du profilage d'appels pour les optimisations de code. Le profilage enregistre des statistiques d'exécution utiles spécifiques à l'application et peut, dans de nombreux cas, améliorer les performances car la JVM peut alors agir sur ces statistiques.

> Remarque: cette option est prise en charge avec JRockit JVM R27.3.0 et les versions ultérieures. Il peut devenir par défaut dans les futures versions.

Usage:

```
java -XXcallProfiling myApp
```

Cette option est désactivée par défaut. Vous devez lui permettre de l'utiliser.

-XXdisableFatSpin

Cette option désactive le code d’exécution de verrouillage du fat en Java, permettant ainsi aux threads qui bloquent d’essayer d’acquérir un fat lock de s’allumer directement.

Les objets en Java deviennent un verrou dès qu'un thread entre dans un bloc synchronisé sur cet objet. Tous les verrous sont maintenus (c'est-à-dire sont restés verrouillés) jusqu'à ce qu'ils soient libérés par le fil de verrouillage. Si le verrou ne va pas être libéré très rapidement, il peut être gonflé à un «gros verrou». La «rotation» se produit lorsqu'un thread qui souhaite un verrou spécifique vérifie en permanence ce verrou pour voir s'il est toujours pris, tournant dans un boucle serrée comme il fait le contrôle. Spinning contre un Fat Lock est généralement bénéfique, bien

que, dans certains cas, il peut être coûteux et peut affecter les performances. `-XXdisableFatSpin` vous permet de désactiver la rotation contre un gros verrou et d'éliminer les problèmes de performances potentiels.

Usage:

```
-XXdisableFatSpin
```

-XXdisableGCHeuristique

Cette option désactive les modifications de la stratégie de récupération de place. Les heuristiques de compactage et les heuristiques de taille de pépinière ne sont pas affectées par cette option. Par défaut, les heuristiques de récupération de place sont activées.

Usage:

```
-XXdisableFatSpin
```

-XXdumpSize

Cette option génère un fichier de vidage et vous permet de spécifier la taille relative de ce fichier (petit, moyen ou grand).

Usage:

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

<taille>	La description
`none`	Ne génère pas de fichier de vidage.
`small`	Sous Windows, un petit fichier de vidage est généré (sous Linux, un fichier de vidage complet est généré). Une petite sauvegarde ne comprend que les piles de threads, y compris leurs traces et très peu. Il s'agissait de la configuration par défaut dans JRVMit JVM 8.1 avec les Service Packs 1 et 2, ainsi que 7.0 avec Service Pack 3 et supérieur.
`normal`	Provoque un vidage normal sur toutes les plates-formes. Ce fichier de vidage inclut toute la mémoire à l'exception du tas Java. C'est la valeur par défaut pour JRVMit JVM 1.4.2 et versions ultérieures.
`large`	Inclut tout ce qui est en mémoire, y compris le tas Java. Cette option rend `-XXdumpSize` équivalent à `-XXdumpFullState` .

-XXexitOnOutOfMemory

Cette option permet à Jock JRockit de sortir à la première occurrence d'une erreur de mémoire

insuffisante. Il peut être utilisé si vous préférez redémarrer une instance de JVM JRockit plutôt que de gérer les erreurs de mémoire. Entrez cette commande au démarrage pour forcer la JVM JRockit à sortir à la première occurrence d'une erreur de mémoire insuffisante.

Usage:

```
-XXexitOnOutOfMemory
```

Lire JVM Flags en ligne: https://riptutorial.com/fr/java/topic/2500/jvm-flags

Chapitre 108: l'audio

Remarques

Au lieu d'utiliser le `Clip` javax.sound.sampled, vous pouvez également utiliser `AudioClip` qui provient de l'API de l'applet. Il est toutefois recommandé d'utiliser `Clip` car `AudioClip` est juste plus ancien et présente des fonctionnalités limitées.

Examples

Lire un fichier audio en boucle

Importations nécessaires:

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

Ce code va créer un clip et le lire en continu une fois démarré:

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

Obtenir un tableau avec tous les types de fichiers pris en charge:

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

Jouer un fichier MIDI

Les fichiers MIDI peuvent être lus en utilisant plusieurs classes du package `javax.sound.midi` . Un `Sequencer` effectue la lecture du fichier MIDI et plusieurs de ses méthodes peuvent être utilisées pour définir des commandes de lecture telles que le nombre de boucles, le tempo, la mise en sourdine et autres.

La lecture générale des données MIDI peut être effectuée de cette manière:

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

Pour arrêter la lecture, utilisez:

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

Un séquenceur peut être configuré pour mettre en sourdine une ou plusieurs pistes de la séquence pendant la lecture, de sorte qu'aucun des instruments de la lecture spécifiée ne soit lu. L'exemple suivant définit la première piste de la séquence à mettre en sourdine:

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

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

Un séquenceur peut jouer une séquence à plusieurs reprises si le compte de boucle est donné. Ce qui suit permet au séquenceur de jouer une séquence quatre fois et indéfiniment:

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

Le séquenceur ne doit pas toujours jouer la séquence depuis le début, ni jouer la séquence jusqu'à la fin. Il peut commencer et se terminer à tout moment en spécifiant la *coche* dans la séquence pour commencer et se terminer. Il est également possible de spécifier manuellement quelle coche dans l'ordre dans lequel le séquenceur doit jouer:

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

Les séquenceurs peuvent également lire un fichier MIDI à un certain tempo, qui peut être contrôlé en spécifiant le tempo en battements par minute (BPM) ou en microsecondes par quart de note (MPQ). Le facteur auquel la séquence est jouée peut également être ajusté.

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

Lorsque vous avez fini d'utiliser le `Sequencer` , rappelez-vous de le fermer

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

Son metal nu

Vous pouvez également aller presque nu-métal lors de la production de son avec Java. Ce code écrira des données binaires brutes dans le tampon audio du système d'exploitation pour générer du son. Il est extrêmement important de comprendre les limites et les calculs nécessaires pour générer un son comme celui-ci. La lecture étant essentiellement instantanée, les calculs doivent être effectués presque en temps réel.

En tant que telle, cette méthode est inutilisable pour un échantillonnage sonore plus compliqué. À cette fin, utiliser des outils spécialisés est la meilleure approche.

La méthode suivante génère et génère directement une onde rectangulaire d'une fréquence donnée dans un volume donné pour une durée donnée.

```
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
}
```

Pour une manière plus différenciée de générer différentes ondes sonores, des calculs sinusaux et éventuellement des tailles d'échantillon plus importantes sont nécessaires. Cela se traduit par un code beaucoup plus complexe et est donc omis ici.

Sortie audio de base

Le bonjour audio! de Java qui joue un fichier audio à partir de stockage local ou Internet ressemble à ceci. Il fonctionne pour les fichiers .wav non compressés et ne doit pas être utilisé pour lire des fichiers MP3 ou compressés.

```
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();
   }
}
```

Lire l'audio en ligne: https://riptutorial.com/fr/java/topic/160/l-audio

Chapitre 109: La classe java.util.Objects

Examples

Utilisation basique de la vérification d'objet null

Pour la méthode d'archivage nul

```
Object nullableObject = methodReturnObject();
if (Objects.isNull(nullableObject)) {
    return;
}
```

Pour la méthode d'archivage non nulle

```
Object nullableObject = methodReturnObject();
if (Objects.nonNull(nullableObject)) {
    return;
}
```

Objects.nonNull () référence de la méthode use dans api stream

À l'ancienne façon pour la collection null check

```
List<Object> someObjects = methodGetList();
for (Object obj : someObjects) {
    if (obj == null) {
        continue;
    }
    doSomething(obj);
}
```

Avec la méthode `Objects.nonNull` et l'API Java8 Stream, nous pouvons procéder comme suit:

```
List<Object> someObjects = methodGetList();
someObjects.stream()
           .filter(Objects::nonNull)
           .forEach(this::doSomething);
```

Lire La classe java.util.Objects en ligne: https://riptutorial.com/fr/java/topic/5768/la-classe-java-util-objects

Chapitre 110: La commande Java - "java" et "javaw"

Syntaxe

- `java [ <opt> ... ] <class-name> [ <argument> ... ]`
- `java [ <opt> ... ] -jar <jar-file-pathname> [ <argument> ... ]`

Remarques

La commande `java` est utilisée pour exécuter une application Java à partir de la ligne de commande. Il est disponible dans le cadre de tout Java SE JRE ou JDK.

Sur les systèmes Windows, il existe deux variantes de la commande `java` :

- La variante `java` lance l'application dans une nouvelle fenêtre de console.
- La variante `javaw` lance l'application sans créer de nouvelle fenêtre de console.

Sur les autres systèmes (par exemple Linux, Mac OSX, UNIX), seule la commande `java` est fournie et ne lance pas de nouvelle fenêtre de console.

Le symbole `<opt>` dans la syntaxe indique une option sur la ligne de commande `java` . Les rubriques "Options Java" et "Options de dimensionnement du tas et de la pile" couvrent les options les plus courantes. D'autres sont abordés dans la rubrique Indicateurs JVM .

Examples

Exécution d'un fichier JAR exécutable

Les fichiers JAR exécutables sont le moyen le plus simple d'assembler du code Java en un seul fichier pouvant être exécuté. * (Note éditoriale: la création de fichiers JAR doit être couverte par une rubrique distincte.) *

En supposant que vous ayez un fichier JAR exécutable avec le chemin `<jar-path>` , vous devriez pouvoir l'exécuter comme suit:

```
java -jar <jar-path>
```

Si la commande nécessite des arguments de ligne de commande, ajoutez-les après le `<jar-path>` . Par exemple:

```
java -jar <jar-path> arg1 arg2 arg3
```

Si vous devez fournir des options de JVM supplémentaires sur la `java` ligne de commande, ils

doivent aller *avant* la `-jar` option. Notez qu'un `-cp` / `-classpath` option sera ignorée si vous utilisez `-jar` . Le chemin de classe de l'application est déterminé par le manifeste du fichier JAR.

Exécution d'une application Java via une classe "principale"

Lorsqu'une application n'a pas été empaquetée en tant que fichier JAR exécutable, vous devez indiquer le nom d'une classe de point d'entrée sur la ligne de commande `java` .

Lancer la classe HelloWorld

L'exemple "HelloWorld" est décrit dans la section Création d'un nouveau programme Java . Il consiste en une classe unique appelée `HelloWorld` qui répond aux exigences d'un point d'entrée.

En supposant que le fichier (compilé) "HelloWorld.class" se trouve dans le répertoire actuel, il peut être lancé comme suit:

```
java HelloWorld
```

Certaines choses importantes à noter sont:

- Nous devons fournir le nom de la classe: pas le chemin d'accès du fichier ".class" ou du fichier ".java".
- Si la classe est déclarée dans un package (comme le sont la plupart des classes Java), le nom de classe que nous fournissons à la commande `java` doit être le nom de classe complet. Par exemple, si `SomeClass` est déclaré dans le package `com.example` , le nom de classe complet sera `com.example.SomeClass` .

Spécifier un classpath

À moins d'utiliser la syntaxe de la commande `java -jar` , la commande `java` recherche la classe à charger en effectuant une recherche dans le chemin de classe; voir le Classpath . La commande ci-dessus repose sur le chemin de classe par défaut étant (ou incluant) le répertoire en cours. Nous pouvons être plus explicites à ce sujet en spécifiant le classpath à utiliser avec l'option `-cp` .

```
java -cp . HelloWorld
```

Cela dit pour faire le répertoire courant (qui est ce que "." Se réfère à) la seule entrée sur le classpath.

Le `-cp` est une option qui est traitée par la commande `java` . Toutes les options destinées à la commande `java` doivent être avant le nom de classe. Tout ce qui suit la classe sera traité comme un argument de ligne de commande pour l'application Java et sera transmis à l'application dans la `String[]` transmise à la méthode `main` .

(Si aucune option `-cp` n'est fournie, le `java` utilisera le classpath qui est fourni par la `CLASSPATH` environnement `CLASSPATH` . Si cette variable n'est pas définie ou vide, `java` utilise "." Comme chemin de classe par défaut.)

Classes de points d'entrée

Une classe de point d'entrée Java possède une méthode `main` avec la signature et les modificateurs suivants:

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

> Sidenote: à cause du fonctionnement des tableaux, cela peut aussi être `(String args[])`

Lorsque la commande `java` démarre la machine virtuelle, elle charge les classes de points d'entrée spécifiées et tente de trouver la classe `main` . En cas de succès, les arguments de la ligne de commande sont convertis en objets Java `String` et assemblés en un tableau. Si `main` est appelée comme ceci, le tableau *ne* sera *pas* `null` et ne contiendra aucune entrée `null` .

Une méthode de classe de point d'entrée valide doit effectuer les opérations suivantes:

- Être nommé `main` (sensible à la casse)
- Être `public` et `static`
- Avoir un type de retour `void`
- Avoir un seul argument avec un tableau `String[]` . L'argument doit être présent et pas plus d'un argument est autorisé.
- Soyez générique: les paramètres de type ne sont pas autorisés.
- Avoir une classe englobante non générique, de premier niveau (non imbriquée ou interne)

Il est classique de déclarer la classe comme `public` mais ce n'est pas strictement nécessaire. A partir de Java 5, le type d'argument de la méthode `main` peut être un `String` varargs au lieu d'un tableau de chaînes. `main` peut éventuellement jeter des exceptions, et son paramètre peut être nommé n'importe quoi, mais conventionnellement, il est `args` .

Points d'entrée JavaFX

A partir de Java 8, la commande `java` peut également lancer directement une application JavaFX. JavaFX est documenté dans la balise JavaFX , mais un point d'entrée JavaFX doit effectuer les opérations suivantes:

- Étendre `javafx.application.Application`
- Être `public` et non `abstract`
- Ne pas être générique ou imbriqué
- Avoir un constructeur `public` explicite ou implicite no-args

Dépannage de la commande 'java'

Cet exemple couvre les erreurs courantes liées à l'utilisation de la commande 'java'.

"Commande non trouvée"

Si vous obtenez un message d'erreur tel que:

```
java: command not found
```

en essayant d'exécuter la commande `java` , cela signifie qu'il n'y a pas de commande `java` sur le chemin de recherche de commandes de votre shell. La cause pourrait être:

- vous n'avez pas de Java JRE ou JDK installé,
- vous n'avez pas mis à jour la variable d'environnement `PATH` (correctement) dans votre fichier d'initialisation de shell, ou
- vous n'avez pas "trouvé" le fichier d'initialisation correspondant dans le shell actuel.

Reportez-vous à la section "Installation de Java" pour connaître les étapes à suivre.

"Impossible de trouver ou de charger la classe principale"

Ce message d'erreur est généré par la commande `java` s'il n'a pas pu trouver / charger la classe de point d'entrée que vous avez spécifiée. En termes généraux, trois raisons principales peuvent expliquer ce phénomène:

- Vous avez spécifié une classe de point d'entrée qui n'existe pas.
- La classe existe, mais vous l'avez spécifiée de manière incorrecte.
- La classe existe et vous l'avez spécifiée correctement, mais Java ne peut pas la trouver car le classpath est incorrect.

Voici une procédure pour diagnostiquer et résoudre le problème:

1. Découvrez le nom complet de la classe de point d'entrée.

    - Si vous avez un code source pour une classe, le nom complet comprend le nom du package et le nom de la classe simple. L'instance dont la classe "Main" est déclarée dans le package "com.example.myapp", puis son nom complet est "com.example.myapp.Main".
    - Si vous avez un fichier de classe compilé, vous pouvez trouver le nom de la classe en lançant `javap` .
    - Si le fichier de classe se trouve dans un répertoire, vous pouvez déduire le nom complet de la classe des noms de répertoire.
    - Si le fichier de classe se trouve dans un fichier JAR ou ZIP, vous pouvez déduire le nom de classe complet du chemin du fichier JAR ou ZIP.

2. Regardez le message d'erreur de la commande `java` . Le message doit se terminer par le nom de classe complet que `java` essaie d'utiliser.

    - Vérifiez qu'il correspond exactement au nom de classe complet de la classe de point d'entrée.
    - Il ne devrait pas se terminer par ".java" ou ".class".
    - Il ne doit pas contenir de barres obliques ou d'autres caractères non légaux dans un identifiant Java [1] .

- L'enveloppe du nom doit correspondre exactement au nom complet de la classe.

3. Si vous utilisez le bon nom de classe, assurez-vous que la classe est bien sur le chemin de classe:

    - Examinez le chemin d'accès auquel le nom de classe correspond; voir Mappage des noms de classe sur les chemins d'accès
    - Examinez ce qu'est le classpath; voir cet exemple: Différentes manières de spécifier le classpath
    - Examinez chacun des fichiers JAR et ZIP du chemin de classe pour voir s'ils contiennent une classe avec le chemin d'accès requis.
    - Examinez chaque répertoire pour voir si le chemin d'accès se résout en un fichier dans le répertoire.

Si la vérification manuelle du `-Xdiag -XshowSettings` n'a pas trouvé le problème, vous pouvez ajouter les options `-Xdiag` et `-XshowSettings` . La première liste toutes les classes chargées et la seconde affiche les paramètres qui incluent le chemin de classe effectif pour la machine virtuelle Java.

Enfin, il existe *des* causes *obscures* à ce problème:

- Un fichier JAR exécutable avec un attribut `Main-Class` qui spécifie une classe qui n'existe pas.
- Un fichier JAR exécutable avec un attribut `Class-Path` incorrect.
- Si vous gâcher [2] les options avant le nom de classe, la `java` commande peut tenter d'interpréter l' un d'entre eux comme le nom de classe.
- Si quelqu'un a ignoré les règles de style Java et utilisé des identifiants de classe ou de package qui ne diffèrent que par la casse des lettres et que vous utilisez une plate-forme qui considère les noms de fichiers des lettres comme non significatifs.
- Problèmes avec les homoglyphes dans les noms de classe dans le code ou sur la ligne de commande.

"Méthode principale introuvable dans la classe <nom>"

Ce problème se produit lorsque la commande `java` est capable de trouver et de charger la classe que vous avez nommée, mais est ensuite incapable de trouver une méthode de point d'entrée.

Il y a trois explications possibles:

- Si vous essayez d'exécuter un fichier JAR exécutable, le manifeste du fichier JAR a un attribut "Main-Class" incorrect qui spécifie une classe qui n'est pas une classe de point d'entrée valide.
- Vous avez dit à la commande `java` une classe qui n'est pas une classe de point d'entrée.
- La classe de point d'entrée est incorrecte. voir Classes de point d'entrée pour plus d'informations.

Autres ressources

- Que signifie "Impossible de trouver ou de charger la classe principale"?
- http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html

---

1 - A partir de Java 8 et versions ultérieures, la commande `java` mappera utilement un séparateur de nom de fichier ("/" ou "") sur un point ("."). Toutefois, ce comportement n'est pas documenté dans les pages de manuel.

2 - Si vous copiez et collez une commande à partir d'un document formaté dans lequel l'éditeur de texte a utilisé un "trait d'union long" au lieu d'un trait d'union régulier, cela est très obscur.

Exécution d'une application Java avec des dépendances de bibliothèque

Ceci est une continuation des exemples "main class" et "executable JAR" .

Les applications Java classiques consistent en un code spécifique à l'application et en divers codes de bibliothèque réutilisables que vous avez implémentés ou mis en œuvre par des tiers. Ces derniers sont généralement appelés dépendances de bibliothèque et sont généralement regroupés sous forme de fichiers JAR.

Java est un langage lié dynamiquement. Lorsque vous exécutez une application Java avec des dépendances de bibliothèque, la machine virtuelle Java doit savoir où se trouvent les dépendances pour pouvoir charger les classes selon vos besoins. D'une manière générale, il y a deux manières de traiter cela:

- L'application et ses dépendances peuvent être regroupées en un seul fichier JAR contenant toutes les classes et ressources requises.
- On peut dire à la JVM où trouver les fichiers JAR dépendants via le classpath d'exécution.

Pour un fichier JAR exécutable, le chemin de classe d'exécution est spécifié par l'attribut de manifeste "Class-Path". *(Note éditoriale: ceci doit être décrit dans une rubrique distincte sur la commande `jar` .)* Sinon, le classpath d'exécution doit être fourni à l'aide de l'option `-cp` ou de la `CLASSPATH` environnement `CLASSPATH` .

Par exemple, supposons que nous ayons une application Java dans le fichier "myApp.jar" dont la classe de point d'entrée est `com.example.MyApp` . Supposons également que l'application dépend des fichiers JAR de la bibliothèque "lib / library1.jar" et "lib / library2.jar". Nous pourrions lancer l'application en utilisant la commande `java` comme suit dans une ligne de commande:

```
$ # 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
```

(Sous Windows, vous utiliseriez `;` au lieu de `:` tant que séparateur de chemin de classe, et vous `CLASSPATH` variable `CLASSPATH` (locale) à l'aide de `set` plutôt que d' `export` .)

Bien qu'un développeur Java soit à l'aise avec cela, il n'est pas "convivial". Il est donc courant d'écrire un script shell simple (ou un fichier de commandes Windows) pour masquer les détails

que l'utilisateur n'a pas besoin de connaître. Par exemple, si vous placez le script shell suivant dans un fichier appelé "myApp", exécutez-le et placez-le dans un répertoire du chemin de recherche:

```
#!/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
```

alors vous pourriez l'exécuter comme suit:

```
$ myApp arg1 arg2 ...
```

Tous les arguments sur la ligne de commande seront transmis à l'application Java via l'extension `"$@"` . (Vous pouvez faire quelque chose de similaire avec un fichier de commandes Windows, bien que la syntaxe soit différente.)

Espaces et autres caractères spéciaux dans les arguments

Tout d'abord, le problème de la gestion des espaces dans les arguments n'est PAS un problème Java. Il s'agit plutôt d'un problème qui doit être traité par le shell de commandes que vous utilisez lorsque vous exécutez un programme Java.

À titre d'exemple, supposons que nous ayons le programme simple suivant qui imprime la taille d'un fichier:

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

Maintenant, supposons que nous voulions imprimer la taille d'un fichier dont le chemin contient des espaces; par exemple `/home/steve/Test File.txt` . Si nous exécutons la commande comme ceci:

```
$ java PrintFileSizes /home/steve/Test File.txt
```

le shell ne saura pas que `/home/steve/Test File.txt` est en fait un chemin d'accès. Au lieu de cela, il transmettra 2 arguments distincts à l'application Java, qui tentera de trouver leur taille de fichier respective, et échouera car les fichiers avec ces chemins (probablement) n'existent pas.

Solutions utilisant un shell POSIX

Coquilles comprennent POSIX `sh` sous forme de dérivés bien tels que `bash` et `ksh` . Si vous utilisez l'un de ces shells, vous pouvez résoudre le problème en *citant* l'argument.

```
$ java PrintFileSizes "/home/steve/Test File.txt"
```

Les guillemets autour du nom de chemin indiquent au shell qu'il doit être transmis en un seul argument. Les citations seront supprimées lorsque cela se produira. Il y a plusieurs autres façons de le faire:

```
$ java PrintFileSizes '/home/steve/Test File.txt'
```

Les guillemets simples (droits) sont traités comme des guillemets, sauf qu'ils suppriment également diverses extensions dans l'argument.

```
$ java PrintFileSizes /home/steve/Test\ File.txt
```

Une barre oblique inverse échappe à l'espace suivant et l'empêche d'être interprétée comme un séparateur d'arguments.

Pour une documentation plus complète, y compris des descriptions sur la manière de traiter d'autres caractères spéciaux dans les arguments, veuillez vous reporter à la rubrique de citation dans la documentation de Bash .

Solution pour Windows

Le problème fondamental de Windows est qu'au niveau du système d'exploitation, les arguments sont transmis à un processus enfant sous la forme d'une chaîne unique (source). Cela signifie que la responsabilité ultime de l'analyse (ou de la nouvelle analyse) de la ligne de commande incombe au programme ou à ses bibliothèques d'exécution. Il y a beaucoup d'incohérence.

Dans le cas de Java, pour faire court:

- Vous pouvez placer des guillemets doubles autour d'un argument dans une commande `java` , ce qui vous permettra de transmettre des arguments avec des espaces.
- Apparemment, la commande `java` elle-même analyse la chaîne de commande
- Cependant, lorsque vous essayez de combiner cela avec l'utilisation de `SET` et de la substitution de variables dans un fichier de commandes, il devient vraiment compliqué de savoir si les guillemets doubles sont supprimés.
- Le shell `cmd.exe` a apparemment d'autres mécanismes d'échappement; par exemple, doubler les guillemets doubles et utiliser `^` échappent.

Pour plus de détails, reportez-vous à la documentation de Batch-File .

Options Java

La commande `java` prend en charge un large éventail d'options:

- Toutes les options commencent par un seul trait d'union ou un signe moins (`-`): la convention GNU / Linux consistant à utiliser `--` pour les options "long" n'est pas prise en charge.
- Les options doivent apparaître avant la `<classname>` ou le `-jar <jarfile>` argument pour être reconnu. Tous les arguments après ceux-ci seront traités comme des arguments à transmettre à l'application Java en cours d'exécution.
- Les options qui ne commencent pas par `-X` ou `-XX` sont des options standard. Vous pouvez compter sur toutes les implémentations Java [1] pour prendre en charge toute option standard.
- Les options commençant par `-X` sont des options non standard et peuvent être retirées d'une version Java à l'autre.
- Les options commençant par `-XX` sont des options avancées et peuvent également être retirées.

Définition des propriétés du système avec `-D`

L'option `-D<property>=<value>` est utilisée pour définir une propriété dans l'objet `Properties` du système. Ce paramètre peut être répété pour définir différentes propriétés.

Options de mémoire, d'empilage et de récupération de mémoire

Les principales options de contrôle du tas et de la taille des piles sont documentées dans la section Définition des tailles de tas, PermGen et Stack . *(Note de la rédaction: les options du récupérateur de place doivent être décrites dans le même sujet.)*

Activation et désactivation des assertions

Les options `-ea` et `-da` activent et désactivent respectivement la vérification Java `assert` :

- Toute vérification des assertions est désactivée par défaut.
- L'option `-ea` permet de vérifier toutes les assertions
- Le `-ea:<packagename>...` permet de vérifier les assertions dans un package *et tous les sous-packages* .
- Le `-ea:<classname>...` permet de vérifier les assertions d'une classe.
- L'option `-da` désactive la vérification de toutes les assertions
- Le `-da:<packagename>...` désactive la vérification des assertions dans un paquet *et dans tous les sous-paquets* .

- `-da:<classname>...` désactive la vérification des assertions dans une classe.
- L'option `-esa` permet de vérifier toutes les classes de système.
- L'option `-dsa` désactive la vérification pour toutes les classes système.

Les options peuvent être combinées. Par exemple.

```
$ # 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
```

Notez que l'activation de la vérification d'assertion est susceptible de modifier le comportement d'une programmation Java.

- Il est responsable de rendre l'application plus lente en général.
- Cela peut entraîner des méthodes plus longues à exécuter, ce qui peut changer la synchronisation des threads dans une application multithread.
- Il peut introduire des relations entre événements par hasard *et* provoquer des anomalies de la mémoire.
- Une mauvaise application `assert` déclaration pourrait avoir des effets secondaires indésirables.

Sélection du type de machine virtuelle

Les options `-client` et `-server` vous permettent de sélectionner deux formes différentes de la machine virtuelle HotSpot:

- Le formulaire "client" est adapté aux applications utilisateur et offre un démarrage plus rapide.
- Le formulaire "serveur" est conçu pour des applications de longue durée. Il faut plus de temps pour capturer les statistiques pendant le "warm-up" de la JVM, ce qui permet au compilateur JIT d'optimiser le code natif.

Par défaut, la JVM fonctionnera en mode 64 bits si possible, en fonction des capacités de la plate-forme. Les options `-d32` et `-d64` vous permettent de sélectionner le mode explicitement.

---

1 - Consultez le manuel officiel de la commande `java` . Parfois, une option *standard* est décrite comme "susceptible de changer".

Lire La commande Java - "java" et "javaw" en ligne: https://riptutorial.com/fr/java/topic/5791/la-commande-java----java--et--javaw-

Chapitre 111: La mise en réseau

Syntaxe

- nouveau Socket ("localhost", 1234); // Se connecte à un serveur à l'adresse "localhost" et au port 1234
- nouveau SocketServer ("localhost", 1234); // Crée un serveur de socket capable d'écouter les nouvelles sockets à l'adresse localhost et au port 1234
- socketServer.accept (); // Accepte un nouvel objet Socket qui peut être utilisé pour communiquer avec le client

Examples

Communication de base entre le client et le serveur à l'aide d'un socket

Serveur: démarrer et attendre les connexions entrantes

```
//Open a listening "ServerSocket" on port 1234.
ServerSocket serverSocket = new ServerSocket(1234);

while (true) {
    // Wait for a client connection.
    // Once a client connected, we get a "Socket" object
    // that can be used to send and receive messages to/from the newly
    // connected client
    Socket clientSocket = serverSocket.accept();

    // Here we'll add the code to handle one specific client.
}
```

Serveur: Gestion des clients

Nous traiterons chaque client dans un thread séparé afin que plusieurs clients puissent interagir avec le serveur en même temps. Cette technique fonctionne bien tant que le nombre de clients est faible (<< 1000 clients, selon l'architecture du système d'exploitation et la charge attendue de chaque 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: Connectez-vous au serveur et envoyez un message

```
// 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();
```

Fermeture des sockets et gestion des exceptions

Les exemples ci-dessus ont laissé de côté certaines choses pour les rendre plus lisibles.

1. Tout comme les fichiers et les autres ressources externes, il est important de dire au système d'exploitation lorsque nous en avons fini avec eux. Lorsque vous avez terminé avec un socket, appelez `socket.close()` pour le fermer correctement.

2. Les sockets gèrent les opérations d'entrée / sortie (E / S) qui dépendent de divers facteurs externes. Par exemple, si l'autre côté se déconnecte soudainement? Que se passe-t-il s'il y a une erreur de réseau? Ces choses sont hors de notre contrôle. C'est la raison pour laquelle de nombreuses opérations de socket peuvent générer des exceptions, en particulier `IOException` .

Un code plus complet pour le client serait donc quelque chose comme ceci:

```
// "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
}
```

Serveur et client de base - exemples complets

Serveur:

```
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();
        }

    }
}
```

Client:

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

Chargement de DOSSIERORE et de KeyStore depuis 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();
            }

    }
}
```

Utiliser un KeyStore fonctionne de la même manière, sauf remplacer n'importe quel mot `Trust` dans un nom d'objet par `Key` . De plus, le `KeyManager[]` doit être transmis au premier argument de `SSLContext.init` . C'est `SSLContext.init(keyMangers, trustMangers, null)`

Exemple de socket - lecture d'une page Web à l'aide d'un socket simple

```
import java.io.*;
import java.net.Socket;

public class Main {

    public static void main(String[] args) throws IOException {//We don't handle Exceptions in
this example
        //Open a socket to stackoverflow.com, port 80
        Socket socket = new Socket("stackoverflow.com",80);

        //Prepare input, output stream before sending request
        OutputStream outStream = socket.getOutputStream();
        InputStream inStream = socket.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(inStream));
        PrintWriter writer = new PrintWriter(new BufferedOutputStream(outStream));

        //Send a basic HTTP header
        writer.print("GET / HTTP/1.1\nHost:stackoverflow.com\n\n");
        writer.flush();

        //Read the response
        System.out.println(readFully(reader));

        //Close the socket
        socket.close();
    }

    private static String readFully(Reader in) {
        StringBuilder sb = new StringBuilder();
        int BUFFER_SIZE=1024;
        char[] buffer = new char[BUFFER_SIZE]; // or some other size,
        int charsRead = 0;
        while ( (charsRead  = rd.read(buffer, 0, BUFFER_SIZE)) != -1) {
          sb.append(buffer, 0, charsRead);
        }
    }
}
```

Vous devriez obtenir une réponse commençant par `HTTP/1.1 200 OK` , qui indique une réponse HTTP normale, suivie du reste de l'en-tête HTTP, suivie de la page Web brute au format HTML.

Notez que la méthode `readFully()` est importante pour éviter une exception EOF prématurée. La dernière ligne de la page Web manque peut-être un retour, pour signaler la fin de la ligne, alors `readLine()` se plaindra, donc il faut le lire à la main ou utiliser des méthodes utilitaires d' Apache commons-io IOUtils

Cet exemple se veut une simple démonstration de connexion à une ressource existante à l'aide d'un socket, ce n'est pas un moyen pratique d'accéder aux pages Web. Si vous devez accéder à une page Web à l'aide de Java, il est préférable d'utiliser une bibliothèque de clients HTTP existante, telle que le client HTTP d'Apache ou le client HTTP de Google.

Communication basique client / serveur via 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);
    }
}
```

Dans ce cas, on passe l'adresse du serveur, via un argument (`args[0]`). Le port que nous utilisons est 4160.

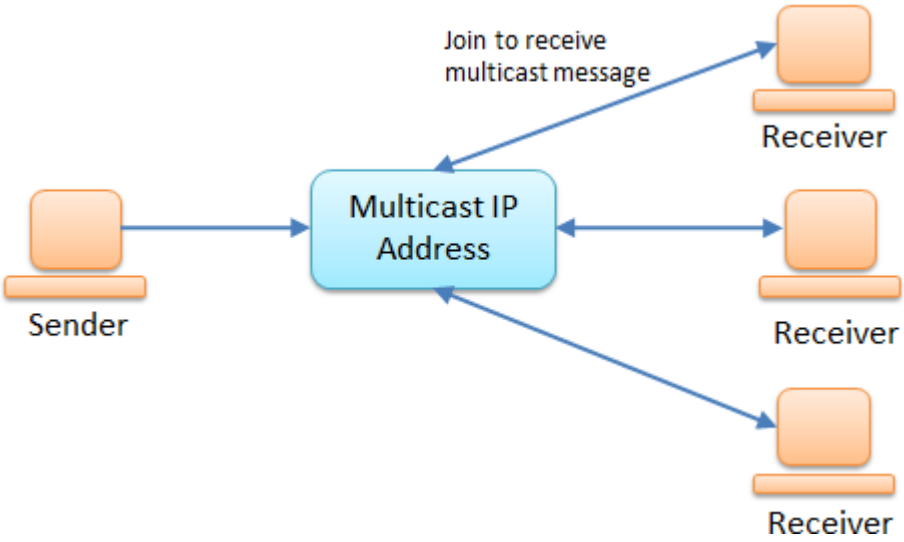
Server.java

```
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);
    }
}
```

Du côté du serveur, déclarez un DatagramSocket sur le même port auquel nous avons envoyé notre message (4160) et attendez une réponse.

Multicast

La multidiffusion est un type de socket de datagramme. Contrairement aux datagrammes classiques, la multidiffusion ne gère pas chaque client individuellement, mais l'envoie à une adresse IP et tous les clients abonnés recevront le message.



Exemple de code pour un côté serveur:

```
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();
    }
}
```

Exemple de code pour un 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();
    }
}
```

Code pour exécuter le serveur:

```
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();
    }
}
```

Code pour exécuter un client:

```
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();
    }
}
```

Exécuter le client d'abord: le client doit s'abonner à l'IP avant de pouvoir commencer à recevoir des paquets. Si vous démarrez le serveur et appelez la méthode `send()` , puis créez un client (& appelez `printMessage()`). Rien ne se passera car le client est connecté après l'envoi du message.

Désactivez temporairement la vérification SSL (à des fins de test)

Parfois, dans un environnement de développement ou de test, la chaîne de certificats SSL n'a peut-être pas encore été entièrement établie.

Pour continuer à développer et à tester, vous pouvez désactiver la vérification SSL par programme en installant un gestionnaire de confiance "fiable":

```
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();
}
```

Télécharger un fichier en utilisant Channel

Si le fichier existe déjà, il sera écrasé!

```
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 */ }
```

Remarques

- Ne laissez pas les blocs de capture vides!
- En cas d'erreur, vérifiez si le fichier distant existe
- Ceci est une opération de blocage, peut prendre beaucoup de temps avec des fichiers volumineux

Lire La mise en réseau en ligne: https://riptutorial.com/fr/java/topic/149/la-mise-en-reseau

Chapitre 112: La sérialisation

Introduction

Java fournit un mécanisme, appelé sérialisation d'objet, dans lequel un objet peut être représenté sous la forme d'une séquence d'octets incluant les données de l'objet, ainsi que des informations sur le type de l'objet et les types de données stockés dans l'objet.

Une fois qu'un objet sérialisé a été écrit dans un fichier, il peut être lu à partir du fichier et désérialisé, ce qui signifie que les informations de type et les octets qui représentent l'objet et ses données peuvent être utilisés pour recréer l'objet en mémoire.

Examples

Sérialisation de base en Java

Qu'est-ce que la sérialisation

La sérialisation est le processus consistant à convertir l'état d'un objet (y compris ses références) en une séquence d'octets, ainsi que le processus de reconstruction de ces octets en un objet actif à une date ultérieure. La sérialisation est utilisée lorsque vous souhaitez conserver l'objet. Il est également utilisé par Java RMI pour transmettre des objets entre JVM, soit en tant qu'arguments dans une invocation de méthode depuis un client vers un serveur, soit en tant que valeurs de retour depuis une invocation de méthode, soit en tant qu'exceptions lancées par des méthodes distantes. En général, la sérialisation est utilisée lorsque nous voulons que l'objet existe au-delà de la durée de vie de la machine virtuelle Java.

`java.io.Serializable` est une interface de marqueur (sans corps). Il est juste utilisé pour "marquer" les classes Java comme sérialisables.

Le runtime de sérialisation associe à chaque classe sérialisable un numéro de version, appelé `serialVersionUID` , utilisé lors *de la* désérialisation pour vérifier que l'expéditeur et le destinataire d'un objet sérialisé ont des classes chargées pour cet objet compatibles avec la sérialisation. Si le récepteur a chargé une classe pour l'objet ayant un `serialVersionUID` différent de celui de la classe de l'expéditeur correspondant, la désérialisation entraînera une `InvalidClassException` . Une classe sérialisable peut déclarer explicitement son propre `serialVersionUID` en déclarant un champ nommé `serialVersionUID` qui doit être `static, final,` et de type `long` :

```
ANY-ACCESS-MODIFIER static final long serialVersionUID = 1L;
```

Comment rendre une classe éligible pour la sérialisation

Pour conserver un objet, la classe respective doit implémenter l'interface `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;
    }
}
```

Comment écrire un objet dans un fichier

Maintenant, nous devons écrire cet objet dans un système de fichiers. Nous utilisons `java.io.ObjectOutputStream` à cette fin.

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

Comment recréer un objet à partir de son état sérialisé

L'objet stocké peut être lu à partir du système de fichiers plus tard en utilisant `java.io.ObjectInputStream` comme indiqué ci-dessous:

```
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 sérialisée est sous forme binaire. La désérialisation peut être problématique si la définition de classe change: voir le chapitre Gestion des versions des objets sérialisés de la spécification de sérialisation Java pour plus de détails.

La sérialisation d'un objet sérialise l'ensemble du graphe d'objet dont il est la racine et fonctionne correctement en présence de graphes cycliques. Une méthode `reset()` est fournie pour forcer `ObjectOutputStream` à oublier les objets qui ont déjà été sérialisés.

Champs transitoires - Sérialisation

Sérialisation avec Gson

La sérialisation avec Gson est facile et produira un JSON correct.

```
public class Employe {

    private String firstName;
    private String lastName;
    private int age;
    private BigDecimal salary;
    private List<String> skills;

    //getters and setters
}
```

(Sérialisation)

```
//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"]}
```

Notez que vous ne pouvez pas sérialiser les objets avec des références circulaires car cela se traduira par une récursion infinie.

(Désérialisation)

```
//it's very simple...
//Assuming that json is the previous String object....

Employe obj2 = gson.fromJson(json, Employe.class); // obj2 is just like obj
```

Sérialisation avec Jackson 2

Voici une implémentation qui montre comment un objet peut être sérialisé dans sa chaîne JSON correspondante.

```
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;
    }
}
```

Sérialisation:

```
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
}
```

Sortie:

```
{
  "idx" : 1,
  "name" : "abc"
}
```

Vous pouvez omettre l'imprimante Pretty Default si vous n'en avez pas besoin.

La dépendance utilisée ici est la suivante:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.6.3</version>
</dependency>
```

Sérialisation personnalisée

Dans cet exemple, nous voulons créer une classe qui générera et sortira en console, un nombre aléatoire compris entre une plage de deux nombres entiers qui seront transmis en tant qu'arguments lors de l'initialisation.

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

Maintenant, si nous voulons rendre cette classe sérialisable, il y aura des problèmes. Le thread est l'une des classes de niveau système qui ne sont pas Serializable. Nous devons donc déclarer le thread comme **transitoire** . En faisant cela, nous serons en mesure de sérialiser les objets de cette classe mais nous aurons toujours un problème. Comme vous pouvez le voir dans le

constructeur, nous définissons les valeurs min et max de notre randomiseur et ensuite nous démarrons le thread qui est responsable de la génération et de l'impression de la valeur aléatoire. Ainsi, lors de la restauration de l'objet persistant en appelant le ***readObject (),*** le constructeur ne sera plus exécuté car il n'y a pas de création d'un nouvel objet. Dans ce cas, nous devons développer une **sérialisation personnalisée** en fournissant deux méthodes à l'intérieur de la classe. Ces méthodes sont:

```
private void writeObject(ObjectOutputStream out) throws IOException;
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

Ainsi, en ajoutant notre implémentation dans ***readObject (),*** nous pouvons lancer et démarrer notre thread:

```
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();
}
}
```

Voici le principal pour notre exemple:

```
public class Main {
public static void main(String[] args) {
    System.out.println("Hello");
    RangeRandom rangeRandom = new RangeRandom(1,10);

    FileOutputStream fos = null;
    ObjectOutputStream out = null;
    try
    {
        fos = new FileOutputStream("test");
        out = new ObjectOutputStream(fos);
        out.writeObject(rangeRandom);
        out.close();
    }
    catch(IOException ex)
    {
        ex.printStackTrace();
    }


    RangeRandom rangeRandom2 = null;
       FileInputStream fis = null;
       ObjectInputStream in = null;
       try
       {
             fis = new FileInputStream("test");
             in = new ObjectInputStream(fis);
           rangeRandom2 = (RangeRandom)in.readObject();
             in.close();
           }
       catch(IOException ex)
       {
             ex.printStackTrace();
           }
       catch(ClassNotFoundException ex)
       {
             ex.printStackTrace();
           }

}
}
```

Si vous exécutez la commande principale, vous verrez qu'il y a deux threads en cours d'exécution pour chaque instance RangeRandom, car la méthode ***Thread.start () se*** trouve maintenant à la fois dans le constructeur et dans ***readObject ()*** .

Versioning et serialVersionUID

Lorsque vous implémentez l'interface `java.io.Serializable` pour rendre une classe sérialisable, le compilateur recherche un `static final` nommé `serialVersionUID` de type `long` . Si la classe n'a pas ce champ déclaré explicitement, le compilateur créera un tel champ et l'attribuera avec une valeur qui sort d'un calcul de `serialVersionUID` dépendant de l' `serialVersionUID` . Ce calcul dépend de divers aspects de la classe et suit les spécifications de sérialisation d'objet données par Sun. Cependant, la valeur n'est pas garantie pour toutes les implémentations du compilateur.

Cette valeur est utilisée pour vérifier la compatibilité des classes par rapport à la sérialisation et

cela lors de la désérialisation d'un objet enregistré. Serialization Runtime vérifie que `serialVersionUID` lu à partir des données `serialVersionUID` et que `serialVersionUID` déclaré dans la classe sont exactement les mêmes. Si ce n'est pas le cas, il génère une `InvalidClassException` .

Il est fortement recommandé de déclarer et d'initialiser explicitement le champ statique de type long et nommé 'serialVersionUID' dans toutes les classes que vous souhaitez rendre Serializable au lieu de compter sur le calcul par défaut de la valeur pour ce champ même si vous ne voulez pas utiliser le contrôle de version. **Le calcul 'serialVersionUID' est extrêmement sensible et peut varier d'une implémentation de compilateur à l'autre. Vous pouvez donc obtenir l' `InvalidClassException` même pour la même classe simplement parce que vous avez utilisé différentes implémentations de compilateur.**

```
public class Example implements Serializable {
    static final long serialVersionUID = 1L /*or some other value*/;
    //...
}
```

Tant que `serialVersionUID` est identique, la sérialisation Java peut gérer différentes versions d'une classe. Les changements compatibles et incompatibles sont;

Changements compatibles

- **Ajouter des champs:** Lorsque la classe en cours de reconstitution comporte un champ qui ne figure pas dans le flux, ce champ dans l'objet sera initialisé à la valeur par défaut pour son type. Si une initialisation spécifique à une classe est nécessaire, la classe peut fournir une méthode readObject capable d'initialiser le champ à des valeurs autres que celles par défaut.
- **Ajout de classes:** le flux contiendra la hiérarchie de types de chaque objet du flux. La comparaison de cette hiérarchie dans le flux avec la classe en cours peut détecter des classes supplémentaires. Comme il n'y a aucune information dans le flux à partir de laquelle initialiser l'objet, les champs de la classe seront initialisés aux valeurs par défaut.
- **Suppression de classes: La** comparaison de la hiérarchie de classes dans le flux avec celle de la classe en cours peut détecter qu'une classe a été supprimée. Dans ce cas, les champs et les objets correspondant à cette classe sont lus dans le flux. Les champs primitifs sont ignorés, mais les objets référencés par la classe supprimée sont créés, car ils peuvent être référencés ultérieurement dans le flux. Ils seront récupérés lorsque le flux est récupéré ou réinitialisé.
- **Ajouter des méthodes writeObject / readObject:** Si la version lisant le flux possède ces méthodes, readObject doit, comme d'habitude, lire les données requises écrites dans le flux par la sérialisation par défaut. Il doit d'abord appeler defaultReadObject avant de lire les données facultatives. La méthode writeObject devrait normalement appeler defaultWriteObject pour écrire les données requises, puis écrire des données facultatives.
- **Ajouter java.io.Serializable:** Cela équivaut à ajouter des types. Il n'y aura aucune valeur dans le flux pour cette classe, donc ses champs seront initialisés aux valeurs par défaut. La prise en charge des sous-classes de classes non sérialisables nécessite que le sur-type de la classe ait un constructeur sans argument et que la classe elle-même soit initialisée aux valeurs par défaut. Si le constructeur no-arg n'est pas disponible, l'exception

InvalidClassException est levée.
- **Modification de l'accès à un champ:** Les modificateurs d'accès public, package, protected et private n'ont aucun effet sur la capacité de la sérialisation à affecter des valeurs aux champs.
- **Changer un champ statique en non statique ou transitoire en non-transitoire:** lorsque vous utilisez la sérialisation par défaut pour calculer les champs sérialisables, cette modification équivaut à ajouter un champ à la classe. Le nouveau champ sera écrit dans le flux, mais les classes précédentes ignoreront la valeur car la sérialisation n'affectera pas de valeurs aux champs statiques ou transitoires.

Changements incompatibles

- **Suppression de champs:** Si un champ est supprimé dans une classe, le flux écrit ne contiendra pas sa valeur. Lorsque le flux est lu par une classe antérieure, la valeur du champ est définie sur la valeur par défaut car aucune valeur n'est disponible dans le flux. Cependant, cette valeur par défaut peut compromettre la capacité de la version antérieure à respecter son contrat.
- **Déplacement des classes vers le haut ou le bas de la hiérarchie:** cela ne peut pas être autorisé car les données du flux apparaissent dans la mauvaise séquence.
- **Changer un champ non statique en statique ou un champ non-transitoire en transitoire: en** cas de sérialisation par défaut, cette modification équivaut à supprimer un champ de la classe. Cette version de la classe n'écrira pas ces données dans le flux, elle ne sera donc pas disponible pour être lue par les versions antérieures de la classe. Comme lors de la suppression d'un champ, le champ de la version antérieure sera initialisé à la valeur par défaut, ce qui peut entraîner un échec inattendu de la classe.
- **Modification du type déclaré d'un champ primitif:** Chaque version de la classe écrit les données avec son type déclaré. Les versions antérieures de la classe qui tentent de lire le champ échoueront car le type des données du flux ne correspond pas au type du champ.
- Changer la méthode writeObject ou readObject pour qu'elle n'écrive plus ou ne lise plus les données de champ par défaut ou ne les modifie pas de manière à ce que celle-ci tente de l'écrire ou de la lire lorsque la version précédente ne l'a pas fait. Les données de champ par défaut doivent toujours apparaître ou ne pas apparaître dans le flux.
- Changer une classe de Serializable à Externalizable ou vice versa est un changement incompatible puisque le flux contiendra des données incompatibles avec l'implémentation de la classe disponible.
- Changer une classe d'un type non-enum en un type enum ou vice versa puisque le flux contiendra des données incompatibles avec l'implémentation de la classe disponible.
- La suppression de Serializable ou Externalizable est une modification incompatible car, une fois écrite, elle ne fournira plus les champs requis par les anciennes versions de la classe.
- L'ajout de la méthode writeReplace ou readResolve à une classe est incompatible si le comportement produit un objet incompatible avec une version antérieure de la classe.

Désérialisation JSON personnalisée avec Jackson

Nous consommons l'API de repos en tant que format JSON, puis la désactivons sur un POJO. Le fichier org.codehaus.jackson.map.ObjectMapper de Jackson fonctionne simplement et nous ne

faisons rien dans la plupart des cas. Mais parfois, nous avons besoin d'un désérialiseur personnalisé pour répondre à nos besoins personnalisés et ce tutoriel vous guidera tout au long du processus de création de votre propre désérialiseur.

Disons que nous avons des entités suivantes.

```
public class User {
    private Long id;
    private String name;
    private String email;

    //getter setter are omitted for clarity
}
```

Et

```
public class Program {
    private Long id;
    private String name;
    private User createdBy;
    private String contents;

    //getter setter are omitted for clarity
}
```

Sérialisons / marshalons un objet en premier.

```
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 (programme); System.out.println (json);

Le code ci-dessus produira après JSON-

```
{
    "id": 1,
    "name": "Program @# 1",
    "createdBy": {
        "id": 1,
        "name": "Bazlur Rahman",
        "email": "example@example.com"
    },
    "contents": "Some contents"
}
```

Maintenant, peut faire le contraire très facilement. Si nous avons ce JSON, nous pouvons supprimer un objet programme en utilisant ObjectMapper comme suit -

Maintenant, disons que ce n'est pas le cas réel, nous allons avoir un JSON différent d'une API qui ne correspond pas à notre classe de `Program` .

```
{
"id": 1,
"name": "Program @# 1",
"ownerId": 1
"contents": "Some contents"
}
```

Regardez la chaîne JSON, vous pouvez voir, il a un champ différent qui est owenerId.

Maintenant, si vous souhaitez sérialiser ce JSON comme nous l'avons fait précédemment, vous aurez des exceptions.

Il existe deux manières d'éviter les exceptions et d'avoir cette publication en série -

Ignorer les champs inconnus

Ignorer le `onwerId` . Ajouter l'annotation suivante dans la classe Programme

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Program {}
```

Ecrire un désérialiseur personnalisé

Mais il y a des cas où vous avez réellement besoin de ce champ `owerId` . Disons que vous voulez le relier comme identifiant de la classe `User` .

Dans ce cas, vous devez écrire un désérialiseur personnalisé.

Comme vous pouvez le voir, vous devez d'abord accéder au `JsonNode` depuis le `JonsParser` . Et puis, vous pouvez facilement extraire des informations d'un `JsonNode` utilisant la méthode `get()` . et vous devez vous assurer du nom du champ. Ce devrait être le nom exact, la faute d'orthographe entraînera des exceptions.

Et enfin, vous devez enregistrer votre ProgramDeserializer dans `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);
```

Vous pouvez également utiliser l'annotation pour enregistrer le désérialiseur directement -

```
@JsonDeserialize(using = ProgramDeserializer.class)
public class Program {
}
```

Lire La sérialisation en ligne: https://riptutorial.com/fr/java/topic/767/la-serialisation

Chapitre 113: Le Classpath

Introduction

Le classpath répertorie les endroits où le runtime Java doit rechercher des classes et des ressources. Le classpath est également utilisé par le compilateur Java pour rechercher les dépendances précédemment compilées et externes.

Remarques

Chargement de classe Java

La machine virtuelle Java (Java Virtual Machine) va charger les classes au fur et à mesure que les classes sont requises (cela s'appelle le chargement différé). Les emplacements des classes à utiliser sont spécifiés à trois endroits: -

1. Ceux requis par la plate-forme Java sont chargés en premier, tels que ceux de la bibliothèque de classes Java et de ses dépendances.
2. Les classes d'extension sont chargées ensuite (c'est-à-dire celles de `jre/lib/ext/`)
3. Les classes définies par l'utilisateur via le classpath sont ensuite chargées

Les classes sont chargées à l'aide de classes qui sont des sous-types de `java.lang.ClassLoader` . Ceci décrit plus en détail dans cette rubrique: Chargeurs de classes .

Chemin de classe

Le classpath est un paramètre utilisé par la JVM ou le compilateur qui spécifie les emplacements des classes et des packages définis par l'utilisateur. Cela peut être défini dans la ligne de commande comme avec la plupart de ces exemples ou via une variable d'environnement (`CLASSPATH`)

Examples

Différentes manières de spécifier le classpath

Il existe trois façons de définir le chemin de classe.

1. Il peut être défini à l'aide de la `CLASSPATH` environnement `CLASSPATH` :

```
set CLASSPATH=...          # Windows and csh
export CLASSPATH=...       # Unix ksh/bash
```

2. Il peut être défini sur la ligne de commande comme suit

```
java -classpath ...
```

```
javac -classpath ...
```

Notez que l' `-classpath` (ou `-cp`) est prioritaire sur la `CLASSPATH` environnement `CLASSPATH` .

3. Le `Class-Path` d'un fichier JAR exécutable est spécifié à l'aide de l'élément `Class-Path` dans `MANIFEST.MF` :

```
Class-Path: jar1-name jar2-name directory-name/jar3-name
```

Notez que cela ne s'applique que lorsque le fichier JAR est exécuté comme ceci:

```
java -jar some.jar ...
```

Dans ce mode d'exécution, l'option `-classpath` et la variable d'environnement CLASSPATH seront ignorées, même si le fichier JAR ne contient pas d'élément `Class-Path` .

Si aucun classpath n'est spécifié, le classpath par défaut est le fichier JAR sélectionné lors de l'utilisation de `java -jar` ou du répertoire en cours.

En relation:

- https://docs.oracle.com/javase/tutorial/deployment/jar/downman.html
- http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html

Ajout de tous les JAR dans un répertoire au classpath

Si vous souhaitez ajouter tous les fichiers JAR dans le répertoire classpath, vous pouvez le faire de manière concise à l'aide de la syntaxe de classe de chemin d'accès au classpath; par exemple:

```
someFolder/*
```

Cela indique à la JVM d'ajouter tous les fichiers JAR et ZIP du répertoire `someFolder` au classpath. Cette syntaxe peut être utilisée dans un `-cp` argument un `CLASSPATH` variable d'environnement ou d' une `Class-Path` attribut dans file.See manifeste d'un fichier exécutable JAR *Réglage* de *la classe Chemin: Classe Chemin Jokers* des exemples et des mises en garde.

Remarques:

1. Les caractères génériques du chemin de classe ont été introduits pour la première fois dans Java 6. Les versions antérieures de Java ne traitaient pas "*" comme un caractère générique.
2. Vous ne pouvez pas mettre d'autres caractères avant ou après le " *"; Par exemple, "someFolder* / .jar" n'est pas un joker.
3. Un caractère générique ne correspond qu'à des fichiers portant le suffixe ".jar" ou ".JAR". Les fichiers ZIP sont ignorés, de même que les fichiers JAR avec un suffixe différent.
4. Un caractère générique ne correspond qu'à des fichiers JAR du répertoire lui-même et non à ses sous-répertoires.
5. Lorsqu'un groupe de fichiers JAR est associé à une entrée générique, leur ordre relatif sur le

chemin de classe n'est pas spécifié.

Syntaxe du chemin de classe

Le classpath est une séquence d'entrées qui sont les noms de chemin d'accès aux répertoires, les chemins d'accès aux fichiers JAR ou ZIP, ou les spécifications de caractères génériques JAR / ZIP.

- Pour un classpath spécifié sur la ligne de commande (par exemple `-classpath`) ou comme variable d'environnement, les entrées doivent être séparées par `;` (point-virgule) caractères sur Windows, ou `:` (deux-points) caractères sur d'autres plates-formes (Linux, UNIX, MacOSX, etc.).
- Pour l'élément `Class-Path` dans `MANIFEST.MF` un fichier JAR, utilisez un seul espace pour séparer les entrées.

Parfois, il est nécessaire d'intégrer un espace dans une entrée classpath

- Lorsque le chemin de classe est spécifié sur la ligne de commande, il suffit d'utiliser les citations de shell appropriées. Par exemple:

  ```
  export CLASSPATH="/home/user/My JAR Files/foo.jar:second.jar"
  ```

  (Les détails peuvent dépendre du shell de commande que vous utilisez.)

- Lorsque le chemin de classe est spécifié dans un fichier JAR, le fichier "MANIFEST.MF" doit être utilisé.

  ```
  Class-Path: /home/user/My%20JAR%20Files/foo.jar second.jar
  ```

Classpath dynamique

Parfois, il ne suffit pas d'ajouter tous les fichiers JAR d'un dossier, par exemple lorsque vous avez du code natif et que vous devez sélectionner un sous-ensemble de fichiers JAR. Dans ce cas, vous avez besoin de deux méthodes `main()` . Le premier construit un classloader et utilise ensuite ce classloader pour appeler le second `main()` .

Voici un exemple qui sélectionne le JAR natif SWT correct pour votre plate-forme, ajoute tous les JAR de votre application, puis appelle la méthode `main()` : Créer une application Java SWT multi-plateforme

Charger une ressource depuis le classpath

Il peut être utile de charger une ressource (image, fichier texte, propriétés, KeyStore, ...) qui est incluse dans un JAR. Pour cela, nous pouvons utiliser les `Class` et `ClassLoader` .

Supposons que nous ayons la structure de projet suivante:

```
program.jar
|
\-com
  \-project
    |
    |-file.txt
    \-Test.class
```

Et nous voulons accéder au contenu de `file.txt` de la classe `Test` . Nous pouvons le faire en demandant au classloader:

```
InputStream is = Test.class.getClassLoader().getResourceAsStream("com/project/file.txt");
```

En utilisant le classloader, nous devons spécifier le chemin complet de notre ressource (chaque paquet).

Ou alternativement, nous pouvons demander l'objet de classe de test directement

```
InputStream is = Test.class.getResourceAsStream("file.txt");
```

En utilisant l'objet class, le chemin est relatif à la classe elle-même. Notre `Test.class` étant dans le package `com.project` , identique à `file.txt` , nous n'avons pas besoin de spécifier de chemin du tout.

Nous pouvons cependant utiliser des chemins absolus à partir de l'objet de classe, comme ceci:

```
 is = Test.class.getResourceAsStream("/com/project/file.txt");
```

Mappage de noms de classes sur des chemins d'accès

La chaîne d'outils Java standard (et les outils tiers conçus pour interagir avec eux) ont des règles spécifiques pour mapper les noms de classes aux noms de fichiers et aux autres ressources qui les représentent.

Les mappages sont les suivants

- Pour les classes du package par défaut, les chemins d'accès sont des noms de fichiers simples.
- Pour les classes d'un package nommé, les composants du nom du package sont mappés aux répertoires.
- Pour les classes imbriquées et internes nommées, le composant filename est formé en joignant les noms de classe avec un caractère `$` .
- Pour les classes internes anonymes, les nombres sont utilisés à la place des noms.

Ceci est illustré dans le tableau suivant:

Nom du cours	Chemin d'accès source	Chemin d'accès au fichier de classe
`SomeClass`	`SomeClass.java`	`SomeClass.class`
`com.example.SomeClass`	`com/example/SomeClass.java`	`com/example/SomeClass.class`
`SomeClass.Inner`	(dans `SomeClass.java`)	`SomeClass$Inner.class`
`SomeClass` anon classes internes	(dans `SomeClass.java`)	`SomeClass$1.class` , `SomeClass$2.class` , etc

Que signifie le classpath: comment les recherches fonctionnent

Le classpath a pour but d'indiquer à une machine virtuelle Java où trouver les classes et autres ressources. La signification du chemin de classe et le processus de recherche sont étroitement liés.

Le classpath est une forme de chemin de recherche qui spécifie une séquence d' *emplacements* pour rechercher des ressources. Dans un classpath standard, ces emplacements sont soit un répertoire dans le système de fichiers hôte, un fichier JAR ou un fichier ZIP. Dans chaque cas, l'emplacement est la racine d'un *espace de noms* qui sera recherché.

La procédure standard pour rechercher une classe sur le classpath est la suivante:

1. Carte du nom de la classe à un chemin d'accès relatif ClassFile `RP` . Le mappage des noms de classe avec les noms de fichiers de classe est décrit ailleurs.

2. Pour chaque entrée `E` dans le classpath:

    - Si l'entrée est un répertoire de système de fichiers:
        - Résoudre `RP` rapport à `E` pour donner un chemin d'accès absolu `AP` .
        - Testez si `AP` est un chemin pour un fichier existant.
        - Si oui, chargez la classe à partir de ce fichier
    - Si l'entrée est un fichier JAR ou ZIP:
        - Recherche `RP` dans l'index du fichier JAR / ZIP.
        - Si l'entrée de fichier JAR / ZIP correspondante existe, chargez la classe à partir de cette entrée.

La procédure de recherche d'une ressource sur le chemin de classe dépend du caractère absolu ou relatif du chemin d'accès à la ressource. Pour un chemin de ressource absolu, la procédure est comme ci-dessus. Pour un chemin de ressource relatif résolu avec `Class.getResource` ou `Class.getResourceAsStream` , le chemin du package de classes est ajouté avant la recherche.

(Notez que ces procédures sont implémentées par les chargeurs de classes Java standard. Un chargeur de classe personnalisé peut effectuer la recherche différemment.)

Le classpath du bootstrap

Les classloaders Java normaux recherchent d'abord les classes dans le chemin de classe bootstrap, avant de rechercher les extensions et le chemin de classe de l'application. Par défaut, le classpath bootstrap se compose du fichier "rt.jar" et d'autres fichiers JAR importants fournis par l'installation JRE. Celles-ci fournissent toutes les classes de la bibliothèque de classes Java SE standard, ainsi que diverses classes d'implémentation "internes".

Dans des circonstances normales, vous n'avez pas à vous en préoccuper. Par défaut, les commandes telles que `java` , `javac` , etc. utiliseront les versions appropriées des bibliothèques d'exécution.

Très occasionnellement, il est nécessaire de remplacer le comportement normal du runtime Java en utilisant une version alternative d'une classe dans les bibliothèques standard. Par exemple, vous pourriez rencontrer un bogue "show stopper" dans les bibliothèques d'exécution que vous ne pouvez pas contourner par des moyens normaux. Dans une telle situation, il est possible de créer un fichier JAR contenant la classe modifiée, puis de l'ajouter au chemin de classe bootstrap qui lance la machine virtuelle Java.

La commande `java` fournit les options `-X` suivantes pour modifier le chemin de classe bootstrap:

- `-Xbootclasspath:<path>` remplace le `-Xbootclasspath:<path>` démarrage actuel par le chemin fourni.
- `-Xbootclasspath/a:<path>` ajoute le chemin fourni au chemin de classe de démarrage actuel.
- `-Xbootclasspath/p:<path>` ajoute le chemin fourni au chemin de classe de démarrage actuel.

Notez que lorsque vous utilisez les options bootclasspath pour remplacer ou remplacer une classe Java (etc.), vous modifiez techniquement Java. Si vous distribuez votre code, cela *peut avoir* des conséquences sur la licence. (Reportez-vous aux conditions générales de la licence binaire Java… et consultez un avocat.)

Lire Le Classpath en ligne: https://riptutorial.com/fr/java/topic/3720/le-classpath

Chapitre 114: Lecteurs et écrivains

Introduction

Les lecteurs et les enregistreurs et leurs sous-classes respectives fournissent des E / S simples pour les données textuelles / basées sur des caractères.

Examples

BufferedReader

introduction

La classe `BufferedReader` est un wrapper pour les autres classes `Reader` qui remplit deux fonctions principales:

1. Un `BufferedReader` fournit une mémoire tampon pour le `Reader` . Cela permet à une application de lire les caractères un par un sans surcharger les E / S.

2. Un `BufferedReader` fournit des fonctionnalités permettant de lire du texte ligne par ligne.

Notions de base sur l'utilisation d'un BufferedReader

Le schéma normal d'utilisation d'un `BufferedReader` est le suivant. Tout d'abord, vous obtenez le `Reader` dont vous souhaitez lire les caractères. Ensuite, vous instanciez un `BufferedReader` qui encapsule le `Reader` . Ensuite, vous lisez les données de caractères. Enfin, vous fermez le `BufferedReader` qui ferme le lecteur `enveloppé. Par exemple:

```
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);
}
```

Vous pouvez appliquer ce modèle à n'importe quel `Reader`

Remarques:

1. Nous avons utilisé Java 7 (ou ultérieur) *avec des ressources* pour garantir que le lecteur sous-jacent est toujours fermé. Cela évite une fuite potentielle de ressources. Dans les versions antérieures de Java, vous fermiez explicitement `BufferedReader` dans un bloc `finally` .

2. Le code à l'intérieur du bloc `try` est pratiquement identique à ce que nous utiliserions si nous `FileReader` directement à partir de `FileReader` . En fait, un `BufferedReader` fonctionne exactement comme le `Reader` qu'il encapsule. La différence est que *cette* version est beaucoup plus efficace.

La taille du tampon BufferedReader

La méthode BufferedReader.readLine ()

Exemple: lecture de toutes les lignes d'un fichier dans une liste

Cela se fait en extrayant chaque ligne d'un fichier et en l'ajoutant à une `List<String>` . La liste est alors renvoyée:

```
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 fournit un moyen plus concis de le faire en utilisant la méthode `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();
}
```

Exemple StringWriter

La classe Java StringWriter est un flux de caractères qui collecte la sortie du tampon de chaîne, qui peut être utilisé pour créer une chaîne.

La classe StringWriter étend la classe Writer.

Dans la classe StringWriter, les ressources système telles que les sockets réseau et les fichiers ne sont pas utilisées. Par conséquent, la fermeture de StringWriter n'est pas nécessaire.

```
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'exemple ci-dessus nous aide à connaître un exemple simple de StringWriter utilisant BufferedReader pour lire des données de fichier à partir du flux.

Lire Lecteurs et écrivains en ligne: https://riptutorial.com/fr/java/topic/10618/lecteurs-et-ecrivains

Chapitre 115: Les opérateurs

Introduction

Les opérateurs en langage de programmation Java sont des symboles spéciaux qui effectuent des opérations spécifiques sur un, deux ou trois opérandes, puis renvoient un résultat.

Remarques

Un *opérateur* est un symbole (ou des symboles) qui indique à un programme Java d'effectuer une *opération* sur un, deux ou trois *opérandes* . Un opérateur et ses opérandes forment une *expression* (voir la rubrique Expressions). Les opérandes d'un opérateur sont eux-mêmes des expressions.

Cette rubrique décrit les quelque 40 opérateurs distincts définis par Java. La rubrique Expressions séparée explique:

- comment les opérateurs, les opérandes et d'autres choses sont combinés en expressions,
- comment les expressions sont évaluées et
- comment fonctionnent le typage, les conversions et l'évaluation des expressions.

Examples

L'opérateur de concaténation de chaînes (+)

Le symbole `+` peut signifier trois opérateurs distincts en Java:

- S'il n'y a pas d'opérande avant le `+` , alors c'est l'opérateur unaire Plus.
- S'il y a deux opérandes, ils sont tous deux numériques. alors c'est l'opérateur Addition binaire.
- S'il y a deux opérandes et qu'au moins l'un d'eux est une `String` , alors il s'agit de l'opérateur de concaténation binaire.

Dans le cas simple, l'opérateur de concaténation joint deux chaînes pour donner une troisième chaîne. Par exemple:

```
String s1 = "a String";
String s2 = "This is " + s1;    // s2 contains "This is a String"
```

Lorsque l'un des deux opérandes n'est pas une chaîne, il est converti en une `String` comme suit:

- Un opérande dont le type est un type primitif est converti *comme* en appelant `toString()` sur la valeur encadrée.

- Un opérande dont le type est un type de référence est converti en appelant la `toString()`

l'opérande. Si l'opérande est `null` ou si la `toString()` renvoie `null` , le littéral de chaîne `"null"` est utilisé à la place.

Par exemple:

```
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"
```

L'explication de l'exemple `s5` est que la `toString()` sur les types de tableau est héritée de `java.lang.Object` et que le comportement consiste à produire une chaîne composée du nom du type et du code de hachage de l'identité de l'objet.

L'opérateur Concatenation est spécifié pour créer un nouvel objet `String` , sauf dans le cas où l'expression est une expression constante. Dans ce dernier cas, l'expression est évaluée au type de compilation et sa valeur d'exécution est équivalente à un littéral de chaîne. Cela signifie qu'il n'y a pas de surcharge d'exécution lors du fractionnement d'un littéral de chaîne longue comme ceci:

```
String typing = "The quick brown fox " +
                "jumped over the " +
                "lazy dog";           // constant expression
```

Optimisation et efficacité

Comme indiqué ci-dessus, à l'exception des expressions constantes, chaque expression de concaténation de chaîne crée un nouvel objet `String` . Considérez ce code:

```
public String stars(int count) {
    String res = "";
    for (int i = 0; i < count; i++) {
        res = res + "*";
    }
    return res;
}
```

Dans la méthode ci-dessus, chaque itération de la boucle créera une nouvelle `String` un caractère plus longue que l'itération précédente. Chaque concaténation copie tous les caractères des chaînes d'opérandes pour former la nouvelle `String` . Ainsi, les `stars(N)` vont:

- créer `N` nouveaux objets `String` , et jeter tous sauf le dernier,
- copier `N * (N + 1) / 2` caractères, et
- générer `O(N^2)` octets de déchets.

C'est très cher pour les gros `N` En effet, tout code concaténant des chaînes dans une boucle est susceptible d'avoir ce problème. Une meilleure façon d'écrire ceci serait la suivante:

```
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();
}
```

Idéalement, vous devez définir la capacité du `StringBuilder` , mais si cela est impossible, la classe se *développera* automatiquement le tableau de sauvegarde que le constructeur utilise pour tenir des caractères. (Remarque: l'implémentation étend le tableau de support de manière exponentielle. Cette stratégie conserve cette quantité de caractères à un `O(N)` plutôt qu'à un `O(N^2)` .)

Certaines personnes appliquent ce modèle à toutes les concaténations de chaînes. Cependant, cela n'est pas nécessaire car le JLS *permet à* un compilateur Java d'optimiser les concaténations de chaînes dans une seule expression. Par exemple:

```
String s1 = ...;
String s2 = ...;
String test = "Hello " + s1 + ". Welcome to " + s2 + "\n";
```

sera *généralement* optimisé par le compilateur bytecode à quelque chose comme ça;

```
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();
```

(Le compilateur JIT peut optimiser cela plus loin s'il peut en déduire que `s1` ou `s2` ne peut pas être `null` .) Mais notez que cette optimisation n'est autorisée que dans une seule expression.

En bref, si vous êtes préoccupé par l'efficacité des concaténations de chaînes:

- Effectuez une optimisation manuelle si vous effectuez une concaténation répétée dans une boucle (ou similaire).
- Ne pas optimiser manuellement une seule expression de concaténation.

Les opérateurs arithmétiques (+, -, *, /,%)

Le langage Java fournit 7 opérateurs effectuant des opérations arithmétiques sur des valeurs entières et à virgule flottante.

- Il y a deux opérateurs `+` :
    - L'opérateur d'addition binaire ajoute un numéro à un autre. (Il y a aussi un opérateur binaire `+` qui effectue la concaténation des chaînes. Ceci est décrit dans un exemple séparé.)
    - L'opérateur unaire plus ne fait rien d'autre que déclencher une promotion numérique

(voir ci-dessous)

- Il y a deux `-` opérateurs:
  - L'opérateur de soustraction binaire soustrait un nombre d'un autre.
  - L'opérateur moins unaire équivaut à soustraire son opérande de zéro.
- L'opérateur de multiplication binaire (*) multiplie un nombre par un autre.
- L'opérateur de division binaire (/) divise un nombre par un autre.
- L'opérateur binaire reste [1] (%) calcule le reste lorsqu'un nombre est divisé par un autre.

1. Ceci est souvent appelé à tort l'opérateur "module". "Reste" est le terme utilisé par JLS. "Module" et "reste" ne sont pas la même chose.

Opérande et types de résultats, et promotion numérique

Les opérateurs nécessitent des opérandes numériques et produisent des résultats numériques. Les types d'opérandes peuvent être tous les types numériques primitifs (`byte` , caractères `short` , caractères `char` , `int` , `long` , `float` ou `double`) ou tout type d'encapsuleur numérique défini dans `java.lang` ; à savoir (`Byte` , `Character` , `Short` , `Integer` , `Long` , `Float` ou `Double` .

Le type de résultat est déterminé sur la base des types de l'opérande ou des opérandes, comme suit:

- Si l'un des opérandes est un `double` ou un `Double` , le type de résultat est `double` .
- Sinon, si l'un des opérandes est un `float` ou un `Float` , le type de résultat est `float` .
- Sinon, si l'un des opérandes est `long` ou `Long` , le type de résultat est `long` .
- Sinon, le type de résultat est `int` . Cela couvre les opérandes `byte` , `short` et `char` ainsi que `int.

Le type de résultat de l'opération détermine comment l'opération arithmétique est effectuée et comment les opérandes sont traités

- Si le type de résultat est `double` , les opérandes sont promus pour `double` , et l'opération est effectuée en utilisant l'arithmétique à virgule flottante IEE 754 64 bits (double précision binaire).
- Si le type de résultat est `float` , les opérandes sont promus à `float` et l'opération est effectuée en utilisant l'arithmétique à virgule flottante IEE 754 32 bits (binaire simple précision).
- Si le type de résultat est `long` , les opérandes sont promus à `long` et l'opération est effectuée en utilisant l'arithmétique d'entier binaire à deux compléments signés 64 bits.
- Si le type de résultat est `int` , les opérandes sont promus en `int` , et l'opération est effectuée en utilisant une arithmétique d'entiers binaires à deux compléments signés 32 bits.

La promotion se déroule en deux étapes:

- Si le type d'opérande est un type wrapper, la valeur de l'opérande est *unboxed* à une valeur du type primitif correspondant.
- Si nécessaire, le type primitif est promu au type requis:
  - La promotion des entiers en `int` ou en `long` est sans perte.
  - La promotion du `float` pour `double` est sans perte.

- La promotion d'un entier sur une valeur à virgule flottante peut entraîner une perte de précision. La conversion est effectuée en utilisant la sémantique IEE 768 "round-to-close".

Le sens de la division

L'opérateur / divise l'opérande gauche `n` *(dividende)* et l'opérande de droite `d` (le *diviseur)* et produit le résultat `q` *(quotient).*

La division Java entière arrondit vers zéro. La section JLS 15.17.2 spécifie le comportement de la division entière Java comme suit:

> Le quotient produit pour les opérandes `n` et `d` est une valeur entière `q` dont la grandeur est la plus grande possible tout en satisfaisant `|d · q| ≤ |n|` . De plus, `q` est positif quand `|n| ≥ |d|` et `n` et `d` ont le même signe, mais `q` est négatif lorsque `|n| ≥ |d|` et `n` et `d` ont des signes opposés.

Il y a quelques cas particuliers:

- Si le `n` est `MIN_VALUE` et que le diviseur est -1, alors un dépassement d'entier se produit et le résultat est `MIN_VALUE` . Aucune exception n'est levée dans ce cas.
- Si `d` vaut 0, alors `ArithmeticException est levée.

La division en virgule flottante Java a plus de cas à prendre en compte. Cependant, l'idée de base est que le résultat `q` est la valeur la plus proche de la satisfaction `d . q = n` .

La division en virgule flottante ne donnera jamais lieu à une exception. Au lieu de cela, les opérations qui divisent par zéro donnent des valeurs INF et NaN; voir ci-dessous.

La signification du reste

Contrairement à C et C ++, l'opérateur restant en Java fonctionne à la fois avec des opérations à nombre entier et à virgule flottante.

Pour les cas entiers, le résultat d' `a % b` est défini comme étant le nombre `r` tel que `(a / b) * b + r` est égal à `a` , où `/` , `*` et `+` sont les opérateurs entiers Java appropriés. Cela s'applique dans tous les cas sauf lorsque `b` est égal à zéro. Ce cas, reste reste une `ArithmeticException` .

Il résulte de la définition ci-dessus qu'un `a % b` ne peut être négatif que si `a` est négatif, et qu'il ne soit positif que si `a` est positif. De plus, l'ampleur d' `a % b` est toujours inférieure à l'ampleur de `b` .

L'opération de reste à virgule flottante est une généralisation de la casse entière. Le résultat d' `a % b` est le reste `r` est défini par la relation mathématique `r = a - (b · q)` où:

- `q` est un entier,
- il est négatif uniquement si `a / b` est négatif et positif uniquement si `a / b` est positif et
- sa magnitude est aussi grande que possible sans dépasser l'ampleur du vrai quotient mathématique de `a` et `b` .

Le reste à virgule flottante peut produire des valeurs `INF` et `NaN` dans les cas limites tels que lorsque `b` est égal à zéro; voir ci-dessous. Il ne jettera pas une exception.

Note importante:

> Le résultat d'une opération de reste à virgule flottante calculée par `%` **n'est pas le même** que celui produit par l'opération restante définie par IEEE 754. Le reste de l'IEEE 754 peut être calculé à l'aide de la méthode de bibliothèque `Math.IEEEremainder` .

Débordement d'entier

Les valeurs entières de Java 32 et 64 bits sont signées et utilisent une représentation binaire en complément à deux. Par exemple, la gamme des nombres représentables comme (32 bits) `int` -2 ^{31} à 2 $^{31-1}$.

Lorsque vous ajoutez, soustrayez ou plusieurs deux entiers N bits (N == 32 ou 64), le résultat de l'opération peut être trop important pour représenter un entier N bits. Dans ce cas, l'opération entraîne un *débordement d'entier* et le résultat peut être calculé comme suit:

- L'opération mathématique est effectuée pour donner une représentation intermédiaire à deux complément du nombre entier. Cette représentation sera supérieure à N bits.
- Les 32 ou 64 bits inférieurs de la représentation intermédiaire sont utilisés comme résultat.

Il convient de noter que le débordement d'entier n'entraîne en aucun cas des exceptions.

Valeurs INF et NAN en virgule flottante

Java utilise des représentations à virgule flottante IEE 754 pour `float` et `double` . Ces représentations ont des valeurs spéciales pour représenter des valeurs qui ne relèvent pas du domaine des nombres réels:

- Les valeurs infinies ou INF indiquent des nombres trop importants. La valeur `+INF` indique des nombres trop grands et positifs. La valeur `-INF` indique des nombres trop grands et négatifs.
- Les "indéfinis" / "pas un nombre" ou NaN désignent des valeurs résultant d'opérations sans signification.

Les valeurs INF sont produites par des opérations flottantes qui provoquent un débordement ou par division par zéro.

Les valeurs NaN sont produites en divisant zéro par zéro ou en calculant le reste zéro.

Étonnamment, il est possible d'effectuer des opérations arithmétiques à l'aide des opérandes INF et NaN sans provoquer d'exceptions. Par exemple:

- Ajouter + INF et une valeur finie donne + INF.
- Ajouter + INF et + INF donne + INF.
- Ajouter + INF et -INF donne NaN.

- La division par INF donne +0.0 ou -0.0.
- Toutes les opérations avec un ou plusieurs opérandes NaN donnent NaN.

Pour plus de détails, veuillez vous référer aux sous-sections pertinentes de JLS 15 . Notez que c'est largement "académique". Pour les calculs typiques, un `INF` ou `NaN` signifie que quelque chose ne va pas; Par exemple, vous avez des données d'entrée incomplètes ou incorrectes, ou le calcul a été programmé de manière incorrecte.

Les opérateurs d'égalité (==,! =)

Les opérateurs `==` et `!=` Sont des opérateurs binaires dont la valeur est `true` ou `false` selon que les opérandes sont égaux ou non. L'opérateur `==` donne `true` si les opérandes sont égaux et `false` sinon. L'opérateur `!=` Donne `false` si les opérandes sont égaux et `true` sinon.

Ces opérateurs peuvent être utilisés avec des opérandes avec des types primitifs et de référence, mais le comportement est sensiblement différent. Selon le JLS, il existe en réalité trois ensembles distincts de ces opérateurs:

- Les booléens `==` et les opérateurs `!=` .
- Les opérateurs numériques `==` et `!=` .
- Les opérateurs de référence `==` et `!=` .

Cependant, dans tous les cas, le type de résultat des opérateurs `==` et `!=` Est `boolean` .

Les opérateurs numériques `==` et `!=`

Lorsque l'un (ou les deux) opérandes d'un opérateur `==` ou `!=` Est un type numérique primitif (`byte` , `short` , `char` , `int`, `long` , `float` ou `double`), l'opérateur est une comparaison numérique. Le second opérande doit être un type numérique primitif ou un type numérique encadré.

Le comportement des autres opérateurs numériques est le suivant:

1. Si l'un des opérandes est un type encadré, il n'est pas enregistré.
2. Si l'une des opérandes maintenant un `byte` , à `short` ou `char` , il est promu à un `int` .
3. Si les types des opérandes ne sont pas les mêmes, l'opérande avec le type "plus petit" est promu au type "plus grand".
4. La comparaison est ensuite effectuée comme suit:
    - Si les opérandes promus sont `int` ou `long` les valeurs sont testées pour voir si elles sont identiques.
    - Si les opérandes promus sont `float` ou `double` alors:
        - les deux versions de zéro (`+0.0` et `-0.0`) sont considérés comme égaux
        - une valeur `NaN` est traitée comme non égal à rien, et
        - les autres valeurs sont égales si leurs représentations IEEE 754 sont identiques.

Remarque: vous devez faire attention lorsque vous utilisez `==` et `!=` Pour comparer des valeurs en virgule flottante.

Le booléen `==` et les opérateurs `!=`

Si les deux opérandes sont `boolean` , ou l'un est `boolean` et l'autre `Boolean` , ces opérateurs sont les opérateurs booléens `==` et `!=` . Le comportement est le suivant:

1. Si l'un des opérandes est un `Boolean` , il n'est pas enregistré.
2. Les opérandes unboxed sont testés et le résultat booléen est calculé selon la table de vérité suivante

UNE	B	A == B	A! = B
faux	faux	vrai	faux
faux	vrai	faux	vrai
vrai	faux	faux	vrai
vrai	vrai	vrai	faux

Il y a deux «pièges» qui recommandent d'utiliser `==` et `!=` Avec parcimonie avec des valeurs de vérité:

- Si vous utilisez `==` ou `!=` Pour comparer deux objets `Boolean` , les opérateurs de référence sont utilisés. Cela peut donner un résultat inattendu. voir Pitfall: utiliser == pour comparer des objets primitifs tels que Entier
- L'opérateur `==` peut facilement être tapé comme `=` . Pour la plupart des types d'opérandes, cette erreur entraîne une erreur de compilation. Cependant, pour `Boolean` opérandes `boolean` et `Boolean` l'erreur conduit à un comportement d'exécution incorrect; voir Pitfall - Utiliser '==' pour tester une valeur booléenne

La référence `==` et les opérateurs `!=`

Si les deux opérandes sont des références d'objet, les opérateurs `==` et `!=` Testent si les deux opérandes **font référence au même objet** . Ce n'est souvent pas ce que vous voulez. Pour tester si deux objets sont égaux *en valeur* , la méthode `.equals()` doit être utilisée à la place.

```
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
```

Attention: utiliser `==` et `!=` Pour comparer les valeurs de `String` est **incorrect** dans la plupart des cas; voir http://www.riptutorial.com/java/example/16290/pitfall--using----to-compare-strings . Un problème similaire s'applique aux types d'encapsuleurs primitifs. voir

http://www.riptutorial.com/java/example/8996/pitfall--using----to-compare-primitive-wrappers-objects-such-as-integer .

À propos des bordures NaN

JLS 15.21.1 indique ce qui suit:

> Si l'un des opérandes est `NaN` , le résultat de `==` est `false` mais le résultat de `!=` Est `true` . En effet, le test `x != x` est `true` si et seulement si la valeur de `x` est `NaN` .

Ce comportement est (pour la plupart des programmeurs) inattendu. Si vous testez si une valeur `NaN` est égale à elle-même, la réponse est "Non, ça ne l'est pas!". En d'autres termes, `==` n'est pas *réflexif* pour les valeurs `NaN` .

Cependant, il ne s'agit pas d'une "bizarrerie" Java, ce comportement est spécifié dans les normes à virgule flottante IEEE 754 et vous constaterez qu'il est implémenté par la plupart des langages de programmation modernes. (Pour plus d'informations, voir http://stackoverflow.com/a/1573715/139985 ... notant que ceci est écrit par quelqu'un qui était "dans la pièce lorsque les décisions ont été prises"!)

Les opérateurs d'incrémentation / décrémentation (++ / -)

Les variables peuvent être incrémentées ou décrémentées de 1 en utilisant respectivement les opérateurs `++` et `--` .

Lorsque les opérateurs `++` et `--` suivent des variables, ils sont appelés respectivement **post-incrémentation** et **post-décrémentation** .

```
int a = 10;
a++; // a now equals 11
a--; // a now equals 10 again
```

Lorsque les opérateurs `++` et `--` précèdent les variables, les opérations sont appelées **pré-incrémentation** et **pré-décrémentation** respectivement.

```
int x = 10;
--x; // x now equals 9
++x; // x now equals 10
```

Si l'opérateur précède la variable, la valeur de l'expression est la valeur de la variable après incrémentation ou décrémentation. Si l'opérateur suit la variable, la valeur de l'expression est la valeur de la variable avant d'être incrémentée ou décrémentée.

```
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
```

Veillez à ne pas écraser les post-incréments ou les décréments. Cela se produit si vous utilisez un opérateur de post-in / decrement à la fin d'une expression qui est réaffecté à la variable in / decremented elle-même. Le in / decrement n'aura aucun effet. Même si la variable du côté gauche est incrémentée correctement, sa valeur sera immédiatement remplacée par le résultat précédemment évalué du côté droit de l'expression:

```
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!)
```

Correct:

```
int x = 0;
x = x++ + 1 + x;        // evaluates to x = 0 + 1 + 1
x++;                    // adds 1
System.out.println(x);  // prints 3
```

L'opérateur conditionnel (? :)

Syntaxe

{condition à évaluer} **?** *{statement-execute-on-true}* **:** *{statement-execute-on-false}*

Comme indiqué dans la syntaxe, l'opérateur conditionnel (également appelé opérateur ternaire [1]) utilise le `?` (point d'interrogation) et `:` (deux-points) caractères pour permettre une expression conditionnelle de deux résultats possibles. Il peut être utilisé pour remplacer des blocs `if-else` plus longs pour renvoyer l'une des deux valeurs en fonction de la condition.

```
result = testCondition ? value1 : value2
```

Est équivalent à

```
if (testCondition) {
    result = value1;
} else {
    result = value2;
}
```

Il peut être lu comme suit: ***«Si testCondition est true, définissez result sur value1; sinon, définissez le résultat sur valeur2 ”.***

Par exemple:

```
// get absolute value using conditional operator
a = -10;
int absValue = a < 0 ? -a : a;
System.out.println("abs = " + absValue); // prints "abs = 10"
```

Est équivalent à

```
// 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"
```

Usage courant

Vous pouvez utiliser l'opérateur conditionnel pour les affectations conditionnelles (comme la vérification null).

```
String x = y != null ? y.toString() : ""; //where y is an object
```

Cet exemple est équivalent à:

```
String x = "";

if (y != null) {
    x = y.toString();
}
```

Étant donné que l'opérateur conditionnel a la deuxième priorité la plus basse, au-dessus des opérateurs d'affectation , il est rarement nécessaire d'utiliser des parenthèses autour de la *condition* , mais des parenthèses sont nécessaires dans toute la structure de l'opérateur conditionnel lorsqu'il est associé à d'autres opérateurs:

```
// no parenthesis needed for expressions in the 3 parts
10 <= a && a < 19 ? b * 5 : b * 7

// parenthesis required
7 * (a > 0 ? 2 : 5)
```

Les opérateurs conditionnels peuvent également être imbriqués dans la troisième partie, où ils fonctionnent plus comme un chaînage ou comme une instruction 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))
```

Note de bas de page:

1 - La spécification de langage Java et le didacticiel Java appellent tous deux l'opérateur (`?` `:` :) *L'opérateur conditionnel* . Le tutoriel dit qu'il est "également connu sous le nom d'opérateur ternaire" car il est (actuellement) le seul opérateur ternaire défini par Java. La terminologie "opérateur conditionnel" est cohérente avec C et C ++ et les autres langages avec un opérateur équivalent.

Les opérateurs binaires et logiques (~, &, |, ^)

Le langage Java fournit 4 opérateurs qui effectuent des opérations binaires ou logiques sur des opérandes entiers ou booléens.

- L'opérateur complément (`~`) est un opérateur unaire qui effectue une inversion binaire ou logique des bits d'un opérande; voir JLS 15.15.5. .
- L'opérateur AND (`&`) est un opérateur binaire qui effectue un "et" deux bits ou logique de deux opérandes; voir JLS 15.22.2. .
- L'opérateur OR (`|`) est un opérateur binaire qui exécute un "inclusive" ou deux "opérandes" binaire ou logique; voir JLS 15.22.2. .
- L'opérateur XOR (`^`) est un opérateur binaire qui effectue une opération "exclusive ou" binaire ou logique de deux opérandes; voir JLS 15.22.2. .

Les opérations logiques effectuées par ces opérateurs lorsque les opérandes sont des booléens peuvent être résumées comme suit:

UNE	B	~ A	UN B	Un | 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

Notez que pour les opérandes entiers, le tableau ci-dessus décrit ce qui se passe pour les bits individuels. Les opérateurs fonctionnent en fait sur les 32 ou 64 bits de l'opérande ou des opérandes en parallèle.

Types d'opérandes et types de résultats.

Les conversions arithmétiques habituelles s'appliquent lorsque les opérandes sont des entiers. Cas d'utilisation courants pour les opérateurs binaires

---

L'opérateur `~` est utilisé pour inverser une valeur booléenne ou modifier tous les bits d'un opérande entier.

L'opérateur `&` est utilisé pour "masquer" certains bits d'un opérande entier. Par exemple:

```
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
```

Le `|` L'opérateur est utilisé pour combiner les valeurs de vérité de deux opérandes. Par exemple:

```
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'opérateur `^` est utilisé pour basculer ou "retourner" les bits:

```
int word3 = 0b00101010;
int word4 = word3 ^ mask;             // -> 0b00101001
```

Pour plus d'exemples d'utilisation des opérateurs binaires , voir Manipulation des bits

L'opérateur d'instance

Cet opérateur vérifie si l'objet est d'un type de classe / interface particulier. **L'** opérateur **instanceof** est écrit comme suit:

```
( Object reference variable ) instanceof  (class/interface type)
```

Exemple:

```
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 );
   }
}
```

Cela produirait le résultat suivant:

```
true
```

Cet opérateur retournera toujours true si l'objet comparé est l'affectation compatible avec le type à droite.

Exemple:

```
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 );
    }
}
```

Cela produirait le résultat suivant:

```
true
```

Les opérateurs d'affectation (=, + =, - =, * =, / =,% =, << =, >> =, >>> =, & =, | = et ^ =)

L'opérande de gauche pour ces opérateurs doit être soit une variable non finale, soit un élément d'un tableau. L'opérande de droite doit être *compatible* avec l'opérande de gauche. Cela signifie que soit les types doivent être identiques, soit le type d'opérande droit doit être convertible en type d'opérandes gauche par une combinaison de boxing, unboxing ou élargissement. (Pour plus de détails, reportez-vous à JLS 5.2 .)

La signification précise des opérateurs "operation and assign" est spécifiée par JLS 15.26.2 comme suit :

> Une expression d'affectation composée de la forme `E1 op= E2` est équivalente à `E1 = (T) ((E1) op (E2))` , où `T` est le type de `E1` , sauf `E1` n'est évaluée qu'une seule fois.

Notez qu'il existe un type implicite avant l'assignation finale.

1. `=`

L'opérateur d'affectation simple: attribue la valeur de l'opérande de droite à l'opérande de gauche.

> Exemple: `c = a + b` va ajouter la valeur de `a + b` à la valeur de `c` et l'assigner à `c`

2. `+=`

L'opérateur "add and assign": ajoute la valeur de l'opérande de droite à la valeur de l'opérande de gauche et affecte le résultat à l'opérande de gauche. Si l'opérande de gauche a le type `String` , alors c'est un opérateur "concatenate and assign".

> Exemple: `c += a` est sensiblement le même que `c = c + a`

3. `-=`

L'opérateur "soustraire et affecter": soustrait la valeur de l'opérande droite de la valeur de l'opérande de gauche et attribue le résultat à l'opérande de gauche.

> Exemple: `c -= a` est sensiblement le même que `c = c - a`

4. `*=`

L'opérateur "multiplier et affecter": multiplie la valeur de l'opérande de droite par la valeur de l'opérande de gauche et attribue le résultat à l'opérande de gauche. .

Exemple: `c *= a` est sensiblement le même que `c = c * a`

5. `/=`

L'opérateur "diviser et assigner": divise la valeur de l'opérande de droite par la valeur de l'opérande de gauche et attribue le résultat à l'opérande de gauche.

Exemple: `c /*= a` est sensiblement le même que `c = c / a`

6. `%=`

L'opérateur "modulus and assign": calcule le module de la valeur de l'opérande de droite par la valeur de l'opérande de gauche et attribue le résultat à l'opérande de gauche.

Exemple: `c %*= a` est sensiblement le même que `c = c % a`

7. `<<=`

L'opérateur "left shift et assign".

Exemple: `c <<= 2` est à peu près le même que `c = c << 2`

8. `>>=`

L'opérateur "arithmétique à droite et assigner".

Exemple: `c >>= 2` est à peu près la même chose que `c = c >> 2`

9. `>>>=`

L'opérateur "décalage droit logique et attribution".

Exemple: `c >>>= 2` est à peu près la même chose que `c = c >>> 2`

10. `&=`

L'opérateur "bitwise and and assign".

Exemple: `c &= 2` est à peu près le même que `c = c & 2`

11. `|=`

L'opérateur "bitwise or and assign".

Exemple: `c |= 2` est à peu près la même chose que `c = c | 2`

12. `^=`

L'opérateur "bitwise exclusive or and assign".

Exemple: `c ^= 2` est à peu près le même que `c = c ^ 2`

Les opérateurs conditionnels et et / ou conditionnels (&& et ||)

Java fournit un opérateur conditionnel et un opérateur conditionnel ou conditionnel, qui prennent tous deux un ou deux opérandes de type `boolean` et produisent un résultat `boolean` . Ceux-ci sont:

- `&&` - l'opérateur AND conditionnel,
- `||` - les opérateurs conditionnels-OU. L'évaluation de `<left-expr> && <right-expr>` est équivalente au pseudo-code suivant:

```
{
   boolean L = evaluate(<left-expr>);
   if (L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return false;
   }
}
```

L'évaluation de `<left-expr> || <right-expr>` est équivalent au pseudo-code suivant:

```
{
   boolean L = evaluate(<left-expr>);
   if (!L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return true;
   }
}
```

Comme le montre le pseudo-code ci-dessus, le comportement des opérateurs de court-circuit équivaut à utiliser des instructions `if` / `else` .

Exemple - utilisation de && comme garde dans une expression

L'exemple suivant montre le modèle d'utilisation le plus courant pour l'opérateur `&&` . Comparez ces deux versions d'une méthode pour tester si un `Integer` fourni est zéro.

```
public boolean isZero(Integer value) {
    return value == 0;
}

public boolean isZero(Integer value) {
    return value != null && value == 0;
}
```

La première version fonctionne dans la plupart des cas, mais si l'argument `value` est `null` , une `NullPointerException` sera lancée.

Dans la deuxième version, nous avons ajouté un test de "garde". La `value != null && value == 0` expression est évaluée en exécutant d'abord le test `value != null` . Si le test `null` réussit (c.-à-d. Qu'il est évalué à `true`), l'expression `value == 0` est évaluée. Si le test `null` échoue, alors l'évaluation de la `value == 0` est ignorée (court-circuitée) et nous n'obtenons *pas* une `NullPointerException` .

Exemple - utilisation de && pour éviter un calcul coûteux

L'exemple suivant montre comment `&&` peut être utilisé pour éviter un calcul relativement coûteux:

```
public boolean verify(int value, boolean needPrime) {
    return !needPrime | isPrime(value);
}

public boolean verify(int value, boolean needPrime) {
    return !needPrime || isPrime(value);
}
```

Dans la première version, les deux opérandes du `|` sera toujours évalué, donc la méthode (coûteuse) `isPrime` sera appelée inutilement. La deuxième version évite l'appel inutile en utilisant `||` au lieu de `|` .

Les opérateurs de quart (<<, >> et >>>)

Le langage Java fournit trois opérateurs pour effectuer un décalage binaire sur des valeurs entières de 32 et 64 bits. Ce sont tous des opérateurs binaires, le premier opérande étant la valeur à déplacer et le second opérande indiquant la distance à déplacer.

- Le `<<` ou l' opérateur *de décalage vers* la *gauche* déplace la valeur donnée par le premier opérande *vers* la *gauche* par le nombre de positions de bits fournis par le deuxième opérande. Les positions vides à droite sont remplies de zéros.
- L'opérateur de décalage « *arithmétique* » ou « *arithmétique»* décale la valeur donnée par le premier opérande *vers la droite* du nombre de positions binaires données par le second opérande. Les positions vides à gauche sont remplies en copiant le bit le plus à gauche. Ce processus est appelé *extension de signe* .
- L'opérateur de *décalage à droite* ">>>" ou *logique* déplace la valeur donnée par le premier opérande *vers la droite* par le nombre de positions de bits fournies par le second opérande. Les positions vides à gauche sont remplies de zéros.

Remarques:

1. Ces opérateurs nécessitent une valeur `int` ou `long` comme premier opérande et produisent une valeur du même type que le premier opérande. (Vous aurez besoin d'utiliser une distribution de type explicite lors de l' affectation du résultat d'un passage à un `byte` , à `short`

ou `char` variable.)

2. Si vous utilisez un opérateur de décalage avec un premier opérande qui est un `byte` , `char` ou `short` , il est promu à un `int` et l'opération produit un `int` .)

3. Le second opérande est réduit *modulo par le nombre de bits de l'opération* pour donner la valeur du décalage. Pour plus d'informations sur le **concept mathématique des mods** , voir Exemples de modules .

4. Les bits décalés de l'extrémité gauche ou droite par l'opération sont ignorés. (Java ne fournit pas un opérateur primitif "rotate".)

5. L'opérateur de décalage arithmétique est équivalent en divisant un nombre (complément à deux) par une puissance de 2.

6. L'opérateur de décalage de gauche est équivalent en multipliant un nombre (complément à deux) par une puissance de 2.

Le tableau suivant vous aidera à voir les effets des trois opérateurs de quart. (Les nombres ont été exprimés en notation binaire pour faciliter la visualisation.)

Opérande1	Opérande2	<<	>>	>>>
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

Il y a des exemples de l'utilisateur des opérateurs de décalage dans la manipulation de bits

L'opérateur Lambda (->)

A partir de Java 8, l'opérateur Lambda (`->`) est l'opérateur utilisé pour introduire une expression Lambda. Il existe deux syntaxes communes, comme illustré par ces exemples:

Java SE 8

```
  a -> a + 1               // a lambda that adds one to its argument
  a -> { return a + 1; }   // an equivalent lambda using a block.
```

Une expression lambda définit une fonction anonyme ou, plus correctement, une instance d'une classe anonyme qui implémente une *interface fonctionnelle* .

(Cet exemple est inclus ici pour être complet. Reportez-vous à la rubrique Expressions Lambda pour le traitement complet.)

Les opérateurs relationnels (<, <=,>,> =)

Les opérateurs `<` , `<=` , `>` et `>=` sont des opérateurs binaires permettant de comparer des types numériques. La signification des opérateurs est celle que vous attendez. Par exemple, si `a` et `b` sont déclarés comme `byte` , `short` , `char` , `int` , `long` , `float` , `double` ou les types de boîte correspondants:

```
- `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`.
```

Le type de résultat pour ces opérateurs est `boolean` dans tous les cas.

Les opérateurs relationnels peuvent être utilisés pour comparer des nombres de types différents. Par exemple:

```
int i = 1;
long l = 2;
if (i < l) {
    System.out.println("i is smaller");
}
```

Les opérateurs relationnels peuvent être utilisés lorsque l'un ou les deux nombres sont des instances de types numériques encadrés. Par exemple:

```
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");
}
```

Le comportement précis est résumé comme suit:

1. Si l'un des opérandes est un type encadré, il n'est pas enregistré.
2. Si l'une des opérandes maintenant un `byte` , à `short` ou `char` , il est promu à un `int` .
3. Si les types des opérandes ne sont pas les mêmes, l'opérande avec le type "plus petit" est promu au type "plus grand".
4. La comparaison est effectuée sur les valeurs `int` , `long` , `float` ou `double` résultantes.

Vous devez faire attention aux comparaisons relationnelles impliquant des nombres à virgule

flottante:

- Les expressions qui calculent des nombres à virgule flottante entraînent souvent des erreurs d'arrondi du fait que les représentations à virgule flottante de l'ordinateur ont une précision limitée.
- Lorsque vous comparez un type entier et un type à virgule flottante, la conversion de l'entier en virgule flottante peut également entraîner des erreurs d'arrondi.

Enfin, Java ne prend pas en charge l'utilisation d'opérateurs relationnels avec d'autres types que ceux répertoriés ci-dessus. Par exemple, vous *ne pouvez pas* utiliser ces opérateurs pour comparer des chaînes, des tableaux de nombres, etc.

Lire Les opérateurs en ligne: https://riptutorial.com/fr/java/topic/176/les-operateurs

Chapitre 116: LinkedHashMap

Introduction

La classe LinkedHashMap est l'implémentation de la table de hachage et de la liste liée de l'interface Map, avec un ordre d'itération prévisible. Il hérite de la classe HashMap et implémente l'interface Map.

Les points importants concernant la classe Java LinkedHashMap sont les suivants: Un LinkedHashMap contient des valeurs basées sur la clé. Il ne contient que des éléments uniques. Il peut avoir une clé NULL et plusieurs valeurs NULL. C'est la même chose que HashMap maintient l'ordre d'insertion.

Examples

Classe Java LinkedHashMap

Points clés:-

- Est la table de hachage et l'implémentation de la liste liée de l'interface Map, avec un ordre d'itération prévisible.
- hérite de la classe HashMap et implémente l'interface Map.
- contient des valeurs basées sur la clé.
- seulement des éléments uniques.
- peut avoir une clé NULL et plusieurs valeurs NULL.
- de même que HashMap maintient l'ordre d'insertion.

Méthodes: -

- annulez clear ().
- boolean containsKey (clé d'objet).
- Objet get (clé d'objet).
- booléen protégé removeEldestEntry (Map.Entry eldest)

Exemple :-

```
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"));
    }
```

Lire LinkedHashMap en ligne: https://riptutorial.com/fr/java/topic/10750/linkedhashmap

Chapitre 117: Liste vs SET

Introduction

Quelles sont les différences entre la collection List et Set au niveau supérieur et Comment choisir quand utiliser List dans Java et quand utiliser Set in Java

Examples

Liste vs Set

```
import java.util.ArrayList;
```

import java.util.HashSet; import java.util.List; import java.util.Set;

Classe publique SetAndListExample {public static void main (String [] args) {System.out.println ("Exemple de liste ....."); List list = new ArrayList (); list.add ("1"); list.add ("2"); list.add ("3"); list.add ("4"); list.add ("1");

```
    for (String temp : list){
        System.out.println(temp);
    }

    System.out.println("Set example .....");
    Set<String> set = new HashSet<String>();
    set.add("1");
    set.add("2");
    set.add("3");
    set.add("4");
    set.add("1");
    set.add("2");
    set.add("5");

    for (String temp : set){
        System.out.println(temp);
    }
}
```

}

Exemple de liste de sortie ..... 1 2 3 4 1 Exemple de réglage 3 2 10 5 4

Lire Liste vs SET en ligne: https://riptutorial.com/fr/java/topic/10125/liste-vs-set

Chapitre 118: Littéraux

Introduction

Un littéral Java est un élément syntaxique (c'est-à-dire quelque chose que vous trouvez dans le *code source* d'un programme Java) qui représente une valeur. Les exemples sont `1` , `0.333F` , `false` , `'X'` et `"Hello world\n"` .

Examples

Littéraux hexadécimaux, octaux et binaires

Un nombre `hexadecimal` est une valeur de base-16. Il y a 16 chiffres, `0-9` et les lettres `AF` (la casse n'a pas d'importance). `AF` représente `10-16` .

Un nombre `octal` est une valeur en base 8 et utilise les chiffres `0-7` .

Un nombre `binary` est une valeur de base-2 et utilise les chiffres `0` et `1` .

Tous ces nombres donnent la même valeur, `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
```

Notez que la syntaxe littérale binaire a été introduite dans Java 7.

Le littéral octal peut facilement être un piège pour les erreurs sémantiques. Si vous définissez un `'0'` vos littéraux décimaux, vous obtiendrez une valeur incorrecte:

```
int a = 0100;        // Instead of 100, a == 64
```

Utiliser le soulignement pour améliorer la lisibilité

Depuis Java 7, il est possible d'utiliser un ou plusieurs traits de soulignement (_) pour séparer des groupes de chiffres dans un littéral de nombre primitif afin d'améliorer leur lisibilité.

Par exemple, ces deux déclarations sont équivalentes:

Java SE 7

```
int i1 = 123456;
int i2 = 123_456;
System.out.println(i1 == i2); // true
```

Cela peut être appliqué à tous les littéraux de nombres primitifs comme indiqué ci-dessous:

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;
```

Cela fonctionne également en utilisant des préfixes pour les bases binaires, octales et hexadécimales:

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;
```

Il y a quelques règles sur les soulignés qui **interdisent** leur placement dans les endroits suivants:

- Au début ou à la fin d'un nombre (par exemple, `_123` ou `123_` *ne* sont *pas* valides)
- Adjacent à un point décimal dans un littéral à virgule flottante (par exemple, `1._23` ou `1_.23` *ne* sont *pas* valides)
- Avant un suffixe F ou L (par exemple, `1.23_F` ou `9999999_L` *ne* sont *pas* valides)
- Dans les positions où une chaîne de chiffres est attendue (par exemple, `0_xFFFF` n'est *pas* valide)

Séquences d'échappement dans les littéraux

Les littéraux de chaîne et de caractère fournissent un mécanisme d'échappement qui permet des codes de caractères express qui ne seraient pas autorisés dans le littéral. Une séquence d'échappement consiste en une barre oblique inverse (`\`) suivie d'un ou plusieurs autres caractères. Les mêmes séquences sont valables dans les deux caractères et les chaînes de caractères.

L'ensemble complet des séquences d'échappement est le suivant:

Séquence d'échappement	Sens
`\\`	Indique une barre oblique inverse (`\`)
`\'`	Indique un caractère de guillemet simple (`'`)
`\"`	Indique un caractère double guillemet (`"`)
`\n`	Indique un caractère de `LF` ligne (`LF`)
`\r`	Indique un caractère de retour chariot (`CR`)
`\t`	Indique un caractère de tabulation horizontale (`HT`)
`\f`	Indique un caractère de flux de formulaire (`FF`)

Séquence d'échappement	Sens
`\b`	Indique un caractère de retour arrière (`BS`)
`\<octal>`	Indique un code de caractère compris entre 0 et 255.

Le `<octal>` ci-dessus consiste en un, deux ou trois chiffres octaux ("0" à "7") qui représentent un nombre compris entre 0 et 255 (décimal).

Notez qu'une barre oblique inverse suivie de tout autre caractère est une séquence d'échappement non valide. Les séquences d'échappement non valides sont traitées comme des erreurs de compilation par le JLS.

Référence:

- JLS 3.10.6. Séquences d'échappement pour les littéraux de caractères et de chaînes

Unicode s'échappe

En plus des séquences d'échappement de chaînes et de caractères décrites ci-dessus, Java possède un mécanisme d'échappement Unicode plus général, tel que défini dans JLS 3.3. Unicode s'échappe . Une sortie Unicode a la syntaxe suivante:

```
'\' 'u' <hex-digit> <hex-digit> <hex-digit> <hex-digit>
```

où `<hex-digit>` est l'un de `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'`, `'9'`, `'a'`, `'b'`, `'c'`, `'d'`, `'e'`, `'f'`, `'A'`, `'B'`, `'C'`, `'D'`, `'E'`, `'F'`.

Un échappement Unicode est mappé par le compilateur Java sur un caractère (à proprement parler une *unité de code* Unicode 16 bits) et peut être utilisé n'importe où dans le code source où le caractère mappé est valide. Il est couramment utilisé dans les littéraux de caractères et de chaînes lorsque vous devez représenter un caractère non-ASCII dans un littéral.

S'échapper dans les regexes

À déterminer

Littéraux décimaux entiers

Fournissent des valeurs des entiers qui peuvent être utilisés lorsque vous avez besoin d' un `byte`, `short`, `int`, à `long` ou `char` par exemple. (Cet exemple se concentre sur les formes décimales simples. D'autres exemples expliquent comment utiliser les littéraux en octal, hexadécimal et binaire, et l'utilisation de traits de soulignement pour améliorer la lisibilité.)

Littéraux entiers ordinaires

La forme la plus simple et la plus commune de littéral entier est un littéral entier décimal. Par

exemple:

```
0       // The decimal number zero       (type 'int')
1       // The decimal number one        (type 'int')
42      // The decimal number forty two  (type 'int')
```

Vous devez faire attention aux premiers zéros. Un zéro en tête entraîne l'interprétation d'un littéral entier en *octal et* non en décimal.

```
077    // This literal actually means 7 x 8 + 7 ... or 63 decimal!
```

Les littéraux entiers sont non signés. Si vous voyez quelque chose comme `-10` ou `+10` , ce sont en fait des *expressions* en utilisant les unaire `-` et unaires `+` opérateurs.

La plage de littéraux entiers de cette forme a un type intrinsèque de `int` et doit être comprise entre zéro et 2 ^{31} ou 2 147 483 648.

Notez que 2 ^{31} est `1` supérieur à `Integer.MAX_VALUE` . Littéraux de 0 à à `2147483647` peut être utilisé partout, mais il est une erreur de compilation à utiliser `2147483648` sans précédent unaire `-` opérateur. (En d'autres termes, il est réservé pour exprimer la valeur de `Integer.MIN_VALUE` .)

```
int max = 2147483647;     // OK
int min = -2147483648;    // OK
int tooBig = 2147483648;  // ERROR
```

Littéraux entiers longs

Les littéraux de type `long` sont exprimés en ajoutant un suffixe `L` Par exemple:

```
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
```

Notez que la distinction entre littéraux `int` et `long` est significative à d'autres endroits. Par exemple

```
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.
```

Référence: JLS 3.10.1 - Littéraux entiers

Littéraux booléens

Les littéraux booléens sont les littéraux les plus simples du langage de programmation Java. Les

deux valeurs `boolean` possibles sont représentées par les littéraux `true` et `false` . Ceux-ci sont sensibles à la casse. Par exemple:

```
boolean flag = true;    // using the 'true' literal
flag = false;           // using the 'false' literal
```

Littéraux de chaîne

Les littéraux de chaîne constituent le moyen le plus pratique de représenter des valeurs de chaîne dans le code source Java. Un littéral de chaîne se compose de:

- Un caractère de guillemet double (`"`).
- Zéro ou plusieurs autres caractères qui ne sont ni des guillemets doubles ni des sauts de ligne. (Un caractère barre oblique inverse (`\`) modifie la signification des caractères suivants; voir Séquences d'échappement dans les littéraux .)
- Un caractère de guillemet double.

Par exemple:

```
"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
```

Notez qu'un littéral de chaîne unique peut ne pas couvrir plusieurs lignes de code source. C'est une erreur de compilation pour qu'un saut de ligne (ou la fin du fichier source) survienne avant le double guillemet de fermeture d'un littéral. Par exemple:

```
"Jello world    // Compilation error (at the end of the line!)
```

Cordes longues

Si vous avez besoin d'une chaîne trop longue pour tenir sur une ligne, la façon conventionnelle de l'exprimer est de la diviser en plusieurs littéraux et d'utiliser l'opérateur de concaténation (`+`) pour les joindre. Par exemple

```
String typingPractice = "The quick brown fox " +
                        "jumped over " +
                        "the lazy dog"
```

Une expression comme ci-dessus constituée de chaînes littérales et de `+` satisfait aux exigences pour être une expression constante . Cela signifie que l'expression sera évaluée par le compilateur et représentée à l'exécution par un seul objet `String` .

Interning des littéraux de chaîne

Lorsque le fichier de classe contenant des littéraux de chaîne est chargé par la machine virtuelle Java, les objets `String` correspondants sont *internés* par le système d'exécution. Cela signifie qu'un littéral de chaîne utilisé dans plusieurs classes n'occupe pas plus d'espace que s'il était utilisé dans une classe.

Pour plus d'informations sur l'internement et le pool de chaînes, reportez-vous à l'exemple du pool de chaînes et du stockage de segments dans la rubrique Chaînes.

Le littéral nul

Le littéral Null (écrit en tant que `null`) représente la valeur unique du type NULL. Voici quelques exemples

```
MyClass object = null;
MyClass[] objects = new MyClass[]{new MyClass(), null, new MyClass()};

myMethod(null);

if (objects != null) {
     // Do something
}
```

Le type null est plutôt inhabituel. Il n'a pas de nom, vous ne pouvez donc pas l'exprimer en code source Java. (Et il n'a pas de représentation à l'exécution non plus.)

Le seul but du type null est d'être le type de `null` . Il est compatible avec tous les types de référence et peut être converti en type de référence. (Dans ce dernier cas, le cast n'implique pas de vérification du type à l'exécution.)

Enfin, `null` a la propriété que `null instanceof <SomeReferenceType>` évaluera à `false` , quel que soit le type.

Littéraux à virgule flottante

Les littéraux à virgule flottante fournissent des valeurs pouvant être utilisées lorsque vous avez besoin d'une instance `float` ou `double` . Il existe trois types de littéraux à virgule flottante.

- Formes décimales simples
- Formes décimales mises à l'échelle
- Formes hexadécimales

(Les règles de syntaxe JLS combinent les deux formes décimales en un seul formulaire. Nous les traitons séparément pour faciliter l'explication.)

Il existe des types littéraux distincts pour les littéraux `float` et `double` , exprimés en utilisant des suffixes. Les différentes formes utilisent des lettres pour exprimer différentes choses. Ces lettres sont insensibles à la casse.

Formes décimales simples

La forme la plus simple du littéral à virgule flottante consiste en un ou plusieurs chiffres décimaux et un point décimal (`.`) Et un suffixe facultatif (`f` , `F` , `d` ou `D`). Le suffixe facultatif vous permet de spécifier que le littéral est une valeur `float` (`f` ou `F`) ou `double` (`d` ou `D`). La valeur par défaut (quand aucun suffixe n'est spécifié) est `double` .

Par exemple

```
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)
```

En fait, les chiffres décimaux suivis d'un suffixe sont également des littéraux à virgule flottante.

```
1F      // means the same thing as 1.0F
```

La signification d'un littéral décimal est le nombre à virgule flottante IEEE le *plus proche* du nombre réel mathématique de précision infinie désigné par la forme décimale à virgule flottante. Cette valeur conceptuelle est convertie en représentation à virgule flottante binaire IEEE en utilisant *arrondi au plus proche* . (La sémantique précise de la conversion décimale est spécifiée dans javadocs pour `Double.valueOf(String)` et `Float.valueOf(String)` , en gardant à l'esprit qu'il existe des différences dans les syntaxes de nombres.)

Formes décimales mises à l'échelle

Les formes décimales mises à l'échelle consistent en une décimale simple avec une partie exposant introduite par un `E` ou un `e` et suivie d'un entier signé. La partie exposant est une main courte pour multiplier la forme décimale par une puissance de dix, comme indiqué dans les exemples ci-dessous. Il existe également un suffixe facultatif permettant de distinguer les littéraux `float` et `double` . Voici quelques exemples:

```
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 taille d'un littéral est limitée par la représentation (`float` ou `double`). C'est une erreur de compilation si le facteur d'échelle donne une valeur trop grande ou trop petite.

Formes hexadécimales

À partir de Java 6, il est possible d'exprimer des littéraux à virgule flottante en hexadécimal. La forme hexadécimale a une syntaxe analogue aux formes décimales simples et mises à l'échelle, avec les différences suivantes:

1. Chaque littéral hexadécimal à virgule flottante commence par un zéro (`0`), puis un `x` ou un `X`
2. Les chiffres du nombre (mais *pas* la partie exposant!) Incluent également les chiffres

hexadécimaux `a` à `f` et leurs équivalents majuscules.
3. L'exposant est *obligatoire* et introduit par la lettre `p` (ou `P`) au lieu de `e` ou `E` L'exposant représente un facteur d'échelle qui est une puissance de 2 au lieu d'une puissance de 10.

Voici quelques exemples:

```
0x0.0p0f    // this is zero expressed in hexadecimal form (`float`)
0xff.0p19   // this is 255.0 x 2^19 (`double`)
```

Conseil: comme la plupart des programmeurs Java ne connaissent pas les formulaires à virgule flottante hexadécimale, il est conseillé de les utiliser avec parcimonie.

Les dessous

À partir de Java 7, les traits de soulignement sont autorisés dans les chaînes de caractères des trois formes de littéral à virgule flottante. Cela s'applique également aux parties "exposant". Voir Utilisation des traits de soulignement pour améliorer la lisibilité .

Cas spéciaux

C'est une erreur de compilation si un littéral à virgule flottante dénote un nombre trop grand ou trop petit pour être représenté dans la représentation sélectionnée; c'est-à-dire si le nombre débordait à + INF ou -INF, ou à 0.0. Cependant, il est légal qu'un littéral représente un nombre dénormalisé non nul.

La syntaxe littérale à virgule flottante ne fournit pas de représentation littérale pour les valeurs spéciales IEEE 754 telles que les valeurs INF et NaN. Si vous devez les exprimer en code source, la méthode recommandée consiste à utiliser les constantes définies par `java.lang.Float` et `java.lang.Double` ; Par exemple `Float.NaN` , `Float.NEGATIVE_INFINITY` et `Float.POSITIVE_INFINITY` .

Littéraux de caractère

Littéraux de caractères constituent le moyen le plus pratique d'exprimer `char` valeurs dans le code source Java. Un littéral de caractère consiste en:

- Un caractère d'ouverture simple (`'`).
- Une représentation d'un personnage. Cette représentation ne peut pas être un caractère simple ou un caractère de saut de ligne, mais il peut s'agir d'une séquence d'échappement introduite par un caractère barre oblique inverse (`\`); voir Séquences d'échappement dans les littéraux .
- Un caractère de guillemet simple (`'`).

Par exemple:

```
char a = 'a';
char doubleQuote = '"';
char singleQuote = '\'';
```

Un saut de ligne dans un littéral de caractère est une erreur de compilation:

```
char newline = '
// Compilation error in previous line
char newLine = '\n';  // Correct
```

Lire Littéraux en ligne: https://riptutorial.com/fr/java/topic/8250/litteraux

Chapitre 119: Localisation et internationalisation

Remarques

Java est fourni avec un mécanisme puissant et flexible pour la localisation de vos applications, mais il est également facile d'utiliser un programme qui ignore ou modifie les paramètres régionaux de l'utilisateur, et donc son comportement.

Vos utilisateurs s'attendent à voir les données localisées dans les formats auxquels ils sont habitués, et tenter de les prendre en charge manuellement est une erreur. Voici juste un petit exemple des différentes manières dont les utilisateurs s'attendent à voir le contenu que vous pouvez supposer être "toujours" affiché d'une certaine manière:

	Rendez-vous	Nombres	Monnaie locale	Monnaie étrangère	Distances
Brésil					
Chine					
Egypte					
Mexique	20/3/16	1.234,56	1 000,50 $	1,000.50 USD	
Royaume-Uni	20/3/16	1 234,56	£ 1,000.50		100 km
Etats-Unis	20/03/16	1 234,56	1 000,50 $	1,000.50 MXN	60 mi

Ressources générales

- Wikipedia: Internationalisation et localisation

Ressources Java

- Tutoriel Java: Internationalisation
- Oracle: Internationalisation: Comprendre les paramètres régionaux dans la plate-forme Java
- JavaDoc: `Locale`

Examples

Format automatique des dates en utilisant "locale"

`SimpleDateFormatter` est génial dans un pincement, mais comme son nom l'indique, il ne s'adapte pas bien.

Si vous codez `"MM/dd/yyyy"` sur votre application, vos utilisateurs internationaux ne seront pas contents.

Laissez Java faire le travail pour vous

Utilisez les méthodes `static` dans `DateFormat` pour récupérer le format correct pour votre utilisateur. Pour une application de bureau (sur laquelle vous comptez sur les paramètres régionaux par défaut), appelez simplement:

```
String localizedDate = DateFormat.getDateInstance(style).format(date);
```

Où `style` est l'une des constantes de formatage (`FULL` , `LONG` , `MEDIUM` , `SHORT` , etc.) spécifiées dans `DateFormat` .

Pour une application côté serveur dans laquelle l'utilisateur spécifie ses paramètres régionaux dans le cadre de la requête, vous devez le transmettre explicitement à `getDateInstance()` place:

```
String localizedDate =
    DateFormat.getDateInstance(style, request.getLocale()).format(date);
```

Comparaison de chaîne

Comparez deux chaînes en ignorant la casse:

```
"School".equalsIgnoreCase("school"); // true
```

Ne pas utiliser

```
text1.toLowerCase().equals(text2.toLowerCase());
```

Les langues ont des règles différentes pour convertir les majuscules et les minuscules. Un "je" serait converti en "i" en anglais. Mais en turc, un «je» devient un «ı». Si vous devez utiliser `toLowerCase()` utiliser la surcharge qui attend une `Locale` : `String.toLowerCase(Locale)` .

En comparant deux chaînes en ignorant les différences mineures:

```
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.PRIMARY);
collator.equals("Gärten", "gaerten"); // returns true
```

Trier les chaînes en respectant l'ordre du langage naturel, en ignorant la casse (utilisez la clé de classement pour:

```
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"}
```

Lieu

La classe `java.util.Locale` est utilisée pour représenter une région "géographique, politique ou culturelle" permettant de localiser un texte, un numéro, une date ou une opération donnés. Un objet Locale peut donc contenir un pays, une région, une langue et également une variante d'une langue, par exemple un dialecte parlé dans une certaine région d'un pays ou parlé dans un pays différent du pays d'origine de la langue.

L'instance de paramètres régionaux est transmise aux composants qui doivent localiser leurs actions, que ce soit pour convertir l'entrée, la sortie ou simplement en avoir besoin pour des opérations internes. La classe Locale ne peut effectuer aucune internationalisation ou localisation par elle-même

La langue

La langue doit être un code de langue ISO 639 2 ou 3 caractères, ou une sous-étiquette de langue enregistrée de 8 caractères maximum. Si une langue a à la fois un code de langue à 2 et 3 caractères, utilisez le code à 2 caractères. Une liste complète des codes de langue se trouve dans le registre des sous-étiquettes de langues de l'IANA.

Les codes de langue sont insensibles à la casse, mais la classe Locale utilise toujours des versions minuscules des codes de langue

Créer une locale

La création d'une instance `java.util.Locale` peut se faire de quatre manières différentes:

```
Locale constants
Locale constructors
Locale.Builder class
Locale.forLanguageTag factory method
```

Java ResourceBundle

Vous créez une instance ResourceBundle comme ceci:

```
Locale locale = new Locale("en", "US");
ResourceBundle labels = ResourceBundle.getBundle("i18n.properties");
System.out.println(labels.getString("message"));
```

Considère que j'ai un fichier de propriétés `i18n.properties` :

```
message=This is locale
```

Sortie:

```
This is locale
```

Réglage des paramètres régionaux

Si vous souhaitez reproduire l'état en utilisant d'autres langages, vous pouvez utiliser la méthode **`setDefault()`** . Son utilisation:

```
 setDefault(Locale.JAPANESE); //Set Japanese
```

Lire Localisation et internationalisation en ligne:
https://riptutorial.com/fr/java/topic/4086/localisation-et-internationalisation

Chapitre 120: log4j / log4j2

Introduction

Apache Log4j est un utilitaire de journalisation basé sur Java, il s'agit de l'un des nombreux frameworks de journalisation Java. Cette rubrique explique comment configurer et configurer Log4j en Java avec des exemples détaillés sur tous les aspects possibles de son utilisation.

Syntaxe

- Logger.debug ("text to log"); // Enregistrement des informations de débogage
- Logger.info ("text to log"); // Enregistrement d'informations communes
- Logger.error ("text to log"); // Informations d'erreur de journalisation
- Logger.warn ("text to log"); // Avertissement de journalisation
- Logger.trace ("text to log"); // Enregistrement des informations de trace
- Logger.fatal ("text to log"); // Enregistrement des erreurs fatales
- Utilisation de Log4j2 avec la journalisation des paramètres:
- Logger.debug ("Paramètres de débogage {} {} {}", param1, param2, param3); // Déboguer le débogage avec les paramètres
- Logger.info ("Info params {} {} {}", param1, param2, param3); // Informations de journalisation avec paramètres
- Logger.error ("Erreur params {} {} {}", param1, param2, param3); // Erreur de journalisation avec les paramètres
- Logger.warn ("Avertir params {} {} {}", param1, param2, param3); // Enregistrement des avertissements avec des paramètres
- Logger.trace ("Trace params {} {} {}", param1, param2, param3); // Trace de trace avec paramètres
- Logger.fatal ("Paramètres fatals {} {} {}", param1, param2, param3); // Enregistrement fatal avec des paramètres
- Logger.error ("Exception Caught:", ex); // Exception de journalisation avec message et stacktrace (sera automatiquement ajouté)

Remarques

Fin de vie pour Log4j 1 atteint

> Le 5 août 2015, le comité de gestion du projet Logging Services a annoncé que Log4j 1.x était en fin de vie. Pour le texte complet de l'annonce, veuillez consulter le blog Apache. **Les utilisateurs de Log4j 1 sont invités à passer à Apache Log4j 2** .

De: http://logging.apache.org/log4j/1.2/

Examples

Comment obtenir Log4j

Version actuelle (log4j2)

En utilisant Maven:
Ajoutez la dépendance suivante à votre fichier `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>
```

En utilisant 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>
```

Utiliser 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'
}
```

Obtenir log4j 1.x

Remarque: Log4j 1.x a atteint la fin de vie (EOL) (voir Remarques).

En utilisant Maven:

Déclarez cette dépendance dans le fichier `POM.xml` :

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
```

```
</dependency>
```

En utilisant Ivy:

```
<dependency org="log4j" name="log4j" rev="1.2.17"/>
```

Utilisez Gradle:

```
compile group: 'log4j', name: 'log4j', version: '1.2.17'
```

Utiliser Buildr:

```
'log4j:log4j:jar:1.2.17'
```

Ajouter manuellement dans la construction du chemin:

Télécharger depuis le projet de site Web Log4j

Comment utiliser Log4j en code Java

Vous devez d'abord créer un objet de `final static logger` :

```
final static Logger logger = Logger.getLogger(classname.class);
```

Ensuite, appelez les méthodes de journalisation:

```
//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);
```

Configuration du fichier de propriétés

Log4j vous offre la possibilité de consigner les données dans la console et les fichiers en même temps. Créez un fichier `log4j.properties` et placez-le dans cette configuration de 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
```

Si vous utilisez maven, placez ce fichier de propriétés dans le chemin:

```
/ProjectFolder/src/java/resources
```

Fichier de configuration de 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>
```

Ceci est une configuration de base log4j2.xml qui a un appender console et un enregistreur racine. La disposition du modèle spécifie quel modèle doit être utilisé pour consigner les instructions.
Pour déboguer le chargement de log4j2.xml, vous pouvez ajouter l'attribut `status = <WARN | DEBUG | ERROR | FATAL | TRACE | INFO>` dans la balise de configuration de votre log4j2.xml.
Vous pouvez également ajouter un intervalle de surveillance pour qu'il charge à nouveau la configuration après la période d'intervalle spécifiée. L'intervalle de surveillance peut être ajouté à la balise de configuration comme suit: `monitorInterval = 30` . Cela signifie que la configuration sera chargée toutes les 30 secondes.

Migration de log4j 1.x à 2.x

Si vous souhaitez migrer de log4j 1.x existant dans votre projet vers log4j 2.x, supprimez toutes les dépendances log4j 1.x existantes et ajoutez la dépendance suivante:

Pont API Log4j 1.x

Maven Build

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-1.2-api</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Ivy 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 Si votre projet utilise Apache Commons Logging qui utilise log4j 1.x et que vous souhaitez le migrer vers log4j 2.x, ajoutez les dépendances suivantes:

Maven Build

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-jcl</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Ivy 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'
}
```

Remarque: Ne supprimez aucune dépendance existante de la journalisation Apache commons

Référence: https://logging.apache.org/log4j/2.x/maven-artifacts.html

Fichier de propriétés pour se connecter à la base de données

Pour que cet exemple fonctionne, vous aurez besoin d'un pilote JDBC compatible avec le système

sur lequel la base de données est exécutée. Un fichier opensource vous permettant de vous connecter aux bases de données DB2 sur un système IBM peut être trouvé ici: JT400

Même si cet exemple est spécifique à DB2, il fonctionne pour presque tous les autres systèmes si vous échangez le pilote et adaptez 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
```

Filtrer le flux de sortie par niveau (log4j 1.x)

Vous pouvez utiliser un filtre pour enregistrer uniquement les messages "inférieurs", par exemple, au niveau `ERROR` . **Mais le filtre n'est pas pris en charge par PropertyConfigurator. Vous devez donc passer à la configuration XML pour l'utiliser** . Voir log4j-Wiki sur les filtres .

Exemple "niveau spécifique"

```
<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>
```

Ou "Level range"

```
<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>
```

Lire log4j / log4j2 en ligne: https://riptutorial.com/fr/java/topic/2472/log4j---log4j2

Chapitre 121: Méthodes de classe d'objet et constructeur

Introduction

Cette page de documentation permet d'afficher des détails avec des exemples concernant les constructeurs de classes java et les méthodes de classe d'objets qui sont automatiquement héritées de l' `Object` superclasse de toute classe nouvellement créée.

Syntaxe

- Classe native finale publique <?> getClass ()
- public final final void notify ()
- public final final void notifyAll ()
- Attente de void final native publique (long timeout) lève InterruptedException
- public final void wait () lève InterruptedException
- attente finale publique vide (long timeout, int nanos) lève InterruptedException
- public natif Int hashCode ()
- booléen public égal à (objet obj)
- public String toString ()
- Objet protégé natif clone () lève CloneNotSupportedException
- protected void finalize () lance Throwable

Examples

méthode toString ()

La `toString()` permet de créer une représentation `String` d'un objet en utilisant le contenu de l'objet. Cette méthode doit être remplacée lors de l'écriture de votre classe. `toString()` est appelé implicitement lorsqu'un objet est concaténé en une chaîne comme dans `"hello " + anObject` .

Considérer ce qui suit:

```
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"
    }
}
```

Ici, `toString()` from `Object` class est remplacé dans la classe `User` pour fournir des données significatives concernant l'objet lors de son impression.

Lors de l'utilisation de `println()` , la `toString()` l'objet est appelée implicitement. Par conséquent, ces déclarations font la même chose:

```
System.out.println(user); // toString() is implicitly called on `user`
System.out.println(user.toString());
```

Si `toString()` n'est pas remplacé dans la classe `User` mentionnée ci-dessus, `System.out.println(user)` peut renvoyer `User@659e0bfd` ou une String similaire avec presque aucune information utile à l'exception du nom de la classe. Cela sera dû au fait que l'appel utilisera l'implémentation de `toString()` de la classe d' `Object` Java de base qui ne connaît rien de la structure ou des règles métier de la classe `User` . Si vous voulez modifier cette fonctionnalité dans votre classe, remplacez simplement la méthode.

méthode equals ()

TL; DR

`==` teste l'égalité de référence (qu'ils soient le ***même objet***)

`.equals()` teste l'égalité des valeurs (qu'elles soient ***logiquement "égales"***)

`equals()` est une méthode utilisée pour comparer deux objets pour l'égalité. L'implémentation par défaut de la méthode `equals()` dans la classe `Object` renvoie `true` si et seulement si les deux références pointent vers la même instance. Il se comporte donc comme la comparaison par `==` .

```
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
    }
}
```

Même si `foo1` et `foo2` sont créés avec les mêmes champs, ils pointent vers deux objets différents en mémoire. Par conséquent, l'implémentation `equals()` par défaut `equals()` évaluée à `false` .

Pour comparer le contenu d'un objet à l'égalité, `equals()` doit être remplacé.

```
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
    }
}
```

Ici, la méthode `equals()` surchargée décide que les objets sont égaux si leurs champs sont identiques.

Notez que la `hashCode()` a également été remplacée. Le contrat de cette méthode indique que lorsque deux objets sont égaux, leurs valeurs de hachage doivent également être identiques. C'est pourquoi il faut presque toujours remplacer `hashCode()` et `equals()` ensemble.

Portez une attention particulière au type d’argument de la méthode `equals` . C'est `Object obj` , pas `Foo obj` . Si vous mettez cette dernière dans votre méthode, cela ne remplace pas la méthode `equals` .

Lors de l'écriture de votre propre classe, vous devrez écrire une logique similaire lors de la substitution de `equals()` et `hashCode()` . La plupart des IDE peuvent automatiquement générer cela pour vous.

Un exemple d'implémentation `equals()` peut être trouvé dans la classe `String` , qui fait partie de l'API Java principale. Plutôt que de comparer des pointeurs, la classe `String` compare le contenu de la `String` .

Java SE 7

Java 1.7 a introduit la classe `java.util.Objects` qui fournit une méthode pratique, `equals` , qui compare deux références potentiellement `null` , afin de simplifier les implémentations de la méthode `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);
}
```

Comparaison de classes

Comme la méthode equals peut s'exécuter sur n'importe quel objet, l'une des premières actions de la méthode (après avoir vérifié la valeur `null`) consiste à vérifier si la classe de l'objet comparé correspond à la classe en cours.

```
@Override
public boolean equals(Object obj) {
    //...check for null
    if (getClass() != obj.getClass()) {
        return false;
    }
    //...compare fields
}
```

Cela se fait généralement comme ci-dessus en comparant les objets de classe. Cependant, cela peut échouer dans quelques cas particuliers qui peuvent ne pas être évidents. Par exemple, certains frameworks génèrent des proxies dynamiques de classes et ces proxy dynamiques sont en fait une classe différente. Voici un exemple d'utilisation de 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 mécanisme pour contourner cette limitation consiste à comparer les classes en utilisant `instanceof`

```
@Override
public final boolean equals(Object obj) {
    if (!(obj instanceof Foo)) {
        return false;
    }
    //...compare fields
}
```

Cependant, il y a quelques pièges à éviter lors de l'utilisation de `instanceof` . Puisque Foo pourrait potentiellement avoir d'autres sous-classes et que ces sous-classes pourraient remplacer `equals()` vous pourriez entrer dans un cas où un `Foo` est égal à un `FooSubclass` mais le `FooSubclass` n'est pas égal à `Foo` .

```
Foo foo = new Foo(7);
FooSubclass fooSubclass = new FooSubclass(7, false);
foo.equals(fooSubclass) //true
fooSubclass.equals(foo) //false
```

Cela viole les propriétés de symétrie et de transitivité et constitue donc une implémentation invalide de la méthode `equals()` . Par conséquent, lorsque vous utilisez `instanceof` , une bonne pratique consiste à rendre la méthode `equals() final` (comme dans l'exemple ci-dessus). Cela garantira qu'aucune sous-classe ne remplace `equals()` et viole les hypothèses clés.

Méthode hashCode ()

Lorsqu'une classe Java remplace la méthode `equals` , elle devrait également remplacer la méthode `hashCode` . Comme défini dans le contrat de la méthode :

- Chaque fois qu'il est appelé sur le même objet plus d'une fois lors de l'exécution d'une application Java, la méthode `hashCode` doit systématiquement renvoyer le même entier, à condition qu'aucune information utilisée dans les comparaisons d'égal à égal sur l'objet ne soit modifiée. Cet entier ne doit pas nécessairement rester cohérent d'une exécution d'une application à une autre exécution de la même application.
- Si deux objets sont égaux selon la méthode `equals(Object)` , alors l'appel de la méthode `hashCode` sur chacun des deux objets doit produire le même résultat entier.
- Il n'est pas obligatoire que si deux objets sont inégaux selon la méthode `equals(Object)` , alors l'appel de la méthode `hashCode` sur chacun des deux objets doit produire des résultats entiers distincts. Cependant, le programmeur doit savoir que produire des résultats entiers distincts pour des objets inégaux peut améliorer les performances des tables de hachage.

Les codes de hachage sont utilisés dans les implémentations de hachage telles que `HashMap` , `HashTable`

et `HashSet` . Le résultat de la fonction `hashCode` détermine le compartiment dans lequel un objet sera placé. Ces implémentations de hachage sont plus efficaces si l'implémentation `hashCode` fournie est bonne. Une propriété importante de l'implémentation de `hashCode` est que la distribution des valeurs `hashCode` est uniforme. En d'autres termes, il existe une faible probabilité que de nombreuses instances soient stockées dans le même compartiment.

Un algorithme de calcul d'une valeur de code de hachage peut être similaire à celui-ci:

```
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;
    }
}
```

Utiliser Arrays.hashCode () comme raccourci

Java SE 1.2

Dans Java 1.2 et versions ultérieures, au lieu de développer un algorithme pour calculer un code de hachage, vous pouvez en générer un en utilisant `java.util.Arrays#hashCode` en fournissant un tableau Object ou primitives contenant les valeurs de champ:

```
@Override
public int hashCode() {
    return Arrays.hashCode(new Object[] {field1, field2, field3});
}
```

Java SE 7

Java 1.7 a introduit la classe `java.util.Objects` qui fournit une méthode pratique, `hash(Object... objects)` , qui calcule un code de hachage basé sur les valeurs des objets qui lui sont fournis. Cette méthode fonctionne comme `java.util.Arrays#hashCode` .

```
@Override
public int hashCode() {
    return Objects.hash(field1, field2, field3);
}
```

Remarque: cette approche est inefficace et produit des objets indésirables chaque fois que votre `hashCode()` personnalisée est appelée:

- Un `Object[]` temporaire `Object[]` est créé. (Dans la version `Objects.hash()` , le tableau est créé par le mécanisme "varargs".)
- Si l'un des champs est de type primitif, il doit être encadré et créer plus d'objets temporaires.
- Le tableau doit être rempli.
- Le tableau doit être itéré par la méthode `Arrays.hashCode` ou `Objects.hash` .
- Les appels à `Object.hashCode() Arrays.hashCode` ou `Objects.hash` doit effectuer (probablement) ne peuvent pas être intégrés.

Mise en cache interne des codes de hachage

Étant donné que le calcul du code de hachage d'un objet peut être coûteux, il peut être intéressant de mettre en cache la valeur du code de hachage dans l'objet la première fois qu'il est calculé. Par exemple

```
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;
    }
}
```

Cette approche permet d'échanger le coût de (répétition) du calcul du code de hachage par rapport à la surcharge d'un champ supplémentaire pour mettre en cache le code de hachage. Que cela soit rentable en tant qu'optimisation des performances dépendra de la fréquence à laquelle un objet donné est haché (recherché) et d'autres facteurs.

Vous remarquerez également que si le vrai hashcode d'un `ImmutableArray` arrive à zéro (une chance sur 2 $^{32)}$, le cache est inefficace.

Enfin, cette approche est beaucoup plus difficile à implémenter correctement si l'objet que nous sommes en train de hacher est mutable. Cependant, il y a de plus grandes préoccupations si les codes de hachage changent; voir le contrat ci-dessus.

Méthodes wait () et notify ()

`wait()` et `notify()` fonctionnent en tandem - quand un thread appelle `wait()` sur un objet, ce thread va bloquer jusqu'à ce qu'un autre thread appelle `notify()` ou `notifyAll()` sur le même objet.

(Voir aussi: 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!");
    }
}
```

Un exemple de sortie:

```
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!
```

Méthode getClass ()

La méthode `getClass()` peut être utilisée pour rechercher le type de classe d'exécution d'un objet. Voir l'exemple ci-dessous:

```
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"
}
```

La méthode `getClass()` renvoie le type de classe le plus spécifique, ce qui explique pourquoi lorsque `getClass()` est appelée sur un `anotherSpecificUser` , la valeur `getClass()` est la `class SpecificUser` car celle-ci est inférieure à celle de l' `User` .

Il est à noter que, tandis que la méthode `getClass` est déclarée comme:

```
public final native Class<?> getClass();
```

Le type statique réel renvoyé par un appel à `getClass` est `Class<? extends T>` où `T` est le type statique de l'objet sur lequel `getClass` est appelé.

c'est à dire que ce qui suit compilera:

```
Class<? extends String> cls = "".getClass();
```

méthode clone ()

La méthode `clone()` est utilisée pour créer et renvoyer une copie d'un objet. Cette méthode discutable devrait être évitée car elle pose problème et un constructeur de copie ou une autre méthode de copie devrait être utilisée en faveur de `clone()` .

Pour que la méthode soit utilisée, toutes les classes appelant la méthode doivent implémenter l'interface `Cloneable` .

L'interface `Cloneable` elle-même n'est qu'une interface de balise utilisée pour modifier le comportement de la méthode `clone() native` qui vérifie si la classe d'objets appelants implémente `Cloneable` . Si l'appelant `CloneNotSupportedException` pas cette interface, une `CloneNotSupportedException` sera lancée.

La classe `Object` elle-même `CloneNotSupportedException` pas cette interface, donc une `CloneNotSupportedException` sera lancée si l'objet appelant est de classe `Object` .

Pour qu'un clone soit correct, il doit être indépendant de l'objet à partir duquel il est cloné. Par conséquent, il peut être nécessaire de modifier l'objet avant qu'il ne soit renvoyé. Cela signifie essentiellement créer une "copie profonde" en copiant également l'un des objets *mutables* qui constituent la structure interne de l'objet en cours de clonage. Si cela n'est pas implémenté correctement, l'objet cloné ne sera pas indépendant et aura les mêmes références aux objets mutables que l'objet à partir duquel il a été cloné. Cela se traduirait par un comportement incohérent, car toute modification de l'un de ces éléments affecterait l'autre.

```
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);
        }
    }
}
```

méthode finalize ()

C'est une méthode *protégée* et *non statique* de la classe `Object` . Cette méthode est utilisée pour effectuer certaines opérations finales ou pour effectuer des opérations de nettoyage sur un objet avant qu'il ne soit supprimé de la mémoire.

> Selon le document, cette méthode est appelée par le ramasse-miettes sur un objet lorsque la récupération de la mémoire détermine qu'il n'y a plus de références à l'objet.

Mais il n'y a aucune garantie que la méthode `finalize()` soit appelée si l'objet est toujours accessible ou si aucun récupérateur de place n'est exécuté lorsque l'objet devient éligible. C'est pourquoi il vaut mieux **ne pas compter** sur cette méthode.

Dans les librairies Java, des exemples d'utilisation ont pu être trouvés, par exemple dans `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();
    }
}
```

Dans ce cas, c'est la dernière chance de fermer la ressource si cette ressource n'a pas encore été fermée.

En général, l'utilisation de la méthode `finalize()` dans des applications de toute nature est considérée comme une mauvaise pratique et devrait être évitée.

Les finaliseurs *ne* sont *pas* destinés à libérer des ressources (par exemple, la fermeture de fichiers). Le ramasse-miettes est appelé lorsque (si!) Le système manque de place sur le tas. Vous ne pouvez pas compter sur elle pour être appelée lorsque le système est à court de descripteurs de fichiers ou pour toute autre raison.

Le cas d'utilisation prévu pour les finaliseurs concerne un objet sur le point d'être récupéré pour notifier un autre objet de son sort imminent. Un meilleur mécanisme existe maintenant à cet effet - la classe `java.lang.ref.WeakReference<T>` . Si vous pensez avoir besoin d'écrire une méthode `WeakReference` `finalize()` , alors vous devriez vérifier si vous pouvez résoudre le même problème en utilisant `WeakReference` place. Si cela ne résout pas votre problème, vous devrez peut-être repenser votre conception à un niveau plus profond.

Pour plus de lecture, voici un article sur la méthode `finalize()` du livre "Effective Java" de Joshua Bloch.

Constructeur d'objet

Tous les constructeurs de Java doivent appeler le constructeur `Object` . Ceci est fait avec l'appel `super()` . Ce doit être la première ligne d'un constructeur. La raison en est que l'objet peut réellement être créé sur le segment de mémoire avant toute initialisation supplémentaire.

Si vous ne spécifiez pas l'appel à `super()` dans un constructeur, le compilateur le mettra pour vous.

Donc, ces trois exemples sont fonctionnellement identiques

avec appel explicite au constructeur `super()`

```
public class MyClass {

    public MyClass() {
        super();
    }
}
```

avec appel implicite au constructeur `super()`

```
public class MyClass {

    public MyClass() {
        // empty
    }
}
```

avec constructeur implicite

```
public class MyClass {

}
```

Qu'en est-il du constructeur-chaînage?

Il est possible d'appeler d'autres constructeurs comme première instruction d'un constructeur. Comme l'appel explicite à un super constructeur et l'appel à un autre constructeur doivent être les deux premières instructions, elles s'excluent mutuellement.

```
public class MyClass {

    public MyClass(int size) {

        doSomethingWith(size);

    }

    public MyClass(Collection<?> initialValues) {

        this(initialValues.size());
        addInitialValues(initialValues);
    }
}
```

L'appel de nouveau `MyClass(Arrays.asList("a", "b", "c"))` appellera le second constructeur avec l'argument List, qui à son tour déléguera au premier constructeur (qui déléguera implicitement à `super()`), puis appelez `addInitialValues(int size)` avec la deuxième taille de la liste. Ceci permet de réduire la duplication de code lorsque plusieurs constructeurs doivent effectuer le même travail.

Comment appeler un constructeur spécifique?

Compte tenu de l'exemple ci-dessus, on peut appeler `new MyClass("argument")` ou `new MyClass("argument", 0)` . En d'autres termes, tout comme la surcharge de méthodes , il vous suffit d'appeler le constructeur avec les paramètres nécessaires au constructeur choisi.

> Que se passera-t-il dans le constructeur de la classe Object?

Rien de plus que cela n'arriverait dans une sous-classe qui a un constructeur vide par défaut (moins l'appel à `super()`).

Le constructeur vide par défaut peut être explicitement défini, mais sinon le compilateur le mettra pour vous tant qu'aucun autre constructeur n'est déjà défini.

> Comment un objet est-il alors créé à partir du constructeur dans Object?

La création réelle des objets incombe à la machine virtuelle Java. Chaque constructeur en Java apparaît comme une méthode spéciale nommée `<init>` qui est responsable de l'initialisation de l'instance. Cette méthode `<init>` est fournie par le compilateur et comme `<init>` n'est pas un identifiant valide en Java, il ne peut pas être utilisé directement dans le langage.

> Comment la JVM appelle-t-elle cette méthode `<init>` ?

La JVM `invokespecial` la méthode `<init>` à l'aide de l'instruction `invokespecial` et ne peut être invoquée que sur des instances de classe non initialisées.

Pour plus d'informations, consultez la spécification JVM et la spécification du langage Java:

- Méthodes spéciales (JVM) - JVMS - 2.9
- Constructeurs - JLS - 8.8

Lire Méthodes de classe d'objet et constructeur en ligne:
https://riptutorial.com/fr/java/topic/145/methodes-de-classe-d-objet-et-constructeur

Chapitre 122: Méthodes de collecte d'usine

Introduction

L'arrivée de Java 9 apporte de nombreuses nouvelles fonctionnalités à l'API Collections de Java, parmi lesquelles les méthodes de collecte. Ces méthodes permettent une initialisation facile des collections **immuables** , qu'elles soient vides ou non.

Notez que ces méthodes d'usine ne sont disponibles que pour les interfaces suivantes: `List<E>` , `Set<E>` et `Map<K, V>`

Syntaxe

- `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)`

Paramètres

Méthode avec paramètre	La description
`List.of(E e)`	Un type générique pouvant être une classe ou une interface.
`Set.of(E e)`	Un type générique pouvant être une classe ou une interface.
`Map.of(K k, V v)`	Une paire clé-valeur de types génériques pouvant chacun être une classe ou une interface.
`Map.of(Map.Entry<? extends K, ? extends V> entry)`	Une instance `Map.Entry` où sa clé peut être `K` ou l'un de ses enfants, et sa valeur peut être `V` ou l'un de ses enfants.

Examples

liste Exemples de méthode d'usine

- `List<Integer> immutableEmptyList = List.of();`
  - Initialise une `List<Integer>` immuable vide `List<Integer>` .
- `List<Integer> immutableList = List.of(1, 2, 3, 4, 5);`
  - Initialise une `List<Integer>` immuable `List<Integer>` avec cinq éléments initiaux.
- `List<Integer> mutableList = new ArrayList<>(immutableList);`
  - Initialise une `List<Integer>` mutable `List<Integer>` partir d'une `List<Integer>` immuable `List<Integer>` .

Ensemble Exemples de méthode d'usine

- `Set<Integer> immutableEmptySet = Set.of();`
  - Initialise un `Set<Integer>` immuable vide `Set<Integer>` .
- `Set<Integer> immutableSet = Set.of(1, 2, 3, 4, 5);`
  - Initialise un `Set<Integer>` immuable `Set<Integer>` avec cinq éléments initiaux.
- `Set<Integer> mutableSet = new HashSet<>(immutableSet);`
  - Initialise un `Set<Integer>` modifiable `Set<Integer>` partir d'un `Set<Integer>` immuable `Set<Integer>` .

Carte Exemples de méthode d'usine

- `Map<Integer, Integer> immutableEmptyMap = Map.of();`
  - Initialise une `Map<Integer, Integer>` immuable vide `Map<Integer, Integer>` .
- `Map<Integer, Integer> immutableMap = Map.of(1, 2, 3, 4);`
  - Initialise une `Map<Integer, Integer>` immuable `Map<Integer, Integer>` avec deux entrées de valeur-clé initiales.
- `Map<Integer, Integer> immutableMap = Map.ofEntries(Map.entry(1, 2), Map.entry(3, 4));`
  - Initialise une `Map<Integer, Integer>` immuable `Map<Integer, Integer>` avec deux entrées de valeur-clé initiales.
- `Map<Integer, Integer> mutableMap = new HashMap<>(immutableMap);`
  - Initialise une `Map<Integer, Integer>` mutable `Map<Integer, Integer>` partir d'une `Map<Integer, Integer>` immuable `Map<Integer, Integer>` .

Lire Méthodes de collecte d'usine en ligne: https://riptutorial.com/fr/java/topic/9783/methodes-de-collecte-d-usine

Chapitre 123: Méthodes par défaut

Introduction

La méthode par défaut introduite dans Java 8 permet aux développeurs d'ajouter de nouvelles méthodes à une interface sans casser les implémentations existantes de cette interface. Il offre une certaine souplesse pour permettre à l'interface de définir une implémentation qui sera utilisée par défaut lorsqu'une classe qui implémente cette interface ne parvient pas à fournir une implémentation de cette méthode.

Syntaxe

- public par défaut void methodName () {/ * method body * /}

Remarques

Méthodes par défaut

- Peut être utilisé dans une interface pour introduire un comportement sans forcer les sous-classes existantes à l'implémenter.
- Peut être remplacé par des sous-classes ou par une sous-interface.
- Ne sont pas autorisés à remplacer les méthodes dans la classe java.lang.Object.
- Si une classe implémentant plusieurs interfaces hérite des méthodes par défaut avec des signatures de méthode identiques pour chacune des interfaces, elle doit alors remplacer et fournir sa propre interface comme si elles n'étaient pas des méthodes par défaut (dans le cadre de la résolution de l'héritage multiple).
- Bien que soient destinés à introduire un comportement sans casser les implémentations existantes, les sous-classes existantes avec une méthode statique avec la même signature de méthode que la méthode par défaut nouvellement introduite seront toujours interrompues. Cependant, cela est vrai même en cas d'introduction d'une méthode d'instance dans une super-classe.

Méthodes statiques

- Peut être utilisé dans une interface, principalement destinée à être utilisée comme méthode utilitaire pour les méthodes par défaut.
- Ne peut pas être remplacé par des sous-classes ou par une sous-interface (qui leur est masquée). Cependant, comme c'est déjà le cas avec les méthodes statiques, chaque classe ou interface peut avoir son propre nom.
- Ne sont pas autorisés à remplacer les méthodes d'instance dans la classe java.lang.Object

(comme c'est également le cas actuellement pour les sous-classes).

Vous trouverez ci-dessous un tableau résumant l'interaction entre la sous-classe et la super-classe.

-	SUPER_CLASS-INSTANCE-METHOD	MÉTHODE SUPER_CLASS-STATIC
SUB_CLASS-INSTANCE-METHOD	*annule*	*génère-compiletime-error*
SUB_CLASS-STATIC-METHOD	*génère-compiletime-error*	*se cache*

Vous trouverez ci-dessous un tableau résumant l'interaction entre l'interface et la classe d'implémentation.

-	METHODE INTERFACE-DEFAULT	METHODE INTERFACE STATIQUE
IMPL_CLASS-INSTANCE-METHOD	*annule*	*se cache*
IMPL_CLASS-STATIC-METHOD	*génère-compiletime-error*	*se cache*

Les références :

- http://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method
- https://docs.oracle.com/javase/tutorial/java/IandI/override.html

Examples

Utilisation basique des méthodes par défaut

```
/**
 * 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" );
    }
}
```

Les déclarations suivantes

```
    new WithDefault().printString();
    new OverrideDefault().printString();
```

Produira cette sortie:

```
default implementation
overridden implementation
```

Accéder à d'autres méthodes d'interface avec la méthode par défaut

Vous pouvez également accéder à d'autres méthodes d'interface à partir de votre méthode par défaut.

```
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 déclaration suivante imprimera *3* :

```
System.out.println(new Sum().calculateSum());
```

Les méthodes par défaut peuvent également être utilisées avec les méthodes statiques d'interface:

```
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 déclaration suivante imprimera également 3:

```
System.out.println(new Sum().calculateSum());
```

Accès aux méthodes par défaut remplacées à partir de la classe d'implémentation

Dans les classes, `super.foo()` recherchera uniquement les superclasses. Si vous souhaitez appeler une implémentation par défaut à partir d'une superinterface, vous devez vous qualifier `super` avec le nom de l'interface: `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
    }
}
```

Pourquoi utiliser les méthodes par défaut?

La réponse simple est que cela vous permet de faire évoluer une interface existante sans casser les implémentations existantes.

Par exemple, vous avez l'interface `Swim` que vous avez publiée il y a 20 ans.

```
public interface Swim {
    void backStroke();
}
```

Nous avons fait un excellent travail, notre interface est très populaire, il y a beaucoup d'implémentations à ce sujet partout dans le monde et vous n'avez pas le contrôle de leur code source.

```
public class FooSwimmer implements Swim {
    public void backStroke() {
         System.out.println("Do backstroke");
    }
}
```

Après 20 ans, vous avez décidé d'ajouter de nouvelles fonctionnalités à l'interface, mais il semble que notre interface soit gelée car elle brisera les implémentations existantes.

Heureusement, Java 8 introduit une toute nouvelle fonctionnalité appelée méthode par défaut.

Nous pouvons maintenant ajouter une nouvelle méthode à l'interface `Swim` .

```
public interface Swim {
    void backStroke();
    default void sideStroke() {
        System.out.println("Default sidestroke implementation. Can be overridden");
    }
}
```

Maintenant, toutes les implémentations existantes de notre interface peuvent encore fonctionner. Mais surtout, ils peuvent mettre en œuvre la méthode nouvellement ajoutée à leur propre rythme.

L'une des principales raisons de ce changement, et l'une de ses utilisations les plus importantes, réside dans le cadre des collections Java. Oracle n'a pas pu ajouter une méthode `foreach` à l'interface Iterable existante sans casser tout le code existant qui implémentait Iterable. En ajoutant des méthodes par défaut, l'implémentation Iterable existante héritera de l'implémentation par défaut.

Classe, classe abstraite et préséance de la méthode d'interface

Les implémentations dans les classes, y compris les déclarations abstraites, ont priorité sur toutes les valeurs par défaut de l'interface.

- La méthode de classe abstraite a priorité sur la méthode d' interface par défaut .

```
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 déclaration suivante

```
new FooSwimmer().backStroke();
```

Produira

```
AbstractSwimmer.backStroke
```

- La méthode de classe est prioritaire sur la méthode d'interface par défaut

```
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 déclaration suivante

```
new FooSwimmer().backStroke();
```

Produira

```
FooSwimmer.backStroke
```

Méthode par défaut collision par héritage multiple

Prenons l'exemple suivant:

```
public interface A {
    default void foo() { System.out.println("A.foo"); }
}

public interface B {
```

```
    default void foo() { System.out.println("B.foo"); }
}
```

Voici deux interfaces déclarant la méthode `default foo` avec la même signature.

Si vous essayez d' `extend` ces deux interfaces dans la nouvelle interface, vous devez en choisir deux, car Java vous oblige à résoudre cette collision explicitement.

Tout d'abord , vous pouvez déclarer la méthode `foo` avec la même signature que `abstract` , ce qui remplacera `B` comportements `A` et `B`

```
public interface ABExtendsAbstract extends A, B {
    @Override
    void foo();
}
```

Et lorsque vous allez `implement ABExtendsAbstract` dans la `class` vous devrez fournir une implémentation `foo` :

```
public class ABExtendsAbstractImpl implements ABExtendsAbstract {
    @Override
    public void foo() { System.out.println("ABImpl.foo"); }
}
```

Ou **deuxièmement** , vous pouvez fournir une implémentation `default` complètement nouvelle. Vous pouvez également réutiliser le code des méthodes `foo` `A` et `B` en accédant aux méthodes par défaut substituées à partir de la classe d'implémentation .

```
public interface ABExtends extends A, B {
    @Override
    default void foo() { System.out.println("ABExtends.foo"); }
}
```

Et lorsque vous `implement ABExtends` dans la `class` vous `not` devrez `not` fournir d'implémentation `foo` :

```
public class ABExtendsImpl implements ABExtends {}
```

Lire Méthodes par défaut en ligne: https://riptutorial.com/fr/java/topic/113/methodes-par-defaut

Chapitre 124: Modèle de mémoire Java

Remarques

Le modèle de mémoire Java est la section du JLS qui spécifie les conditions dans lesquelles un thread est assuré de voir les effets des écritures mémoire effectuées par un autre thread. La section pertinente dans les éditions récentes est "Modèle de mémoire JLS 17.4" (en Java 8 , Java 7 , Java 6)

Il y a eu une refonte majeure du modèle de mémoire Java dans Java 5 qui (entre autres choses) a changé la façon dont le `volatile` fonctionnait. Depuis lors, le modèle de mémoire n'a pratiquement pas changé.

Examples

Motivation pour le modèle de mémoire

Prenons l'exemple suivant:

```
public class Example {
    public int a, b, c, d;

    public void doIt() {
       a = b + 1;
       c = d + 1;
    }
}
```

Si cette classe est utilisée comme une application à un seul thread, le comportement observable sera exactement comme prévu. Par exemple:

```
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);
    }
}
```

va sortir:

```
0, 0
1, 1
```

Dans la mesure où le thread "main" peut le savoir , les instructions de la méthode `main()` et de la méthode `doIt()` seront exécutées dans l'ordre où elles sont écrites dans le code source. Ceci est une exigence claire de la spécification de langage Java (JLS).

Considérons maintenant la même classe utilisée dans une application multithread.

```
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);
        }
    }
}
```

Qu'est-ce que cette impression?

En fait, selon le JLS, il n'est pas possible de prédire que cela va imprimer:

- Vous allez probablement voir quelques lignes de `0, 0` pour commencer.
- Ensuite, vous voyez probablement des lignes comme `N, N` ou `N, N + 1` .
- Vous pourriez voir des lignes comme `N + 1, N`
- En théorie, vous pourriez même voir que les lignes `0, 0` continuent pour toujours [1] .

1 - En pratique, la présence des instructions `println` est susceptible de provoquer des synchronisations et des vidages de mémoire cache inopinés. Cela risque de masquer certains des effets qui provoqueraient le comportement ci-dessus.

Alors, comment pouvons-nous les expliquer?

Réorganisation des missions

Une explication possible des résultats inattendus est que le compilateur JIT a modifié l'ordre des affectations dans la méthode `doIt()` . Le JLS exige que les instructions *apparaissent pour* s'exécuter dans *la perspective du* thread en cours. Dans ce cas, rien dans le code de la méthode `doIt()` ne peut observer l'effet d'une réorganisation (hypothétique) de ces deux instructions. Cela signifie que le compilateur JIT serait autorisé à le faire.

Pourquoi ça ferait ça?

Sur du matériel moderne typique, les instructions de la machine sont exécutées en utilisant un pipeline d'instructions qui permet à une séquence d'instructions de se trouver à différents stades. Certaines phases d'exécution des instructions prennent plus de temps que d'autres et les opérations de mémoire prennent plus de temps. Un compilateur intelligent peut optimiser le débit d'instructions du pipeline en ordonnant les instructions afin d'optimiser la quantité de chevauchement. Cela peut conduire à l'exécution de parties de relevés hors service. Le JLS permet cela à condition que cela n'affecte pas le résultat du calcul *du point de vue du thread en cours* .

Effets des caches de mémoire

Une deuxième explication possible est l'effet de la mise en cache de la mémoire. Dans une architecture informatique classique, chaque processeur possède un petit ensemble de registres et une plus grande quantité de mémoire. L'accès aux registres est beaucoup plus rapide que l'accès à la mémoire principale. Dans les architectures modernes, il existe des caches mémoire plus lents que les registres, mais plus rapides que la mémoire principale.

Un compilateur exploitera cela en essayant de conserver des copies des variables dans les registres ou dans les caches de mémoire. Si vous ne *devez* pas vider une variable dans la mémoire principale ou si vous *n'avez* pas *besoin* de la lire depuis la mémoire, le fait de ne pas le faire présente des avantages importants en termes de performances. Dans les cas où le JLS n'exige pas que les opérations de mémoire soient visibles par un autre thread, le compilateur Java JIT risque de ne pas ajouter les instructions "read barrier" et "write barrier" qui forceront les lectures et écritures de la mémoire principale. Encore une fois, les avantages de cette performance sont importants.

Bonne synchronisation

Jusqu'à présent, nous avons vu que le JLS permet au compilateur JIT de générer du code qui accélère le code mono-thread en réordonnant ou en évitant les opérations de mémoire. Mais que se passe-t-il lorsque d'autres threads peuvent observer l'état des variables (partagées) dans la mémoire principale?

La réponse est que les autres threads sont susceptibles d'observer des états de variables qui sembleraient impossibles ... sur la base de l'ordre de code des instructions Java. La solution consiste à utiliser la synchronisation appropriée. Les trois approches principales sont:

- Utilisation de mutex primitifs et des constructions `synchronized` .
- Utiliser `volatile` variables `volatile` .
- Utiliser un support concurrentiel de niveau supérieur; par exemple des classes dans les paquets `java.util.concurrent` .

Mais même avec cela, il est important de comprendre où la synchronisation est nécessaire et quels sont les effets sur lesquels vous pouvez compter. C'est là qu'intervient le modèle de mémoire Java.

Le modèle de mémoire

Le modèle de mémoire Java est la section du JLS qui spécifie les conditions dans lesquelles un thread est assuré de voir les effets des écritures mémoire effectuées par un autre thread. Le modèle de mémoire est spécifié avec un degré raisonnable de *rigueur formelle* et, par conséquent, nécessite une lecture détaillée et attentive pour comprendre. Mais le principe de base est que certaines constructions créent une relation "arrive-avant" entre l'écriture d'une variable par un thread et une lecture ultérieure de la même variable par un autre thread. Si la relation "arrive avant" existe, le compilateur JIT est *obligé* de générer du code qui garantira que l'opération de

lecture voit la valeur écrite par l'écriture.

Armé de cela, il est possible de raisonner sur la cohérence de la mémoire dans un programme Java, et de décider si cela sera prévisible et cohérent pour *toutes* les plates-formes d'exécution.

Des relations heureuses

(Ce qui suit est une version simplifiée de ce que dit Java Language Specification. Pour une compréhension plus approfondie, vous devez lire la spécification elle-même.)

Les relations heureuses sont la partie du modèle de mémoire qui nous permet de comprendre et de raisonner sur la visibilité de la mémoire. Comme le dit le JLS (JLS 17.4.5):

> "Deux *actions* peuvent être ordonnées par une relation *avant*- passé. Si une action *se produit avant* une autre, la première est visible et ordonnée avant la seconde."

Qu'est-ce que ça veut dire?

actes

Les actions auxquelles la citation ci-dessus fait référence sont spécifiées dans JLS 17.4.2 . Il y a 5 types d'action énumérés par la spécification:

- Lecture: lecture d'une variable non volatile.
- Écrire: écrire une variable non volatile.
- Actions de synchronisation:
  - Lecture volatile: Lecture d'une variable volatile.
  - Écriture volatile: écriture d'une variable volatile.
  - Fermer à clé. Verrouiller un moniteur
  - Ouvrir. Déverrouiller un moniteur.
  - Les premières et dernières actions (synthétiques) d'un thread.
  - Actions qui démarrent un thread ou détectent la fin d'un thread.
- Actions externes Une action qui a un résultat qui dépend de l'environnement dans lequel le programme.
- Actions de divergence de threads. Ils modélisent le comportement de certains types de boucle infinie.

Ordre de programme et ordre de synchronisation

Ces deux ordres (JLS 17.4.3 et JLS 17.4.4) régissent l'exécution des instructions dans un Java

L'ordre des programmes décrit l'ordre d'exécution des instructions dans un seul thread.

L'ordre de synchronisation décrit l'ordre d'exécution des instructions pour deux instructions connectées par une synchronisation:

- Une action de déverrouillage sur le moniteur se *synchronise avec* toutes les actions de verrouillage ultérieures sur ce moniteur.
- Une écriture dans une variable volatile se *synchronise avec* toutes les lectures ultérieures de la même variable par n'importe quel thread.
- Une action qui lance un thread (c'est-à-dire l'appel à `Thread.start()`) se *synchronise avec* la première action du thread qu'il démarre (c'est-à-dire l'appel à la méthode `run()` du thread).
- L'initialisation par défaut des champs se *synchronise avec* la première action de chaque thread. (Voir le JLS pour une explication à ce sujet.)
- L'action finale dans un thread se *synchronise avec* toute action dans un autre thread qui détecte la terminaison; Par exemple, le retour d'un appel `join()` ou d'un appel `isTerminated()` qui renvoie `true` .
- Si un thread interrompt un autre thread, l'appel d'interruption dans le premier thread se *synchronise avec* le point où un autre thread détecte que le thread a été interrompu.

Happens-before Order

Cet ordre (JLS 17.4.5) est ce qui détermine si une écriture en mémoire est garantie d'être visible pour une lecture de mémoire ultérieure.

Plus précisément, une lecture d'une variable `v` garantit une écriture dans `v` si et seulement si `write(v)` *se produit avant* `read(v)` ET il n'y a pas d'écriture intermédiaire dans `v` . S'il y a des écritures intermédiaires, alors la `read(v)` peut voir les résultats plutôt que la précédente.

Les règles qui définissent les *événements avant la* commande sont les suivantes:

- **Règle Happens-Before # 1** - Si x et y sont des actions du même thread et que x arrive avant y dans *l'ordre du programme* , alors x *se produit avant* y.
- **Règle Happens-Before # 2** - Il y a un bord de passe avant de la fin d'un constructeur d'un objet au début d'un finaliseur pour cet objet.
- **Règle Happens-Before # 3** - Si une action x se *synchronise avec* une action ultérieure y, alors x *se produit avant* y.
- **Happens-Before Rule # 4** - Si x *arrive -avant* y et y *se produit -avant* z alors x *arrive-avant* z.

De plus, diverses classes dans les bibliothèques standard Java sont spécifiées comme définissant

les relations *avant-terme* . Vous pouvez interpréter cela comme signifiant que cela se produit en *quelque sorte* , sans avoir besoin de savoir exactement comment la garantie va être satisfaite.

Happens-before raisonnement appliqué à quelques exemples

Nous présenterons quelques exemples pour montrer comment appliquer les *événements-avant de* raisonner pour vérifier que les écritures sont visibles lors des lectures suivantes.

Code mono-thread

Comme vous vous en doutez, les écritures sont toujours visibles lors des lectures suivantes dans un programme à thread unique.

```
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)
    }
}
```

Par Happens-Before Rule # 1:

1. L'action `write(a)` *se produit avant* l'action `write(b)` .
2. L'action `write(b)` *se produit avant* l'action `read(a)` .
3. L'action `read(a)` *se produit avant* l'action `read(a)` .

Par Happens-Before Rule # 4:

4. `write(a)` *arrive-avant* `write(b)` ET `write(b)` *arrive-avant de* `read(a)` IMPLIES `write(a)` *arrive-avant de* `read(a)` .
5. `write(b)` *arrive-avant* `read(a)` AND `read(a)` *arrive-avant* `read(b)` IMPLIES `write(b)` *arrive-avant* `read(b)` .

En résumé:

6. La relation `write(a)` *arrive-avant* `read(a)` signifie que l'instruction a `a + b` est garantie de voir la valeur correcte de `a` .
7. La relation `write(b)` *arrive avant* `read(b)` signifie que l'instruction a `a + b` est garantie de voir la valeur correcte de `b` .

Comportement de 'volatile' dans un exemple avec 2 threads

Nous allons utiliser l'exemple de code suivant pour explorer certaines implications du modèle de mémoire pour `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)
    }
}
```

Tout d'abord, considérez la séquence suivante d'instructions impliquant 2 threads:

1. Une seule instance de `VolatileExample` est créée. appelez ça `ve`
2. `ve.update(1, 2)` est appelé dans un thread, et
3. `ve.observe()` est appelé dans un autre thread.

Par Happens-Before Rule # 1:

1. L'action `write(a)` *se produit avant* l'action `volatile-write(a)` .
2. L'action `volatile-read(a)` *se produit avant* l'action `read(b)` .

Par Happens-Before Rule # 2:

3. L'action `volatile-write(a)` dans le premier thread *se produit avant* l'action `volatile-read(a)` dans le deuxième thread.

Par Happens-Before Rule # 4:

4. L'action `write(b)` dans le premier thread *arrive avant* l'action `read(b)` dans le second thread.

En d'autres termes, pour cette séquence particulière, il est garanti que le deuxième thread verra la mise à jour de la variable non volatile `b` créée par le premier thread. Cependant, il devrait également être clair que si les affectations dans la méthode de `update` étaient inverses ou si la méthode `observe()` lisait la variable `b` avant `a` , alors la chaîne « *passe avant*» serait rompue. La chaîne serait également brisée si `volatile-read(a)` dans le deuxième thread n'était pas postérieur à la `volatile-write(a)` dans le premier thread.

Lorsque la chaîne est rompue, il n'y a aucune *garantie* que d' `observe()` verra la valeur correcte de `b` .

Volatile à trois fils

Supposons que nous ajoutions un troisième thread dans l'exemple précédent:

1. Une seule instance de `VolatileExample` est créée. appelez ça `ve`
2. `update` appels à deux threads:
    - `ve.update(1, 2)` est appelé dans un thread,
    - `ve.update(3, 4)` est appelé dans le deuxième thread,

3. `ve.observe()` est ensuite appelé dans un troisième thread.

Pour analyser cela complètement, nous devons considérer tous les liens possibles entre les instructions du thread un et le thread deux. Au lieu de cela, nous n'en considérerons que deux.

Scénario n ° 1 - supposons que la `update(1, 2)` précède la `update(3,4)` nous obtenons cette séquence:

```
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
```

Dans ce cas, il est facile de voir qu'il y a une chaîne ininterrompue *avant-* `write(b, 3)` de `write(b, 3)` à `read(b)` . De plus, il n'y a pas d'écriture à `b` . Ainsi, pour ce scénario, le troisième thread est assuré de voir que `b` valeur `3` .

Scénario n ° 2 - supposons que la `update(1, 2)` et la `update(3,4)` chevauchent et que les ations soient entrelacées comme suit:

```
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
```

Maintenant, bien qu'il y ait une chaîne de *passe-avant* de `write(b, 3)` à `read(b)` , il y a une action `write(b, 1)` intermédiaire exécutée par l'autre thread. Cela signifie que nous ne pouvons pas être certains que la valeur `read(b)` sera visible.

(Mis à part: cela démontre que nous ne pouvons pas compter sur la `volatile` pour assurer la visibilité des variables non volatiles, sauf dans des situations très limitées.)

Comment éviter d'avoir à comprendre le modèle de mémoire

Le modèle de mémoire est difficile à comprendre et difficile à appliquer. C'est utile si vous avez besoin de raisonner sur l'exactitude du code multi-thread, mais vous ne voulez pas avoir à faire ce raisonnement pour chaque application multithread que vous écrivez.

Si vous adoptez les principes suivants lors de l' écriture du code concurrent en Java, vous pouvez éviter en *grande partie* la nécessité de recourir à un raisonnement *qui se passe-avant.*

- Utilisez des structures de données immuables lorsque cela est possible. Une classe immuable correctement implémentée sera compatible avec les threads et n'introduira pas de problèmes de sécurité des threads lorsque vous l'utiliserez avec d'autres classes.
- Comprendre et éviter les "publications dangereuses".
- Utilisez des mutex primitifs ou des objets `Lock` pour synchroniser l'accès à l'état des objets mutables qui doivent être thread-safe [1] .

- Utilisez `Executor` / `ExecutorService` ou le framework de jointure de fork plutôt que de tenter de créer des threads directement.
- Utilisez les classes `java.util.concurrent qui fournissent des verrous, des sémaphores, des verrous et des verrous avancés, au lieu d'utiliser directement wait / notify / notifyAll.
- Utilisez les versions `java.util.concurrent` de cartes, d'ensembles, de listes, de files d'attente et de deques plutôt que la synchronisation externe de collections non concurrentes.

Le principe général est d'essayer d'utiliser les bibliothèques de concurrence intégrées de Java plutôt que de "déployer votre propre" concurrence. Vous pouvez compter sur leur fonctionnement, si vous les utilisez correctement.

---

1 - Tous les objets ne doivent pas être thread-safe. Par exemple, si un objet ou des objets sont *confinés* dans un thread (c'est -*à*- dire qu'il n'est accessible qu'à un seul thread), sa sécurité de thread n'est pas pertinente.

Lire Modèle de mémoire Java en ligne: https://riptutorial.com/fr/java/topic/6829/modele-de-memoire-java

Chapitre 125: Modificateurs de non-accès

Introduction

Les modificateurs de non-accès ***ne modifient pas l'accessibilité des variables*** et des méthodes, mais leur fournissent ***des propriétés spéciales*** .

Examples

final

`final` en Java peut faire référence à des variables, des méthodes et des classes. Il y a trois règles simples:

- la variable finale ne peut pas être réaffectée
- la méthode finale ne peut pas être remplacée
- la classe finale ne peut pas être prolongée

Coutumes

Bonne pratique de programmation

Certains développeurs considèrent comme une bonne pratique de marquer une variable finale lorsque vous le pouvez. Si vous avez une variable qui ne devrait pas être changée, vous devriez la marquer finale.

Une utilisation importante du mot-clé `final` pour les paramètres de méthode. Si vous voulez souligner qu'une méthode ne modifie pas ses paramètres d'entrée, marquez les propriétés comme étant finales.

```
public int sumup(final List<Integer> ints);
```

Cela souligne que la méthode `sumup` ne va pas changer les `ints` .

Accès en classe interne

Si votre classe interne anonyme veut accéder à une variable, la variable doit être marquée comme `final`

```
  public IPrintName printName(){
    String name;
    return new IPrintName(){
        @Override
        public void printName(){
            System.out.println(name);
        }
    };
}
```

Cette classe ne compile pas, comme la variable `name` , n'est pas définitif.

Java SE 8

Les variables finales sont une exception. Ce sont des variables locales qui ne sont écrites qu'une seule fois et peuvent donc être rendues définitives. Les variables finales sont accessibles à partir de classes anonymes.

variable `final static`

Même si le code ci-dessous est totalement légal lorsque la variable `final foo` n'est pas `static` , en cas de `static` elle ne compilera pas:

```
class TestFinal {
    private final static List foo;

    public Test() {
        foo = new ArrayList();
    }
}
```

La raison en est, répétons-le encore, la *variable finale ne peut pas être réaffectée* . `foo` étant statique, il est partagé entre toutes les instances de la classe `TestFinal` . Lorsqu'une nouvelle instance d'une classe `TestFinal` est créée, son constructeur est appelé et, par conséquent, foo est réaffecté, ce que le compilateur ne permet pas. Une manière correcte d'initialiser la variable `foo` dans ce cas est soit:

```
class TestFinal {
    private static final List foo = new ArrayList();
    //..
}
```

ou en utilisant un initialiseur statique:

```
class TestFinal {
    private static final List foo;
    static {
        foo = new ArrayList();
    }
    //..
}
```

`final` méthodes `final` sont utiles lorsque la classe de base implémente des fonctionnalités importantes que la classe dérivée n'est pas censée modifier. Ils sont également plus rapides que les méthodes non finales, car aucun concept de table virtuelle n'est impliqué.

Toutes les classes de wrapper de Java sont finales, telles que `Integer` , `Long` etc. Les créateurs de ces classes ne voulaient pas que quiconque, par exemple, étend Integer dans sa propre classe et modifie le comportement de base de la classe Integer. L'une des exigences pour rendre une classe immuable est que les sous-classes ne peuvent pas remplacer les méthodes. La manière la plus simple de le faire est de déclarer la classe comme `final` .

volatil

Le modificateur `volatile` est utilisé dans la programmation multi-thread. Si vous déclarez un champ comme `volatile` cela signifie aux threads qu'ils doivent lire la valeur la plus récente, et non celle mise en cache localement. De plus, `volatile` lectures et écritures `volatile` sont garanties atomiques (l'accès à un `long` ou `double` non `volatile` n'est pas atomique), évitant ainsi certaines erreurs de lecture / écriture entre plusieurs threads.

```
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;
    }
}
```

statique

Le mot-clé `static` est utilisé sur une classe, une méthode ou un champ pour les faire fonctionner indépendamment de toute instance de la classe.

- Les champs statiques sont communs à toutes les instances d'une classe. Ils n'ont pas besoin d'instance pour y accéder.
- Les méthodes statiques peuvent être exécutées sans instance de la classe dans laquelle elles se trouvent. Cependant, elles ne peuvent accéder qu'aux champs statiques de cette classe.
- Les classes statiques peuvent être déclarées à l'intérieur d'autres classes. Ils n'ont pas besoin d'une instance de la classe dans laquelle ils sont instanciés.

```
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
```

abstrait

Abstraction est un processus consistant à masquer les détails de la mise en œuvre et à ne montrer que les fonctionnalités à l'utilisateur. Une classe abstraite ne peut jamais être instanciée. Si une classe est déclarée comme abstraite, le seul but est que la classe soit étendue.

```
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");
    }
}
```

synchronisé

Le modificateur synchronisé est utilisé pour contrôler l'accès d'une méthode particulière ou d'un bloc par plusieurs threads. Un seul thread peut entrer dans une méthode ou un bloc déclaré synchronisé. Le mot-clé synchronisé fonctionne sur le verrouillage intrinsèque d'un objet, dans le cas d'une méthode synchronisée, le verrouillage des objets en cours et la méthode statique utilisent un objet de classe. Tout thread essayant d'exécuter un bloc synchronisé doit d'abord acquérir le verrou d'objet.

```
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();
    }
}
```

transitoire

Une variable déclarée comme transitoire ne sera pas sérialisée lors de la sérialisation des objets.

```
public transient int limit = 55;   // will not persist
public int b; // will persist
```

strictfp

Java SE 1.2

Le modificateur strictfp est utilisé pour les calculs en virgule flottante. Ce modificateur rend les variables à virgule flottante plus cohérentes sur plusieurs plates-formes et garantit que tous les calculs à virgule flottante sont conformes aux normes IEEE 754 afin d'éviter les erreurs de calcul (erreurs d'arrondi), les dépassements et Cela ne peut pas être appliqué aux méthodes abstraites, aux variables ou aux constructeurs.

```
// strictfp keyword can be applied on methods, classes and interfaces.

strictfp class A{}

strictfp interface M{}

class A{
    strictfp void m(){}
}
```

Lire Modificateurs de non-accès en ligne: https://riptutorial.com/fr/java/topic/4401/modificateurs-

de-non-acces

Chapitre 126: Modules

Syntaxe

- nécessite java.xml;
- nécessite public java.xml; # expose le module aux dépendants pour utilisation
- exportations com.example.foo; # dépendants peuvent utiliser les types publics dans ce paquet
- exporte com.example.foo.impl vers com.example.bar; # restreindre l'utilisation à un module

Remarques

L'utilisation des modules est encouragée mais n'est pas obligatoire, cela permet au code existant de continuer à fonctionner en Java 9. Il permet également une transition progressive vers du code modulaire.

Tout code non modulaire est placé dans un *module sans nom* lorsqu'il est compilé. C'est un module spécial capable d'utiliser des types de tous les autres modules, mais **uniquement à partir de packages qui ont une déclaration d'** `exports` .

Tous les paquets du *module sans nom* sont exportés automatiquement.

Les mots-clés, par exemple, `module` etc ..., sont restreints dans la déclaration de module mais peuvent continuer à être utilisés comme identificateurs ailleurs.

Examples

Définir un module de base

Les modules sont définis dans un fichier nommé `module-info.java` , nommé descripteur de module. Il doit être placé dans la racine du code source:

```
|-- module-info.java
|-- com
    |-- example
        |-- foo
            |-- Foo.java
        |-- bar
            |-- Bar.java
```

Voici un descripteur de module simple:

```
module com.example {
    requires java.httpclient;
    exports com.example.foo;
}
```

Le nom du module doit être unique et il est recommandé d'utiliser la même notation de nommage Reverse-DNS que celle utilisée par les packages pour garantir cela.

Le module `java.base` , qui contient les classes de base de Java, est implicitement visible par tous les modules et n'a pas besoin d'être inclus.

La `requires` déclaration nous permet d'utiliser d' autres modules, dans l'exemple le module `java.httpclient` est importé.

Un module peut également spécifier les paquets qu'il `exports` et le rend donc visible aux autres modules.

Le package `com.example.foo` déclaré dans la clause `exports` sera visible par les autres modules. Les sous-packages de `com.example.foo` ne seront pas exportés, ils ont besoin de leurs propres déclarations d' `export` .

À l'inverse, `com.example.bar` qui n'est pas répertorié dans `exports` clauses `exports` ne sera pas visible par les autres modules.

Lire Modules en ligne: https://riptutorial.com/fr/java/topic/5286/modules

Chapitre 127: Monnaie et argent

Examples

Ajouter une devise personnalisée

JAR requis sur classpath:

- javax.money:money-api:1.0 (argent JSR354 et API de la monnaie)
- org.javamoney: moneta: 1.0 (implémentation de référence)
- javax: annotation-api: 1.2. (Annotations communes utilisées par l'implémentation de référence)

```
// 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
```

Lire Monnaie et argent en ligne: https://riptutorial.com/fr/java/topic/8359/monnaie-et-argent

Chapitre 128: Moteur JavaScript Nashorn

Introduction

Nashorn est un moteur JavaScript développé en Java par Oracle, et a été libéré avec Java 8. nashorn permet d' incorporer le Javascript dans les applications Java via JSR-223 et permet de développer des applications autonomes Javascript, et il offre une meilleure performance d' exécution et une meilleure conformité à la ECMA spécification Javascript normalisée.

Syntaxe

- ScriptEngineManager // Fournit un mécanisme de découverte et d'installation pour les classes ScriptEngine; utilise une interface SPI (Service Provider Interface)
- ScriptEngineManager.ScriptEngineManager () // Constructeur recommandé
- ScriptEngine // Fournit l'interface au langage de script
- ScriptEngine ScriptEngineManager.getEngineByName (String shortName) // Méthode d'usine pour l'implémentation donnée
- Object ScriptEngine.eval (Script de chaîne) // Exécute le script spécifié
- Object ScriptEngine.eval (Reader Reader) // Charge puis exécute un script à partir de la source spécifiée
- ScriptContext ScriptEngine.getContext () // Retourne les liaisons, les lecteurs et le fournisseur d'écriture par défaut
- void ScriptContext.setWriter (Writer writer) // Définit la destination pour envoyer la sortie du script à

Remarques

Nashorn est un moteur JavaScript écrit en Java et inclus dans Java 8. Tout ce dont vous avez besoin est inclus dans le package `javax.script` .

Notez que `ScriptEngineManager` fournit une API générique vous permettant d'obtenir des moteurs de script pour différents langages de script (c.-à-d. Pas seulement Nashorn, pas seulement JavaScript).

Examples

Définir des variables globales

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Define a global variable
engine.put("textToPrint", "Data defined in Java.");
```

```
// Print the global variable
try {
    engine.eval("print(textToPrint);");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
// 'Data defined in Java.' printed on standard output
```

Bonjour 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
```

Exécuter le fichier 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!');
```

Intercepter la sortie du 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!'
```

Évaluer les chaînes arithmétiques

```
// 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.
```

Utilisation d'objets Java en JavaScript dans Nashorn

Il est possible de passer des objets Java au moteur Nashorn pour qu'ils soient traités en code Java. Dans le même temps, il existe des constructions spécifiques à JavaScript (et à Nashorn), et leur fonctionnement avec les objets Java n'est pas toujours clair.

Vous trouverez ci-dessous un tableau qui décrit le comportement des objets Java natifs dans les constructions JavaScript.

Constructions testées:

1. Expression dans la clause if. Dans JS expression in, la clause if n'a pas besoin d'être booléenne contrairement à Java. Il est évalué comme faux pour ce que l'on appelle des valeurs de falsification (null, non défini, 0, chaînes vides, etc.)
2. Pour chaque instruction, Nashorn a un type particulier de boucle - pour chacun - qui peut parcourir différents objets JS et Java.
3. Obtenir la taille de l'objet Dans JS, les objets ont une longueur de propriété qui renvoie la taille d'un tableau ou d'une chaîne.

Résultats:

Type	Si	pour chaque	.longueur
Java null	faux	Pas d'itérations	**Exception**
Chaîne vide Java	faux	Pas d'itérations	0
Chaîne Java	vrai	Itère sur les caractères de chaîne	Longueur de la chaîne
Entier Java / Long	valeur! = 0	Pas d'itérations	indéfini
Java ArrayList	vrai	Itère sur des éléments	Longueur de la liste
Java HashMap	vrai	Itère sur les valeurs	nul
Java HashSet	vrai	Itère sur les articles	indéfini

Recommandations:

- Il est conseillé d'utiliser `if (some_string)` pour vérifier si une chaîne n'est pas nulle et non vide
- `for each` peut être utilisé en toute sécurité pour parcourir n'importe quelle collection, et ne déclenche pas d'exceptions si la collection n'est pas itérable, nulle ou indéfinie
- Avant d'obtenir la longueur d'un objet, il doit être vérifié pour null ou undefined (la même chose est vraie pour toute tentative d'appeler une méthode ou d'obtenir une propriété d'objet Java)

Implémentation d'une interface à partir d'un 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
    }
}
```

Définir et obtenir des variables globales

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

try {
    // Set value in the global name space of the engine
    engine.put("name","Nashorn");
    // Execute an hardcoded script
    engine.eval("var value='Hello '+name+'!';");
    // Get value
    String value=(String)engine.get("value");
    System.out.println(value);
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Hello Nashorn!' printed on standard output
```

Lire Moteur JavaScript Nashorn en ligne: https://riptutorial.com/fr/java/topic/166/moteur-javascript-nashorn

Chapitre 129: NIO - Mise en réseau

Remarques

`SelectionKey` définit les différentes opérations et informations sélectionnables entre son sélecteur et son canal . En particulier, la pièce jointe peut être utilisée pour stocker des informations liées à la connexion.

La gestion de `OP_READ` est assez simple. Cependant, vous devez faire attention lorsque vous traitez avec `OP_WRITE` : la plupart du temps, les données peuvent être écrites sur les sockets pour que l'événement continue à se déclencher. Assurez-vous d'enregistrer `OP_WRITE` uniquement avant de vouloir écrire des données (voir cette réponse).

En outre, `OP_CONNECT` doit être annulé une fois que le canal est connecté (car, bien, il *est* connecté. Voir ceci et cela répond sur SO). D' où l' `OP_CONNECT` enlèvement après `finishConnect()` a réussi.

Examples

Utilisation du sélecteur pour attendre les événements (exemple avec OP_CONNECT)

NIO est apparu dans Java 1.4 et a introduit le concept de "canaux", censés être plus rapides que les E / S classiques. Du point de vue du réseau, le `SelectableChannel` est le plus intéressant car il permet de surveiller différents états du canal. Il fonctionne de la même manière que l'appel système C `select()` : nous sommes réveillés lorsque certains types d'événements se produisent:

- connexion reçue (`OP_ACCEPT`)
- connexion réalisée (`OP_CONNECT`)
- données disponibles en lecture FIFO (`OP_READ`)
- les données peuvent être poussées pour écrire FIFO (`OP_WRITE`)

Il permet la séparation entre la *détection des* E / S de socket (quelque chose peut être lu / écrit / ...) et l' *exécution* des E / S (lecture / écriture / ...). En particulier, toutes les détections d'E / S peuvent être effectuées dans un seul thread pour plusieurs sockets (clients), tandis que l'exécution d'E / S peut être gérée dans un pool de threads ou ailleurs. Cela permet à une application de s'adapter facilement au nombre de clients connectés.

L'exemple suivant montre les bases:

1. Créer un `Selector`
2. Créer un `SocketChannel`
3. Enregistrez le `SocketChannel` sur le `Selector`
4. Boucle avec le `Selector` pour détecter les événements

```
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");
```

Donnerait la sortie suivante:

```
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
```

Lire NIO - Mise en réseau en ligne: https://riptutorial.com/fr/java/topic/5513/nio---mise-en-reseau

Chapitre 130: Nouveau fichier E / S

Syntaxe

- Paths.get (String first, String ... plus) // Crée une instance Path par ses éléments String
- Paths.get (URI URI) // Crée une instance Path par un URI

Examples

Créer des chemins

La classe `Path` est utilisée pour programmer un chemin dans le système de fichiers (et peut donc pointer vers des fichiers aussi bien que des répertoires, même vers des répertoires inexistants)

Un chemin peut être obtenu en utilisant la classe helper `Paths` :

```
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");
```

Récupération d'informations sur un chemin

Les informations sur un chemin peuvent être obtenues à l'aide des méthodes d'un objet `Path` :

- `toString()` renvoie la représentation sous forme de chaîne du chemin

  ```
  Path p1 = Paths.get("/var/www"); // p1.toString() returns "/var/www"
  ```

- `getFileName()` renvoie le nom du fichier (ou plus précisément le dernier élément du chemin

  ```
  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()` renvoie le nombre d'éléments formant le chemin

  ```
  Path p1 = Paths.get("/var/www"); // p1.getNameCount() returns 2
  ```

- `getName(int index)` renvoie l'élément à l'index donné

  ```
  Path p1 = Paths.get("/var/www"); // p1.getName(0) returns "var", p1.getName(1) returns
  "www"
  ```

- `getParent()` renvoie le chemin du répertoire parent

```
Path p1 = Paths.get("/var/www"); // p1.getParent().toString() returns "/var"
```

- `getRoot()` renvoie la racine du chemin

```
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:\\"
```

Manipulation des chemins

Rejoindre deux chemins

Les chemins peuvent être joints en utilisant la méthode `resolve()` . Le chemin d'accès doit être un chemin partiel, qui ne comprend pas l'élément racine.

```
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"
```

Normaliser un chemin

Les chemins peuvent contenir les éléments `.` (qui pointe vers le répertoire dans lequel vous vous trouvez actuellement) et `..` (qui pointe vers le répertoire parent).

Lorsqu'il est utilisé dans un chemin, `.` peut être supprimé à tout moment sans changer la destination du chemin, et `..` peut être supprimé avec l'élément précédent.

Avec l'API Paths, cela se fait en utilisant la méthode `.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"
```

Récupération d'informations à l'aide du système de fichiers

Pour interagir avec le système de fichiers, utilisez les méthodes de la classe `Files` .

Vérification de l'existence

Pour vérifier l'existence du fichier ou du répertoire indiqué par un chemin d'accès, utilisez les méthodes suivantes:

```
Files.exists(Path path)
```

et

```
Files.notExists(Path path)
```

`!Files.exists(path)` ne doit pas nécessairement être égal à `Files.notExists(path)` , car il existe trois scénarios possibles:

- Est vérifié l'existence est un fichier ou un répertoire (`exists` des rendements `true` et `notExists` retourne `false` dans ce cas)
- La non-existence d'un fichier ou d'un répertoire est vérifiée (`exists` renvoie `false` et `notExists` renvoie `true`)
- Ni l'existence ni la non-existence d'un fichier ou d'un répertoire ne peuvent être vérifiées (par exemple en raison de restrictions d'accès): Les deux `exists` et `nonExists` renvoie false.

Vérifier si un chemin pointe vers un fichier ou un répertoire

Ceci est fait en utilisant `Files.isDirectory(Path path)` et `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
```

Obtenir des propriétés

Cela peut être fait en utilisant les méthodes suivantes:

```
Files.isReadable(Path path)
Files.isWritable(Path path)
Files.isExecutable(Path path)

Files.isHidden(Path path)
Files.isSymbolicLink(Path path)
```

Obtenir le type MIME

```
Files.probeContentType(Path path)
```

Cela tente d'obtenir le type MIME d'un fichier. Il retourne un type MIME String, comme ceci:

- `text/plain` pour les fichiers texte
- `text/html` pour les pages HTML
- `application/pdf` pour les fichiers PDF
- `image/png` pour les fichiers PNG

Lecture de fichiers

Les fichiers peuvent être lus en octets et en lignes en utilisant 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()` prend éventuellement un jeu de caractères comme paramètre (`StandardCharsets.UTF_8` par défaut):

```
List<String> linesOfContent = Files.readAllLines(p2, StandardCharsets.ISO_8859_1);
```

Ecrire des fichiers

Les fichiers peuvent être écrits en mordant et en ligne en utilisant 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)
```

Les fichiers existants seront remplacés, des fichiers non existants seront créés.

Lire Nouveau fichier E / S en ligne: https://riptutorial.com/fr/java/topic/5519/nouveau-fichier-e---s

Chapitre 131: Objets immuables

Remarques

Les objets immuables ont un état fixe (aucun paramètre), donc tout état doit être connu au moment de la création de l'objet.

Bien que ce ne soit pas techniquement requis, il est recommandé de rendre tous les champs `final` . Cela rendra la classe immuable sûre (cf. Concurrence Java en pratique, 3.4.1).

Les exemples montrent plusieurs modèles qui peuvent aider à atteindre cet objectif.

Examples

Création d'une version immuable d'un type utilisant la copie défensive.

Certains types et classes de base en Java sont fondamentalement mutables. Par exemple, tous les types de tableau sont modifiables, tout comme les classes `java.util.Data` . Cela peut être gênant dans les situations où un type immuable est obligatoire.

Une façon de gérer cela est de créer un wrapper immuable pour le type mutable. Voici un wrapper simple pour un tableau d'entiers

```
public class ImmutableIntArray {
    private final int[] array;

    public ImmutableIntArray(int[] array) {
        this.array = array.clone();
    }

    public int[] getValue() {
        return this.clone();
    }
}
```

Cette classe fonctionne en utilisant la *copie défensive* pour isoler l'état mutable (l' `int[]`) de tout code qui pourrait le muter:

- Le constructeur utilise `clone()` pour créer une copie distincte du tableau de paramètres. Si l'appelant du constructeur suivant modifiait le tableau de paramètres, cela n'affecterait pas l'état de la `ImmutableIntArray` .
- La méthode `getValue()` utilise également `clone()` pour créer le tableau renvoyé. Si l'appelant changeait le tableau de résultats, cela n'affecterait pas l'état du tableau `ImmutableIntArray` .

Nous pourrions également ajouter des méthodes à `ImmutableIntArray` pour effectuer des opérations en lecture seule sur le tableau encapsulé; par exemple, obtenir sa longueur, obtenir la valeur à un index particulier, etc.

Notez qu'un type de wrapper immuable implémenté de cette manière n'est pas compatible avec le type d'origine. Vous ne pouvez pas simplement substituer le premier au second.

La recette d'une classe immuable

Un objet immuable est un objet dont l'état ne peut pas être modifié. Une classe immuable est une classe dont les instances sont immuables par conception et implémentation. La classe Java la plus communément présentée comme exemple d'immuabilité est java.lang.String .

Voici un exemple stéréotypé:

```
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;
    }
}
```

Une variante consiste à déclarer le constructeur comme `private` et à fournir une méthode de fabrique `public static` .

La *recette standard* pour une classe immuable est la suivante:

- Toutes les propriétés doivent être définies dans les constructeurs ou les méthodes d'usine.
- Il ne devrait y avoir aucun installateur.
- S'il est nécessaire d'inclure des paramètres pour des raisons de compatibilité d'interface, ils doivent soit ne rien faire ou lancer une exception.
- Toutes les propriétés doivent être déclarées comme `private` et `final` .
- Pour toutes les propriétés qui sont des références à des types mutables:
    - la propriété doit être initialisée avec une copie profonde de la valeur transmise via le constructeur, et
    - le getter de la propriété doit retourner une copie profonde de la valeur de la propriété.
- La classe doit être déclarée comme `final` pour empêcher quelqu'un de créer une sous-classe mutable d'une classe immuable.

Quelques autres choses à noter:

- Immutabilité n'empêche pas l'objet d'être nullable; Par exemple, `null` peut être affecté à une variable `String` .
- Si une propriété de classes immuables est déclarée comme `final` , les instances sont

intrinsèquement sûres. Cela fait des classes immuables un bon bloc de construction pour l'implémentation d'applications multithread.

Des défauts de conception typiques qui empêchent une classe d'être immuable

En utilisant certains paramètres, sans définir toutes les propriétés nécessaires dans le ou les constructeurs

```
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); }
}
```

Il est facile de montrer que la classe `Person` n'est pas immuable:

```
Person person = new Person("Joe");
person.setSurname("Average"); // NOT OK, change surname field after creation
```

Pour résoudre ce problème, supprimez simplement `setSurname()` et `setSurname()` le constructeur comme suit:

```
public Person(String name, String surname) {
    this.name = name;
    this.surname = surname;
  }
```

Ne pas marquer les variables d'instance comme privées et finales

Jetez un oeil à la classe suivante:

```
public final class Person {
    public String name;
    public Person(String name) {
        this.name = name;
     }
    public String getName() {
        return name;
    }

}
```

L'extrait suivant montre que la classe ci-dessus n'est pas immuable:

```
Person person = new Person("Average Joe");
```

```
person.name = "Magic Mike"; // not OK, new name for person after creation
```

Pour y remédier, marquez simplement nom propriété comme `private` et `final` .

Exposer un objet mutable de la classe dans un getter

Jetez un oeil à la classe suivante:

```
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 des `Names` semble immuable à première vue, mais ce n'est pas ce que montre le code suivant:

```
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"
```

Cela s'est produit car une modification apportée à la liste de références renvoyée par `getNames()` peut modifier la liste de `Names` .

Pour résoudre ce problème, évitez simplement de renvoyer des références qui font référence aux objets mutables de la classe, *soit* en créant des copies défensives, comme suit:

```
public List<String> getNames() {
   return new ArrayList<String>(this.names); // copies elements
}
```

ou en concevant des getters de manière à ne renvoyer que les autres *objets* et *primitives immuables* , comme suit:

```
public String getName(int index) {
    return names.get(index);
}
public int size() {
    return names.size();
}
```

Injecter un constructeur avec un ou des objets pouvant être modifiés en dehors de la classe immuable

Ceci est une variation du défaut précédent. Jetez un oeil à la classe suivante:

```
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();
    }
}
```

Comme la classe `Names` avant, la classe `NewNames` semble également immuable à première vue, mais ce n'est pas le cas: l'extrait suivant prouve le contraire:

```
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"
```

Pour corriger cela, comme dans la faille précédente, faites simplement des copies défensives de l'objet sans l'affecter directement à la classe immuable, c'est-à-dire que le constructeur peut être modifié comme suit:

```
    public Names(List<String> names) {
        this.names = new ArrayList<String>(names);
    }
```

Laisser les méthodes de la classe être surpassées

Jetez un oeil à la classe suivante:

```
public class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
      }
    public String getName() { return name;}
}
```

`Person` classe des `Person` semble immuable à première vue, mais supposons qu'une nouvelle sous-classe de `Person` soit définie:

```
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;
    }
}
```

maintenant, la mutabilité de `Person` (im) peut être exploitée par le polymorphisme en utilisant la nouvelle sous-classe:

```
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
```

Pour résoudre ce problème, *soit* marquer la classe comme `final` ne peut donc pas être étendu *ou* déclarer l' ensemble de son constructeur (s) comme `private` .

Lire Objets immuables en ligne: https://riptutorial.com/fr/java/topic/2807/objets-immuables

Chapitre 132: Objets sécurisés

Syntaxe

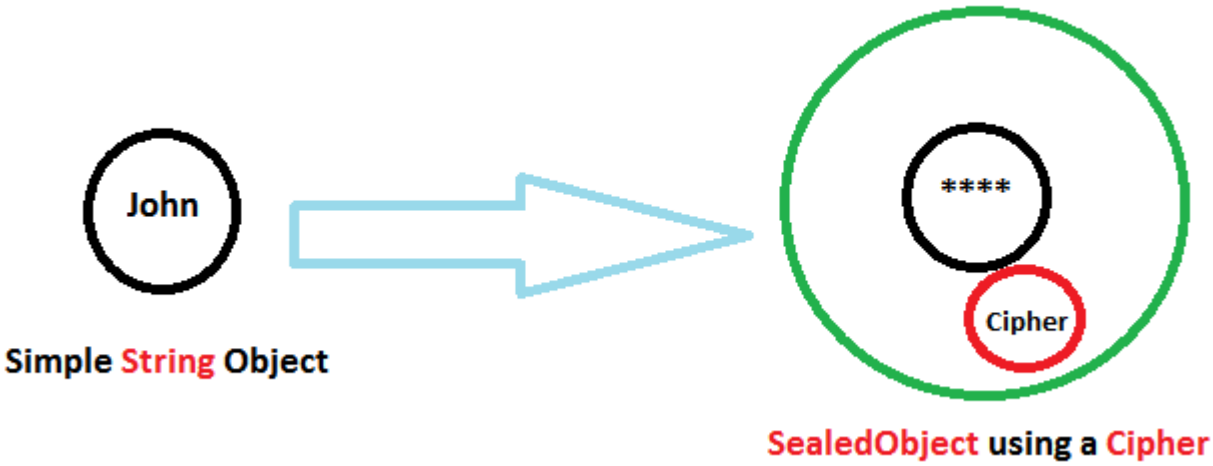
- SealedObject sealedObject = new SealedObject (obj, cipher);
- SignedObject signedObject = new SignedObject (obj, signedKey, signedEngine);

Examples

SealedObject (javax.crypto.SealedObject)

Cette classe permet à un programmeur de créer un objet et de protéger sa confidentialité avec un algorithme cryptographique.

Étant donné tout objet Serializable, on peut créer un objet **SealedObject** qui encapsule l'objet d'origine, au format sérialisé (c.-à-d. Une copie profonde), et scelle (crypte) son contenu sérialisé en utilisant un algorithme cryptographique tel que AES, DES. sa confidentialité. Le contenu chiffré peut ensuite être déchiffré (avec l'algorithme correspondant à l'aide de la clé de déchiffrement correcte) et désérialisé, ce qui permet d'obtenir l'objet d'origine.

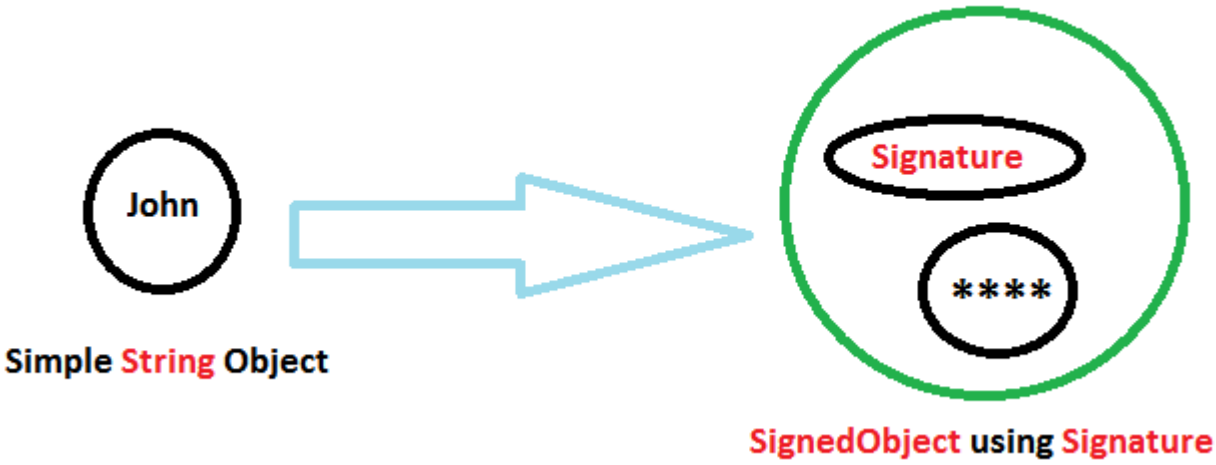



```
Serializable obj = new String("John");
// Generate key
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128);
SecretKey aesKey = kgen.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, aesKey);
SealedObject sealedObject = new SealedObject(obj, cipher);
System.out.println("sealedObject-" + sealedObject);
System.out.println("sealedObject Data-" + sealedObject.getObject(aesKey));
```

SignedObject (java.security.SignedObject)

SignedObject est une classe destinée à créer des objets d'exécution authentiques dont l'intégrité ne peut être compromise sans être détectée.

Plus précisément, un objet SignedObject contient un autre objet Serializable, l'objet (à signer) et sa signature.




```
//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());
```

Lire Objets sécurisés en ligne: https://riptutorial.com/fr/java/topic/5528/objets-securises

Chapitre 133: Opérations en virgule flottante Java

Introduction

Les nombres à virgule flottante sont des nombres comportant des parties fractionnaires (généralement exprimées par un point décimal). En Java, il existe deux types de primitives pour les nombres à virgule flottante: `float` (utilise 4 octets) et `double` (utilise 8 octets). Cette page de documentation permet de détailler avec des exemples des opérations pouvant être effectuées sur des points flottants en Java.

Examples

Comparer des valeurs en virgule flottante

Vous devez faire attention lorsque vous comparez des valeurs à virgule flottante (`float` ou `double`) en utilisant des opérateurs relationnels: `==` `!=` , `<` Et ainsi de suite. Ces opérateurs donnent des résultats en fonction des représentations binaires des valeurs à virgule flottante. Par exemple:

```
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"
    }
}
```

Le calcul `oneThird` a introduit une erreur d'arrondi minuscule, et lorsque nous multiplions un `oneThird` par `3` nous obtenons un résultat légèrement différent de `1.0` .

Ce problème des représentations inexactes est plus flagrant lorsque nous essayons de mélanger `double` et `float` dans les calculs. Par exemple:

```
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
    }
}
```

Les représentations en virgule flottante utilisées en Java pour les types `float` et `double` ont un nombre limité de chiffres de précision. Pour le type `float` , la précision est de 23 chiffres binaires ou d'environ 8 chiffres décimaux. Pour le type `double` , il s'agit de 52 bits ou d'environ 15 chiffres décimaux. De plus, certaines opérations arithmétiques introduiront des erreurs d'arrondi. Par conséquent, lorsqu'un programme compare des valeurs à virgule flottante, il est pratique courante de définir un **delta acceptable** pour la comparaison. Si la différence entre les deux nombres est inférieure au delta, ils sont considérés égaux. Par exemple

```
if (Math.abs(v1 - v2) < delta)
```

Delta comparer exemple:

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

Résultat:

```
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
```

Pour la comparaison des types primitifs `double` et `float` on peut également utiliser une méthode de `compare` statique du type boxe correspondant. Par exemple:

```
double a = 1.0;
double b = 1.0001;

System.out.println(Double.compare(a, b));//-1
System.out.println(Double.compare(b, a));//1
```

Enfin, il peut être difficile de déterminer quels sont les deltas les plus appropriés pour une comparaison. Une approche couramment utilisée consiste à choisir des valeurs delta qui, selon notre intuition, sont à peu près correctes. Cependant, si vous connaissez l'échelle et la précision des valeurs d'entrée, ainsi que les calculs effectués, il est possible de définir mathématiquement des limites solides sur la précision des résultats et, par conséquent, sur les deltas. (Il existe une branche formelle des mathématiques connue sous le nom d’analyse numérique qui était enseignée à des scientifiques spécialisés dans ce type d’analyse.)

OverFlow et UnderFlow

Type de données **flottant**

Le type de données float est un virgule flottante IEEE 754 32 bits simple précision.

`Float`

La valeur maximale possible est `3.4028235e+38` , quand elle dépasse cette valeur, elle produit `Infinity`

```
float f = 3.4e38f;
float result = f*2;
System.out.println(result); //Infinity
```

`Float` **UNDERFLOW**

La valeur minimale est 1.4e-45f, quand est en dessous de cette valeur, elle produit `0.0`

```
    float f = 1e-45f;
    float result = f/1000;
    System.out.println(result);
```

type de données **double**

Le type de données double est un virgule flottante IEEE 754 `64-bit` double précision.

`Double` **OverFlow**

La valeur maximale possible est `1.7976931348623157e+308` , lorsqu'elle dépasse cette valeur, elle produit `Infinity`

```
double d = 1e308;
double result=d*2;
System.out.println(result); //Infinity
```

`Double` **UnderFlow**

La valeur minimale est 4.9e-324, quand elle est inférieure à cette valeur, elle produit `0.0`

```
    double d = 4.8e-323;
    double result = d/1000;
    System.out.println(result); //0.0
```

Formatage des valeurs à virgule flottante

Les nombres à virgule flottante peuvent être formatés sous la forme d'un nombre décimal à l'aide de `String.format` avec l' `String.format 'f'`

```
    //Two digits in fracttional part are rounded
    String format1 = String.format("%.2f", 1.2399);
    System.out.println(format1); // "1.24"

    // three digits in fractional part are rounded
    String format2 = String.format("%.3f", 1.2399);
    System.out.println(format2); // "1.240"

    //rounded to two digits, filled with zero
    String format3 = String.format("%.2f", 1.2);
    System.out.println(format3); // returns "1.20"

    //rounder to two digits
    String format4 = String.format("%.2f", 3.19999);
    System.out.println(format4); // "3.20"
```

Les nombres à virgule flottante peuvent être formatés en nombre décimal à l'aide de `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
```

Adhésion stricte à la spécification IEEE

Par défaut, les opérations à virgule flottante sur `float` et `double` *ne* respectent *pas* strictement les règles de la spécification IEEE 754. Une expression est autorisée à utiliser des extensions spécifiques à l'implémentation pour la plage de ces valeurs; leur permettant essentiellement d'être *plus* précis que nécessaire.

`strictfp` désactive ce comportement. Il est appliqué à une classe, une interface ou une méthode et s'applique à tout ce qui y est contenu, comme les classes, interfaces, méthodes, constructeurs, initialiseurs de variables, etc. Avec `strictfp` , les valeurs intermédiaires d'une expression à virgule flottante *doivent* être comprises entre l'ensemble de valeurs flottantes ou l'ensemble de valeurs doubles. Les résultats de ces expressions sont donc exactement ceux prévus par la spécification IEEE 754.

Toutes les expressions constantes sont implicitement strictes, même si elles ne sont pas `strictfp` dans une portée `strictfp` .

Par conséquent, `strictfp` a pour effet net de parfois faire certains calculs coin cas *moins* précis, et peut également *plus lent* opérations en virgule flottante (comme la CPU est en train de faire plus de travail pour assurer une précision supplémentaire native ne modifie pas le résultat). Cependant, les résultats sont également identiques sur toutes les plates-formes. Il est donc utile dans des choses comme les programmes scientifiques, où la reproductibilité est plus importante que la vitesse.

```
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;
        }
    }
}
```

Lire Opérations en virgule flottante Java en ligne:
https://riptutorial.com/fr/java/topic/6167/operations-en-virgule-flottante-java

Chapitre 134: Optionnel

Introduction

`Optional` est un objet conteneur qui peut ou non contenir une valeur non nulle. Si une valeur est présente, `isPresent()` retournera `true` et `get()` retournera la valeur.

Des méthodes supplémentaires dépendant de la présence de la valeur contenue sont fournies, telles que `orElse()` , qui renvoie une valeur par défaut si value not present, et `ifPresent()` qui exécute un bloc de code si la valeur est présente.

Syntaxe

- Optional.empty () // Crée une instance facultative vide.
- Facultatif.of (valeur) // Renvoie une option avec la valeur non nulle spécifiée. Une exception NullPointerException sera lancée si la valeur transmise est nulle.
- Facultatif.ofNullable (value) // Renvoie une option avec la valeur spécifiée pouvant être nulle.

Examples

Renvoie la valeur par défaut si Facultatif est vide

Ne vous contentez pas d'utiliser `Optional.get()` car cela peut `NoSuchElementException` . Les méthodes `Optional.orElse(T)` et `Optional.orElseGet(Supplier<? extends T>)` permettent de fournir une valeur par défaut dans le cas où l'option Facultatif est vide.

```
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 différence cruciale entre `orElse` et `orElseGet` fait que ce dernier n'est évalué que lorsque le `orElseGet` Optional est vide, tandis que l'argument fourni au précédent est évalué même si le paramètre Facultatif n'est pas vide. L' `orElse` ne doit donc être utilisé que pour les constantes et jamais pour fournir une valeur basée sur un calcul quelconque.

Carte

Utilisez la méthode `map()` de `Optional` pour travailler avec des valeurs pouvant être `null` sans effectuer de vérifications `null` explicites:

(Notez que les opérations `map()` et `filter()` sont évaluées immédiatement, contrairement à leurs homologues Stream qui ne sont évalués que lors d'une *opération de terminal* .)

Syntaxe:

```
public <U> Optional<U> map(Function<? super T,? extends U> mapper)
```

Exemples de code:

```
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"
```

Etant donné que Optional.map () renvoie une valeur facultative vide lorsque sa fonction de mappage renvoie la valeur null, vous pouvez chaîner plusieurs opérations map () sous forme de déréférencement null-safe. Ceci est également connu sous le nom de **chaînage Null-safe** .

Prenons l'exemple suivant:

```
String value = foo.getBar().getBaz().toString();
```

Tout ce qui `getBar` , `getBaz` et `toString` peut potentiellement lancer une `NullPointerException` .

Voici une autre manière d'obtenir la valeur de `toString()` utilisant `Optional` :

```
String value = Optional.ofNullable(foo)
                       .map(Foo::getBar)
                       .map(Bar::getBaz)
                       .map(Baz::toString)
                       .orElse("");
```

Cela renverra une chaîne vide si l'une des fonctions de mappage retourne null.

Voici un autre exemple, mais légèrement différent. Il imprimera la valeur uniquement si aucune des fonctions de mappage n'a renvoyé la valeur null.

```
Optional.ofNullable(foo)
        .map(Foo::getBar)
        .map(Bar::getBaz)
```

```
        .map(Baz::toString)
        .ifPresent(System.out::println);
```

Lancer une exception, s'il n'y a pas de valeur

Utilisez la méthode `orElseThrow()` de `Optional` pour obtenir la valeur contenue ou lancer une exception, si elle n'a pas été définie. Ceci est similaire à l'appel de `get()` , sauf qu'il permet des types d'exceptions arbitraires. La méthode prend un fournisseur qui doit renvoyer l'exception à lancer.

Dans le premier exemple, la méthode renvoie simplement la valeur contenue:

```
Optional optional = Optional.of("something");

return optional.orElseThrow(IllegalArgumentException::new);
// returns "something" string
```

Dans le deuxième exemple, la méthode renvoie une exception car une valeur n'a pas été définie:

```
Optional optional = Optional.empty();

return optional.orElseThrow(IllegalArgumentException::new);
// throws IllegalArgumentException
```

Vous pouvez également utiliser la syntaxe lambda si une exception avec un message est nécessaire:

```
optional.orElseThrow(() -> new IllegalArgumentException("Illegal"));
```

Filtre

`filter()` est utilisé pour indiquer que vous souhaitez la valeur *uniquement si* elle correspond à votre prédicat.

Pensez-y comme `if (!somePredicate(x)) { x = null; }` .

Exemples de code:

```
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
```

Utilisation de conteneurs facultatifs pour les types de nombres primitifs

`OptionalDouble` , `OptionalInt` et `OptionalLong` fonctionnent comme `Optional` , mais sont spécifiquement conçus pour envelopper les types primitifs:

```
OptionalInt presentInt = OptionalInt.of(value);
OptionalInt absentInt = OptionalInt.empty();
```

Comme les types numériques ont une valeur, il n'y a pas de traitement spécial pour null. Les conteneurs vides peuvent être vérifiés avec:

```
presentInt.isPresent(); // Is true.
absentInt.isPresent(); // Is false.
```

De même, des raccourcis existent pour faciliter la gestion de la valeur:

```
// 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();
```

N'exécutez le code que s'il y a une valeur présente

```
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.
```

Fournissez une valeur par défaut en utilisant un fournisseur

La méthode `orElse` *normale* prend un `Object` , donc vous pourriez vous demander pourquoi il existe une option pour fournir un `Supplier` ici (la méthode `orElseGet`).

Considérer:

```
String value = "something";
return Optional.ofNullable(value)
               .orElse(getValueThatIsHardToCalculate()); // returns "something"
```

Il appellerait toujours `getValueThatIsHardToCalculate()` même si son résultat n'est pas utilisé car l'option n'est pas vide.

Pour éviter cette pénalité, vous fournissez un fournisseur:

```
String value = "something";
return Optional.ofNullable(value)
               .orElseGet(() -> getValueThatIsHardToCalculate()); // returns "something"
```

De cette manière, `getValueThatIsHardToCalculate()` sera uniquement appelé si le `getValueThatIsHardToCalculate() Optional` est vide.

FlatMap

`flatMap` est similaire à `map` . La différence est décrite par le javadoc comme suit:

> Cette méthode est similaire à `map(Function)` , mais le mappeur fourni est un mappeur dont le résultat est déjà `Optional` . `flatMap` est `flatMap` , `flatMap` ne l'enveloppe pas avec un `Optional` supplémentaire.

En d'autres termes, lorsque vous enchaînez un appel de méthode qui renvoie un `Optional` , l'utilisation de `Optional.flatMap` évite de créer des options `Optionals` imbriquées.

Par exemple, compte tenu des classes suivantes:

```
public class Foo {
    Optional<Bar> getBar(){
        return Optional.of(new Bar());
    }
}

public class Bar {
}
```

Si vous utilisez `Optional.map` , vous obtiendrez un `Optional` imbriqué; c'est-à-dire `Optional<Optional<Bar>>` .

```
Optional<Optional<Bar>> nestedOptionalBar =
    Optional.of(new Foo())
        .map(Foo::getBar);
```

Cependant, si vous utilisez `Optional.flatMap` , vous obtiendrez un simple `Optional` ; c'est-à-dire `Optional<Bar>` .

```
Optional<Bar> optionalBar =
    Optional.of(new Foo())
        .flatMap(Foo::getBar);
```

Lire Optionnel en ligne: https://riptutorial.com/fr/java/topic/152/optionnel

Chapitre 135: Paquets

Introduction

package in java est utilisé pour regrouper les classes et les interfaces. Cela aide le développeur à éviter les conflits lorsqu'il y a un grand nombre de classes. Si nous utilisons ce paquet les classes, nous pouvons créer une classe / interface avec le même nom dans différents packages. En utilisant des paquets, nous pouvons importer le morceau de nouveau dans une autre classe. Il y a beaucoup *de paquets intégrés* dans java comme> 1.java.util> 2.java.lang> 3.java.io Nous pouvons définir nos propres *paquets définis par l'utilisateur* .

Remarques

Les packages fournissent une protection d'accès.

> La déclaration de package doit être la première ligne du code source. Il ne peut y avoir qu'un seul paquet dans un fichier source.

Avec l'aide de paquets, les conflits entre les différents modules peuvent être évités.

Examples

Utilisation de packages pour créer des classes portant le même nom

Premier test.classe:

```
package foo.bar

public class Test {

}
```

Aussi Test.class dans un autre package

```
package foo.bar.baz

public class Test {

}
```

Ce qui précède est correct car les deux classes existent dans des packages différents.

Utiliser la portée protégée du paquet

En Java, si vous ne fournissez pas de modificateur d'accès, la portée par défaut des variables est le niveau protégé par package. Cela signifie que les classes peuvent accéder aux variables

d'autres classes dans le même package que si ces variables étaient publiquement disponibles.

```
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
}
```

Cette méthode ne fonctionnera pas pour une classe dans un autre package:

```
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
}
```

Lire Paquets en ligne: https://riptutorial.com/fr/java/topic/8273/paquets

Chapitre 136: Pièges de Java - Nulls et NullPointerException

Remarques

La valeur `null` est la valeur par défaut pour une valeur non initialisée d'un champ dont le type est un type de référence.

`NullPointerException` (ou NPE) est l'exception levée lorsque vous tentez d'effectuer une opération inappropriée sur la référence d'objet `null` . Ces opérations comprennent:

- appeler une méthode d'instance sur un objet cible `null` ,
- accéder à un champ d'un objet cible `null` ,
- tenter d'indexer un objet de tableau `null` ou d'accéder à sa longueur,
- utiliser une référence d'objet `null` comme mutex dans un bloc `synchronized` ,
- lancer une référence à un objet `null` ,
- désencapsuler une référence d'objet `null` et
- lancer une référence d'objet `null` .

Les causes profondes les plus courantes des NPE:

- oublier d'initialiser un champ avec un type de référence,
- oublier d'initialiser des éléments d'un tableau d'un type de référence, ou
- ne pas tester les résultats de certaines méthodes API *spécifiées* comme renvoyant `null` dans certaines circonstances.

Les exemples de méthodes couramment utilisées qui renvoient `null` incluent:

- La méthode `get(key)` de l'API `Map` renvoie une valeur `null` si vous l'appelez avec une clé sans mappage.
- Les `getResource(path)` et `getResourceAsStream(path)` dans les API `ClassLoader` et `Class` `ClassLoader` `null` si la ressource est introuvable.
- La méthode `get()` de l'API de `Reference` renvoie `null` si le ramasse-miettes a effacé la référence.
- Diverses méthodes `getXxxx` API de servlet Java EE renvoient `null` si vous tentez de récupérer un paramètre de requête, un attribut de session ou de session inexistant, etc.

Il existe des stratégies pour éviter les NPE indésirables, comme tester explicitement `null` ou utiliser "Yoda Notation", mais ces stratégies ont souvent pour résultat indésirable de *cacher des* problèmes dans votre code qui devraient vraiment être résolus.

Examples

Piège - L'utilisation inutile de Wrappers primitifs peut mener à des exceptions

NullPointerExceptions

Parfois, les programmeurs qui sont de nouveaux Java utilisent des types et des encapsuleurs primitifs de manière interchangeable. Cela peut entraîner des problèmes. Considérez cet exemple:

```
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
```

Notre classe `MyRecord` [1] repose sur une initialisation par défaut pour initialiser les valeurs sur ses champs. Ainsi, lorsque nous avons de `new` record, les `a` et `b` les champs seront mis à zéro, et les `c` et `d` les champs seront mis à `null` .

Lorsque nous essayons d'utiliser les champs initialisés par défaut, nous voyons que les champs `int` fonctionnent tout le temps, mais les champs `Integer` fonctionnent dans certains cas et pas dans d'autres. Spécifiquement, dans le cas où cela échoue (avec `d`), ce qui se passe est que l'expression du côté droit tente de décomprimer une référence `null` , et c'est ce qui provoque la levée de l'exception `NullPointerException` .

Il y a plusieurs façons de regarder ceci:

- Si les champs `c` et `d` doivent être des wrappers primitifs, alors nous ne devrions pas nous fier à l'initialisation par défaut, ou nous devrions tester `null` . Pour le premier est l'approche correcte, à *moins* qu'il y ait une signification précise pour les champs dans l'état `null` .
- Si les champs ne doivent pas nécessairement être des enveloppes primitives, il est erroné de les transformer en enveloppes primitives. En plus de ce problème, les wrappers primitifs ont des surcharges supplémentaires par rapport aux types primitifs.

La leçon ici est de ne pas utiliser les types d’encapsuleurs primitifs, sauf si vous en avez vraiment besoin.

---

1 - Ce cours n’est pas un exemple de bonne pratique de codage. Par exemple, une classe bien conçue n'aurait pas de champs publics. Cependant, ce n'est pas le but de cet exemple.

Pitfall - Utiliser null pour représenter un tableau ou une collection vide

Certains programmeurs pensent que c'est une bonne idée d'économiser de l'espace en utilisant un `null` pour représenter un tableau ou une collection vide. S'il est vrai que vous pouvez économiser une petite quantité d'espace, le revers de la médaille est que cela rend votre code plus compliqué et plus fragile. Comparez ces deux versions d'une méthode de sommation d'un

tableau:

La première version est la façon dont vous devez normalement coder la méthode:

```
/**
 * 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 deuxième version est la manière dont vous devez coder la méthode si vous avez l'habitude d'utiliser `null` pour représenter un tableau vide.

```
/**
 * 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;
}
```

Comme vous pouvez le constater, le code est un peu plus compliqué. Ceci est directement attribuable à la décision d'utiliser `null` de cette manière.

Maintenant, considérez si ce tableau qui pourrait être `null` est utilisé dans de nombreux endroits. À chaque endroit où vous l'utilisez, vous devez déterminer si vous devez tester `null` . Si vous manquez un test `null` devant être présent, vous risquez une `NullPointerException` . Ainsi, la stratégie d'utilisation de `null` de cette manière rend votre application plus fragile; c'est-à-dire plus vulnérable aux conséquences des erreurs de programmation.

La leçon ici est d'utiliser des tableaux vides et des listes vides lorsque c'est ce que vous voulez dire.

```
int[] values = new int[0];                    // always empty
List<Integer> list = new ArrayList();         // initially empty
List<Integer> list = Collections.emptyList(); // always empty
```

La surcharge d'espace est faible et il existe d'autres moyens de le minimiser si cela s'avère utile.

Piège - "Réussir" des nulls inattendus

Sur StackOverflow, nous voyons souvent du code comme celui-ci dans Réponses:

```
public String joinStrings(String a, String b) {
    if (a == null) {
        a = "";
    }
    if (b == null) {
        b = "";
    }
    return a + ": " + b;
}
```

Souvent, cela est accompagné d'une assertion qui est la "meilleure pratique" pour tester `null` comme ceci pour éviter `NullPointerException` .

Est-ce la meilleure pratique? En bref: non

Certaines hypothèses sous-jacentes doivent être remises en question avant de pouvoir dire si c'est une bonne idée de le faire dans nos `joinStrings` :

Qu'est-ce que cela signifie pour que "a" ou "b" soit nul?

Une valeur de `String` peut être zéro ou plusieurs caractères, nous avons donc déjà un moyen de représenter une chaîne vide. Est-ce que `null` signifie quelque chose de différent de `""` ? Si non, il est alors problématique d'avoir deux manières de représenter une chaîne vide.

Est-ce que le null provient d'une variable non initialisée?

Un `null` peut provenir d'un champ non initialisé ou d'un élément de tableau non initialisé. La valeur peut être non initialisée par conception ou par accident. Si c'était par accident, alors c'est un bug.

Le null représente-t-il un "ne sait pas" ou une "valeur manquante"?

Parfois, un `null` peut avoir un sens véritable; Par exemple, la valeur réelle d'une variable est inconnue ou indisponible ou "facultative". Dans Java 8, la classe `Optional` offre un meilleur moyen d’exprimer cela.

S'il s'agit d'un bogue (ou d'une erreur de conception), faut-il "réparer"?

Une interprétation du code est que nous "réparons" un `null` inattendu en utilisant une chaîne vide à sa place. La stratégie est-elle correcte? Serait-il préférable de laisser l'exception `NullPointerException` se produire, puis d'attraper l'exception plus haut dans la pile et de

l'enregistrer en tant que bogue?

Le problème avec "rendre bon" est que cela risque de cacher le problème ou de rendre le diagnostic plus difficile.

Est-ce efficace / bon pour la qualité du code?

Si l'approche "make good" est utilisée de manière cohérente, votre code contiendra beaucoup de tests null "défensifs". Cela va le rendre plus long et plus difficile à lire. En outre, tous ces tests et «correctifs» sont susceptibles d'avoir un impact sur les performances de votre application.

En résumé

Si `null` est une valeur significative, alors le test du cas `null` est la bonne approche. Le corollaire est que si une valeur `null` est significative, cela devrait être clairement documenté dans les javadocs de toutes les méthodes qui acceptent la valeur `null` ou la renvoient.

Sinon, il est préférable de traiter un `null` inattendu en tant qu'erreur de programmation et de laisser le `NullPointerException` se produire pour que le développeur sache qu'il y a un problème dans le code.

Pitfall - Renvoyer null au lieu de lancer une exception

Certains programmeurs Java ont une aversion générale pour lancer ou propager des exceptions. Cela conduit à un code comme celui-ci:

```
public Reader getReader(String pathname) {
    try {
        return new BufferedReader(FileReader(pathname));
    } catch (IOException ex) {
        System.out.println("Open failed: " + ex.getMessage());
        return null;
    }
```

}

Alors, quel est le problème avec ça?

Le problème est que `getReader` renvoie une valeur `null` tant que valeur spéciale pour indiquer que le `Reader` n'a pas pu être ouvert. Maintenant, la valeur renvoyée doit être testée pour voir si elle est `null` avant d'être utilisée. Si le test est omis, le résultat sera une `NullPointerException` .

Il y a en fait trois problèmes ici:

1. L' `IOException` été prise trop tôt.
2. La structure de ce code signifie qu'il existe un risque de fuite d'une ressource.
3. Un `null` été utilisé, puis renvoyé car aucun "vrai" `Reader` n'était disponible pour revenir.

En fait, en supposant que l'exception devait être détectée tôt comme cela, il y avait quelques

alternatives à la restitution de `null` :

1. Il serait possible d'implémenter une classe `NullReader` ; Par exemple, une opération où les opérations de l'API se comportent comme si le lecteur était déjà à la position "fin du fichier".
2. Avec Java 8, il serait possible de déclarer `getReader` comme renvoyant un `Optional<Reader>` .

Pitfall - Ne pas vérifier si un flux d'E / S n'est même pas initialisé lors de la fermeture

Pour éviter les fuites de mémoire, il ne faut pas oublier de fermer un flux d'entrée ou un flux de sortie dont le travail est effectué. Cela se fait généralement avec une instruction `try` - `catch` - `finally` sans la partie `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();
    }
}
```

Bien que le code ci-dessus puisse paraître innocent, il présente une faille qui peut rendre le débogage impossible. Si la ligne où `out` est initialisée (`out = new FileOutputStream(filename)`) génère une exception, alors `out` sera `null` lorsque `out.close()` sera exécuté, entraînant une méchante `NullPointerException` !

Pour éviter cela, assurez-vous simplement que le flux n'est pas `null` avant de le fermer.

```
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();
    }
}
```

Une approche encore meilleure consiste à `try` avec des ressources, car cela ferme automatiquement le flux avec une probabilité de 0 pour lancer un NPE sans avoir besoin d'un bloc `finally` .

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    try (FileOutputStream out = new FileOutputStream(filename)) {
        for(; count > 0; count--)
            out.write(0);
    }
}
```

Pitfall - Utiliser la notation "Yoda" pour éviter une exception NullPointerException

Un grand nombre d'exemples de code postés sur StackOverflow incluent des extraits comme ceci:

```
if ("A".equals(someString)) {
    // do something
}
```

Cela "empêche" ou "évite" une possible `NullPointerException` dans le cas où `someString` est `null` . En outre, il est discutable que

```
    "A".equals(someString)
```

est mieux que:

```
    someString != null && someString.equals("A")
```

(Il est plus concis et, dans certaines circonstances, il pourrait être plus efficace. Cependant, comme nous le soutenons plus loin, la concision pourrait être négative.)

Cependant, le véritable piège consiste à utiliser le test Yoda **pour éviter les `NullPointerExceptions`** .

Lorsque vous écrivez `"A".equals(someString)` vous `"A".equals(someString)` " le cas où `someString` est `null` . Mais comme un autre exemple (Pitfall - "Faire de bonnes" nulls inattendus) explique, "faire" `null` valeurs `null` peut être nocif pour diverses raisons.

Cela signifie que les conditions de Yoda ne sont pas les "meilleures pratiques" [1] . À moins que la valeur `null` soit attendue, il est préférable de laisser l' `NullPointerException` se produire pour obtenir un échec de test d'unité (ou un rapport de bogue). Cela vous permet de trouver et de corriger le bogue qui a provoqué l'apparition de la `null` inattendue / indésirable.

Les conditions Yoda ne doivent être utilisées que dans les cas où la valeur `null` est *attendue* car l'objet que vous testez provient d'une API *documentée* comme renvoyant une valeur `null` . Et sans doute, il serait préférable d'utiliser l'une des méthodes les moins jolies pour exprimer le test, car cela aide à mettre en évidence le test `null` pour quelqu'un qui examine votre code.

---

1 - Selon Wikipedia : *"Les meilleures pratiques de codage sont un ensemble de règles informelles que la communauté du développement de logiciels a appris au fil du temps, ce qui peut aider à améliorer la qualité des logiciels."* . Utiliser la notation Yoda ne permet pas d'atteindre cet objectif. Dans beaucoup de situations, cela aggrave le code.

Lire Pièges de Java - Nulls et NullPointerException en ligne:
https://riptutorial.com/fr/java/topic/5680/pieges-de-java---nulls-et-nullpointerexception

Chapitre 137: Pièges Java - Problèmes de performances

Introduction

Cette rubrique décrit un certain nombre de "pièges" (c.-à-d. Les erreurs commises par les programmeurs novices java) liés aux performances des applications Java.

Remarques

Cette rubrique décrit quelques "micro" pratiques de codage Java inefficaces. Dans la plupart des cas, les inefficacités sont relativement faibles, mais il est toujours préférable de les éviter.

Examples

Pitfall - Les frais généraux de création de messages de journal

`TRACE` niveaux de journalisation `TRACE` et `DEBUG` sont là pour transmettre des informations détaillées sur le fonctionnement du code donné lors de l'exécution. La définition du niveau de journalisation au-dessus de ces paramètres est généralement recommandée. Toutefois, vous devez veiller à ce que ces instructions n'affectent pas les performances, même si elles sont apparemment désactivées.

Considérez cette déclaration de journal:

```
// Processing a request of some kind, logging the parameters
LOG.debug("Request coming from " + myInetAddress.toString()
          + " parameters: " + Arrays.toString(veryLongParamArray));
```

Même lorsque le niveau de journalisation est défini sur `INFO` , les arguments transmis à `debug()` seront évalués à chaque exécution de la ligne. Cela le rend inutile inutilement à plusieurs égards:

- Concaténation de `String` : plusieurs instances de `String` seront créées
- `InetAddress` peut même faire une recherche DNS.
- `veryLongParamArray` peut être très long - créer un String à partir de celui-ci consomme de la mémoire, prend du temps

Solution

La plupart des structures de journalisation permettent de créer des messages de journalisation à l'aide de chaînes de correctifs et de références d'objets. Le message de journal sera évalué uniquement si le message est réellement consigné. Exemple:

```
// No toString() evaluation, no string concatenation if debug is disabled
LOG.debug("Request coming from {} parameters: {}", myInetAddress, parameters));
```

Cela fonctionne très bien tant que tous les paramètres peuvent être convertis en chaînes en utilisant String.valueOf (Object) . Si le calcul du message de journal est plus complexe, le niveau de journalisation peut être vérifié avant la journalisation:

```
if (LOG.isDebugEnabled()) {
    // Argument expression evaluated only when DEBUG is enabled
    LOG.debug("Request coming from {}, parameters: {}", myInetAddress,
              Arrays.toString(veryLongParamArray);
}
```

Ici, `LOG.debug()` avec le `Arrays.toString(Obect[])` coûteux `Arrays.toString(Obect[])` est traité uniquement lorsque `DEBUG` est effectivement activé.

Pitfall - La concaténation de chaînes dans une boucle ne s'adapte pas

Considérez le code suivant comme illustration:

```
public String joinWords(List<String> words) {
    String message = "";
    for (String word : words) {
        message = message + " " + word;
    }
    return message;
}
```

Malheureusement, ce code est inefficace si la liste de `words` est longue. La racine du problème est cette déclaration:

```
message = message + " " + word;
```

Pour chaque itération de boucle, cette instruction crée une nouvelle chaîne de `message` contenant une copie de tous les caractères de la chaîne de `message` origine avec des caractères supplémentaires. Cela génère beaucoup de chaînes temporaires et fait beaucoup de copie.

Lorsque nous analysons `joinWords` , en supposant qu'il y ait N mots avec une longueur moyenne de M, nous trouvons que O (N) chaînes temporaires sont créées et que O (MN^2) caractères seront copiés dans le processus. La composante N^2 est particulièrement préoccupante.

L'approche recommandée pour ce type de problème [1] consiste à utiliser un `StringBuilder` au lieu de la concaténation de chaîne comme suit:

```
public String joinWords2(List<String> words) {
    StringBuilder message = new StringBuilder();
    for (String word : words) {
        message.append(" ").append(word);
    }
    return message.toString();
}
```

L'analyse de `joinWords2` doit prendre en compte les frais généraux `joinWords2` à la "croissance" du tableau de commandes `StringBuilder` qui contient les caractères du générateur. Cependant, il s'avère que le nombre de nouveaux objets créés est O (logN) et que le nombre de caractères copiés est O (MN). Ce dernier inclut les caractères copiés dans l'appel `toString()` final.

(Il est peut-être possible d'optimiser ce réglage en créant le `StringBuilder` avec la capacité correcte pour commencer. Cependant, la complexité globale reste la même.)

En revenant à la méthode `joinWords` origine, il s'avère que la déclaration critique sera optimisée par un compilateur Java typique en quelque chose comme ceci:

```
StringBuilder tmp = new StringBuilder();
tmp.append(message).append(" ").append(word);
message = tmp.toString();
```

Cependant, le compilateur Java ne "retirera" pas le `StringBuilder` de la boucle, comme nous l'avons fait à la main dans le code de `joinWords2` .

Référence:

- "Est-ce que l'opérateur String de Java '+' est en boucle?"

---

1 - Dans Java 8 et `Joiner` ultérieures, la classe `Joiner` peut être utilisée pour résoudre ce problème particulier. Cependant, ce n'est pas ce que cet exemple est *vraiment censé être* .

Pitfall - Utiliser 'new' pour créer des instances d'emballages primitives est inefficace

Le langage Java vous permet d'utiliser `new` pour créer des instances `Integer` , `Boolean` , etc., mais c'est généralement une mauvaise idée. Il est préférable d'utiliser la méthode d'autoboxing (Java 5 et versions ultérieures) ou la méthode `valueOf` .

```
Integer i1 = new Integer(1);      // BAD
Integer i2 = 2;                   // BEST (autoboxing)
Integer i3 = Integer.valueOf(3);  // OK
```

La raison pour laquelle l'utilisation de `new Integer(int)` explicitement est une mauvaise idée est qu'il crée un nouvel objet (à moins d'être optimisé par le compilateur JIT). En revanche, lors de l'utilisation de la mise en file d'attente automatique ou d'un appel `valueOf` explicite, le runtime Java tente de réutiliser un objet `Integer` partir d'un cache d'objets préexistants. Chaque fois que le runtime a un cache "hit", cela évite de créer un objet. Cela permet également d'économiser de la mémoire de tas et de réduire les frais généraux du GC causés par le roulement de l'objet.

Remarques:

1. Dans les implémentations Java récentes, l'autoboxing est implémenté en appelant `valueOf` , et il existe des caches pour `Boolean` , `Byte` , `Short` , `Integer` , `Long` et `Character` .
2. Le comportement de mise en cache pour les types intégraux est requis par la spécification

de langage Java.

Piège - Appeler 'new String (String)' est inefficace

Utiliser `new String(String)` pour dupliquer une chaîne est inefficace et presque toujours inutile.

- Les objets String sont immuables, il n'est donc pas nécessaire de les copier pour les protéger contre les modifications.
- Dans certaines versions antérieures de Java, les objets `String` peuvent partager des tableaux de sauvegarde avec d'autres objets `String` . Dans ces versions, il est possible de fuir de la mémoire en créant une (petite) sous-chaîne d'une (grande) chaîne et en la conservant. Cependant, à partir de Java 7, les matrices de sauvegarde de `String` ne sont pas partagées.

En l'absence d'avantage tangible, l'appel de `new String(String)` est tout simplement inutile:

- Faire la copie prend du temps CPU.
- La copie utilise plus de mémoire, ce qui augmente l'encombrement de l'application et / ou augmente les frais généraux du GC.
- Les opérations telles que `equals(Object)` et `hashCode()` peuvent être plus lentes si des objets String sont copiés.

Piège - L'appel de System.gc () est inefficace

C'est (presque toujours) une mauvaise idée d'appeler `System.gc()` .

Le javadoc pour la méthode `gc()` spécifie ce qui suit:

> "L'appel de la méthode `gc` suggère que la machine virtuelle Java déploie des efforts pour recycler les objets inutilisés afin de pouvoir réutiliser rapidement la mémoire qu'elle occupe actuellement. Lorsque le contrôle revient de l'appel de méthode, la machine virtuelle Java s'efforce de récupérer espace de tous les objets jetés. "

On peut en tirer quelques points importants:

1. L'utilisation du mot "suggère" plutôt que (dis) "raconte" signifie que la JVM est libre d'ignorer la suggestion. Le comportement par défaut de la JVM (versions récentes) doit suivre la suggestion, mais cela peut être remplacé par la définition de `-XX:+DisableExplicitGC` lors du lancement de la JVM.

2. La phrase "un meilleur effort pour récupérer de l'espace à partir de tous les objets ignorés" implique que l'appel de `gc` déclenchera une récupération de `gc` "complète".

Alors pourquoi appeler `System.gc()` une mauvaise idée?

Tout d'abord, exécuter une collecte de place complète est coûteux. Un GC complet implique la visite et le "marquage" de chaque objet encore accessible; c'est-à-dire chaque objet qui n'est pas une poubelle. Si vous déclenchez cela alors qu'il n'y a pas beaucoup de déchets à collecter, le GC fait beaucoup de travail pour relativement peu d'avantages.

Deuxièmement, une récupération de place complète risque de perturber les propriétés de "localité" des objets non collectés. Les objets alloués par le même thread à peu près au même moment ont tendance à être alloués de manière rapprochée en mémoire. C'est bon. Les objets alloués en même temps sont susceptibles d'être liés; c'est-à-dire se référencer Si votre application utilise ces références, il est probable que l'accès à la mémoire sera plus rapide en raison des divers effets de mémoire et de mise en cache des pages. Malheureusement, une récupération de mémoire complète a tendance à déplacer des objets, de sorte que les objets qui étaient autrefois fermés sont désormais plus éloignés.

Troisièmement, l'exécution d'une récupération de place complète risque de mettre votre application en pause jusqu'à ce que la collecte soit terminée. Pendant que cela se produit, votre demande ne sera pas recevable.

En fait, la meilleure stratégie consiste à laisser la JVM décider du moment où exécuter le GC et du type de collection à exécuter. Si vous n'intervenez pas, la machine virtuelle Java choisira un type de temps et de collection qui optimise le débit ou minimise les temps de pause du GC.

Au début, nous avons dit "(presque toujours) une mauvaise idée ...". En fait, il existe quelques scénarios dans lesquels cela *pourrait* être une bonne idée:

1. Si vous implémentez un test unitaire pour un code sensible au ramassage des ordures (par exemple, quelque chose impliquant des finaliseurs ou des références faibles / douces / fantômes), il peut être nécessaire d'appeler `System.gc()` .

2. Dans certaines applications interactives, il peut y avoir des moments particuliers où l'utilisateur ne se soucie pas de savoir s'il y a une pause de récupération de place. Un exemple est un jeu où il y a des pauses naturelles dans le "jeu"; par exemple lors du chargement d'un nouveau niveau.

Piège - La surutilisation des types d'emballages primitifs est inefficace

Considérez ces deux morceaux de code:

```
int a = 1000;
int b = a + 1;
```

et

```
Integer a = 1000;
Integer b = a + 1;
```

Question: Quelle version est la plus efficace?

Réponse: Les deux versions sont presque identiques, mais la première version est beaucoup plus efficace que la deuxième.

La deuxième version utilise une représentation des nombres qui utilise plus d'espace, et s'appuie sur la mise en boîte automatique et le désencapsulation automatique en arrière-plan. En fait, la

deuxième version est directement équivalente au code suivant:

```
Integer a = Integer.valueOf(1000);               // box 1000
Integer b = Integer.valueOf(a.intValue() + 1);   // unbox 1000, add 1, box 1001
```

En comparant ceci à l'autre version qui utilise `int` , il y a clairement trois appels de méthode supplémentaires quand `Integer` est utilisé. Dans le cas de `valueOf` , les appels vont chacun créer et initialiser un nouvel objet `Integer` . Tout ce travail supplémentaire de boxe et de déballage va probablement rendre la deuxième version plus lente que la première.

En plus de cela, la deuxième version alloue des objets sur le tas dans chaque appel `valueOf` . Bien que l'utilisation de l'espace soit spécifique à la plate-forme, il est probable qu'il soit de l'ordre de 16 octets pour chaque objet `Integer` . En revanche, la version `int` nécessite un espace de pile supplémentaire, en supposant que `a` et `b` sont des variables locales.

Une autre grande raison pour laquelle les primitives sont plus rapides que leur équivalent en boîte est la manière dont leurs types de tableau respectifs sont disposés en mémoire.

Si vous prenez `int[]` et `Integer[]` comme exemple, dans le cas d'un `int[]` les *valeurs* `int` sont contiguës en mémoire. Mais dans le cas d'un `Integer[]` ce ne sont pas les valeurs qui sont mises en page, mais les références (pointeurs) aux objets `Integer` , qui contiennent à leur tour les valeurs `int` réelles.

En plus d'être un niveau supplémentaire d'indirection, il peut s'agir d'un gros réservoir lorsqu'il s'agit de mettre en cache une localité lors d'une itération sur les valeurs. Dans le cas d'un `int[]` le processeur peut récupérer toutes les valeurs du tableau, dans son cache, car elles sont contiguës en mémoire. Mais dans le cas d'un `Integer[]` le processeur doit éventuellement effectuer une extraction de mémoire supplémentaire pour chaque élément, car le tableau contient uniquement des références aux valeurs réelles.

En bref, l'utilisation de types d'encapsuleurs primitifs est relativement coûteuse à la fois en termes de ressources processeur et mémoire. Les utiliser inutilement est efficace.

Piège - Itérer les clés d'une carte peut être inefficace

L'exemple de code suivant est plus lent que nécessaire:

```
Map<String, String> map = new HashMap<>();
for (String key : map.keySet()) {
    String value = map.get(key);
    // Do something with key and value
}
```

En effet, il nécessite une recherche de carte (la méthode `get()`) pour chaque clé de la carte. Cette recherche peut ne pas être efficace (dans un HashMap, cela implique d'appeler `hashCode` sur la clé, puis de rechercher le bon compartiment dans les structures de données internes, et parfois même d'appeler des `equals`). Sur une grande carte, ceci peut ne pas être une surcharge triviale.

La manière correcte d'éviter ceci est d'itérer sur les entrées de la carte, ce qui est détaillé dans la rubrique Collections.

Piège - L'utilisation de size () pour tester si une collection est vide est inefficace.

Java Collections Framework fournit deux méthodes connexes pour tous les objets `Collection` :

- `size()` renvoie le nombre d'entrées dans une `Collection` et
- `isEmpty()` méthode `isEmpty()` renvoie true si (et seulement si) la `Collection` est vide.

Les deux méthodes peuvent être utilisées pour tester la vacuité de la collecte. Par exemple:

```
Collection<String> strings = new ArrayList<>();
boolean isEmpty_wrong = strings.size() == 0; // Avoid this
boolean isEmpty = strings.isEmpty();         // Best
```

Bien que ces approches se ressemblent, certaines implémentations de collections ne stockent pas la taille. Pour une telle collection, l'implémentation de `size()` doit calculer la taille à chaque appel. Par exemple:

- Une simple classe de liste liée (mais pas `java.util.LinkedList`) peut avoir besoin de parcourir la liste pour compter les éléments.
- La classe `ConcurrentHashMap` doit additionner les entrées de tous les "segments" de la carte.
- Une implémentation paresseuse d'une collection peut nécessiter de réaliser toute la collection en mémoire afin de compter les éléments.

En revanche, une méthode `isEmpty()` doit uniquement tester s'il existe *au moins un* élément dans la collection. Cela ne nécessite pas de compter les éléments.

Alors que `size() == 0` n'est pas toujours moins efficace que `isEmpty()` , il est inconcevable qu'un `isEmpty()` correctement implémenté soit moins efficace que `size() == 0` . Par conséquent, `isEmpty()` est préféré.

Piège - Problèmes d’efficacité avec les expressions régulières

La correspondance d'expressions régulières est un outil puissant (en Java et dans d'autres contextes), mais elle présente certains inconvénients. Une de ces expressions que les expressions régulières ont tendance à être assez chère.

Les instances Pattern et Matcher doivent être réutilisées

Prenons l'exemple suivant:

```
/**
 * 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;
}
```

Ce code est correct, mais il est inefficace. Le problème réside dans l'appel de `matches(...)` . Sous le capot, `s.matches("[A-Za-z0-9]*")` est équivalent à ceci:

```
Pattern.matches(s, "[A-Za-z0-9]*")
```

ce qui équivaut à son tour à

```
Pattern.compile("[A-Za-z0-9]*").matcher(s).matches()
```

L' `Pattern.compile("[A-Za-z0-9]*")` analyse l'expression régulière, l'analyse et construit un objet `Pattern` contenant la structure de données qui sera utilisée par le moteur regex. C'est un calcul non trivial. Ensuite, un objet `Matcher` est créé pour envelopper l'argument `s` . Enfin, nous appelons `match()` pour faire la correspondance de motif réelle.

Le problème est que ce travail est répété pour chaque itération de boucle. La solution consiste à restructurer le code comme suit:

```
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;
}
```

Notez que le javadoc pour `Pattern` indique:

> Les instances de cette classe sont immuables et peuvent être utilisées par plusieurs threads simultanés. Les instances de la classe `Matcher` ne sont pas sûres pour une telle utilisation.

N'utilisez pas match () quand vous devriez utiliser find ()

Supposons que vous voulez tester si une chaîne `s` contient trois chiffres ou plus dans une rangée. Vous pouvez l'exprimer de différentes manières, notamment:

```
 if (s.matches(".*[0-9]{3}.*")) {
     System.out.println("matches");
 }
```

ou

```
 if (Pattern.compile("[0-9]{3}").matcher(s).find()) {
     System.out.println("matches");
 }
```

Le premier est plus concis, mais il est également susceptible d'être moins efficace. À première vue, la première version va essayer de faire correspondre la chaîne entière au motif. De plus, puisque ". *" Est un modèle "gourmand", le gestionnaire de motifs est susceptible de faire avancer "avidement" la fin de la chaîne et de revenir en arrière jusqu'à ce qu'il trouve une correspondance.

En revanche, la deuxième version recherchera de gauche à droite et cessera de chercher dès qu'elle trouvera les 3 chiffres à la suite.

Utiliser des alternatives plus efficaces aux expressions régulières

Les expressions régulières sont un outil puissant, mais elles ne doivent pas être votre seul outil. Beaucoup de tâches peuvent être effectuées de manière plus efficace par d'autres moyens. Par exemple:

```
Pattern.compile("ABC").matcher(s).find()
```

fait la même chose que:

```
s.contains("ABC")
```

sauf que ce dernier est beaucoup plus efficace. (Même si vous pouvez amortir le coût de compilation de l'expression régulière)

Souvent, la forme non-regex est plus compliquée. Par exemple, le test effectué par l'appel `matches()` la méthode `allAlplanumeric` antérieure peut être réécrit comme `allAlplanumeric` :

```
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;
}
```

Maintenant, c'est plus de code que d'utiliser un `Matcher` , mais cela va être beaucoup plus rapide.

Retournement catastrophique

(Ceci est potentiellement un problème avec toutes les implémentations d'expressions régulières, mais nous allons le mentionner ici car c'est un piège pour l'utilisation de `Pattern` .)

Considérons cet exemple (artificiel):

```
Pattern pat = Pattern.compile("(A+)+B");
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAB").matches());
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAC").matches());
```

Le premier `println` appel va rapidement imprimer `true` . Le second imprimera `false` . Finalement. En effet, si vous testez le code ci-dessus, vous verrez que chaque fois que vous ajoutez un `A` avant le `C` , le temps nécessaire doublera.

Ce comportement est un exemple de *retour en arrière catastrophique* . Le moteur correspondant de modèle qui met en oeuvre la mise en correspondance regex tente infructueusement toutes les façons *possibles* que le modèle *pourrait* correspondre.

Regardons ce que `(A+)+B` signifie réellement. Superficiellement, il semble dire "un ou plusieurs caractères `A` suivis d'une valeur `B` ", mais en réalité, il s'agit d'un ou de plusieurs groupes, chacun composé d'un ou plusieurs caractères `A` Donc, par exemple:

- "AB" correspond à un sens seulement: "(A) B"
- "AAB" correspond à deux manières: "(AA) B" ou "(A) (A) B"
- "AAAB" correspond à quatre méthodes: "(AAA) B" ou "(AA) (A) B `or '(A)(AA)B` ou "(A) (A) (A) B"
- etc

En d'autres termes, le nombre de correspondances possibles est 2 ^{N} où N est le nombre de caractères `A`

L'exemple ci-dessus est clairement inventé, mais les modèles qui présentent ce type de caractéristiques de performance (c'est-à-dire `O(2^N)` ou `O(N^K)` pour un grand `K`) apparaissent fréquemment lorsque des expressions régulières mal considérées sont utilisées. Il existe des remèdes standard:

- Évitez d'imbriquer des motifs répétés dans d'autres motifs répétés.
- Évitez d'utiliser trop de motifs répétés.
- Utilisez la répétition sans retour en arrière si nécessaire.
- N'utilisez pas les expressions rationnelles pour les tâches d'analyse complexes. (Écrivez un analyseur approprié à la place.)

Enfin, méfiez-vous des situations où un utilisateur ou un client API peut fournir une chaîne d'expression régulière présentant des caractéristiques pathologiques. Cela peut entraîner un "déni de service" accidentel ou délibéré.

Les références:

- Le tag Expressions régulières , en particulier http://www.riptutorial.com/regex/topic/259/getting-started-with-regular-expressions/977/backtracking#t=20161001033913136116 3 et http://www.riptutorial.com/regex / topic / 259 / getting-started-with-regular-expressions / 4527 / lorsque-you-should-not-use-regular-expressions # t = 20161001033959356491 3
- "Regex Performance" par Jeff Atwood.
- "Comment tuer Java avec une expression régulière" par Andreas Haufler.

Pitfall - Interner des chaînes pour que vous puissiez utiliser == est une mauvaise idée

Quand certains programmeurs voient ce conseil:

> "Tester des chaînes avec `==` est incorrect (à moins que les chaînes ne soient internées)"

leur première réaction consiste à utiliser des chaînes internes pour pouvoir utiliser `==` . (Après tout, `==` est plus rapide que d'appeler `String.equals(...)` , n'est-ce pas?)

C'est la mauvaise approche, sous plusieurs angles:

Fragilité

Tout d'abord, vous ne pouvez utiliser qu'en toute sécurité `==` si vous savez que *tous* les objets `String` vous testez ont été internés. Le JLS garantit que les littéraux String de votre code source auront été internés. Cependant, aucune des API Java SE standard ne garantit de renvoyer des chaînes internes, à l'exception de `String.intern(String)` elle-même. Si vous ne manquez qu'une seule source d'objets `String` qui n'ont pas été internés, votre application ne sera pas fiable. Ce manque de fiabilité se traduira par de faux négatifs plutôt que des exceptions susceptibles de rendre la détection plus difficile.

Coûts d'utilisation de 'intern ()'

Sous le capot, l'internement fonctionne en maintenant une table de hachage qui contient des objets `String` précédemment internés. Une sorte de mécanisme de référence faible est utilisé pour que la table de hachage interne ne devienne pas une fuite de stockage. Alors que la table de hachage est implémentée en code natif (contrairement à `HashMap` , `HashTable` , etc.), les appels `intern` sont encore relativement coûteux en termes de CPU et de mémoire.

Ce coût doit être comparé à celui que nous allons obtenir en utilisant `==` au lieu d' `equals` . En fait, nous n'allons pas à la rupture à moins que chaque chaîne interne soit comparée à d'autres chaînes "quelques fois".

(Mis à part: les quelques situations où l'internat est utile ont tendance à réduire l'empreinte mémoire d'une application où les mêmes chaînes se répètent plusieurs fois, *et* ces chaînes ont une longue durée de vie.)

L'impact sur la collecte des ordures

Outre les coûts directs de processeur et de mémoire décrits ci-dessus, les chaînes internes affectent les performances du ramasse-miettes.

Pour les versions de Java antérieures à Java 7, les chaînes internes sont conservées dans l'espace "PermGen", qui est rarement collecté. Si PermGen doit être collecté, cela déclenche généralement une récupération de place complète. Si l'espace PermGen se remplit complètement, la machine virtuelle Java se bloque, même s'il y avait de l'espace libre dans les espaces de pile standard.

Dans Java 7, le pool de chaînes a été déplacé de "PermGen" dans le tas normal. Cependant, la table de hachage sera toujours une structure de données à long terme, ce qui entraînera une longue durée de vie des chaînes internes. (Même si les objets de chaîne internes étaient alloués dans l'espace Eden, ils seraient très probablement promus avant d'être collectés.)

Ainsi, dans tous les cas, l'installation d'une ficelle va prolonger sa durée de vie par rapport à une ficelle ordinaire. Cela augmentera les frais généraux de la récupération de place pendant la durée de vie de la machine virtuelle Java.

Le deuxième problème est que la table de hachage doit utiliser un mécanisme de référence faible afin d'empêcher que la chaîne ne contienne de la mémoire. Mais un tel mécanisme est plus utile pour le ramasse-miettes.

Il est difficile de quantifier ces frais généraux de récupération de place, mais il ne fait aucun doute qu'ils existent. Si vous utilisez beaucoup de `intern` , ils pourraient être importants.

La taille de la table de hachage

Selon cette source , à partir de Java 6, le pool de chaînes de caractères est implémenté sous la forme d'une table de hachage de taille fixe avec des chaînes pour gérer les chaînes qui hachent le même compartiment. Dans les premières versions de Java 6, la table de hachage avait une taille constante (câblée). Un paramètre de réglage (`-XX:StringTableSize`) a été ajouté en tant que mise à jour à mi-vie à Java 6. Dans une mise à jour à mi-vie de Java 7, la taille par défaut du pool est passée de `1009` à `60013` .

L'essentiel est que si vous avez l'intention d'utiliser intensivement `intern` dans votre code, il est *conseillé* de choisir une version de Java où la taille hashtable est réglable et assurez-vous de régler la taille de manière appropriée. Sinon, les performances du `intern` risquent de se dégrader à mesure que le pool augmente.

Interning en tant que vecteur potentiel de déni de service

L'algorithme de hachage pour les chaînes est bien connu. Si vous stockez des chaînes fournies par des utilisateurs ou des applications malveillants, cela peut être utilisé dans le cadre d'une attaque par déni de service (DoS). Si l'agent malveillant organise le même code de hachage pour

toutes les chaînes qu'il fournit, cela peut entraîner une table de hachage non équilibrée et des performances `O(N)` pour `intern` ... où `N` est le nombre de chaînes en collision.

(Il existe des moyens plus simples et plus efficaces pour lancer une attaque DoS contre un service. Toutefois, ce vecteur pourrait être utilisé si l'objectif de l'attaque DoS était de briser la sécurité ou d'éviter les défenses DoS de première ligne.)

Piège - Les petites lectures / écritures sur les flux non tamponnés sont inefficaces

Considérez le code suivant pour copier un fichier vers un autre:

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

(Nous avons délibérément omis de vérifier les arguments normaux, de signaler les erreurs, etc., car ils ne sont pas pertinents pour le *point* de cet exemple.)

Si vous compilez le code ci-dessus et l'utilisez pour copier un fichier volumineux, vous remarquerez qu'il est très lent. En fait, il sera au moins deux fois plus lent que les utilitaires de copie de fichiers standard.

(*Ajouter des mesures de performances réelles ici!*)

La principale raison pour laquelle l'exemple ci-dessus est lent (dans le cas des fichiers volumineux) est qu'il effectue des lectures d'un octet et des écritures d'un octet sur les flux d'octets sans tampon. La manière simple d'améliorer les performances consiste à envelopper les flux avec des flux tamponnés. Par exemple:

```
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);
```

```
            }
        }
    }
}
```

Ces petits changements amélioreront le taux de copie des données d' *au moins* deux ordres de grandeur, en fonction de divers facteurs liés à la plate-forme. Les wrappers de flux en mémoire tampon entraînent la lecture et l'écriture des données en gros morceaux. Les instances ont toutes deux des tampons implémentés en tant que tableaux d'octets.

- Avec `is` , les données sont lues quelques kilo - octets à la fois du fichier dans la mémoire tampon. Lorsque `read()` est appelée, l'implémentation retourne généralement un octet du tampon. Il ne lira que dans le flux d'entrée sous-jacent si le tampon a été vidé.

- Le comportement de `os` est analogue. Les appels à `os.write(int)` écrivent des octets simples dans le tampon. Les données ne sont écrites dans le flux de sortie que lorsque le tampon est plein ou lorsque `os` est vidé ou fermé.

Qu'en est-il des flux basés sur des caractères?

Comme vous devez le savoir, Java I / O fournit différentes API pour lire et écrire des données binaires et textuelles.

- `InputStream` et `OutputStream` sont les API de base pour les E / S binaires basées sur les flux
- `Reader` et `Writer` sont les API de base pour les E / S de texte basées sur les flux.

Pour le texte I / O, `BufferedReader` et `BufferedWriter` sont les équivalents de `BufferedInputStream` et `BufferedOutputStream` .

Pourquoi les flux tamponnés font-ils autant de différence?

La véritable raison pour laquelle les flux mis en mémoire tampon aident les performances est liée à la manière dont une application communique avec le système d'exploitation:

- La méthode Java dans une application Java ou les appels de procédure natifs dans les bibliothèques d'exécution natives de la JVM sont rapides. Ils prennent généralement quelques instructions de la machine et ont un impact minimal sur les performances.

- En revanche, les appels d'exécution JVM au système d'exploitation ne sont pas rapides. Ils impliquent quelque chose appelé un "syscall". Le schéma type d'un appel système est le suivant:

    1. Placez les arguments syscall dans des registres.
    2. Exécutez une instruction d'interruption SYSENTER.
    3. Le gestionnaire d'interruptions passe à l'état privilégié et modifie les mappages de mémoire virtuelle. Ensuite, il envoie au code pour gérer l'appel système spécifique.
    4. Le gestionnaire syscall vérifie les arguments en veillant à ne pas avoir accès à la mémoire que le processus utilisateur ne doit pas voir.

5. Le travail spécifique à l'appel système est effectué. Dans le cas d'un appel système en `read` , cela peut impliquer:
    1. vérifier qu'il y a des données à lire à la position actuelle du descripteur de fichier
    2. appeler le gestionnaire de système de fichiers pour qu'il récupère les données requises sur le disque (ou partout où il est stocké) dans le cache tampon,
    3. copier des données du cache tampon vers l'adresse fournie par la JVM
    4. ajuster la position du descripteur de fichier pointé thstream
6. Revenez de l'appel système. Cela implique de modifier à nouveau les mappages de VM et de sortir de l'état privilégié.

Comme vous pouvez l'imaginer, exécuter un seul appel système peut contenir des milliers d'instructions de machine. De manière conservatrice, *au moins* deux ordres de grandeur plus longs qu'un appel de méthode régulier. (Probablement trois ou plus.)

Compte tenu de cela, la raison pour laquelle les flux en mémoire tampon font une grande différence est qu'ils réduisent considérablement le nombre d'appels système. Au lieu de faire un appel système pour chaque appel `read()` , le flux d'entrée en mémoire tampon lit une grande quantité de données dans un tampon, selon les besoins. La plupart des appels `read()` sur le flux en mémoire tampon effectuent des vérifications simples et renvoient un `byte` lu précédemment. Un raisonnement similaire s'applique dans le cas du flux de sortie, ainsi que dans les cas de flux de caractères.

(Certaines personnes pensent que les performances d'E / S mises en mémoire tampon proviennent de l'incompatibilité entre la taille de la requête de lecture et la taille d'un bloc de disque, la latence de rotation des disques et d'autres facteurs. l'application n'a *généralement* pas besoin d'attendre le disque, ce n'est pas la vraie explication.

Les flux tamponnés sont-ils toujours une victoire?

Pas toujours. Les flux en mémoire tampon sont certainement une victoire si votre application va faire beaucoup de "petites" lectures ou écritures. Cependant, si votre application n'a besoin que d'effectuer des lectures ou des écritures importantes sur / à partir d'un grand `byte[]` ou `char[]` , alors les flux mis en mémoire tampon ne vous apporteront aucun avantage réel. En effet, il pourrait même y avoir une pénalité de performance (minuscule).

Est-ce le moyen le plus rapide de copier un fichier en Java?

Non ce n'est pas Lorsque vous utilisez les API basées sur les flux Java pour copier un fichier, vous devez assumer le coût d'au moins une copie de la mémoire vers la mémoire supplémentaire des données. Il est possible d'éviter cela si vous utilisez les `ByteBuffer` NIO `ByteBuffer` et `Channel` . (*Ajouter un lien vers un exemple séparé ici.*)

Lire Pièges Java - Problèmes de performances en ligne:
https://riptutorial.com/fr/java/topic/5455/pieges-java---problemes-de-performances

Chapitre 138: Pièges Java - Syntaxe du langage

Introduction

Plusieurs abus de langage de programmation Java peuvent conduire à un programme pour générer des résultats incorrects malgré la compilation correcte. Ce sujet a pour objectif principal d'énumérer les écueils les plus courants de leurs causes et de proposer la manière correcte d'éviter de tomber dans de tels problèmes.

Remarques

Cette rubrique concerne des aspects spécifiques de la syntaxe du langage Java qui sont susceptibles de générer des erreurs ou qui ne doivent pas être utilisés de certaines manières.

Examples

Pitfall - Ignorer la visibilité de la méthode

Même les développeurs Java expérimentés ont tendance à penser que Java ne possède que trois modificateurs de protection. La langue a en fait quatre! Le niveau de visibilité du **paquet privé** (aka par défaut) est souvent oublié.

Vous devriez faire attention à quelles méthodes vous rendre public. Les méthodes publiques d'une application sont l'API visible de l'application. Cela devrait être aussi petit et compact que possible, surtout si vous écrivez une bibliothèque réutilisable (voir aussi le principe SOLID). Il est important de considérer de la même manière la visibilité de toutes les méthodes et d'utiliser uniquement l'accès privé protégé ou groupé, le cas échéant.

Lorsque vous déclarez des méthodes qui doivent être **privées** comme publiques, vous exposez les détails d'implémentation internes de la classe.

Un corollaire de ceci est que vous ne unité tester les méthodes publiques de votre classe - en fait , vous **ne** pouvez tester les méthodes publiques. C'est une mauvaise pratique d'augmenter la visibilité des méthodes privées simplement pour pouvoir exécuter des tests unitaires contre ces méthodes. Tester des méthodes publiques appelant les méthodes avec une visibilité plus restrictive devrait suffire à tester une API complète. Vous ne devez **jamais** développer votre API avec davantage de méthodes publiques uniquement pour autoriser les tests unitaires.

Pitfall - Manquer un "break" dans un cas de "switch"

Ces problèmes de Java peuvent être très embarrassants et restent parfois inexplorés jusqu'à la production. Un comportement irréversible dans les instructions de commutation est souvent utile. Cependant, l'absence d'un mot-clé «pause» lorsqu'un tel comportement n'est pas souhaité peut

entraîner des résultats désastreux. Si vous avez oublié de mettre un «break» dans «case 0» dans l'exemple de code ci-dessous, le programme écrira «Zero» suivi de «One», car le flux de contrôle à l'intérieur il atteint une «pause». Par exemple:

```
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");
        }
}
```

Dans la plupart des cas, la solution la plus propre consisterait à utiliser des interfaces et à déplacer le code avec un comportement spécifique dans des implémentations distinctes (*composition sur héritage*)

Si une instruction switch est inévitable, il est recommandé de documenter les retombées "attendues" si elles se produisent. De cette façon, vous montrez aux autres développeurs que vous êtes au courant de la rupture manquante et que cela est un comportement attendu.

```
switch(caseIndex) {
    [...]
    case 2:
        System.out.println("Two");
        // fallthrough
    default:
        System.out.println("Default");
```

Pitfall - Les points-virgules mal placés et les accolades manquantes

C'est une erreur qui crée une véritable confusion pour les débutants de Java, du moins la première fois qu'ils le font. Au lieu d'écrire ceci:

```
if (feeling == HAPPY)
    System.out.println("Smile");
else
    System.out.println("Frown");
```

ils écrivent accidentellement ceci:

```
if (feeling == HAPPY);
    System.out.println("Smile");
else
    System.out.println("Frown");
```

et sont perplexes lorsque le compilateur Java leur dit que le `else` est mal placé. Le compilateur Java avec interpréter ce qui suit comme suit:

```
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");
```

Dans d'autres cas, il n'y aura pas d'erreurs de compilation, mais le code ne fera pas ce que le programmeur a l'intention de faire. Par exemple:

```
for (int i = 0; i < 5; i++);
    System.out.println("Hello");
```

n'imprime que "Hello" une fois. Encore une fois, le faux point-virgule signifie que le corps de la boucle `for` est une instruction vide. Cela signifie que l'appel `println` suivant est inconditionnel.

Une autre variante:

```
for (int i = 0; i < 5; i++);
    System.out.println("The number is " + i);
```

Cela donnera une erreur "Impossible de trouver le symbole" pour `i` . La présence du point-virgule erroné signifie que l'appel `println` tente d'utiliser `i` dehors de son champ d'application.

Dans ces exemples, il existe une solution simple: supprimez simplement le demi-point erroné. Cependant, il y a des leçons plus profondes à tirer de ces exemples:

1. Le point-virgule en Java n'est pas un "bruit syntaxique". La présence ou l'absence d'un point-virgule peut changer le sens de votre programme. Ne vous contentez pas de les ajouter à la fin de chaque ligne.

2. Ne faites pas confiance à l'indentation de votre code. En langage Java, le compilateur ignore les espaces blancs en début de ligne.

3. Utilisez un pénétrateur automatique. Tous les IDE et de nombreux éditeurs de texte simples comprennent comment mettre correctement en retrait le code Java.

4. C'est la leçon la plus importante. Suivez les dernières directives de style Java et placez des accolades autour des instructions "then" et "else" et de la déclaration de corps d'une boucle. L'attache ouverte (`{`) ne doit pas figurer sur une nouvelle ligne.

Si le programmeur suivait les règles de style, l'exemple `if` avec des points-virgules égarés ressemblerait à ceci:

```
if (feeling == HAPPY); {
    System.out.println("Smile");
} else {
```

```
    System.out.println("Frown");
}
```

Cela semble étrange à un œil expérimenté. Si vous indentez automatiquement ce code, cela ressemblera probablement à ceci:

```
if (feeling == HAPPY); {
                           System.out.println("Smile");
                       } else {
                           System.out.println("Frown");
                       }
```

qui devrait se démarquer même mal pour un débutant.

Pitfall - Quitter les accolades: les problèmes de "pendants si" et de "pendants"

La dernière version du guide de style Java Oracle exige que les instructions "then" et "else" d'une instruction `if` soient toujours placées entre "accolades" ou "accolades". Des règles similaires s'appliquent aux corps des différentes instructions de boucle.

```
if (a) {           // <- open brace
    doSomething();
    doSomeMore();
}                  // <- close brace
```

Ce n'est pas réellement requis par la syntaxe du langage Java. En effet, si la partie "alors" d'une déclaration `if` est une déclaration unique, il est légal de laisser de côté les accolades

```
if (a)
    doSomething();
```

ou même

```
if (a) doSomething();
```

Cependant, il y a des dangers à ignorer les règles de style Java et à laisser de côté les accolades. Plus précisément, vous augmentez considérablement le risque que le code avec une indentation erronée soit mal interprété.

Le problème du "dangling if":

Considérez l'exemple de code ci-dessus, réécrit sans accolades.

```
if (a)
   doSomething();
   doSomeMore();
```

Ce code *semble dire* que les appels à `doSomething` et `doSomeMore` se produiront tous les deux *si et*

seulement si `a` est `true` . En fait, le code est indenté incorrectement. La spécification de langage Java que l'appel `doSomeMore()` est une instruction distincte suivant l'instruction `if` . L'indentation correcte est la suivante:

```
if (a)
   doSomething();
doSomeMore();
```

Le problème du "dangling else"

Un deuxième problème apparaît lorsque l' on ajoute le `else` au mélange. Prenons l'exemple suivant avec des accolades manquantes.

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
else
   doZ();
```

Le code ci-dessus *semble dire* que `doZ` sera appelé quand `a` est `false` . En fait, l'indentation est incorrecte encore une fois. L'indentation correcte du code est la suivante:

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
   else
      doZ();
```

Si le code était écrit conformément aux règles de style Java, cela ressemblerait à ceci:

```
if (a) {
   if (b) {
      doX();
   } else if (c) {
      doY();
   } else {
      doZ();
   }
}
```

Pour illustrer pourquoi cela est mieux, supposez que vous ayez accidentellement induit le code en erreur. Vous pourriez vous retrouver avec quelque chose comme ça:

```
if (a) {                        if (a) {
   if (b) {                         if (b) {
      doX();                           doX();
   } else if (c) {                  } else if (c) {
      doY();                           doY();
} else {                        } else {
```

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

Mais dans les deux cas, le code mal indenté "semble erroné" aux yeux d'un programmeur Java expérimenté.

Piège - Surcharge au lieu de dépasser

Prenons l'exemple suivant:

```
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();
    }
}
```

Ce code ne va pas se comporter comme prévu. Le problème est que les méthodes `equals` et `hashcode` pour `Person` ne remplacent pas les méthodes standard définies par `Object` .

- La méthode `equals` a la mauvaise signature. Il doit être déclaré comme `equals(Object)` non `equals(String)` .
- La méthode de `hashcode` n'a pas le bon nom. Ce devrait être `hashCode()` (notez le **C** majuscule).

Ces erreurs signifient que nous avons déclaré des surcharges accidentelles, et celles-ci ne seront pas utilisées si `Person` est utilisé dans un contexte polymorphe.

Cependant, il existe un moyen simple de gérer cela (à partir de Java 5). Utilisez la `@Override` annotation chaque fois que vous avez l' *intention* de votre méthode pour être un remplacement:

Java SE 5

```
public final class Person {
    ...

    @Override
```

```
    public boolean equals(String other) {
        ....
    }

    @Override
    public hashcode() {
        ....
    }
}
```

Lorsque nous ajoutons une `@Override` annotation à une déclaration de méthode, le compilateur vérifiera que la méthode *ne* remplace (ou mettre en œuvre) une méthode déclarée dans une superclasse ou de l' interface. Ainsi, dans l'exemple ci-dessus, le compilateur nous donnera deux erreurs de compilation, ce qui devrait suffire à nous alerter de l'erreur.

Piège - littéraux octaux

Considérez l'extrait de code suivant:

```
// 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 débutant Java pourrait être surpris de savoir que le programme ci-dessus imprime la mauvaise réponse. Il affiche en fait la somme des chiffres 1 à 8.

La raison en est qu'un littéral entier qui commence par le chiffre zéro ('0') est interprété par le compilateur Java comme un littéral octal, et non comme un littéral décimal. Ainsi, `010` est le nombre octal 10, qui est 8 en décimal.

Piège - Déclaration de classes avec les mêmes noms que les classes standard

Parfois, les programmeurs novices en Java font l'erreur de définir une classe avec un nom identique à une classe largement utilisée. Par exemple:

```
package com.example;

/**
 * My string utilities
 */
public class String {
    ....
}
```

Ensuite, ils se demandent pourquoi ils obtiennent des erreurs inattendues. Par exemple:

```
package com.example;
```

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

Si vous compilez puis essayez d'exécuter les classes ci-dessus, vous obtiendrez une erreur:

```
$ 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
```

Quelqu'un qui regarde le code de la classe `Test` verra la déclaration de `main` et regardera sa signature et se demandera de quoi la commande `java` se plaint. Mais en fait, la commande `java` dit la vérité.

Lorsque nous déclarons une version de `String` dans le même package que `Test` , cette version est prioritaire sur l'importation automatique de `java.lang.String` . Ainsi, la signature de la méthode `Test.main` est en fait

```
void main(com.example.String[] args)
```

au lieu de

```
void main(java.lang.String[] args)
```

et la `java` commande ne reconnaît pas *que* comme une méthode de point d' entrée.

Leçon: Ne définissez pas les classes qui portent le même nom que les classes existantes dans `java.lang` ou d'autres classes couramment utilisées dans la bibliothèque Java SE. Si vous faites cela, vous vous exposez à toutes sortes d'erreurs obscures.

Pitfall - Utiliser '==' pour tester un booléen

Parfois, un nouveau programmeur Java écrit un code comme celui-ci:

```
public void check(boolean ok) {
    if (ok == true) {           // Note 'ok == true'
        System.out.println("It is OK");
    }
}
```

Un programmeur expérimenté le trouverait maladroit et voudrait le réécrire comme suit:

```
public void check(boolean ok) {
    if (ok) {
       System.out.println("It is OK");
    }
}
```

Cependant, il y a plus de tort avec `ok == true` que la simple maladresse. Considérez cette variation:

```
public void check(boolean ok) {
    if (ok = true) {           // Oooops!
        System.out.println("It is OK");
    }
}
```

Ici, le programmeur a mal interprété `==` comme `=` ... et maintenant le code a un bogue subtil. L'expression `x = true` assigne inconditionnellement `true` à `x` et évalue ensuite `true` . En d'autres termes, la méthode de `check` affichera désormais "Il est correct", quel que soit le paramètre.

La leçon ici est de sortir de l'habitude d'utiliser `== false` et `== true` . En plus d'être verbeux, ils rendent votre codage plus susceptible aux erreurs.

Note: Une alternative possible à `ok == true` qui évite le piège est d'utiliser les conditions de Yoda ; c'est-à-dire mettre le littéral à gauche de l'opérateur relationnel, comme dans `true == ok` . Cela fonctionne, mais la plupart des programmeurs seraient probablement d'accord pour dire que les conditions de Yoda semblent étranges. Certes, `ok` (ou `!ok`) est plus concis et plus naturel.

Pitfall - Les importations de Wildcard peuvent rendre votre code fragile

Prenons l'exemple partiel suivant:

```
import com.example.somelib.*;
import com.acme.otherlib.*;

public class Test {
    private Context x = new Context();   // from com.example.somelib
    ...
}
```

Supposons que lorsque vous avez développé le code pour la première fois avec la version 1.0 de `somelib` et la version 1.0 de `otherlib` . Ensuite, vous devrez mettre à niveau vos dépendances vers des versions ultérieures et décider d'utiliser `otherlib` version 2.0. Supposons également que l'une des modifications apportées à `otherlib` entre 1.0 et 2.0 consistait à ajouter une classe de `Context` .

Maintenant, lorsque vous recompilez `Test` , vous obtenez une erreur de compilation indiquant que le `Context` est une importation ambiguë.

Si vous êtes familier avec la base de code, cela représente probablement un inconvénient mineur. Sinon, vous avez du travail à faire pour résoudre ce problème, ici et potentiellement ailleurs.

Le problème ici est les importations de caractères génériques. D'une part, l'utilisation de caractères génériques peut rendre vos cours plus courts. D'autre part:

- Des modifications compatibles vers le haut vers d'autres parties de votre base de code, vers des bibliothèques standard Java ou vers des bibliothèques tierces peuvent entraîner des

erreurs de compilation.

- La lisibilité en souffre. À moins d'utiliser un IDE, il peut être difficile de déterminer les importations de caractères génériques dans une classe nommée.

La leçon est que c'est une mauvaise idée d'utiliser des importations de caractères génériques dans un code qui doit durer longtemps. Les importations spécifiques (non génériques) ne nécessitent pas beaucoup d'efforts si vous utilisez un IDE, et l'effort en vaut la peine.

Piège: Utiliser 'assert' pour la validation des arguments ou des entrées utilisateur

Une question qui se pose parfois sur StackOverflow est de savoir s'il convient d'utiliser `assert` pour valider les arguments fournis à une méthode, ou même les entrées fournies par l'utilisateur.

La réponse simple est que ce n'est pas approprié.

Les meilleures alternatives incluent:

- Lancer une exception IllegalArgumentException à l'aide du code personnalisé.
- Utilisation des méthodes `Preconditions` disponibles dans la bibliothèque Google Guava.
- Utilisation des méthodes `Validate` disponibles dans la bibliothèque Apache Commons Lang3.

Voici ce que la spécification de langage Java (JLS 14.10, pour Java 8) conseille à cet égard:

> En règle générale, la vérification des assertions est activée lors du développement et du test du programme et désactivée pour le déploiement afin d'améliorer les performances.
>
> Les assertions pouvant être désactivées, les programmes ne doivent pas supposer que les expressions contenues dans les assertions seront évaluées. Ainsi, ces expressions booléennes devraient généralement être exemptes d'effets secondaires. L'évaluation d'une telle expression booléenne ne devrait affecter aucun état visible une fois l'évaluation terminée. Il n'est pas illégal qu'une expression booléenne contenue dans une assertion ait un effet secondaire, mais elle est généralement inappropriée, car elle pourrait faire varier le comportement du programme selon que les assertions ont été activées ou désactivées.
>
> À la lumière de cela, les assertions ne doivent pas être utilisées pour vérifier les arguments dans les méthodes publiques. La vérification des arguments fait généralement partie du contrat d'une méthode et ce contrat doit être respecté, que les assertions soient activées ou désactivées.
>
> Un problème secondaire lié à l'utilisation d'assertions pour la vérification d'arguments est que des arguments erronés doivent entraîner une exception d'exécution appropriée (telle `IllegalArgumentException` , `ArrayIndexOutOfBoundsException` ou `NullPointerException`). Un échec d'assertion ne déclenchera pas une exception appropriée. Encore une fois, il n'est pas illégal d'utiliser des assertions pour vérifier les arguments sur des méthodes publiques, mais cela est généralement inapproprié. Il est prévu que `AssertionError` ne

soit jamais intercepté, mais il est possible de le faire, donc les règles pour les instructions try doivent traiter les assertions apparaissant dans un bloc try de la même manière que le traitement actuel des instructions throw.

Piège des objets nuls auto-unboxing dans les primitifs

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

        // example:
        Boolean ignore = null;
        if (ignore == false) {
            System.out.println("Do not ignore!");
        }
    }
}
```

Le piège est que `null` est comparé à `false` . Étant donné que nous comparons un `boolean` primitif à un `Boolean` , Java tente de *déballer* l' `Object Boolean` dans un équivalent primitif, prêt pour la comparaison. Cependant, puisque cette valeur est `null` , une `NullPointerException` est levée.

Java est incapable de comparer les types primitifs aux valeurs `null` , ce qui provoque une `NullPointerException` à l'exécution. Considérons le cas primitif de la condition `false == null` ; Cela générerait une erreur de *compilation de* `incomparable types: int and <null>` .

Lire Pièges Java - Syntaxe du langage en ligne: https://riptutorial.com/fr/java/topic/5382/pieges-java---syntaxe-du-langage

Chapitre 139: Pièges Java - Threads et accès concurrents

Examples

Piège: utilisation incorrecte de wait () / notify ()

Les méthodes `object.wait()` , `object.notify()` et `object.notifyAll()` sont destinées à être utilisées de manière très spécifique. (voir http://stackoverflow.com/documentation/java/5409/wait-notify#t=20160811161648303307)

Le problème "Notification perdue"

Une erreur courante des débutants consiste à appeler inconditionnellement `object.wait()`

```
private final Object lock = new Object();

public void myConsumer() {
    synchronized (lock) {
        lock.wait();     // DON'T DO THIS!!
    }
    doSomething();
}
```

La raison en est que cela dépend d'un autre thread à appeler `lock.notify()` ou `lock.notifyAll()` , mais rien ne garantit que l'autre thread n'a pas effectué cet appel *avant* le thread consommateur appelé `lock.wait()` .

`lock.notify()` et `lock.notifyAll()` ne font rien du tout si un autre thread n'attend pas *déjà* la notification. Le thread qui appelle `myConsumer()` dans cet exemple sera bloqué pour toujours s'il est trop tard pour intercepter la notification.

Le bogue "État de surveillance illégal"

Si vous appelez `wait()` ou `notify()` sur un objet sans le verrouiller, la machine `IllegalMonitorStateException` .

```
public void myConsumer() {
    lock.wait();      // throws exception
    consume();
}

public void myProducer() {
    produce();
    lock.notify();    // throws exception
}
```

(La conception de `wait()` / `notify()` exige que le verrou soit maintenu car cela est nécessaire pour éviter les conditions de concurrence systémiques. S'il était possible d'appeler `wait()` ou `notify()` sans verrouiller, il serait impossible de l'implémenter le principal cas d'utilisation de ces primitives: attendre qu'une condition se produise.)

Attendre / notifier est trop bas

La *meilleure* façon d'éviter les problèmes avec `wait()` et `notify()` est de ne pas les utiliser. La plupart des problèmes de synchronisation peuvent être résolus en utilisant les objets de synchronisation de niveau supérieur (files d'attente, barrières, sémaphores, etc.) disponibles dans le package `java.utils.concurrent` .

Piège - Extension de 'java.lang.Thread'

Le javadoc pour la classe `Thread` montre deux façons de définir et d'utiliser un thread:

Utiliser une classe de thread personnalisée:

```
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();
```

Utiliser 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();
```

(Source: `java.lang.Thread` javadoc .)

L’approche de classe de thread personnalisée fonctionne, mais elle pose quelques problèmes:

1. Il est difficile d'utiliser `PrimeThread` dans un contexte qui utilise un pool de threads classique, un exécuteur ou le framework ForkJoin. (Ce n'est pas impossible, car `PrimeThread` implémente indirectement `Runnable` , mais l'utilisation d'une classe `Thread` personnalisée en tant que `Runnable` est certainement maladroite et peut ne pas être viable ... selon les autres aspects de la classe.)

2. Il y a plus de possibilités d'erreurs dans d'autres méthodes. Par exemple, si vous déclarez un `PrimeThread.start()` sans déléguer à `Thread.start()` , vous obtiendrez un "thread" qui s'exécutera sur le thread en cours.

L'approche consistant à placer la logique de thread dans un `Runnable` évite ces problèmes. En effet, si vous utilisez une classe anonyme (Java 1.1 et supérieur) pour implémenter le `Runnable` le résultat est plus succinct et plus lisible que les exemples ci-dessus.

```
final long minPrime = ...
new Thread(new Runnable() {
    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}.start();
```

Avec une expression lambda (Java 8 et suivants), l'exemple ci-dessus deviendrait encore plus élégant:

```
final long minPrime = ...
new Thread(() -> {
   // compute primes larger than minPrime
    . . .
}).start();
```

Pitfall - Trop de threads rendent une application plus lente.

Beaucoup de personnes novices en matière de multithread pensent que l'utilisation de threads accélère le processus. En fait, c'est beaucoup plus compliqué que ça. Mais une chose que nous pouvons affirmer avec certitude est que, pour tout ordinateur, le nombre de threads pouvant être exécutés simultanément est limité:

- Un ordinateur a un nombre fixe de *cœurs* (ou *hyperthreads*).
- Un thread Java doit être *planifié* sur un core ou un hyperthread pour pouvoir s'exécuter.
- Si les threads Java exécutables sont plus nombreux que les cores / hyperthreads (disponibles), certains doivent attendre.

Cela nous indique que la création de plus en plus de threads Java *ne permet pas d'* accélérer l'application. Mais il y a aussi d'autres considérations:

- Chaque thread nécessite une région de mémoire hors tas pour sa pile de threads. La taille de pile de threads standard (par défaut) est de 512 Ko ou 1 Mo. Si vous avez un nombre important de threads, l'utilisation de la mémoire peut être importante.

- Chaque thread actif fera référence à un certain nombre d'objets dans le tas. Cela augmente l'ensemble de travail des objets *accessibles* , ce qui a un impact sur la récupération de place et l'utilisation de la mémoire physique.
- Les frais généraux de commutation entre les threads ne sont pas triviaux. Cela implique généralement un basculement dans l'espace du noyau du système d'exploitation pour prendre une décision de planification des threads.
- Les surcharges de la synchronisation des threads et de la signalisation inter-thread (par exemple wait (), notify () / notifyAll) *peuvent être* importantes.

Selon les détails de votre application, ces facteurs signifient généralement qu'il existe un «point idéal» pour le nombre de threads. Au-delà de cela, l'ajout de nouveaux threads apporte une amélioration minime des performances et peut aggraver les performances.

Si votre application crée pour chaque nouvelle tâche, une augmentation imprévue de la charge de travail (par exemple un taux de demande élevé) peut entraîner un comportement catastrophique.

Une meilleure façon de gérer cela est d'utiliser un pool de threads borné dont vous pouvez contrôler la taille (de manière statique ou dynamique). Lorsqu'il y a trop de travail à faire, l'application doit mettre les demandes en attente. Si vous utilisez un `ExecutorService` , il se chargera de la gestion du pool de threads et de la mise en file d'attente des tâches.

Piège - La création de fils est relativement coûteuse

Considérons ces deux micro-repères:

Le premier benchmark crée, démarre et rejoint simplement les threads. Le thread `Runnable` ne fonctionne pas.

```
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
```

Sur un PC moderne typique fonctionnant sous Linux avec Java 8 bits 64 u101, ce test montre un temps moyen de création, de démarrage et de jonction de threads compris entre 33,6 et 33,9 microsecondes.

Le second test fait l'équivalent du premier mais utilise un `ExecutorService` pour soumettre des tâches et un `Future` pour rejoindre la fin de la tâche.

```
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
```

Comme vous pouvez le voir, les moyennes se situent entre 5,3 et 5,6 microsecondes.

Bien que les temps réels dépendent de divers facteurs, la différence entre ces deux résultats est significative. Il est clairement plus rapide d'utiliser un pool de threads pour recycler des threads

que de créer de nouveaux threads.

Piège: les variables partagées nécessitent une synchronisation correcte

Considérez cet exemple:

```
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.
    }
}
```

L'intention de ce programme est de démarrer un thread, de le laisser fonctionner pendant 1000 millisecondes, puis de l'arrêter en définissant l'indicateur d' `stop` .

Cela fonctionnera-t-il comme prévu?

Peut-être que oui, peut-être que non.

Une application ne s'arrête pas nécessairement lorsque la méthode `main` revient. Si un autre thread a été créé et que ce thread n'a pas été marqué comme un thread de démon, l'application continuera à s'exécuter une fois le thread principal terminé. Dans cet exemple, cela signifie que l'application continuera à s'exécuter jusqu'à la fin du thread enfant. Cela devrait se `tt.stop` lorsque `tt.stop` est défini sur `true` .

Mais ce n'est pas vraiment vrai. En fait, le thread enfant s'arrête après avoir *observé l'* `stop` avec la valeur `true` . Est-ce que ça va arriver? Peut-être que oui, peut-être que non.

La spécification de langage Java *garantit* que les lectures et écritures de mémoire effectuées dans un thread sont visibles pour ce thread, conformément à l'ordre des instructions dans le code source. Cependant, en général, cela n'est PAS garanti lorsqu'un thread écrit et qu'un autre thread (ultérieurement) lit. Pour que la visibilité soit garantie, il doit exister une chaîne d' *événements-avant les* relations entre une écriture et une lecture ultérieure. Dans l'exemple ci-dessus, il n'existe pas de chaîne de ce type pour la mise à jour de l'indicateur d' `stop` . Par conséquent, il n'est pas garanti que le thread enfant verra `stop` change à `true` .

(Note aux auteurs: il devrait y avoir une rubrique distincte sur le modèle de mémoire Java pour

entrer dans les détails techniques approfondis.)

Comment pouvons-nous résoudre le problème?

Dans ce cas, il existe deux méthodes simples pour s'assurer que l' `stop` mise à jour est visible:

1. Déclarez `stop` d'être `volatile` ; c'est à dire

   ```
   private volatile boolean stop = false;
   ```

   Pour une variable `volatile` , le JLS spécifie qu'il y a une relation " *passe-avant"* entre un thread écrit par un et une lecture ultérieure par un second thread.

2. Utilisez un mutex pour synchroniser comme suit:

```
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.
        }
    }
}
```

En plus de s'assurer de l'exclusion mutuelle, le JLS spécifie qu'il existe une relation *avant-après* entre la libération d'un mutex dans un thread et l'obtention du même mutex dans un second thread.

Mais l'assignation n'est-elle pas atomique?

Oui, ça l'est!

Cependant, cela ne signifie pas que les effets de la mise à jour seront visibles simultanément sur tous les threads. Seule une chaîne appropriée de relations *préalables* garantira cela.

Pourquoi ont-ils fait ça?

Les programmeurs qui font de la programmation multithread en Java pour la première fois trouvent le modèle de mémoire difficile. Les programmes se comportent de manière non intuitive car l'attente naturelle est que les écritures soient visibles de manière uniforme. Alors, pourquoi les concepteurs Java conçoivent le modèle de mémoire de cette manière.

Cela se résume à un compromis entre performance et facilité d'utilisation (pour le programmeur).

Une architecture informatique moderne se compose de plusieurs processeurs (cœurs) avec des ensembles de registres individuels. La mémoire principale est accessible à tous les processeurs ou à des groupes de processeurs. Une autre propriété du matériel informatique moderne est que l'accès aux registres est généralement plus rapide que l'accès à la mémoire principale. À mesure que le nombre de cœurs évolue, il est facile de voir que lire et écrire dans la mémoire principale peut devenir le principal goulot d'étranglement des performances d'un système.

Cette incompatibilité est résolue en implémentant un ou plusieurs niveaux de mémoire cache entre les cœurs du processeur et la mémoire principale. Chaque cœur accède aux cellules mémoire via son cache. Normalement, une lecture en mémoire principale se produit uniquement en cas d'échec du cache et une écriture en mémoire principale ne se produit que si une ligne de cache doit être vidée. Pour une application dans laquelle le jeu d'emplacements de mémoire de chaque cœur peut tenir dans son cache, la vitesse de base n'est plus limitée par la vitesse / la bande passante de la mémoire principale.

Mais cela nous pose un nouveau problème lorsque plusieurs cœurs lisent et écrivent des variables partagées. La dernière version d'une variable peut se trouver dans le cache d'un cœur. À moins que ce noyau vide la ligne de cache dans la mémoire principale ET que d'autres cœurs invalident leur copie en cache des versions antérieures, certains d'entre eux risquent de voir des versions obsolètes de la variable. Mais si les caches étaient vidés en mémoire chaque fois qu'il y avait une écriture en cache ("juste au cas où" il y aurait une lecture par un autre noyau), cela consommerait inutilement la bande passante de la mémoire principale.

La solution standard utilisée au niveau du jeu d'instructions matérielles consiste à fournir des instructions pour l'invalidation du cache et la mise en cache du cache, et laisse le compilateur décider du moment où il doit les utiliser.

Retour à Java Le modèle de mémoire est conçu pour que les compilateurs Java ne soient pas obligés d'émettre des instructions d'invalidation de cache et d'écriture directe lorsqu'ils ne sont pas vraiment nécessaires. L'hypothèse est que le programmeur utilisera un mécanisme de synchronisation approprié (par exemple des mutex primitifs, `volatile` classes de concurrence de niveau supérieur `volatile` , etc.) pour indiquer qu'il a besoin de visibilité de la mémoire. En l'absence d'une relation *se produit avant* , les compilateurs Java sont libres de *supposer* qu'aucune opération de cache (ou similaire) n'est requise.

Cela présente des avantages significatifs en termes de performances pour les applications multithread, mais l'inconvénient est que l'écriture d'applications multithread correctes n'est pas simple. Le programmeur *doit* comprendre ce qu'il fait.

Pourquoi ne puis-je pas reproduire cela?

Il existe un certain nombre de raisons pour lesquelles de tels problèmes sont difficiles à reproduire:

1. Comme expliqué ci-dessus, le fait de ne pas traiter correctement les problèmes de visibilité de la mémoire signifie *généralement* que votre application compilée ne gère pas correctement les caches de mémoire. Cependant, comme nous l'avons mentionné plus haut, les caches de mémoire sont souvent vidés.

2. Lorsque vous modifiez la plate-forme matérielle, les caractéristiques des caches mémoire peuvent changer. Cela peut entraîner un comportement différent si votre application ne se synchronise pas correctement.

3. Vous observez peut-être les effets d'une synchronisation *fortuite* . Par exemple, si vous ajoutez des traces de trace, il se produit généralement une synchronisation en arrière-plan dans les flux d'E / S qui provoque des vidages de cache. Ainsi, l'ajout de traces rend *souvent* l'application différente.

4. L'exécution d'une application sous un débogueur le compile différemment par le compilateur JIT. Les points d'arrêt et les étapes simples aggravent la situation. Ces effets changeront souvent le comportement d'une application.

Ces problèmes rendent les bogues dus à une synchronisation inadéquate particulièrement difficile à résoudre.

Lire Pièges Java - Threads et accès concurrents en ligne:
https://riptutorial.com/fr/java/topic/5567/pieges-java---threads-et-acces-concurrents

Chapitre 140: Pièges Java - Utilisation des exceptions

Introduction

Plusieurs abus de langage de programmation Java peuvent conduire à un programme pour générer des résultats incorrects malgré la compilation correcte. L'objectif principal de ce sujet est de répertorier les **pièges** courants liés à la **gestion des exceptions** et de proposer la manière correcte d'éviter de tels pièges.

Examples

Piège - Ignorer ou écraser les exceptions

Cet exemple concerne le fait d'ignorer ou d'écraser délibérément des exceptions. Ou, pour être plus précis, il s'agit de savoir comment capturer et gérer une exception de manière à l'ignorer. Cependant, avant de décrire comment faire cela, nous devons d'abord souligner que les exceptions pour écraser ne sont généralement pas la bonne façon de les gérer.

Des exceptions sont généralement émises (par quelque chose) pour notifier aux autres parties du programme qu'un événement significatif ("exceptionnel") s'est produit. Généralement (mais pas toujours) une exception signifie que quelque chose a mal tourné. Si vous codez votre programme pour écraser l'exception, il y a de fortes chances que le problème réapparaisse sous une autre forme. Pour aggraver les choses, lorsque vous écrasez l'exception, vous jetez les informations dans l'objet exception et sa trace de pile associée. Cela risque de rendre plus difficile l'identification de la source du problème.

En pratique, les squashings d'exceptions se produisent souvent lorsque vous utilisez la fonctionnalité de correction automatique de l'EDI pour "corriger" une erreur de compilation provoquée par une exception non gérée. Par exemple, vous pourriez voir du code comme ceci:

```
try {
    inputStream = new FileInputStream("someFile");
} catch (IOException e) {
    /* add exception handling code here */
}
```

Clairement, le programmeur a accepté la suggestion de l'IDE de supprimer l'erreur de compilation, mais la suggestion était inappropriée. (Si le fichier ouvert a échoué, le programme devrait probablement faire quelque chose à ce sujet. Avec la "correction" ci-dessus, le programme risque d'échouer ultérieurement, par exemple avec une `inputStream` `NullPointerException` car `inputStream` est maintenant `null` .)

Cela dit, voici un exemple d'écrasement délibéré d'une exception. (Aux fins de l'argumentation, supposons que nous avons déterminé qu'une interruption lors de l'affichage du selfie est

inoffensive.) Le commentaire indique au lecteur que nous avons écrasé l'exception délibérément et pourquoi nous l'avons fait.

```
try {
    selfie.show();
} catch (InterruptedException e) {
    // It doesn't matter if showing the selfie is interrupted.
}
```

Une autre manière conventionnelle de souligner que nous avons *délibérément* écrasé une exception sans dire pourquoi est d'indiquer ceci avec le nom de la variable d'exception, comme ceci:

```
try {
    selfie.show();
} catch (InterruptedException ignored) {  }
```

Certains IDE (comme IntelliJ IDEA) n'afficheront pas d'avertissement concernant le bloc catch vide si le nom de la variable est défini sur `ignored` .

Piège - Attraper une exception pouvant être lancée, une exception ou une erreur d'exécution

Un modèle de pensée commune pour les programmeurs Java inexpérimentés est que les exceptions sont « un problème » ou « un fardeau » et la meilleure façon de traiter ce problème est de les attraper tous [1] le plus tôt possible. Cela conduit à un code comme celui-ci:

```
....
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (Exception ex) {
    System.out.println("Could not open file " + fileName);
}
```

Le code ci-dessus présente un défaut important. Le `catch` va en fait attraper plus d'exceptions que ce que le programmeur attend. Supposons que la valeur du nom de `fileName` est `null` , en raison d'un bogue ailleurs dans l'application. Cela entraînera le constructeur `FileInputStream` à lancer une `FileInputStream` `NullPointerException` . Le gestionnaire intercepte ceci et signale à l'utilisateur:

```
    Could not open file null
```

ce qui est inutile et déroutant. Pire encore, supposons que ce soit le code "traiter l'entrée" qui a déclenché l'exception inattendue (cochée ou décochée!). Maintenant, l'utilisateur recevra le message trompeur pour un problème qui ne s'est pas produit lors de l'ouverture du fichier, et peut ne pas avoir de lien avec les E / S.

La racine du problème est que le programmeur a codé un gestionnaire pour `Exception` . C'est presque toujours une erreur:

- Catching `Exception` intercepte toutes les exceptions vérifiées et la plupart des exceptions non vérifiées.
- Attraper `RuntimeException` intercepte la plupart des exceptions non `RuntimeException` .
- Catching `Error` détecte les exceptions non vérifiées qui signalent les erreurs internes de la JVM. Ces erreurs ne sont généralement pas récupérables et ne doivent pas être interceptées.
- Catching `Throwable` interceptera toutes les exceptions possibles.

Le problème lié à un ensemble trop large d'exceptions est que le gestionnaire ne peut généralement pas tous les gérer correctement. Dans le cas de l' `Exception` , etc., il est difficile pour le programmeur de prédire ce qui *pourrait* être intercepté. c'est-à-dire à quoi s'attendre.

En général, la bonne solution est de traiter les exceptions levées. Par exemple, vous pouvez les attraper et les manipuler sur place:

```
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (FileNotFoundException ex) {
    System.out.println("Could not open file " + fileName);
}
```

ou vous pouvez les déclarer comme `thrown` par la méthode

Il existe très peu de situations où la capture d' `Exception` est appropriée. Le seul qui se présente couramment est quelque chose comme ceci:

```
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);
    }
}
```

Ici, nous voulons vraiment gérer toutes les exceptions, donc capturer `Exception` (ou même `Throwable`) est correct.

---

1 - Aussi connu sous le nom de gestion des exceptions Pokemon .

Piège - Lancer Throwable, Exception, Error ou RuntimeException

Bien que la capture des `Exception` `Throwable` , `Exception` , `Error` et `RuntimeException` soit mauvaise, leur `RuntimeException` est encore pire.

Le problème de base est que lorsque votre application doit gérer des exceptions, la présence des exceptions de niveau supérieur rend difficile la distinction entre différentes conditions d'erreur. Par

exemple

```
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
}
```

Le problème est que parce que nous avons lancé une instance `Exception` , nous sommes obligés de l'attraper. Cependant, comme décrit dans un autre exemple, la détection des `Exception` est mauvaise. Dans ce cas, il devient difficile de faire la distinction entre le cas "attendu" d'une `Exception` déclenchée si `somethingBad` est `true` et le cas inattendu où nous interceptons une exception non vérifiée telle que `NullPointerException` .

Si l'exception de niveau supérieur est autorisée à se propager, nous rencontrons d'autres problèmes:

- Nous devons maintenant nous souvenir de toutes les différentes raisons pour lesquelles nous avons lancé le plus haut niveau et les discriminer / gérer.
- Dans le cas d' `Exception` et `Throwable` nous devons aussi ajouter ces exceptions à la `throws` clause de méthodes si nous voulons l'exception de se propager. Ceci est problématique, comme décrit ci-dessous.

En bref, ne jetez pas ces exceptions. Jetez une exception plus spécifique qui décrit plus précisément "l'événement exceptionnel" qui s'est produit. Si vous devez, définissez et utilisez une classe d'exception personnalisée.

Déclarer Throwable ou Exception dans les "lancers" d'une méthode est problématique.

Il est tentant de remplacer une longue liste d'exceptions lancées dans une méthode de `throws` clause avec `Exception` ou même `Throwable. C'est une mauvaise idée:

1. Il force l'appelant à gérer (ou propager) les `Exception` .
2. Nous ne pouvons plus compter sur le compilateur pour nous informer des exceptions vérifiées spécifiques à gérer.
3. Manipuler correctement l' `Exception` est difficile. Il est difficile de savoir quelles exceptions réelles peuvent être interceptées et, si vous ne savez pas ce qui pourrait être capturé, il est difficile de savoir quelle stratégie de rétablissement est appropriée.
4. Manipulation `Throwable` est encore plus difficile, car vous devez maintenant faire face à des défaillances potentielles qui ne devraient jamais être récupérées.

Ce conseil signifie que certains autres modèles doivent être évités. Par exemple:

```
try {
    doSomething();
} catch (Exception ex) {
    report(ex);
    throw ex;
}
```

Les tentatives ci-dessus tentent de consigner toutes les exceptions au fur et à mesure qu'elles passent, sans les traiter définitivement. Malheureusement, avant Java 7, le `throw ex;` La déclaration a amené le compilateur à penser que toute `Exception` pouvait être lancée. Cela pourrait vous obliger à déclarer la méthode englobante comme faisant `throws Exception` . A partir de Java 7, le compilateur sait que l'ensemble des exceptions qui pourraient être (re-lancées) est plus petit.

Pitfall - Catching InterruptedException

Comme déjà souligné dans d’autres pièges, en rattrapant toutes les exceptions en utilisant

```
try {
    // Some code
} catch (Exception) {
    // Some error handling
}
```

Livré avec beaucoup de problèmes différents. Mais un problème particulier est qu’il peut entraîner des blocages lorsqu’il interrompt le système d’interruption lorsqu’il écrit des applications multithread.

Si vous lancez un thread, vous devez également pouvoir l'arrêter brusquement pour diverses raisons.

```
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
```

Le `t.interrupt()` InterruptedException dans ce thread, car il est destiné à arrêter le thread. Mais

que se passe-t-il si le thread doit nettoyer certaines ressources avant qu'il ne soit complètement arrêté? Pour cela, il peut intercepter l'exception InterruptedException et effectuer un nettoyage.

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

Mais si vous avez une expression catch-all dans votre code, l'exception InterruptedException sera également prise en compte et l'interruption ne se poursuivra pas. Ce qui dans ce cas pourrait conduire à un blocage car le thread parent attend indéfiniment que ce thead s'arrête avec `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();
        }
    }
}
```

Il est donc préférable d'attraper les exceptions individuellement, mais si vous insistez pour utiliser un catch-all, prenez au moins l'interception d'InterruptedException au préalable.

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

Piège - Utilisation des exceptions pour un contrôle de flux normal

Il y a un mantra que certains experts Java récitent:

> "Les exceptions ne doivent être utilisées que dans des cas exceptionnels."

(Par exemple: http://programmers.stackexchange.com/questions/184654)

L'essentiel est que c'est une mauvaise idée (en Java) d'utiliser la gestion des exceptions et des exceptions pour implémenter un contrôle de flux normal. Par exemple, comparez ces deux manières de traiter un paramètre qui pourrait être nul.

```
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 "";
    }
}
```

Dans cet exemple, nous traitons (par conception) le cas où le `word` est `null` comme s'il s'agissait d'un mot vide. Les deux versions traitent de `null` soit en utilisant *if ... else* et ou *try ... catch* . Comment devrions-nous décider quelle version est la meilleure?

Le premier critère est la lisibilité. Bien que la lisibilité soit difficile à quantifier objectivement, la plupart des programmeurs conviendraient que la signification essentielle de la première version est plus facile à discerner. En effet, pour bien comprendre la seconde forme, vous devez comprendre qu'une `NullPointerException` ne peut pas être lancée par les méthodes `Math.min` ou `String.substring` .

Le deuxième critère est l'efficacité. Dans les versions de Java antérieures à Java 8, la deuxième version est significativement plus lente que la première version. En particulier, la construction d'un objet d'exception implique la capture et l'enregistrement des stackframes, au cas où la trace de

pile serait requise.

D'autre part, il existe de nombreuses situations où l'utilisation des exceptions est plus lisible, plus efficace et (parfois) plus correcte que l'utilisation d'un code conditionnel pour gérer des événements "exceptionnels". En effet, il existe de rares situations où il est nécessaire de les utiliser pour des événements "non exceptionnels"; c'est-à-dire des événements relativement fréquents. Pour ces derniers, il convient de chercher des moyens de réduire les frais généraux liés à la création d'objets d'exception.

Piège - Empilement excessif ou inapproprié

Une des choses les plus ennuyeuses que les programmeurs puissent faire est de disperser les appels à `printStackTrace()` dans tout leur code.

Le problème est que `printStackTrace()` va écrire le stacktrace sur la sortie standard.

- Pour une application destinée aux utilisateurs finaux qui ne sont pas des programmeurs Java, un stacktrace est, au mieux, non informatif et au pire alarmant.
- Pour une application côté serveur, il y a des chances que personne ne regarde la sortie standard.

Une meilleure idée est de ne pas appeler directement `printStackTrace` ou, si vous l'appelez, de le faire de manière à ce que la trace de la pile soit écrite dans un fichier journal ou un fichier d'erreur plutôt que sur la console de l'utilisateur final.

Pour ce faire, vous pouvez utiliser une structure de journalisation et transmettre l'objet exception en tant que paramètre de l'événement de journal. Cependant, même l'enregistrement de l'exception peut être nuisible si elle est effectuée de manière abusive. Considérer ce qui suit:

```
public void method1() throws SomeException {
    try {
        method2();
        // Do something
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method1", ex);
        throw ex;
    }
}

public void method2() throws SomeException {
    try {
        // Do something else
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method2", ex);
        throw ex;
    }
}
```

Si l'exception est `method2` dans `method2` , vous êtes susceptible de voir deux copies de la même trace dans le fichier journal, correspondant au même échec.

En bref, consignez l'exception ou relancez-la (éventuellement avec une autre exception). Ne faites pas les deux.

Piège - Sous-classement direct «Throwable»

`Throwable` a deux sous-classes directes, `Exception` et `Error` . Bien qu'il soit possible de créer une nouvelle classe qui étend `Throwable` directement, cela est déconseillé, car de nombreuses applications supposent que seules les `Exception` et les `Error` existent.

Plus `Throwable` , il n'y a aucun avantage pratique à sous- `Throwable` directement `Throwable` , car la classe résultante est en fait simplement une exception vérifiée. L' `Exception` sous- `Exception` entraînera plutôt le même comportement, mais traduira plus clairement votre intention.

Lire Pièges Java - Utilisation des exceptions en ligne:
https://riptutorial.com/fr/java/topic/5381/pieges-java---utilisation-des-exceptions

Chapitre 141: Pièges Java communs

Introduction

Cette rubrique présente certaines des erreurs courantes commises par les débutants en Java.

Cela inclut toutes les erreurs courantes dans l'utilisation du langage Java ou la compréhension de l'environnement d'exécution.

Les erreurs associées à des API spécifiques peuvent être décrites dans des rubriques spécifiques à ces API. Les cordes sont un cas particulier; ils sont couverts dans la spécification de langage Java. Les détails autres que les erreurs courantes peuvent être décrits dans cette rubrique sur les chaînes .

Examples

Piège: utiliser == pour comparer des objets d'emballage primitifs tels que Entier

(Ce piège s'applique également à tous les types d'emballages primitifs, mais nous allons l'illustrer pour `Integer` et `int` .)

Lorsque vous travaillez avec des objets `Integer` , il est tentant d'utiliser `==` pour comparer les valeurs, car c'est ce que vous feriez avec les valeurs `int` . Et dans certains cas, cela semble fonctionner:

```
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
```

Ici, nous avons créé deux objets `Integer` avec la valeur `1` et nous les avons `Integer` (dans ce cas, nous en avons créé un à partir d'un `String` et un à partir d'un littéral `int` . Il existe d'autres alternatives). En outre, nous observons que les deux méthodes de comparaison (`==` et `equals`) produisent toutes deux une valeur `true` .

Ce comportement change lorsque nous choisissons des valeurs différentes:

```
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
```

Dans ce cas, seule la comparaison `equals` donne le résultat correct.

La raison de cette différence de comportement est que la machine virtuelle Java conserve un cache d'objets `Integer` compris entre -128 et 127. (La valeur supérieure peut être remplacée par la propriété système "java.lang.Integer.IntegerCache.high" ou la propriété Argument JVM "-XX: AutoBoxCacheMax = size"). Pour les valeurs de cette plage, `Integer.valueOf()` retournera la valeur mise en cache plutôt que d'en créer une nouvelle.

Ainsi, dans le premier exemple, les `Integer.valueOf(1)` et `Integer.valueOf("1")` renvoyé la même instance `Integer` cache. En revanche, dans le deuxième exemple, `Integer.valueOf(1000)` et `Integer.valueOf("1000")` tous deux créé et renvoyé de nouveaux objets `Integer` .

L'opérateur `==` pour les types de référence teste l'égalité de référence (c'est-à-dire le même objet). Par conséquent, dans le premier exemple, `int1_1 == int1_2` est `true` car les références sont les mêmes. Dans le deuxième exemple, `int2_1 == int2_2` est faux car les références sont différentes.

Piège: oublier les ressources gratuites

Chaque fois qu'un programme ouvre une ressource, telle qu'une connexion de fichier ou de réseau, il est important de libérer la ressource une fois que vous l'utilisez. Des précautions similaires devraient être prises si une exception devait être levée pendant des opérations sur de telles ressources. On pourrait soutenir que `FileInputStream` a un finaliseur qui appelle la méthode `close()` sur un événement de récupération de place; Cependant, comme nous ne pouvons pas être sûrs du démarrage d'un cycle de récupération de place, le flux d'entrée peut consommer des ressources informatiques pour une durée indéterminée. La ressource doit être fermée dans une `finally` partie d'un bloc 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();
        }
    }
}
```

Depuis Java 7, Java 7 contient une instruction particulièrement utile, particulièrement pour ce cas, appelée 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'instruction *try-with-resources* peut être utilisée avec n'importe quel objet qui implémente l'interface `Closeable` ou `AutoCloseable` . Il s'assure que chaque ressource est fermée à la fin de l'instruction. La différence entre les deux interfaces est que la méthode `close()` de `Closeable` lance une `Closeable IOException` qui doit être gérée d'une certaine manière.

Dans les cas où la ressource a déjà été ouverte mais doit être fermée en toute sécurité après l'utilisation, on peut l'attribuer à une variable locale à l'intérieur de try-with-resources.

Java SE 7

```
private static void printFileJava7(InputStream extResource) throws IOException {
    try (InputStream input = extResource) {
        ... //access resource
    }
}
```

La variable de ressource locale créée dans le constructeur try-with-resources est effectivement finale.

Piège: fuites de mémoire

Java gère la mémoire automatiquement. Vous n'êtes pas obligé de libérer de la mémoire manuellement. La mémoire d'un objet sur le tas peut être libérée par un garbage collector lorsque l'objet n'est plus *accessible* par un thread en direct.

Cependant, vous pouvez empêcher la libération de la mémoire, en permettant aux objets d'être accessibles qui ne sont plus nécessaires. Que vous appeliez cela une fuite de mémoire ou un empaquetage de mémoire, le résultat est le même: une augmentation inutile de la mémoire allouée.

Les fuites de mémoire dans Java peuvent se produire de différentes manières, mais la raison la plus courante est la référence permanente aux objets, car le ramasse-miettes ne peut pas supprimer les objets du tas alors qu'il y a encore des références.

Champs statiques

On peut créer une telle référence en définissant la classe avec un champ `static` contenant une collection d'objets et en oubliant de définir ce champ `static` sur `null` après que la collection ne soit plus nécessaire. `static` champs `static` sont considérés comme des racines GC et ne sont jamais collectés. Un autre problème concerne les fuites dans la mémoire non-tas lorsque JNI est utilisé.

Fuite de classloader

De loin, le type de fuite de mémoire le plus insidieux est la fuite du chargeur de classe . Un

classloader contient une référence à chaque classe qu'il a chargée, et chaque classe contient une référence à son classloader. Chaque objet contient également une référence à sa classe. Par conséquent, même si un *seul* objet d'une classe chargée par un chargeur de classe n'est pas un déchet, aucune classe chargée par ce chargeur de classes ne peut être collectée. Comme chaque classe fait également référence à ses champs statiques, ils ne peuvent pas non plus être collectés.

Fuite d' accumulation L'exemple de fuite d'accumulation pourrait ressembler à ceci:

```
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();
}
```

Cet exemple crée deux tâches planifiées. La première tâche prend le dernier numéro d'un `numbers` appelé deque et, si le nombre est divisible par 51, elle imprime le nombre et la taille du deque. La deuxième tâche met des chiffres dans le deque. Les deux tâches sont planifiées à un taux fixe et elles s'exécutent toutes les 10 ms.

Si le code est exécuté, vous verrez que la taille de la police augmente en permanence. Cela entraînera éventuellement le remplissage de deque avec des objets consommant toute la mémoire de tas disponible.

Pour éviter cela tout en préservant la sémantique de ce programme, nous pouvons utiliser une méthode différente pour prendre des nombres à partir de deque: `pollLast` . Contrairement à la méthode `peekLast` , `pollLast` renvoie l'élément et le supprime de deque tandis que `peekLast` ne renvoie que le dernier élément.

Piège: utiliser == pour comparer des chaînes

Une erreur courante pour les débutants Java est d'utiliser l'opérateur `==` pour tester si deux chaînes sont égales. Par exemple:

```
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?");
            }
        }
    }
}
```

Le programme ci-dessus est censé tester le premier argument de la ligne de commande et imprimer différents messages lorsqu'il ne s'agit pas du mot "bonjour". Mais le problème est que cela ne fonctionnera pas. Ce programme produira "Êtes-vous grincheux aujourd'hui?" quel que soit le premier argument de la ligne de commande.

Dans ce cas particulier, la `String` "hello" est placée dans le pool de chaînes pendant que les arguments `String` [0] résident sur le tas. Cela signifie qu'il y a deux objets représentant le même littéral, chacun avec sa référence. Étant donné que `==` teste les références et non l'égalité réelle, la comparaison produira un faux la plupart du temps. Cela ne signifie pas qu'il le fera toujours.

Lorsque vous utilisez `==` pour tester des chaînes, ce que vous testez en réalité est si deux objets `String` sont le même objet Java. Malheureusement, cela ne signifie pas l'égalité des chaînes en Java. En fait, la méthode correcte pour tester les chaînes consiste à utiliser la méthode `equals(Object)` . Pour une paire de chaînes, nous voulons généralement tester si elles sont composées des mêmes caractères dans le même ordre.

```
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?");
            }
        }
    }
}
```

Mais en réalité, ça empire. Le problème est que `==` donnera la réponse attendue dans certaines circonstances. Par exemple

```
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");
        }
    }
}
```

Il est intéressant de noter que cela affichera "identique", même si nous testons les chaînes dans le

mauvais sens. Pourquoi donc? Parce que la spécification de langage Java (Section 3.10.5: Littéraux de chaîne) stipule que deux chaînes >> littérales << composées des mêmes caractères seront effectivement représentées par le même objet Java. Par conséquent, le test `==` sera vrai pour les littéraux égaux. (Les littéraux de chaîne sont "internés" et ajoutés à un "pool de chaînes" partagé lorsque votre code est chargé, mais il s'agit en fait d'un détail d'implémentation.)

Pour ajouter à la confusion, la spécification de langage Java stipule également que lorsque vous avez une expression constante de compilation qui concatène deux littéraux de chaîne, cela équivaut à un seul littéral. Ainsi:

```
    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");
        }
    }
}
```

Cela produira "1. mêmes" et "2. différents". Dans le premier cas, l'expression `+` est évaluée au moment de la compilation et nous comparons un objet `String` avec lui-même. Dans le second cas, il est évalué à l'exécution et nous comparons deux objets `String` différents

En résumé, l'utilisation de `==` pour tester des chaînes en Java est presque toujours incorrecte, mais il n'est pas certain que la réponse soit incorrecte.

Piège: tester un fichier avant d'essayer de l'ouvrir.

Certaines personnes recommandent d'appliquer divers tests à un fichier avant de tenter de l'ouvrir pour fournir de meilleurs diagnostics ou pour éviter de traiter des exceptions. Par exemple, cette méthode tente de vérifier si le `path` correspond à un fichier lisible:

```
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;
}
```

Vous pourriez utiliser la méthode ci-dessus comme ceci:

```
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.
}
```

Le premier problème réside dans la signature de `FileInputStream(File)` car le compilateur insistera toujours pour intercepter `IOException` ici ou plus haut dans la pile.

Le second problème est que les vérifications effectuées par `getValidatedFile` ne garantissent pas la `FileInputStream` .

- Conditions de course: un autre thread ou un processus séparé peut renommer le fichier, supprimer le fichier ou supprimer l'accès en lecture après le retour de `getValidatedFile` . Cela conduirait à une `IOException` "simple" sans le message personnalisé.
- Il existe des cas marginaux non couverts par ces tests. Par exemple, sur un système avec SELinux en mode "Forçage", une tentative de lecture d'un fichier peut échouer malgré le retour de `true canRead()` .

Le troisième problème est que les tests sont inefficaces. Par exemple, le `exists` , `isFile` et `canRead` appels feront chacun un syscall pour effectuer le contrôle nécessaire. Un autre appel système est alors effectué pour ouvrir le fichier, qui répète les mêmes vérifications dans les coulisses.

En bref, les méthodes telles que `getValidatedFile` sont erronées. Il est préférable d'essayer d'ouvrir le fichier et de gérer l'exception:

```
try (InputStream is = new FileInputStream("somefile")) {
    // Read data etc.
} catch (IOException ex) {
    System.err.println("IO Error processing 'somefile': " + ex.getMessage());
    return;
}
```

Si vous voulez distinguer les erreurs IO générées lors de l'ouverture et de la lecture, vous pouvez utiliser un try / catch imbriqué. Si vous voulez produire de meilleurs diagnostics pour les échecs ouverts, vous pouvez effectuer les `exists` , `isFile` et `canRead` contrôles dans le gestionnaire.

Piège: penser les variables comme des objets

Aucune variable Java ne représente un objet.

```
String foo;   // NOT AN OBJECT
```

Aucun tableau Java ne contient non plus d'objets.

```
String bar[] = new String[100];  // No member is an object.
```

Si vous pensez à tort que les variables sont des objets, le comportement réel du langage Java vous surprendra.

- Pour les variables Java qui ont un type primitif (tel que `int` ou `float`), la variable contient une copie de la valeur. Toutes les copies d'une valeur primitive sont indiscernables. c'est-à-dire qu'il n'y a qu'une seule valeur `int` pour le numéro un. Les valeurs primitives ne sont pas des objets et ne se comportent pas comme des objets.
- Pour les variables Java qui ont un type de référence (soit un type de classe, soit un type de tableau), la variable contient une référence. Toutes les copies d'une référence sont indiscernables. Les références peuvent pointer sur des objets ou être `null` ce qui signifie qu'elles ne pointent vers aucun objet. Cependant, ils ne sont pas des objets et ils ne se comportent pas comme des objets.

Les variables ne sont pas des objets dans les deux cas et elles ne contiennent aucun objet dans les deux cas. Ils peuvent contenir des *références à des objets* , mais cela dit quelque chose de différent.

Exemple classe

Les exemples suivants utilisent cette classe, qui représente un point dans un espace 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;
   }
}
```

Une instance de cette classe est un objet qui a deux champs `x` et `y` qui ont le type `int` .

Nous pouvons avoir plusieurs instances de la classe `MutableLocation` . Certains représenteront les mêmes emplacements dans un espace 2D; c'est-à-dire que les valeurs respectives de `x` et `y` correspondent. D'autres représenteront des endroits différents.

Plusieurs variables peuvent pointer vers le même objet

```
MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);
```

Dans ce qui précède, nous avons déclaré trois variables `here` , `there` et `elsewhere` pouvant contenir des références aux objets `MutableLocation` .

Si vous pensez (à tort) que ces variables sont des objets, vous risquez de ne pas les interpréter comme suit:

1. Copiez l'emplacement "[1, 2]" `here`
2. Copiez l'emplacement « [1, 2] » pour `there`
3. Copier l'emplacement "[1, 2]" vers `elsewhere`

À partir de cela, vous pouvez en déduire que nous avons trois objets indépendants dans les trois variables. En fait, il n'y a *que deux objets créés* par ce qui précède. Les variables `here` et `there` se réfèrent effectivement au même objet.

Nous pouvons le démontrer. En supposant les déclarations de variable comme ci-dessus:

```
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);
```

Cela va afficher les éléments suivants:

```
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
```

Nous avons assigné une nouvelle valeur à `here.x` et cela a changé la valeur que nous voyons `there.x` Ils font référence au même objet. Mais la valeur que nous voyons via `elsewhere.x` n'a pas changé, donc `elsewhere` doit faire référence à un objet différent.

Si une variable était un objet, l'affectation `here.x = 42` ne changerait pas `there.x` . `there.x`

L'opérateur d'égalité ne teste PAS que deux objets sont égaux

L'application de l'opérateur d'égalité (`==`) pour référencer les valeurs teste si les valeurs font référence au même objet. Il ne teste *pas* si deux objets (différents) sont "égaux" au sens intuitif.

```
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");
}
```

Ceci imprimera "ici est là", mais il n'imprimera pas "ici est ailleurs". (Les références `here` et `elsewhere` concernent deux objets distincts.)

En revanche, si nous appelons la méthode `equals(Object)` que nous avons implémentée ci-dessus, nous allons tester si deux instances de `MutableLocation` ont un emplacement égal.

```
if (here.equals(there)) {
    System.out.println("here equals there");
}
if (here.equals(elsewhere)) {
    System.out.println("here equals elsewhere");
}
```

Cela imprimera les deux messages. En particulier, `here.equals(elsewhere)` renvoie `true` car les critères sémantiques que nous avons choisis pour l'égalité de deux objets `MutableLocation` ont été satisfaits.

Les appels de méthode ne transmettent PAS d'objets du tout

Les appels de méthode Java utilisent la *valeur par défaut* [1] pour transmettre les arguments et renvoyer un résultat.

Lorsque vous transmettez une valeur de référence à une méthode, vous transmettez en réalité une référence à un objet *par valeur* , ce qui signifie qu'il crée une copie de la référence d'objet.

Tant que les deux références d'objet pointent toujours vers le même objet, vous pouvez modifier cet objet à partir de l'une ou l'autre référence, ce qui est source de confusion pour certaines.

Cependant, vous ne passez *pas* d'objet par référence [2] . La distinction est que si la copie de référence d'objet est modifiée pour pointer vers un autre objet, la référence d'objet d'origine pointe toujours vers l'objet d'origine.

```
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".
}
```

Vous ne transmettez pas non plus une copie de l'objet.

```
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 - Dans les langages comme Python et Ruby, le terme "pass by sharing" est préféré pour "passer par valeur" d'un objet / référence.

2 - Le terme "passer par référence" ou "appeler par référence" a une signification très spécifique dans la terminologie des langages de programmation. En effet, cela signifie que vous passez l'adresse *d'une variable ou d'un élément de tableau* , de sorte que lorsque la méthode appelée assigne une nouvelle valeur à l'argument formel, elle modifie la valeur de la variable d'origine. Java ne supporte pas cela. Pour une description plus complète des différents mécanismes de transmission des paramètres, reportez-vous à https://en.wikipedia.org/wiki/Evaluation_strategy .

Piège: combiner affectation et effets secondaires

Occasionnellement, nous voyons des questions sur StackOverflow Java (et des questions sur C ou C ++) qui demandent ce qui suit:

```
i += a[i++] + b[i--];
```

évalue à ... pour certains états initiaux connus de `i` , `a` et `b` .

En général:

- pour Java la réponse est toujours spécifiée [1] , mais non évidente et souvent difficile à comprendre
- pour C et C ++, la réponse est souvent non spécifiée.

Ces exemples sont souvent utilisés dans les examens ou les entretiens d'embauche pour tenter de voir si l'étudiant ou la personne interrogée comprend comment l'évaluation de l'expression fonctionne réellement dans le langage de programmation Java. Cela est sans doute légitime en tant que "test de connaissance", mais cela ne signifie pas que vous devriez le faire dans un vrai programme.

Pour illustrer cela, l'exemple apparemment simple suivant est apparu à quelques reprises dans les questions de StackOverflow (comme celle-ci). Dans certains cas, il apparaît comme une véritable erreur dans le code de quelqu'un.

```
int a = 1;
a = a++;
System.out.println(a);    // What does this print.
```

La plupart des programmeurs (y compris les experts Java) lisant ces déclarations *rapidement* diraient qu'ils produisent `2` . En fait, il produit `1` . Pour une explication détaillée des raisons, veuillez lire cette réponse .

Cependant , la véritable plats à emporter à partir de ce et des exemples similaires est que *toute* déclaration Java qui affecte à la *fois et* les effets secondaires de la même variable va être *au mieux* difficile à comprendre, et *au pire* carrément trompeur. Vous devriez éviter d'écrire du code comme celui-ci.

---

1 - problèmes potentiels modulo avec le modèle de mémoire Java si les variables ou objets sont visibles par les autres threads.

Piège: ne pas comprendre que String est une classe immuable

Les nouveaux programmeurs Java oublient souvent, ou échouent à comprendre, que la classe Java `String` est immuable. Cela conduit à des problèmes comme celui de l'exemple suivant:

```
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();
    }
}
```

Le code ci-dessus est censé imprimer les arguments de ligne de commande en majuscule. Malheureusement, cela ne fonctionne pas, la casse des arguments n'est pas modifiée. Le problème est cette déclaration:

```
s.toUpperCase();
```

Vous pourriez penser que l'appel à `toUpperCase()` va changer `s` en une chaîne majuscule. Ce n'est pas le cas. Ça ne peut pas! `String` objets `String` sont immuables. Ils ne peuvent pas être changés.

En réalité, la méthode `toUpperCase()` *renvoie* un objet `String` qui est une version majuscule de la `String` laquelle vous l'appelez. Ce sera probablement un nouvel objet `String` , mais si `s` était déjà en majuscule, le résultat pourrait être la chaîne existante.

Donc, pour utiliser cette méthode efficacement, vous devez utiliser l'objet renvoyé par l'appel de la méthode. par exemple:

```
s = s.toUpperCase();
```

En fait, la règle "strings never change" s'applique à toutes les méthodes `String` . Si vous vous en souvenez, vous pouvez éviter toute une catégorie d'erreurs du débutant.

Lire Pièges Java communs en ligne: https://riptutorial.com/fr/java/topic/4388/pieges-java-communs

Chapitre 142: Plans

Introduction

L' interface java.util.Map représente un mappage entre les clés et leurs valeurs. Une carte ne peut pas contenir de clés en double; et chaque clé peut correspondre à au plus une valeur.

Comme `Map` est une interface, vous devez instancier une implémentation concrète de cette interface pour pouvoir l'utiliser. Il y a plusieurs implémentations `Map` , et les plus utilisées sont `java.util.HashMap` et `java.util.TreeMap`

Remarques

Une *carte* est un objet qui stocke des *clés* avec une *valeur* associée pour chaque clé. Une clé et sa valeur sont parfois appelées une *paire clé / valeur* ou une *entrée* . Les cartes fournissent généralement ces fonctionnalités:

- Les données sont stockées dans la carte par paires clé / valeur.
- La carte peut contenir une seule entrée pour une clé particulière. Si une carte contient une entrée avec une clé particulière et que vous essayez de stocker une deuxième entrée avec la même clé, la deuxième entrée remplacera la première. En d'autres termes, cela changera la valeur associée à la clé.
- Les cartes fournissent des opérations rapides pour vérifier si une clé existe dans la carte, pour récupérer la valeur associée à une clé et pour supprimer une paire clé / valeur.

L'implémentation de carte la plus couramment utilisée est HashMap . Cela fonctionne bien avec des clés qui sont des chaînes ou des nombres.

Les cartes simples telles que HashMap ne sont pas ordonnées. Les itérations sur les paires clé / valeur peuvent renvoyer des entrées individuelles dans n'importe quel ordre. Si vous devez parcourir les entrées de carte de manière contrôlée, vous devez regarder les éléments suivants:

- Les cartes triées, telles que TreeMap , parcourent les clés dans leur ordre naturel (ou dans un ordre que vous pouvez spécifier, en fournissant un comparateur). Par exemple, une carte triée utilisant des nombres comme clés devrait pouvoir parcourir ses entrées dans l'ordre numérique.

- LinkedHashMap permet de parcourir les entrées dans l'ordre dans lequel elles ont été insérées dans la carte ou dans l'ordre d'accès le plus récent.

Examples

Ajouter un élément

1. Une addition

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
System.out.println(map.get(1));
```

Sortie: `First element.`

2. Passer outre

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
map.put(1, "New element.");
System.out.println(map.get(1));
```

Sortie: `New element.`

`HashMap` est utilisé comme exemple. D'autres implémentations qui implémentent l'interface `Map` peuvent également être utilisées.

Ajouter plusieurs éléments

On peut utiliser `V put(K key,V value)` :

> Associe la valeur spécifiée à la clé spécifiée dans cette carte (opération facultative). Si la carte contenait précédemment un mappage pour la clé, l'ancienne valeur est remplacée par la valeur spécifiée.

```
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());
```

Sortie:

```
3
```

Pour ajouter de nombreux éléments, vous pouvez utiliser une classe interne comme celle-ci:

```
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");
}};
```

Gardez à l'esprit que la création d'une classe interne anonyme n'est pas toujours efficace et peut entraîner des fuites de mémoire. Dans la mesure du possible, utilisez plutôt un bloc d'initialisation:

```
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'exemple ci-dessus rend la carte statique. Il peut également être utilisé dans un contexte non statique en supprimant toutes les occurrences de `static` .

En plus de cela, la plupart des implémentations prennent en charge `putAll` , qui peut ajouter toutes les entrées d'une carte à une autre, comme ceci:

```
another.putAll(one);
```

Utilisation des méthodes de mappage par défaut de Java 8

Exemples d'utilisation des méthodes par défaut introduites dans l'interface Java 8 dans Map

1. Utiliser **getOrDefault**

Renvoie la valeur associée à la clé ou, si la clé n'est pas présente, renvoie la valeur par défaut

```
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. Utilisation de **forEach**

Permet d'effectuer l'opération spécifiée dans l'action sur chaque entrée de carte

```
  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. Utiliser **replaceAll**

Remplacera par new-value uniquement si la clé est présente

```
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. Utiliser **putIfAbsent**

La paire valeur-clé est ajoutée à la carte si la clé n'est pas présente ou associée à la valeur 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. En utilisant **remove**

   Supprime la clé uniquement si elle est associée à la valeur donnée

   ```
   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. En utilisant **replace**

   Si la clé est présente, la valeur est remplacée par une nouvelle valeur. Si la clé n'est pas présente, ne fait rien.

```
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. Utiliser **computeIfAbsent**

Cette méthode ajoute une entrée dans la carte. la clé est spécifiée dans la fonction et la valeur est le résultat de l'application de la fonction de mappage

```
    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. Utiliser **computeIfPresent**

Cette méthode ajoute une entrée ou modifie une entrée existante dans la carte. Ne fait rien si une entrée avec cette clé n'est pas présente

```
    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. Utiliser le **calcul**

Cette méthode remplace la valeur d'une clé par la valeur nouvellement calculée

```
    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. Utiliser la **fusion**

Ajoute la paire clé-valeur à la carte, si la clé n'est pas présente ou que la valeur de la clé est nulle Remplace la valeur par la nouvelle valeur calculée, si la clé est présente La clé est supprimée si la nouvelle valeur calculée est nulle

```
    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}
```

Effacer la carte

```
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
```

En parcourant le contenu d'une carte

Les cartes fournissent des méthodes qui vous permettent d'accéder aux clés, aux valeurs ou aux paires clé-valeur de la carte en tant que collections. Vous pouvez parcourir ces collections. Compte tenu de la carte suivante par exemple:

```
Map<String, Integer> repMap = new HashMap<>();
repMap.put("Jon Skeet", 927_654);
repMap.put("BalusC", 708_826);
repMap.put("Darin Dimitrov", 715_567);
```

Itération à travers les clés de la carte:

```
for (String key : repMap.keySet()) {
    System.out.println(key);
}
```

Impressions:

Darin Dimitrov
Jon Skeet
BalusC

`keySet()` fournit les clés de la carte sous la forme d'un `Set` . `Set` est utilisé car les clés ne peuvent pas contenir de valeurs en double. Itérer à travers l'ensemble donne chaque clé à tour de rôle. Les HashMaps ne sont pas commandés. Dans cet exemple, les clés peuvent être renvoyées dans n'importe quel ordre.

Itération à travers les valeurs de la carte:

```
for (Integer value : repMap.values()) {
    System.out.println(value);
}
```

Impressions:

715567
927654
708826

`values()` renvoie les valeurs de la carte en tant que `Collection` . Itérer à travers la collection donne chaque valeur à tour de rôle. Encore une fois, les valeurs peuvent être renvoyées dans n'importe

quel ordre.

Itérer à travers les clés et les valeurs ensemble

```
for (Map.Entry<String, Integer> entry : repMap.entrySet()) {
    System.out.printf("%s = %d\n", entry.getKey(), entry.getValue());
}
```

Impressions:

> Darin Dimitrov = 715567
> Jon Skeet = 927654
> BalusC = 708826

`entrySet()` renvoie une collection d'objets `Map.Entry` . Map.Entry donne accès à la clé et à la valeur pour chaque entrée.

Fusion, combinaison et composition de cartes

Utilisez `putAll` pour mettre chaque membre d'une carte dans une autre. Les clés déjà présentes dans la carte auront leurs valeurs correspondantes écrasées.

```
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)
```

Cela donne la cartographie suivante en `numbers` :

```
"One" -> 1
"Two" -> 2
"Three" -> 4 //old value 3 was overwritten by new value 4
```

Si vous souhaitez combiner des valeurs au lieu de les écraser, vous pouvez utiliser `Map.merge` , ajouté à Java 8, qui utilise une fonction `BiFunction` fournie par l' `BiFunction` pour fusionner les valeurs des clés en double. `merge` opère sur des clés et des valeurs individuelles, vous devrez donc utiliser une boucle ou `Map.forEach` . Ici, nous concaténons des chaînes pour des clés en double:

```
for (Map.Entry<String, Integer> e : other_numbers.entrySet())
    numbers.merge(e.getKey(), e.getValue(), Integer::sum);
//or instead of the above loop
other_numbers.forEach((k, v) -> numbers.merge(k, v, Integer::sum));
```

Si vous souhaitez appliquer la contrainte, il n'y a pas de clés en double, vous pouvez utiliser une fonction de fusion qui renvoie une `AssertionError` :

```
mapA.forEach((k, v) ->
    mapB.merge(k, v, (v1, v2) ->
        {throw new AssertionError("duplicate values for key: "+k);}));
```

Composer la carte <X, Y> et la carte <Y, Z> pour obtenir la carte <X, Z>

Si vous voulez composer deux mappages, vous pouvez le faire comme suit

```
    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)));
```

Cela donne la cartographie suivante

```
    "key1" -> 1.0
    "key2" -> 2.0
    "key3" -> 3.0
```

Vérifier si la clé existe

```
Map<String, String> num = new HashMap<>();
num.put("one", "first");

if (num.containsKey("one")) {
    System.out.println(num.get("one")); // => first
}
```

Les cartes peuvent contenir des valeurs nulles

Pour les cartes, il faut faire preuve de prudence pour ne pas confondre "contenant une clé" avec "avoir une valeur". Par exemple, `HashMap` s peut contenir null, ce qui signifie que le comportement suivant est parfaitement normal:

```
Map<String, String> map = new HashMap<>();
map.put("one", null);
if (map.containsKey("one")) {
    System.out.println("This prints !"); // This line is reached
```

```
}
if (map.get("one") != null) {
    System.out.println("This is never reached !"); // This line is never reached
}
```

Plus formellement, il n'y a aucune garantie que `map.contains(key) <=> map.get(key)!=null`

Itérer efficacement les entrées de carte

Cette section fournit des codes et des tests d'évaluation pour dix implémentations d'exemples uniques qui parcourent les entrées d'une `Map<Integer, Integer>` et génèrent la somme des valeurs `Integer` . Tous les exemples ont une complexité algorithmique de `Θ(n)` , cependant, les tests de performances sont toujours utiles pour fournir des informations sur les implémentations les plus efficaces dans un environnement "réel".

1. Implémentation utilisant **`Iterator`** avec **`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. Implémentation à l'aide **`for`** avec **`Map.Entry`**

```
    for (Map.Entry<Integer, Integer> pair : map.entrySet()) {
        sum += pair.getKey() + pair.getValue();
    }
```

3. Implémentation à l'aide de **`Map.forEach`** (Java 8+)

```
    map.forEach((k, v) -> sum[0] += k + v);
```

4. Implémentation à l'aide de **`Map.keySet`** avec **`for`**

```
    for (Integer key : map.keySet()) {
        sum += key + map.get(key);
    }
```

5. Implémentation à l'aide de **`Map.keySet`** avec **`Iterator`**

```
    Iterator<Integer> it = map.keySet().iterator();
    while (it.hasNext()) {
        Integer key = it.next();
        sum += key + map.get(key);
    }
```

6. Implémentation utilisant **`for`** avec **`Iterator`** et **`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. Implémentation à l'aide de `Stream.forEach` (Java 8+)

```
map.entrySet().stream().forEach(e -> sum += e.getKey() + e.getValue());
```

8. Implémentation à l'aide de `Stream.forEach` avec `Stream.parallel` (Java 8+)

```
map.entrySet()
   .stream()
   .parallel()
   .forEach(e -> sum += e.getKey() + e.getValue());
```

9. Implémentation à l'aide d' `IterableMap` des collections Apache

```
MapIterator<Integer, Integer> mit = iterableMap.mapIterator();
while (mit.hasNext()) {
    sum += mit.next() + it.getValue();
}
```

10. Implémentation à l'aide de `MutableMap` des collections Eclipse

```
 mutableMap.forEachKeyValue((key, value) -> {
     sum += key + value;
 });
```

Tests de performance (*code disponible sur Github*)
Environnement de test: Windows 8.1 64 bits, Intel i7-4790 3,60 GHz, 16 Go

1. Performance moyenne de 10 essais (100 éléments) Meilleur: 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. Performance moyenne de 10 essais (10000 éléments) Meilleur: 37.606 ± 0.790 µs / op

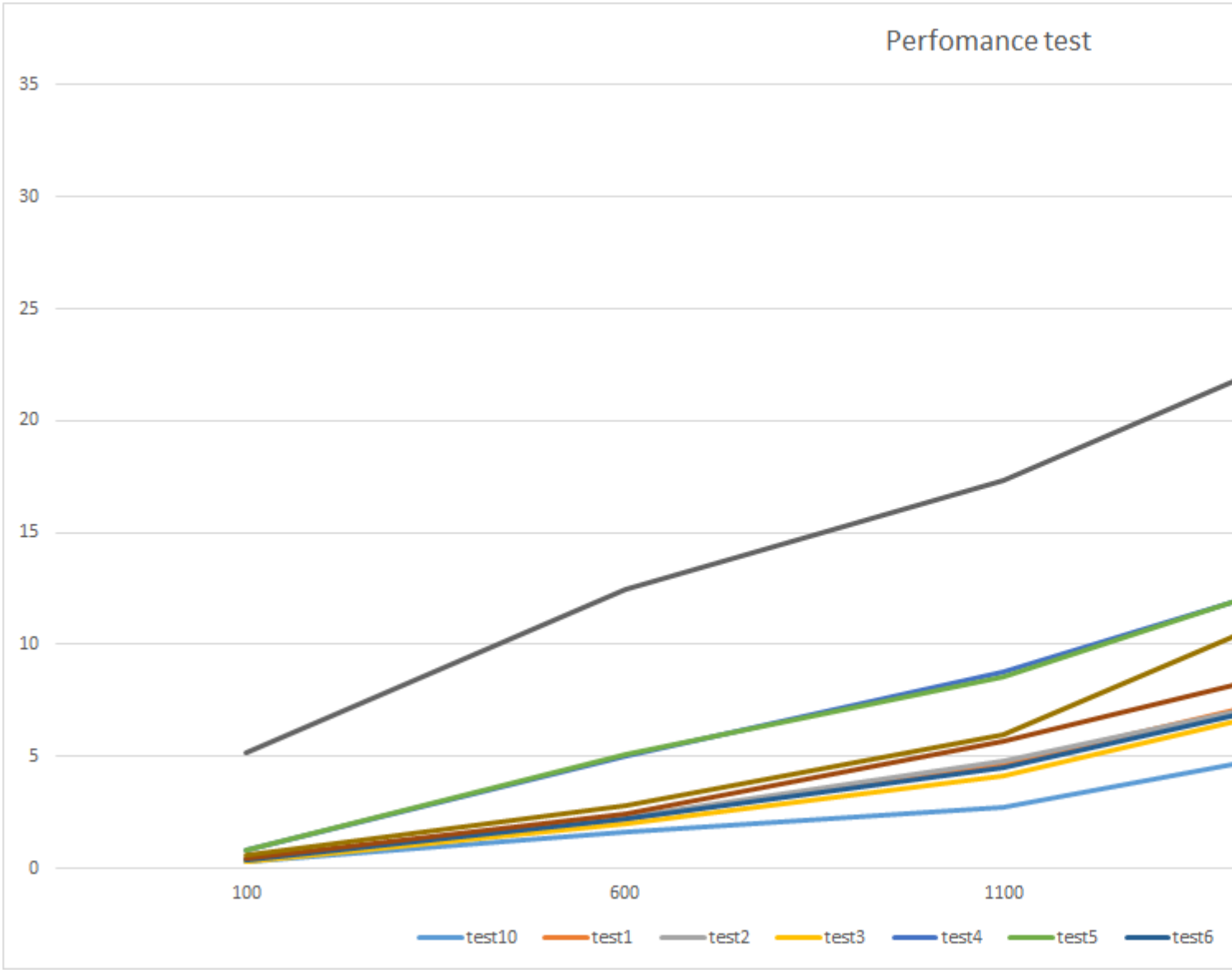
```
Benchmark                          Score     Error  Units
test10_UsingEclipseMutableMap      37606  ±    790  ns/op
test3_UsingForEachAndJava8         50368  ±    887  ns/op
test6_UsingForAndIterator          50332  ±    507  ns/op
```

```
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. Rendement moyen de 10 essais (100 000 éléments) Meilleur: 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. Une comparaison des variations de performance en fonction de la taille de la carte




```
      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
```

Utiliser un objet personnalisé comme clé

Avant d'utiliser votre propre objet comme clé, vous devez remplacer la méthode hashCode () et equals () de votre objet.

Dans le cas simple, vous auriez quelque chose comme:

```
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` décidera quel seau hachage la clé appartient et `equals` à décidera quel objet dans le seau de hachage.

Sans cette méthode, la référence de votre objet sera utilisée pour la comparaison ci-dessus, ce qui ne fonctionnera que si vous utilisez la même référence d'objet à chaque fois.

Utilisation de HashMap

HashMap est une implémentation de l'interface Map qui fournit une structure de données pour stocker des données dans des paires Key-Value.

1. Déclarer HashMap

```
Map<KeyType, ValueType> myMap = new HashMap<KeyType, ValueType>();
```

KeyType et ValueType doivent être des types valides en Java, tels que - String, Integer, Float ou toute classe personnalisée comme Employee, Student, etc.

Par exemple: `Map<String,Integer> myMap = new HashMap<String,Integer>();`

2. Mettre des valeurs dans HashMap.

Pour mettre une valeur dans HashMap, nous devons appeler la méthode `put` sur l'objet HashMap en transmettant la clé et la valeur en tant que paramètres.

```
myMap.put("key1", 1);
myMap.put("key2", 2);
```

Si vous appelez la méthode put avec la clé qui existe déjà dans la carte, la méthode remplace sa valeur et renvoie l'ancienne valeur.

3. Obtenir des valeurs de HashMap.

Pour obtenir la valeur d'un HashMap, vous devez appeler la méthode `get` en transmettant la clé en tant que paramètre.

```
myMap.get("key1");    //return 1 (class Integer)
```

Si vous transmettez une clé qui n'existe pas dans HashMap, cette méthode renverra `null`

4. Vérifiez si la clé est dans la carte ou non.

```
myMap.containsKey(varKey);
```

5. Vérifiez si la valeur est dans la carte ou non.

```
myMap.containsValue(varValue);
```

Les méthodes ci-dessus `boolean` une valeur `boolean` true ou false si key, la valeur existe dans la map ou non.

Création et initialisation de cartes

introduction

`Maps` stocke les paires clé / valeur, chaque clé ayant une valeur associée. Étant donné une clé particulière, la carte peut rechercher la valeur associée très rapidement.

`Maps` , également appelées tableau associé, sont des objets qui stockent les données sous forme de clés et de valeurs. En Java, les cartes sont représentées à l'aide de l'interface Map, qui n'est pas une extension de l'interface de collecte.

- Voie 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);
```

- Way 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);
```

- Voie 3: -

```
Map<String, Object> map = new HashMap<String, Object>(){{
    put("name", "A");
    put("address", "Malviya-Nagar");
    put("city", "Jaipur");
}};
System.out.println(map);
```

- Voie 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);
```

- Chemin 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);
```

- Voie 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");
}
```

- Voie 7: - Créer une carte clé-valeur unique immuable.

```
  //Immutable single key-value map
  Map<String, String> singletonMap = Collections.singletonMap("key", "value");
```

Veuillez noter **qu'il est impossible de modifier une telle carte** .

Tout objet de modification de la carte entraînera une exception 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
```

Lire Plans en ligne: https://riptutorial.com/fr/java/topic/105/plans

Chapitre 143: Polymorphisme

Introduction

Le polymorphisme est l'un des principaux concepts de programmation orientée objet (OOP). Le mot polymorphisme était dérivé des mots grecs "poly" et "morphs". Poly signifie "beaucoup" et morphs signifie "formes" (nombreuses formes).

Il y a deux façons d'effectuer un polymorphisme. **Surcharge de méthode** et **remplacement de méthode** .

Remarques

`Interfaces` sont un autre moyen d'obtenir un polymorphisme en Java, hormis l'héritage basé sur les classes. Les interfaces définissent une liste de méthodes formant l'API du programme. Les classes doivent `implement` une `interface` en remplaçant toutes ses méthodes.

Examples

Surcharge de méthode

La surcharge de méthode , également appelée **surcharge de fonction** , est la capacité d'une classe à avoir plusieurs méthodes portant le même nom, à condition qu'elles diffèrent en nombre ou en type d'arguments.

Le compilateur vérifie la ***signature*** de la méthode pour la surcharge de la méthode.

La signature de la méthode consiste en trois choses -

1. Nom de la méthode
2. Nombre de paramètres
3. Types de paramètres

Si ces trois méthodes sont identiques pour deux méthodes d'une classe, alors le compilateur génère une **erreur de méthode en double** .

Ce type de polymorphisme est appelé polymorphisme *statique* ou *temps de compilation* car la méthode appropriée à appeler est décidée par le compilateur pendant la compilation à partir de la liste des arguments.

```
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));
    }

}
```

Cela se traduira par:

```
2
6
4.000000
```

Les méthodes surchargées peuvent être statiques ou non statiques. Cela n'affecte pas non plus la surcharge de la méthode.

```
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
    }
}
```

De plus, si vous modifiez le type de méthode de retour, nous ne pouvons pas l'obtenir en tant que surcharge de méthode.

```
public class Polymorph {
```

```
void methodOverloaded(){
    //No argument and No return type
}

int methodOverloaded(){
    //No argument and int return type
    return 0;
}
```

Dérogation de méthode

La substitution de méthode est la capacité des sous-types à redéfinir (outrepasser) le comportement de leurs sur-types.

En Java, cela se traduit par des sous-classes remplaçant les méthodes définies dans la super classe. En Java, toutes les variables non primitives sont en fait des `references` , qui s'apparentent à des pointeurs vers l'emplacement de l'objet réel en mémoire. Les `references` ont un seul type, qui est le type avec lequel elles ont été déclarées. Cependant, ils peuvent pointer vers un objet de leur type déclaré ou de l'un de ses sous-types.

Lorsqu'une méthode est appelée sur une `reference` , la **méthode** correspondante **de l'objet réel pointé est appelée** .

```
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
    }
}
```

Règles à garder à l'esprit

Pour remplacer une méthode dans la sous-classe, la méthode de substitution (c.-à-d. Celle de la sous-classe) **DOIT AVOIR** :

- même nom
- même type de retour en cas de primitives (une sous-classe est autorisée pour les classes, c'est ce qu'on appelle les types de retour covariants).
- même type et ordre de paramètres
- il ne peut lancer que les exceptions déclarées dans la clause throws de la méthode de la superclasse ou les exceptions qui sont des sous-classes des exceptions déclarées. Il peut également choisir de ne lancer aucune exception. Les noms des types de paramètres n'ont pas d'importance. Par exemple, void methodX (int i) est identique à void methodX (int k)
- Nous ne pouvons pas remplacer les méthodes finales ou statiques. Seule chose que nous ne pouvons faire que changer de corps de méthode.

Ajout de comportement en ajoutant des classes sans toucher au code existant

```
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");
    }
}
```

Explication

a) La classe `MakeThingsFly` peut fonctionner avec tout ce qui est de type `FlyingMachine` .

b) La méthode `letTheMachinesFly` fonctionne également sans aucun changement (!) lorsque vous ajoutez une nouvelle classe, par exemple `PropellerPlane` :

```
public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
        for (FlyingMachine flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
    }
}
```

C'est le pouvoir du polymorphisme. Vous pouvez implémenter le principe d' ouverture-fermeture avec celui - ci.

Fonctions virtuelles

Les méthodes virtuelles sont des méthodes en Java non statiques et sans le mot-clé Final en tête. Toutes les méthodes par défaut sont virtuelles en Java. Les méthodes virtuelles jouent un rôle important dans le polymorphisme, car les classes enfants de Java peuvent remplacer les méthodes de leurs classes parentes si la fonction remplacée est non statique et possède la même signature de méthode.

Il existe cependant des méthodes qui ne sont pas virtuelles. Par exemple, si la méthode est déclarée privée ou avec le mot clé final, la méthode n'est pas virtuelle.

Considérez l'exemple modifié suivant d'héritage avec les méthodes virtuelles de cette publication StackOverflow Comment les fonctions virtuelles fonctionnent-elles en C # et en Java? :

```
public class A{
    public void hello(){
        System.out.println("Hello");
    }

    public void boo(){
        System.out.println("Say boo");
```

```
    }
}

public class B extends A{
     public void hello(){
        System.out.println("No");
     }

    public void boo(){
        System.out.println("Say haha");

    }
}
```

Si nous invoquons la classe B et appelons hello () et boo (), nous obtiendrions "No" et "Say haha" comme résultat car B remplace les mêmes méthodes de A. Même si l'exemple ci-dessus est presque identique à la méthode outrepassant, il est important de comprendre que les méthodes de la classe A sont toutes, par défaut, virtuelles.

De plus, nous pouvons implémenter des méthodes virtuelles en utilisant le mot-clé abstract. Les méthodes déclarées avec le mot-clé "abstract" n'ont pas de définition de méthode, ce qui signifie que le corps de la méthode n'est pas encore implémenté. Prenons à nouveau l'exemple ci-dessus, sauf que la méthode boo () est déclarée abstraite:

```
public class A{
   public void hello(){
        System.out.println("Hello");
    }

    abstract void boo();
}

public class B extends A{
     public void hello(){
        System.out.println("No");
     }

    public void boo(){
        System.out.println("Say haha");

    }
}
```

Si nous invoquons boo () à partir de B, la sortie sera toujours "Say haha" car B hérite de la méthode abstraite boo () et rend la sortie boo () "Say haha".

Sources utilisées et lectures supplémentaires:

Comment fonctionnent les fonctions virtuelles en C # et en Java?

Découvrez cette excellente réponse qui fournit des informations beaucoup plus complètes sur les fonctions virtuelles:

Pouvez-vous écrire des fonctions / méthodes virtuelles en Java?

Polymorphisme et différents types de dépassement

De tutoriel Java

> La définition du dictionnaire du polymorphisme fait référence à un principe en biologie dans lequel un organisme ou une espèce peut avoir plusieurs formes ou stades différents. Ce principe peut également être appliqué à la programmation orientée objet et à des langages tels que le langage Java. ***Les sous-classes d'une classe peuvent définir leurs propres comportements uniques tout en partageant certaines des mêmes fonctionnalités de la classe parente.***

Regardez cet exemple pour comprendre différents types de dépassement.

1. La classe de base ne fournit aucune implémentation et la sous-classe doit remplacer la méthode complète - (résumé)
2. La classe de base fournit une implémentation par défaut et la sous-classe peut changer le comportement
3. La sous-classe ajoute l'extension à l'implémentation de la classe de base en appelant `super.methodName()` tant que première instruction
4. La classe de base définit la structure de l'algorithme (méthode Template) et la sous-classe remplacera une partie de l'algorithme

extrait de code:

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

sortie:

```
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:
```

Lire Polymorphisme en ligne: https://riptutorial.com/fr/java/topic/980/polymorphisme

Chapitre 144: Pools d'executor, d'executorService et de threads

Introduction

L'interface Executor de Java offre un moyen de découpler la soumission des tâches des mécanismes d'exécution de chaque tâche, y compris des détails sur l'utilisation des threads, la planification, etc. Un exécuteur est normalement utilisé au lieu de créer explicitement des threads. Avec les exécuteurs, les développeurs n'auront pas à réécrire leur code de manière significative pour pouvoir ajuster facilement la stratégie d'exécution des tâches de leur programme.

Remarques

Pièges

- Lorsque vous planifiez une tâche pour une exécution répétée, en fonction du ScheduledExecutorService utilisé, votre tâche peut être suspendue de toute exécution ultérieure, si une exécution de votre tâche provoque une exception qui n'est pas gérée. Voir Mère F ** k le ScheduledExecutorService!

Examples

Fire and Forget - Tâches exécutables

Les exécuteurs acceptent un `java.lang.Runnable` qui contient du code (potentiellement computationnel ou autre long ou lourd) à exécuter dans un autre thread.

L'utilisation serait:

```
Executor exec = anExecutor;
exec.execute(new Runnable() {
    @Override public void run() {
        //offloaded work, no need to get result back
    }
});
```

Notez qu'avec cet exécuteur, vous n'avez aucun moyen de récupérer une valeur calculée.
Avec Java 8, on peut utiliser lambdas pour raccourcir l'exemple de code.

Java SE 8

```
Executor exec = anExecutor;
exec.execute(() -> {
    //offloaded work, no need to get result back
});
```

ThreadPoolExecutor

Un exécuteur commun utilisé est le `ThreadPoolExecutor` , qui prend en charge la gestion des threads. Vous pouvez configurer la quantité minimale de threads que l'exécuteur doit toujours maintenir lorsqu'il n'y a pas grand-chose à faire (la taille de base) et une taille de thread maximale à laquelle le pool peut évoluer, s'il reste du travail à faire. Une fois que la charge de travail diminue, le pool réduit à nouveau le nombre de threads jusqu'à ce qu'il atteigne la taille minimale.

```
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
    }
});
```

Remarque Si vous configurez `ThreadPoolExecutor` avec une file d'attente *illimitée* , le nombre de threads ne dépassera pas `corePoolSize` car les nouveaux threads ne sont créés que si la file d'attente est pleine:

ThreadPoolExecutor avec tous les paramètres:

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

de JavaDoc

> S'il y a plus de corePoolSize mais moins de threads maximumPoolSize, un nouveau thread sera créé uniquement si la file d'attente est pleine.

Avantages:

1. La taille de BlockingQueue peut être contrôlée et des scénarios de mémoire insuffisante peuvent être évités. Les performances des applications ne seront pas dégradées avec une taille de file d'attente limitée.

2. Vous pouvez utiliser des stratégies existantes ou créer de nouvelles stratégies de rejet.

    1. Dans le ThreadPoolExecutor.AbortPolicy par défaut, le gestionnaire renvoie une exception RejectedExecutionException à l'exécution lors du rejet.

    2. Dans `ThreadPoolExecutor.CallerRunsPolicy` , le thread qui appelle exécute lui-même exécute la tâche. Cela fournit un mécanisme de contrôle de rétroaction simple qui ralentira le taux de soumission de nouvelles tâches.

3. Dans `ThreadPoolExecutor.DiscardPolicy` , une tâche qui ne peut pas être exécutée est simplement supprimée.

4. Dans `ThreadPoolExecutor.DiscardOldestPolicy` , si l'exécuteur n'est pas arrêté, la tâche en tête de la file d'attente de travail est supprimée, puis l'exécution est exécutée à nouveau (ce qui peut échouer à nouveau et entraîner sa répétition).

3. Custom `ThreadFactory` peut être configuré, ce qui est utile:

    1. Pour définir un nom de thread plus descriptif
    2. Pour définir le statut du démon de thread
    3. Pour définir la priorité des threads

Voici un exemple d'utilisation de ThreadPoolExecutor

Récupération de la valeur du calcul - Appelable

Si votre calcul produit une valeur de retour qui est requise plus tard, une simple tâche Runnable ne suffit pas. Dans ce cas, vous pouvez utiliser `ExecutorService.submit( Callable <T>)` qui renvoie une valeur une fois l'exécution terminée.

Le service renverra un `Future` que vous pourrez utiliser pour récupérer le résultat de l'exécution de la tâche.

```
// 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
```

Lorsque vous avez besoin d'obtenir le résultat du futur, appelez `future.get()`

- Attendez indéfiniment pour l'avenir pour finir avec un résultat.

```
try {
    // Blocks current thread until future is completed
    Integer result = future.get();
catch (InterruptedException || ExecutionException e) {
    // handle appropriately
}
```

- Attendez que l'avenir se termine, mais pas plus que le temps spécifié.

```
try {
    // Blocks current thread for a maximum of 500 milliseconds.
    // If the future finishes before that, result is returned,
    // otherwise TimeoutException is thrown.
    Integer result = future.get(500, TimeUnit.MILLISECONDS);
```

```
catch (InterruptedException || ExecutionException || TimeoutException e) {
    // handle appropriately
}
```

Si le résultat d'une tâche planifiée ou en cours d'exécution n'est plus requis, vous pouvez appeler `Future.cancel(boolean)` pour l'annuler.

- L'appel de `cancel(false)` supprime simplement la tâche de la file d'attente des tâches à exécuter.
- L'appel à `cancel(true)` interrompt *également* la tâche si elle est en cours d'exécution.

Planification de tâches à exécuter à une heure fixe, après un délai ou à plusieurs reprises

La classe `ScheduledExecutorService` fournit une méthode permettant de planifier des tâches uniques ou répétées de plusieurs manières. L'exemple de code suivant suppose que le `pool` a été déclaré et initialisé comme suit:

```
ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);
```

Outre les méthodes `ExecutorService` normales, l'API `ScheduledExecutorService` ajoute 4 méthodes qui planifient les tâches et renvoient `ScheduledFuture` objets `ScheduledFuture` . Ce dernier peut être utilisé pour récupérer des résultats (dans certains cas) et annuler des tâches.

Démarrer une tâche après un délai fixe

L'exemple suivant programme une tâche pour qu'elle démarre après dix minutes.

```
ScheduledFuture<Integer> future = pool.schedule(new Callable<>() {
        @Override public Integer call() {
            // do something
            return 42;
        }
    },
    10, TimeUnit.MINUTES);
```

Démarrer des tâches à un taux fixe

L'exemple suivant permet de planifier une tâche pour qu'elle démarre après dix minutes, puis de manière répétée à raison d'une fois par minute.

```
ScheduledFuture<?> future = pool.scheduleAtFixedRate(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

L'exécution de la tâche se poursuivra en fonction de la planification jusqu'à ce que le `pool` soit fermé, que le `future` soit annulé ou que l'une des tâches rencontre une exception.

Il est garanti que les tâches planifiées par un appel `scheduledAtFixedRate` donné ne se chevaucheront pas dans le temps. Si une tâche prend plus de temps que la période prescrite, les exécutions de tâches suivantes et suivantes peuvent commencer tard.

Démarrer des tâches avec un délai fixe

L'exemple suivant permet de planifier une tâche pour qu'elle démarre après dix minutes, puis plusieurs fois avec un délai d'une minute entre la fin d'une tâche et la suivante.

```
ScheduledFuture<?> future = pool.scheduleWithFixedDelay(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

L'exécution de la tâche se poursuivra en fonction de la planification jusqu'à ce que le `pool` soit fermé, que le `future` soit annulé ou que l'une des tâches rencontre une exception.

Gérer l'exécution rejetée

Si

1. vous essayez de soumettre des tâches à un exécuteur d'arrêt ou
2. la file d'attente est saturée (uniquement possible avec des bornes) et le nombre maximum de threads a été atteint,

`RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor)` sera appelé.

Le comportement par défaut est que vous obtiendrez une RejectedExecutionException lancée sur l'appelant. Mais il existe plus de comportements prédéfinis disponibles:

- **ThreadPoolExecutor.AbortPolicy** (par défaut, va lancer REE)
- **ThreadPoolExecutor.CallerRunsPolicy** (exécute une tâche sur le thread de l'appelant - le *bloquant*)
- **ThreadPoolExecutor.DiscardPolicy** (tâche silencieuse)
- **ThreadPoolExecutor.DiscardOldestPolicy** (**supprimer en** silence la tâche la **plus ancienne** dans la file d'attente et réessayer l'exécution de la nouvelle tâche)

Vous pouvez les définir en utilisant l'un des constructeurs 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) // <--
```

Vous pouvez aussi implémenter votre propre comportement en étendant l'interface RejectedExecutionHandler :

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
```

différences de gestion des exceptions submit () vs execute ()

Généralement, la commande execute () est utilisée pour les appels d'incendie et d'oubli (sans analyse du résultat) et la commande submit () est utilisée pour analyser le résultat d'un objet Future.

Nous devons être conscients de la différence clé entre les mécanismes de gestion des exceptions entre ces deux commandes.

Les exceptions de submit () sont avalées par framework si vous ne les avez pas capturées.

Exemple de code pour comprendre la différence:

Cas 1: soumettez la commande Runnable with execute (), qui signale l'exception.

```
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);
   }
 }
```

sortie:

```
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)
```

Cas 2: Remplacez execute () par submit (): `service.submit(new Runnable(){` Dans ce cas, les exceptions sont englouties par framework car la méthode run () ne les a pas capturées explicitement.

sortie:

```
creating service
a and b=4:0
a and b=4:0
```

Cas 3: Modifiez le newFixedThreadPool en ExtendedExecutor

```
//ExecutorService service = Executors.newFixedThreadPool(2);
 ExtendedExecutor service = new ExtendedExecutor();
```

sortie:

```
creating service
a and b=4:0
java.lang.ArithmeticException: / by zero
a and b=4:0
java.lang.ArithmeticException: / by zero
```

J'ai montré cet exemple pour couvrir deux sujets: Utilisez votre ThreadPoolExecutor personnalisé et gérez Exectpion avec ThreadPoolExecutor personnalisé.

Autre solution simple au problème ci-dessus: Lorsque vous utilisez la commande normale ExecutorService & submit, obtenez l'objet Future de la commande submit (), appelez get () API sur Future. Catch les trois exceptions, qui ont été citées dans l'implémentation de la méthode afterExecute. Avantage de ThreadPoolExecutor personnalisé par rapport à cette approche: vous devez gérer le mécanisme de gestion des exceptions dans un seul endroit - Custom ThreadPoolExecutor.

Cas d'utilisation pour différents types de constructions de concurrence

1. ExecutorService

   `ExecutorService executor = Executors.newFixedThreadPool(50);`

   C'est simple et facile à utiliser. Il cache les détails de bas niveau de `ThreadPoolExecutor` .

   Je préfère celui-ci lorsque le nombre de tâches `Callable/Runnable` est faible et que le cumul de tâches dans une file d'attente illimitée n'augmente pas la mémoire et ne dégrade pas les performances du système. Si vous avez des contraintes `CPU/Memory` , je préfère utiliser `ThreadPoolExecutor` avec des contraintes de capacité et `RejectedExecutionHandler` pour gérer le rejet des tâches.

2. CountDownLatch

   `CountDownLatch` sera initialisé avec un nombre donné. Ce nombre est décrémenté par des appels à la méthode `countDown()` . Les threads attendant ce nombre pour atteindre zéro peuvent appeler l'une des méthodes `await()` . L'appel de `await()` bloque le thread jusqu'à ce que le compte atteigne zéro. *Cette classe permet à un thread java d'attendre qu'un autre ensemble de threads termine ses tâches.*

   Cas d'utilisation:

   1. Atteindre un parallélisme maximal: nous souhaitons parfois lancer plusieurs threads en même temps pour obtenir un parallélisme maximal
   2. Attendre que N threads se termine avant de commencer l'exécution
   3. Détection de blocage

3. ThreadPoolExecutor : Il fournit plus de contrôle. Si l'application est limitée par le nombre de

tâches Runnable / Callable en attente, vous pouvez utiliser la file d'attente limitée en définissant la capacité maximale. Une fois que la file d'attente atteint la capacité maximale, vous pouvez définir RejectionHandler. Java fournit quatre types de stratégies `RejectedExecutionHandler` .

1. `ThreadPoolExecutor.AbortPolicy` , le gestionnaire renvoie une exception RejectedExecutionException lors du rejet.
2. ThreadPoolExecutor.CallerRunsPolicy`, le thread qui appelle exécute lui-même exécute la tâche. Cela fournit un mécanisme de contrôle de rétroaction simple qui ralentira le taux de soumission de nouvelles tâches.
3. Dans `ThreadPoolExecutor.DiscardPolicy` , une tâche qui ne peut pas être exécutée est simplement supprimée.
4. `ThreadPoolExecutor.DiscardOldestPolicy` , si l'exécuteur n'est pas arrêté, la tâche en tête de la file d'attente de travail est supprimée, puis l'exécution est exécutée à nouveau (ce qui peut échouer à nouveau, entraînant la répétition de l'opération).

Si vous souhaitez simuler le comportement `CountDownLatch` , vous pouvez utiliser la méthode `invokeAll()` .

4. Un autre mécanisme que vous n'avez pas cité est ForkJoinPool

   Le `ForkJoinPool` été ajouté à Java en Java 7. Le `ForkJoinPool` est similaire au Java `ExecutorService` mais avec une différence. Le `ForkJoinPool` permet aux tâches de fractionner leur travail en tâches plus petites qui sont ensuite soumises à `ForkJoinPool` . Le vol de tâche se produit dans `ForkJoinPool` lorsque des threads de travail libres volent des tâches de la file d'attente de threads de travail occupé.

   Java 8 a introduit une API supplémentaire dans ExecutorService pour créer un pool de vol de travail. Vous n'avez pas besoin de créer `RecursiveTask` et `RecursiveAction` mais vous pouvez toujours utiliser `ForkJoinPool` .

   ```
   public static ExecutorService newWorkStealingPool()
   ```

   > Crée un pool de threads voleur de travail en utilisant tous les processeurs disponibles comme niveau de parallélisme cible.

   Par défaut, il faudra un nombre de cœurs de CPU en paramètre.

Tous ces quatre mécanismes sont complémentaires. Selon le niveau de granularité que vous souhaitez contrôler, vous devez choisir les bons.

Attendez la fin de toutes les tâches dans ExecutorService

Jetons un coup d'oeil aux différentes options pour attendre l'achèvement des tâches soumises à l' exécuteur

1. ExecutorService `invokeAll()`

   Exécute les tâches données, en retournant une liste de contrats à terme conservant leur statut et leurs résultats lorsque tout est terminé.

Exemple:

```
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

   Une aide à la synchronisation qui permet à un ou plusieurs threads d'attendre qu'un ensemble d'opérations soit exécuté dans d'autres threads.

   Un **CountDownLatch** est initialisé avec un nombre donné. Les méthodes d'attente bloquent jusqu'à ce que le nombre actuel atteigne zéro en raison des `countDown()` méthode `countDown()` , après quoi tous les threads en attente sont libérés et toutes les invocations d'attente suivantes sont `countDown()` immédiatement. Ceci est un phénomène à un coup - le compte ne peut pas être réinitialisé. Si vous avez besoin d'une version qui réinitialise le compte,

envisagez d'utiliser un **CyclicBarrier** .

3. ForkJoinPool ou `newWorkStealingPool()` dans les exécuteurs
4. Parcourez tous les objets `Future` créés après la soumission à `ExecutorService`
5. Méthode recommandée pour arrêter la page de documentation d'Oracle d' 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()` : lance un arrêt ordonné au cours duquel les tâches précédemment soumises sont exécutées, mais aucune nouvelle tâche ne sera acceptée.

`shutdownNow()` : tente d'arrêter toutes les tâches en cours d'exécution, arrête le traitement des tâches en attente et renvoie une liste des tâches en attente d'exécution.

Dans l'exemple ci-dessus, si vos tâches prennent plus de temps, vous pouvez changer si condition à condition

Remplacer

```
if (!pool.awaitTermination(60, TimeUnit.SECONDS))
```

avec

```
while(!pool.awaitTermination(60, TimeUnit.SECONDS)) {
  Thread.sleep(60000);
```

}

Cas d'utilisation pour différents types d'ExecutorService

Les exécuteurs retournent différents types de ThreadPools répondant à des besoins spécifiques.

1. `public static ExecutorService newSingleThreadExecutor()`

   Crée un exécuteur qui utilise un seul thread de travail opérant sur une file d'attente illimitée

Il y a une différence entre `newFixedThreadPool(1)` et `newSingleThreadExecutor()` comme le dit le doc java pour ce dernier:

> Contrairement à newFixedThreadPool (1), par ailleurs équivalent, l'exécuteur renvoyé est garanti ne pas pouvoir être reconfiguré pour utiliser des threads supplémentaires.

Ce qui signifie qu'un `newFixedThreadPool` peut être reconfiguré plus tard dans le programme par: `((ThreadPoolExecutor) fixedThreadPool).setMaximumPoolSize(10)` Ceci n'est pas possible pour `newSingleThreadExecutor`

Cas d'utilisation:

1. Vous souhaitez exécuter les tâches soumises dans une séquence.
2. Vous n'avez besoin que d'un seul thread pour gérer toute votre demande

Les inconvénients:

1. La file d'attente sans limite est nuisible

2. `public static ExecutorService newFixedThreadPool(int nThreads)`

   > Crée un pool de threads qui réutilise un nombre fixe de threads exécutant une file d'attente sans limites partagée. À tout moment, au plus les threads nThreads seront des tâches de traitement actives. Si des tâches supplémentaires sont soumises lorsque tous les threads sont actifs, ils attendent dans la file d'attente jusqu'à ce qu'un thread soit disponible

   Cas d'utilisation:

   1. Utilisation efficace des noyaux disponibles. Configurez `nThreads` en `Runtime.getRuntime().availableProcessors()`
   2. Lorsque vous décidez que ce nombre de thread ne doit pas dépasser un nombre dans le pool de threads

   Les inconvénients:

   1. La file d'attente sans limite est nuisible.

3. `public static ExecutorService newCachedThreadPool()`

   > Crée un pool de threads qui crée de nouveaux threads en fonction des besoins, mais réutilise les threads précédemment construits lorsqu'ils sont disponibles

   Cas d'utilisation:

   1. Pour les tâches asynchrones de courte durée

   Les inconvénients:

   1. La file d'attente sans limite est nuisible.
   2. Chaque nouvelle tâche créera un nouveau thread si tous les threads existants sont

occupés. Si la tâche dure longtemps, un plus grand nombre de threads sera créé, ce qui dégradera les performances du système. Alternative dans ce cas: `newFixedThreadPool`

4. `public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`

   Crée un pool de threads pouvant planifier des commandes à exécuter après un délai donné ou à exécuter périodiquement.

   Cas d'utilisation:

   1. Gestion des événements récurrents avec des retards, qui se produiront à un certain intervalle de temps

   Les inconvénients:

   1. La file d'attente sans limite est nuisible.

5. `public static ExecutorService newWorkStealingPool()`

   Crée un pool de threads voleur de travail en utilisant tous les processeurs disponibles comme niveau de parallélisme cible

   Cas d'utilisation:

   1. Pour diviser et conquérir type de problèmes.
   2. Utilisation efficace des threads inactifs. Les threads inactifs volent les tâches des threads occupés.

   Les inconvénients:

   1. La taille de la file d'attente non liée est dangereuse.

Vous pouvez voir un inconvénient commun à tous ces ExecutorService: la file d'attente illimitée. Ceci sera adressé avec ThreadPoolExecutor

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

Avec `ThreadPoolExecutor` , vous pouvez

1. Contrôler dynamiquement la taille du pool de threads
2. Définir la capacité de `BlockingQueue`
3. Définir `RejectionExecutionHander` lorsque la file d'attente est pleine
4. `CustomThreadFactory` pour ajouter des fonctionnalités supplémentaires lors de la création de threads `(public Thread newThread(Runnable r)`

Utilisation des pools de threads

Les pools de threads sont principalement utilisés pour appeler des méthodes dans `ExecutorService`

.

Les méthodes suivantes peuvent être utilisées pour soumettre du travail à l'exécution:

Méthode	La description
`submit`	Exécute le travail soumis et retourne un futur qui peut être utilisé pour obtenir le résultat
`execute`	Exécutez la tâche dans le futur sans obtenir de valeur de retour
`invokeAll`	Exécutez une liste de tâches et retournez une liste de futures
`invokeAny`	Exécute tout mais retourne uniquement le résultat de celui qui a réussi (sans exceptions)

Une fois que vous avez terminé avec le pool de threads, vous pouvez appeler `shutdown()` pour terminer le pool de threads. Cela exécute toutes les tâches en attente. Attendre que toutes les tâches `awaitTermination` `isShutdown()` vous pouvez faire une boucle autour de `awaitTermination` ou `isShutdown()` .

Lire Pools d'executor, d'executorService et de threads en ligne:
https://riptutorial.com/fr/java/topic/143/pools-d-executor--d-executorservice-et-de-threads

Chapitre 145: Préférences

Examples

Ajouter des écouteurs d'événement

Il existe deux types d'événements émis par un `Preferences` objet: `PreferenceChangeEvent` et `NodeChangeEvent` .

PreferenceChangeEvent

Un `PreferenceChangeEvent` est émis par un objet `Properties` chaque fois que l'une des paires clé-valeur du nœud change. `PreferenceChangeEvent` s peut être écouté avec `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();
    }
});
```

Cet écouteur n'écoutera pas les paires clé-valeur modifiées des nœuds enfants.

NodeChangeEvent

Cet événement sera déclenché chaque fois qu'un nœud enfant d'un nœud `Properties` est ajouté ou supprimé.

```
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();
```

```
    }
});
```

Obtenir des sous-noeuds de préférences

`Preferences` objets de `Preferences` représentent toujours un nœud spécifique dans une arborescence de `Preferences` entière, un peu comme ceci:

```
/userRoot
├── com
│   └── mycompany
│       └── myapp
│           ├── darkApplicationMode=true
│           ├── showExitConfirmation=false
│           └── windowMaximized=true
└── org
    └── myorganization
        └── anotherapp
            ├── defaultFont=Helvetica
            ├── defaultSavePath=/home/matt/Documents
            └── exporting
                ├── defaultFormat=pdf
                └── openInBrowserAfterExport=false
```

Pour sélectionner le noeud /com/mycompany/myapp :

1. Par convention, basée sur le package d'une 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. Par chemin relatif:

   ```
   Preferences myApp = Preferences.userRoot().node("com/mycompany/myapp");
   ```

   L'utilisation d'un chemin relatif (un chemin ne commençant pas par /) entraînera la résolution du chemin par rapport au nœud parent sur lequel il est résolu. Par exemple, l'exemple suivant renvoie le noeud du chemin /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. Par chemin absolu:

   ```
   Preferences myApp = Preferences.userRoot().node("/com/mycompany/myapp");
   ```

   L'utilisation d'un chemin absolu sur le nœud racine ne sera pas différente de l'utilisation d'un chemin relatif. La différence est que, s'il est appelé sur un sous-nœud, le chemin sera résolu par rapport au nœud racine.

```
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
```

Accès aux préférences de coordination sur plusieurs instances d'application

Toutes les instances de Preferences sont toujours thread-safe sur les threads d'une seule machine virtuelle Java (JVM). Comme les Preferences peuvent être partagées entre plusieurs machines virtuelles, il existe des méthodes spéciales qui traitent de la synchronisation des modifications entre les machines virtuelles.

Si vous avez une application qui est censée s'exécuter dans une **seule instance , aucune synchronisation externe** n'est requise.

Si une application s'exécute dans **plusieurs instances** sur un seul système et que, par conséquent, l'accès aux Preferences doit être coordonné entre les JVM sur le système, la **méthode sync()** de tout noeud Preferences peut être utilisée pour garantir que les modifications apportées au noeud Preferences visible aux autres JVM sur le système:

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

Préférences d'exportation

Preferences nœuds de Preferences peuvent être exportés dans un document XML représentant ce nœud. L'arborescence XML résultante peut être importée à nouveau. Le document XML résultant se souviendra s'il a été exporté à partir des Preferences de l'utilisateur ou du système.

Pour exporter un seul noeud, mais **pas ses noeuds enfants** :

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) {}
    }
}
```

Pour exporter un seul noeud **avec ses noeuds enfants** :

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) {}
    }
}
```

Importation de préférences

Preferences nœuds de Preferences peuvent être importés à partir d'un document XML. L'importation est destinée à être utilisée avec la fonctionnalité d'exportation de Preferences , car elle crée les documents XML correspondants.

Les documents XML se souviendront s'ils ont été exportés à partir des Preferences de l'utilisateur ou du système. Par conséquent, ils peuvent être importés à nouveau dans leurs arborescences de Preferences respectives, sans que vous ayez à comprendre ou à savoir d'où ils viennent. La fonction statique détectera automatiquement si le document XML a été exporté à partir des Preferences de l'utilisateur ou du système et les importera automatiquement dans l'arborescence à partir de laquelle ils ont été exportés.

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) {}
    }
}
```

Suppression des écouteurs d'événements

Les écouteurs d'événements peuvent être supprimés à nouveau à partir de n'importe quel noeud Properties , mais l'instance de l'écouteur doit être conservée pour cela.

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);
```

La même chose s'applique pour NodeChangeListener .

Obtenir des valeurs de préférences

Une valeur d'un noeud Preferences peut être du type String , boolean , byte[] , double , float , int ou long . Toutes les invocations doivent fournir une valeur par défaut, au cas où la valeur spécifiée ne serait pas présente dans le noeud 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);
```

Définition des valeurs de préférences

Pour stocker une valeur dans le noeud Preferences , l'une des méthodes putXXX() est utilisée. Une valeur d'un noeud Preferences peut être du type String , boolean , byte[] , double , float , int ou 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);
```

Utiliser les préférences

Preferences peuvent être utilisées pour stocker les paramètres utilisateur reflétant les paramètres d'application personnels d'un utilisateur, par exemple sa police d'éditeur, si elles préfèrent que l'application soit lancée en mode plein écran, si elles ont coché une case à cocher comme ça.

```
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);
        }
    }
}
```

Lire Préférences en ligne: https://riptutorial.com/fr/java/topic/582/preferences

Chapitre 146: Processus

Remarques

Notez que l'API recommande qu'à partir de la version 1.5, la méthode préférée pour créer un processus utilise ProcessBuilder.start() .

Une autre remarque importante est que la valeur de sortie produite par waitFor dépend du programme / script exécuté. Par exemple, les codes de sortie produits par **calc.exe** sont différents de **notepad.exe** .

Examples

Exemple simple (version Java <1.5)

Cet exemple appellera la calculatrice Windows. Il est important de noter que le code de sortie variera en fonction du programme / script appelé.

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

Utiliser la classe ProcessBuilder

La classe ProcessBuilder facilite l'envoi d'une commande via la ligne de commande. Tout ce qu'il faut, c'est une liste de chaînes qui composent les commandes à saisir. Vous appelez simplement la méthode start () sur votre instance ProcessBuilder pour exécuter la commande.

Si vous avez un programme appelé Add.exe qui prend deux arguments et les ajoute, le code ressemblera à ceci:

```
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();
}
```

Quelques points à garder en tête:

- Le tableau de commandes doit tous être un tableau String
- Les commandes doivent être dans l'ordre (dans le tableau) qu'elles seraient si vous appeliez le programme dans la ligne de commande (c.-à-d. Que le nom du fichier
- Lorsque vous définissez le répertoire de travail, vous devez transmettre un objet File et pas uniquement le nom du fichier en tant que String.

Appels bloquants ou non bloquants

En général, lors d'un appel à la ligne de commande, le programme envoie la commande puis continue son exécution.

Cependant, vous voudrez peut-être attendre que le programme appelé se termine avant de poursuivre votre propre exécution (par exemple, le programme appelé écrira des données dans un fichier et votre programme en aura besoin pour accéder à ces données).

Cela peut facilement être fait en appelant la méthode waitFor() partir de l'instance Process retournée.

Exemple d'utilisation:

```
//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

Le lancement direct de processus externes à partir de Java en utilisant directement l'API java.lang.ProcessBuilder peut être un peu compliqué. La bibliothèque Apache Commons Exec facilite les choses. La bibliothèque ch.vorburger.exec s'étend sur Commons Exec pour le rendre vraiment pratique:

```
 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();
```

Piège: Runtime.exec, Process et ProcessBuilder ne comprennent pas la syntaxe du shell

Les Runtime.exec(String ...) et Runtime.exec(String) vous permettent d'exécuter une commande en tant que processus externe [1] . Dans la première version, vous indiquez le nom de la commande et les arguments de la commande en tant qu'éléments distincts du tableau de chaînes et le runtime Java demande au système d'exécution du système d'exploitation de démarrer la commande externe. La deuxième version est facile à utiliser, mais elle comporte des pièges.

Tout d'abord, voici un exemple d'utilisation de exec(String) utilisé en toute sécurité:

```
Process p = Runtime.exec("mkdir /tmp/testDir");
p.waitFor();
if (p.exitValue() == 0) {
    System.out.println("created the directory");
}
```

Espaces dans les chemins

Supposons que nous généralisions l'exemple ci-dessus pour pouvoir créer un répertoire arbitraire:

```
Process p = Runtime.exec("mkdir " + dirPath);
// ...
```

Cela fonctionnera généralement, mais cela échouera si dirPath est (par exemple) "/ home / user / My Documents". Le problème est que exec(String) divise la chaîne en une commande et les arguments en recherchant simplement des espaces. La chaîne de commande:

```
"mkdir /home/user/My Documents"
```

sera divisé en:

```
"mkdir", "/home/user/My", "Documents"
```

et cela provoquera l'échec de la commande "mkdir" car elle attend un argument, pas deux.

Face à cela, certains programmeurs essaient d'ajouter des guillemets autour du chemin. Cela ne fonctionne pas non plus:

```
"mkdir \"/home/user/My Documents\""
```

sera divisé en:

```
"mkdir", "\"/home/user/My", "Documents\""
```

Les caractères supplémentaires entre guillemets qui ont été ajoutés pour tenter de "citer" les espaces sont traités comme tous les autres caractères non blancs. En effet, tout ce que nous citons ou échappons dans les espaces va échouer.

La manière de gérer ces problèmes particuliers consiste à utiliser la surcharge exec(String ...) .

```
Process p = Runtime.exec("mkdir", dirPath);
// ...
```

Cela fonctionnera si dirpath inclut des caractères d' dirpath car cette surcharge de exec ne tente pas de diviser les arguments. Les chaînes sont transmises à l'appel du système d' exec système d'exploitation tel exec .

Redirection, pipelines et autres syntaxes de shell

Supposons que nous voulions rediriger l'entrée ou la sortie d'une commande externe ou exécuter un pipeline. Par exemple:

```
Process p = Runtime.exec("find / -name *.java -print 2>/dev/null");
```

ou

```
Process p = Runtime.exec("find source -name *.java | xargs grep package");
```

(Le premier exemple répertorie les noms de tous les fichiers Java du système de fichiers et le second affiche les instructions du package [2] dans les fichiers Java de l'arborescence "source".)

Ceux-ci ne vont pas fonctionner comme prévu. Dans le premier cas, la commande "find" sera exécutée avec "2> / dev / null" comme argument de commande. Il ne sera pas interprété comme une redirection. Dans le deuxième exemple, le caractère de tuyau ("|") et les travaux suivants seront donnés à la commande "find".

Le problème est que les méthodes exec et ProcessBuilder ne comprennent aucune syntaxe de shell. Cela inclut les redirections, les pipelines, l'expansion des variables, la mise en mémoire, etc.

Dans quelques cas (par exemple, une simple redirection), vous pouvez facilement obtenir l'effet souhaité en utilisant ProcessBuilder . Cependant, ce n'est pas vrai en général. Une autre approche consiste à exécuter la ligne de commande dans un shell; par exemple:

```
Process p = Runtime.exec("bash", "-c",
                         "find / -name *.java -print 2>/dev/null");
```

ou

```
Process p = Runtime.exec("bash", "-c",
                         "find source -name \\*.java | xargs grep package");
```

Mais notez que dans le deuxième exemple, nous avons dû échapper au caractère générique ("*") car nous voulons que le caractère générique soit interprété par "trouver" plutôt que par le shell.

Les commandes intégrées du shell ne fonctionnent pas

Supposons que les exemples suivants ne fonctionnent pas sur un système avec un shell de type UNIX:

```
Process p = Runtime.exec("cd", "/tmp");     // Change java app's home directory
```

ou

```
Process p = Runtime.exec("export", "NAME=value");  // Export NAME to the java app's
environment
```

Il y a quelques raisons pour lesquelles cela ne fonctionnera pas:

1. Sur "cd" et "export", les commandes sont des commandes intégrées au shell. Ils n'existent pas en tant qu'exécutables distincts.

2. Pour que les commandes intégrées à la coquille fassent ce qu'elles sont censées faire (par exemple, modifier le répertoire de travail, mettre à jour l'environnement), elles doivent changer l'emplacement de cet état. Pour une application normale (y compris une application Java), l'état est associé au processus d'application. Ainsi, par exemple, le processus fils qui exécuterait la commande "cd" ne pourrait pas modifier le répertoire de travail de son processus parent "java". De même, un exec processus « d ne peut pas changer le répertoire de travail pour un processus qui suit.

Ce raisonnement s'applique à toutes les commandes intégrées du shell.

---

1 - Vous pouvez également utiliser ProcessBuilder , mais cela n'est pas pertinent au sens de cet exemple.

2 - C'est un peu dur et prêt ... mais encore une fois, les défauts de cette approche ne sont pas pertinents pour l'exemple.

Lire Processus en ligne: https://riptutorial.com/fr/java/topic/4682/processus

Chapitre 147: Programmation concurrente (threads)

Introduction

L'informatique concurrente est une forme de calcul dans laquelle plusieurs calculs sont exécutés simultanément au lieu d'être séquentiels. Le langage Java est conçu pour prendre en charge la programmation simultanée via l'utilisation de threads. Les objets et les ressources sont accessibles par plusieurs threads; chaque thread peut potentiellement accéder à n'importe quel objet du programme et le programmeur doit s'assurer que les accès en lecture et en écriture aux objets sont correctement synchronisés entre les threads.

Remarques

Sujets apparentés sur StackOverflow:

- Types atomiques
- Pools d'executor, d'executorService et de threads
- Extension du Thread par rapport à l'implémentation de Runnable

Examples

Multithreading de base

Si vous avez beaucoup de tâches à exécuter et que toutes ces tâches ne dépendent pas du résultat des précédentes, vous pouvez utiliser **Multithreading** pour que votre ordinateur effectue toutes ces tâches en même temps en utilisant plus de processeurs si votre ordinateur le peut. Cela peut **accélérer l'** exécution de votre programme si vous avez d'importantes tâches indépendantes.

```
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);
        }
    }
}
```

Le code de la méthode d'exécution des différentes instances de CountAndPrint s'exécutera dans un ordre non prévisible. Un extrait d'un exemple d'exécution peut ressembler à ceci:

```
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
...
```

Producteur-consommateur

Un exemple simple de solution du problème producteur-consommateur. Notez que les classes JDK (AtomicBoolean et BlockingQueue) sont utilisées pour la synchronisation, ce qui réduit le risque de création d'une solution non valide. Consultez Javadoc pour différents types de BlockingQueue ; Le choix d'une implémentation différente peut modifier radicalement le comportement de cet exemple (comme DelayQueue ou 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();
    }
}
```

Utiliser ThreadLocal

Un outil utile dans la concurrence Java est ThreadLocal - cela vous permet d'avoir une variable qui sera unique à un thread donné. Ainsi, si le même code s'exécute dans des threads différents, ces exécutions ne partageront pas la valeur, mais chaque thread aura sa propre variable *locale au thread* .

Par exemple, cela est fréquemment utilisé pour établir le contexte (tel que les informations d'autorisation) du traitement d'une demande dans une servlet. Vous pourriez faire quelque chose comme ceci:

```
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
    }
}
```

Maintenant, au lieu de passer MyUserContext à chaque méthode, vous pouvez utiliser MyServlet.getContext() là où vous en avez besoin. Maintenant, bien sûr, cela introduit une variable qui doit être documentée, mais elle est adaptée aux threads, ce qui élimine beaucoup des inconvénients liés à l'utilisation d'une telle variable.

Le principal avantage est que chaque thread a sa propre variable locale de thread dans ce conteneur de contexts . Tant que vous l'utilisez à partir d'un point d'entrée défini (comme si vous exigiez que chaque servlet conserve son contexte, ou peut-être en ajoutant un filtre de servlet), vous pouvez compter sur ce contexte lorsque vous en avez besoin.

CountDownLatch

CountDownLatch

> Une aide à la synchronisation qui permet à un ou plusieurs threads d'attendre qu'un ensemble d'opérations soit exécuté dans d'autres threads.

1. Un CountDownLatch est initialisé avec un nombre donné.
2. Les méthodes d'attente bloquent jusqu'à ce que le nombre actuel atteigne zéro en raison des countDown() méthode countDown() , après quoi tous les threads en attente sont libérés et toutes les invocations d'attente suivantes sont countDown() immédiatement.
3. Il s'agit d'un phénomène ponctuel: le compte ne peut pas être réinitialisé. Si vous avez besoin d'une version qui réinitialise le compte, envisagez d'utiliser un CyclicBarrier .

Méthodes clés:

```
public void await() throws InterruptedException
```

> Fait en sorte que le thread en cours attende que le verrou ait été ramené à zéro, à moins que le thread ne soit interrompu.

```
public void countDown()
```

> Diminue le nombre de verrous, libérant tous les threads en attente si le nombre atteint zéro.

Exemple:

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

sortie:

```
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
```

Explication:

1. CountDownLatch est initialisé avec un compteur de 5 dans le thread principal
2. Le thread principal attend en utilisant la méthode await() .
3. Cinq instances de DoSomethingInAThread ont été créées. Chaque instance décrémente le compteur avec la méthode countDown() .
4. Une fois que le compteur devient nul, le thread principal reprendra

Synchronisation

En Java, il existe un mécanisme de verrouillage intégré au niveau de la langue: le bloc synchronized , qui peut utiliser n'importe quel objet Java en tant que verrou intrinsèque (chaque objet Java peut être associé à un moniteur).

Les verrous intrinsèques fournissent l'atomicité à des groupes d'instructions. Pour comprendre ce que cela signifie pour nous, examinons un exemple où la synchronized est 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();
}
```

Dans ce cas, s'il n'y avait pas le bloc synchronized , il y aurait eu plusieurs problèmes de simultanéité. Le premier serait avec l'opérateur post-incrémentation (ce n'est pas atomique en lui-même), et le second serait que nous observerions la valeur de t après qu'une quantité arbitraire d'autres threads ait eu la chance de le modifier. Cependant, comme nous avons acquis un verrou intrinsèque, il n'y aura pas de conditions de course ici et la sortie contiendra des nombres de 1 à 100 dans leur ordre normal.

Les verrous intrinsèques de Java sont des *mutex* (c'est-à-dire des verrous d'exécution mutuelle). L'exécution mutuelle signifie que si un thread a acquis le verrou, le second sera obligé d'attendre que le premier le libère avant de pouvoir acquérir le verrou pour lui-même. Remarque: Une opération pouvant placer le thread dans l'état wait (sleep) est appelée *opération de blocage* . Ainsi, l'acquisition d'un verrou est une opération de blocage.

Les serrures intrinsèques en Java sont *réentrantes* . Cela signifie que si un thread tente d'acquérir un verrou qu'il possède déjà, il ne le bloquera pas et l'acquérera avec succès. Par exemple, le code suivant *ne* bloquera *pas* lorsqu'il sera appelé:

```
public void bar(){
    synchronized(this){
        ...
    }
}
public void foo(){
    synchronized(this){
        bar();
    }
}
```

Outre synchronized blocs synchronized , il existe également synchronized méthodes synchronized .

Les blocs de code suivants sont pratiquement équivalents (même si le bytecode semble être différent):

1. bloc synchronized sur this :

```
public void foo() {
    synchronized(this) {
        doStuff();
    }
}
```

2. méthode synchronized :

```
    public synchronized void foo() {
        doStuff();
    }
```

De même pour static méthodes static , ceci:

```
class MyClass {
    ...
    public static void bar() {
        synchronized(MyClass.class) {
            doSomeOtherStuff();
        }
    }
}
```

a le même effet que celui-ci:

```
class MyClass {
    ...
    public static synchronized void bar() {
        doSomeOtherStuff();
    }
}
```

Opérations atomiques

Une opération atomique est une opération exécutée "tout à la fois", sans aucune chance que d'autres threads observent ou modifient l'état pendant l'exécution de l'opération atomique.

Considérons un **mauvais exemple** .

```
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();
}
```

Dans ce cas, il y a deux problèmes. Le premier problème est que l'opérateur de post-incrémentation n'est *pas* atomique. Il est composé de plusieurs opérations: obtenir la valeur, ajouter 1 à la valeur, définir la valeur. C'est pourquoi si nous lançons l'exemple, il est probable que nous ne verrons pas t: 100 dans la sortie - deux threads peuvent obtenir simultanément la valeur, l'incrémenter et la définir: disons que la valeur de t est 10 et deux les threads incrémentent t. Les deux threads définiront la valeur de t à 11, puisque le deuxième thread observe la valeur de t avant que le premier thread ne l'ait fini.

Le deuxième problème concerne la façon dont nous observons t. Lorsque nous imprimons la valeur de t, la valeur peut avoir déjà été modifiée par un thread différent après l'opération d'incrémentation de ce thread.

Pour résoudre ces problèmes, nous utiliserons le java.util.concurrent.atomic.AtomicInteger , qui

a plusieurs opérations atomiques à utiliser.

```
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();
}
```

La méthode incrementAndGet d' AtomicInteger incrémente et retourne la nouvelle valeur, éliminant ainsi la condition de course précédente. Veuillez noter que dans cet exemple, les lignes seront toujours hors service car nous ne faisons aucun effort pour séquencer les appels println et que cela sort du cadre de cet exemple, car cela nécessiterait une synchronisation et l'objectif de cet exemple est de montrer comment utiliser AtomicInteger pour éliminer les conditions de course concernant l'état.

Création d'un système bloqué de base

Une impasse se produit lorsque deux actions concurrentes attendent que l'autre se termine et que, par conséquent, aucune de ces actions ne se produit jamais. En Java, un verrou est associé à chaque objet. Pour éviter les modifications simultanées effectuées par plusieurs threads sur un seul objet, nous pouvons utiliser synchronized mot clé synchronized , mais tout est payant. L'utilisation incorrecte de mots clés synchronized peut conduire à des systèmes bloqués appelés systèmes bloqués.

Considérons qu'il y a 2 threads travaillant sur 1 instance, Lets appelle les threads comme First et Second, et disons que nous avons 2 ressources R1 et R2. First acquiert R1 et a également besoin de R2 pour son achèvement tandis que Second acquiert R2 et a besoin de R1 pour son achèvement.

donc à l'instant t = 0,

Le premier a R1 et le second a R2. maintenant First attend R2 alors que Second attend R1. cette attente est indéfinie et cela conduit à une impasse.

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

Sortie de ce programme:

```
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
```

Faire une pause d'exécution

Thread.sleep provoque la suspension de l'exécution du thread en cours pour une période spécifiée. C'est un moyen efficace de rendre le temps processeur disponible pour les autres threads d'une application ou d'autres applications pouvant s'exécuter sur un système informatique. Il existe deux méthodes de sleep surchargées dans la classe Thread.

Un qui spécifie le temps de sommeil à la milliseconde

```
public static void sleep(long millis) throws InterruptedException
```

Celui qui spécifie le temps de sommeil à la nanoseconde

```
public static void sleep(long millis, int nanos)
```

Mettre en pause l'exécution pendant 1 seconde

```
Thread.sleep(1000);
```

Il est important de noter que ceci est un indice pour le planificateur du noyau du système d'exploitation. Cela n'est pas forcément précis, et certaines implémentations ne prennent même pas en compte le paramètre nanoseconde (arrondissant éventuellement à la milliseconde près).

Il est recommandé de joindre un appel à Thread.sleep dans try / catch et intercepter InterruptedException .

Visualisation des barrières de lecture / écriture lors de l'utilisation de synchronized / volatile

Comme nous le savons, nous devons utiliser le mot-clé synchronized pour rendre l'exécution d'une méthode ou d'un bloc exclusive. Mais peu d'entre nous ne sont peut-être pas au courant d'un autre aspect important de l'utilisation de mots clés synchronized et volatile : en *plus de créer une unité de code atomique, elle fournit également une barrière en lecture / écriture* . Quelle est cette barrière de lecture / écriture? Discutons-en en utilisant un exemple:

```
class Counter {

  private Integer count = 10;

  public synchronized void incrementCount() {
    count++;
  }

  public Integer getCount() {
    return count;
  }
}
```

Supposons qu'un thread *A* appelle incrementCount() puis un autre thread *B* appelle getCount() . Dans ce scénario, il n'y a aucune garantie que B verra la valeur actualisée du count . Il peut toujours voir count comme 10 , même s'il est également possible qu'il ne voit jamais la valeur mise à jour du count jamais.

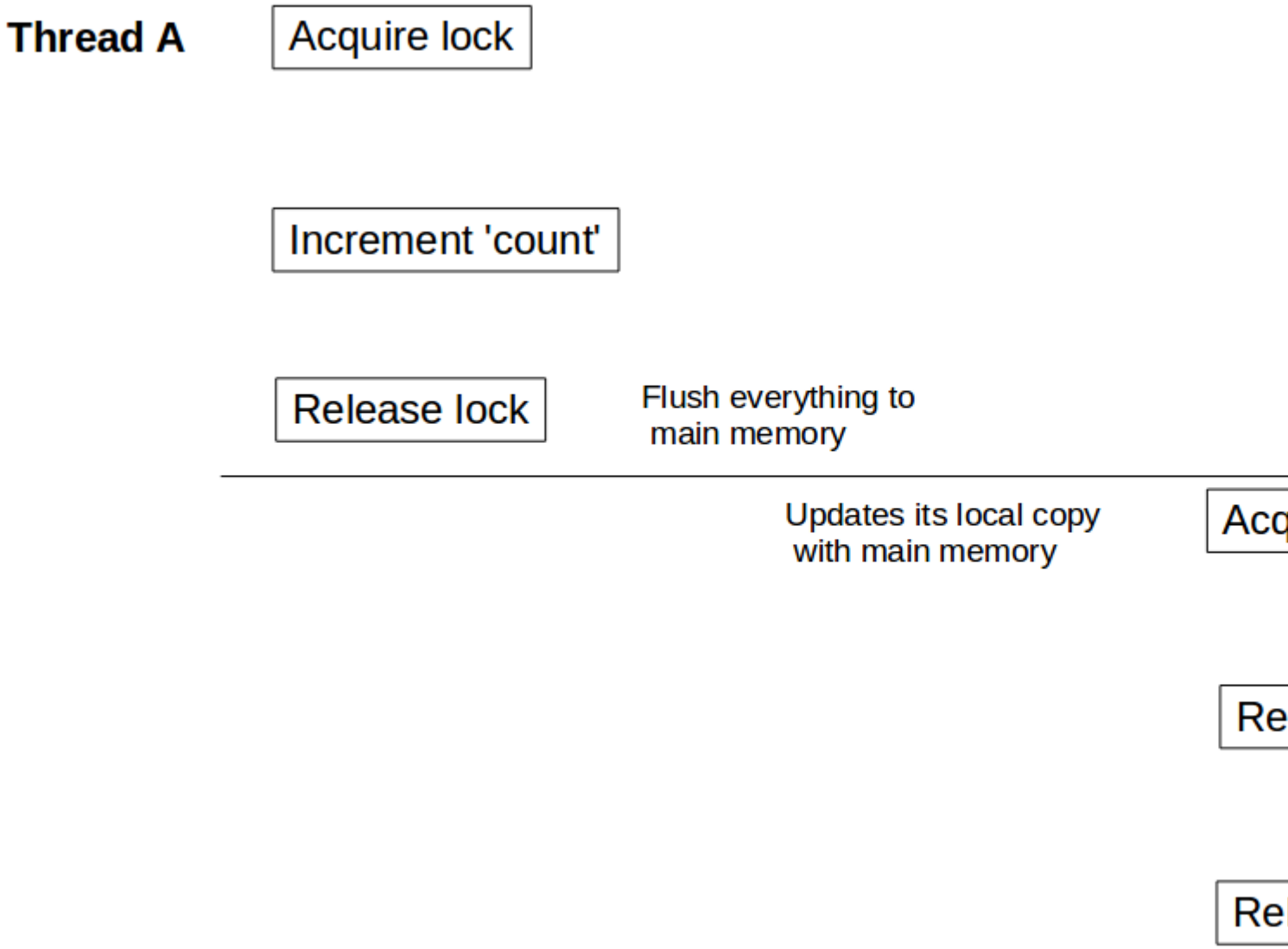
Pour comprendre ce comportement, nous devons comprendre comment le modèle de mémoire Java s'intègre à l'architecture matérielle. En Java, chaque thread a sa propre pile de threads. Cette pile contient: la pile des appels de méthode et la variable locale créée dans ce thread. Dans un système multi-core, il est fort possible que deux threads s'exécutent simultanément dans des cœurs distincts. Dans un tel scénario, il est possible qu'une partie de la pile d'un thread se trouve dans le registre / cache d'un core. Si dans un thread, on accède à un objet en utilisant

le mot-clé synchronized (ou volatile), après le blocage synchronized , le thread synchronise sa copie locale de cette variable avec la mémoire principale. Cela crée une barrière de lecture / écriture et s'assure que le thread voit la dernière valeur de cet objet.

Mais dans notre cas, puisque le thread B n'a pas utilisé l' accès synchronisé à count , il est peut - être la valeur du référant count stocké dans le registre et ne peut jamais voir les mises à jour de fil A. Pour vous assurer que B voit dernière valeur de comptage que nous devons faire getCount() synchronisé également.

```
public synchronized Integer getCount() {
  return count;
}
```

Désormais, lorsque le thread A est terminé avec le count mise à jour, il déverrouille l'instance Counter , crée en même temps une barrière d'écriture et vide toutes les modifications effectuées à l'intérieur de ce bloc dans la mémoire principale. De même, lorsque le thread B acquiert le verrou sur la même instance de Counter , il entre dans la barrière de lecture et lit la valeur de count dans la mémoire principale et voit toutes les mises à jour.



Même effet de visibilité pour volatile lectures / écritures volatile . Toutes les variables mises à jour avant d'écrire dans volatile seront vidées dans la mémoire principale et toutes les lectures après lecture d' volatile variable volatile proviendront de la mémoire principale.

Créer une instance java.lang.Thread

Il existe deux approches principales pour créer un thread en Java. En résumé, créer un thread

est aussi simple que d'écrire le code qui sera exécuté. Les deux approches diffèrent dans la définition de ce code.

En Java, un thread est représenté par un objet - une instance de java.lang.Thread ou de sa sous-classe. La première approche consiste donc à créer cette sous-classe et à remplacer la méthode *run ()* .

Remarque : j'utiliserai *Thread* pour faire référence à la classe java.lang.Thread et au *thread* pour faire référence au concept logique des threads.

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread running!");
        }
    }
}
```

Maintenant que nous avons déjà défini le code à exécuter, le thread peut être créé simplement comme:

```
MyThread t = new MyThread();
```

La classe Thread contient également un constructeur acceptant une chaîne, qui sera utilisée comme nom du thread. Cela peut être particulièrement utile lors du débogage d'un programme 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");
```

La deuxième approche consiste à définir le code à l'aide de java.lang.Runnable et de sa seule méthode *run ()* . La classe Thread vous permet alors d'exécuter cette méthode dans un thread séparé. Pour ce faire, créez le thread en utilisant un constructeur acceptant une instance de l'interface Runnable .

```
Thread t = new Thread(aRunnable);
```

Cela peut être très puissant lorsqu'il est combiné avec des références lambdas ou méthodes (Java 8 uniquement):

```
Thread t = new Thread(operator::hardWork);
```

Vous pouvez également spécifier le nom du thread.

```
Thread t = new Thread(operator::hardWork, "Pi operator");
```

Pratiquement parlant, vous pouvez utiliser les deux approches sans soucis. Cependant, la sagesse générale dit d'utiliser ce dernier.

Pour chacun des quatre constructeurs mentionnés, il existe également une alternative acceptant une instance de java.lang.ThreadGroup comme premier paramètre.

```
ThreadGroup tg = new ThreadGroup("Operators");
Thread t = new Thread(tg, operator::hardWork, "PI operator");
```

Le ThreadGroup représente un ensemble de threads. Vous ne pouvez ajouter un thread à un ThreadGroup qu'en utilisant le constructeur d'un thread . Le ThreadGroup peut alors être utilisé pour gérer tous ses threads ensemble, de même que le thread peut obtenir des informations à partir de son ThreadGroup .

Donc, pour résumer, le thread peut être créé avec l'un de ces constructeurs publics:

```
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)
```

Le dernier nous permet de définir la taille de pile souhaitée pour le nouveau thread.

Souvent, la lisibilité du code souffre lors de la création et de la configuration de nombreux threads avec les mêmes propriétés ou à partir du même modèle. C'est à ce moment que java.util.concurrent.ThreadFactory peut être utilisé. Cette interface vous permet d'encapsuler la procédure de création du thread via le modèle de fabrique et sa seule méthode *newThread (Runnable)* .

```
class WorkerFactory implements ThreadFactory {
    private int id = 0;

    @Override
    public Thread newThread(Runnable r) {
        return new Thread(r, "Worker " + id++);
    }
}
```

Thread Interruption / Stopping Threads

Chaque thread Java a un indicateur d'interruption, qui est initialement faux. Interrompre un thread, ce n'est rien d'autre que de lui attribuer la valeur true. Le code exécuté sur ce thread peut vérifier le drapeau à l'occasion et agir en conséquence. Le code peut également l'ignorer complètement. Mais pourquoi chaque fil aurait-il un tel drapeau? Après tout, avoir un drapeau booléen sur un fil est quelque chose que nous pouvons simplement organiser nous-mêmes, si et quand nous en avons besoin. Eh bien, il existe des méthodes qui se comportent de manière particulière lorsque le thread sur lequel elles s'exécutent est interrompu. Ces méthodes sont appelées méthodes de blocage. Ce sont des méthodes qui placent le thread dans l'état WAITING ou TIMED_WAITING. Lorsqu'un thread est dans cet état, l'interrompre provoquera une exception InterruptedException sur le thread interrompu, au lieu que l'indicateur d'interruption soit défini sur true et que le thread redevienne RUNNABLE. Le code qui appelle une méthode de blocage

est obligé de gérer l'exception Interrupted, car il s'agit d'une exception vérifiée. Donc, et par conséquent son nom, une interruption peut avoir pour effet d'interrompre un WAIT, le terminant effectivement. Notez que toutes les méthodes en attente (par exemple, le blocage des E / S) ne réagissent pas de cette manière aux interruptions, car elles ne mettent pas le thread en attente. Enfin, un thread dont l'indicateur d'interruption est défini, qui entre dans une méthode de blocage (c'est-à-dire qui tente d'entrer dans un état en attente), lancera immédiatement une exception InterruptedException et l'indicateur d'interruption sera effacé.

Outre ces mécanismes, Java n'attribue aucune signification sémantique particulière à l'interruption. Le code est libre d'interpréter une interruption comme bon lui semble. Mais le plus souvent, l'interruption est utilisée pour signaler à un thread qu'il doit cesser de fonctionner dès que possible. Mais, comme il ressort clairement de ce qui précède, il appartient au code de ce thread de réagir de manière appropriée à cette interruption afin de ne plus fonctionner. Arrêter un thread est une collaboration. Lorsqu'un thread est interrompu, son code en cours peut avoir plusieurs niveaux dans la trace de la pile. La plupart du code n'appelle pas de méthode de blocage et se termine suffisamment rapidement pour ne pas retarder indûment l'arrêt du thread. Le code qui devrait principalement concerner la réactivité aux interruptions, est le code qui est en boucle et gère les tâches jusqu'à ce qu'il n'y en ait plus, ou jusqu'à ce qu'un indicateur soit défini pour l'interrompre. Les boucles qui traitent des tâches éventuellement infinies (c'est-à-dire qu'elles continuent à fonctionner en principe) devraient vérifier l'indicateur d'interruption afin de quitter la boucle. Pour les boucles finies, la sémantique peut exiger que toutes les tâches soient terminées avant de se terminer, ou il peut être approprié de laisser certaines tâches non gérées. Le code qui appelle les méthodes de blocage sera obligé de gérer l'exception Interrupted. S'il est sémantiquement possible, il peut simplement propager l'InterruptedException et déclarer le lancer. En tant que tel, il devient une méthode de blocage en ce qui concerne ses appelants. S'il ne peut pas propager l'exception, il doit au minimum définir l'indicateur interrompu, afin que les appelants plus haut dans la pile sachent également que le thread a été interrompu. Dans certains cas, la méthode doit continuer d'attendre quelle que soit l'interruption, auquel cas elle doit retarder la définition de l'indicateur interrompu jusqu'à la fin de l'attente, cela peut impliquer la définition d'une variable locale à vérifier avant de quitter la méthode. puis interrompre son fil.

Exemples :

Exemple de code qui arrête la gestion des tâches lors de l'interruption

```
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
    }
}
```

Exemple de code qui retarde la définition de l'indicateur d'interruption jusqu'à sa complète exécution:

```
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
    }
}
```

Exemple de code qui a une liste de tâches fixe mais qui peut quitter tôt lorsqu'il est interrompu

```
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
    }
}
```

Exemple de producteur / consommateur multiple avec file d'attente globale partagée

Le code ci-dessous présente plusieurs programmes Producteur / Consommateur. Les threads Producteur et Consommateur partagent la même file d'attente 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();
            }
        }
    }
}
```

sortie:

```
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
```

etc ....

Explication:

1. sharedQueue , qui est un LinkedBlockingQueue est partagé entre tous les threads Producer et Consumer.
2. Les threads de producteurs produisent un entier pour 200 millisecondes en continu et l'ajoutent à sharedQueue
3. Consumer thread Consumer consomme un nombre entier de sharedQueue continu.
4. Ce programme est implémenté sans constructions explicites synchronized ou Lock . BlockingQueue est la clé pour y parvenir.

   Les implémentations BlockingQueue sont conçues pour être utilisées principalement pour les files d'attente producteur-consommateur.

   Les implémentations BlockingQueue sont sécurisées pour les threads. Toutes les méthodes de mise en file d'attente atteignent leurs effets de manière atomique en utilisant des verrous internes ou d'autres formes de contrôle de concurrence.

Écriture exclusive / accès en lecture simultanée

Un processus doit parfois écrire et lire simultanément les mêmes "données".

L'interface ReadWriteLock et son implémentation ReentrantReadWriteLock permettent un modèle d'accès pouvant être décrit comme suit:

1. Il peut y avoir un nombre quelconque de lecteurs simultanés des données. Si au moins un accès au lecteur est autorisé, aucun accès au graveur n'est possible.
2. Il peut y avoir au plus un seul auteur pour les données. Si un accès en écriture est accordé, aucun lecteur ne peut accéder aux données.

Une implémentation pourrait ressembler à:

```
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();
    }
}

}
```

NOTE 1 : Ce cas d'utilisation précis a une solution plus propre utilisant AtomicInteger , mais ce qui est décrit ici est un modèle d'accès, qui fonctionne indépendamment du fait que les données ici sont un entier atomique.

NOTE 2 : Le verrouillage de la partie lecture est vraiment nécessaire, même si cela peut ne pas paraître au lecteur occasionnel. En effet, si vous ne verrouillez pas le lecteur, un certain nombre de problèmes peuvent survenir, parmi lesquels:

1. Les écritures de valeurs primitives ne sont pas forcément atomiques sur toutes les JVM, de sorte que le lecteur pourrait voir, par exemple, que seuls 32 bits sur 64 bits écrivent si les data étaient de type 64 bits
2. La visibilité de l'écriture à partir d'un thread qui ne l'a pas effectuée est garantie par la JVM uniquement si nous établissons une *relation Happen Before* entre les écritures et les lectures. Cette relation est établie lorsque les lecteurs et les écrivains utilisent leurs verrous respectifs, mais pas autrement

Java SE 8

StampedLock des performances plus élevées sont requises, sous certains types d'utilisation, il existe un type de verrouillage plus rapide, appelé StampedLock , qui, entre autres, implémente un mode de verrouillage optimiste. Ce verrou fonctionne très différemment du ReadWriteLock , et cet exemple n'est pas transposable.

Objet Runnable

L'interface Runnable définit une méthode unique, run() , destinée à contenir le code exécuté dans le thread.

L'objet Runnable est transmis au constructeur Thread . Et la méthode start() de Thread est appelée.

Exemple

```
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();
    }
}
```

Exemple dans Java8:

```
public static void main(String[] args) {
    Runnable r = () -> System.out.println("Hello world");
    new Thread(r).start();
}
```

Sous-classe Runnable vs Thread

Un emploi d'objet Runnable est plus général, car l'objet Runnable peut sous-classer une classe autre que Thread .

Thread sous- Thread est plus facile à utiliser dans des applications simples, mais il est limité par le fait que votre classe de tâches doit être un descendant de Thread .

Un objet Runnable est applicable aux API de gestion des threads de haut niveau.

Sémaphore

Un sémaphore est un synchroniseur de haut niveau qui conserve un ensemble d' *autorisations* pouvant être acquises et libérées par les threads. Un sémaphore peut être imaginé comme un compteur de *permis* qui sera décrémenté lorsqu'un fil acquiert et incrémenté lors de la sortie d'un thread. Si le nombre de *permis* est *égal à* 0 lorsqu'un thread tente d'acquérir, le thread se bloque jusqu'à ce qu'un permis soit disponible (ou jusqu'à ce que le thread soit interrompu).

Un sémaphore est initialisé comme:

Semaphore semaphore = new Semaphore(1); // The int value being the number of permits

Le constructeur Semaphore accepte un paramètre booléen supplémentaire pour l'équité. Lorsqu'elle est définie sur false, cette classe ne garantit pas l'ordre dans lequel les threads acquièrent l'autorisation. Lorsque l'équité est définie sur true, le sémaphore garantit que les threads appelant l'une des méthodes d'acquisition sont sélectionnés pour obtenir les autorisations dans l'ordre dans lequel leur appel de ces méthodes a été traité. Il est déclaré de la manière suivante:

Semaphore semaphore = new Semaphore(1, true);

Examinons maintenant un exemple de javadocs, où Semaphore est utilisé pour contrôler l'accès à un pool d'éléments. Un sémaphore est utilisé dans cet exemple pour fournir une fonctionnalité de blocage afin de s'assurer qu'il ya toujours des éléments à obtenir lorsque getItem() est appelé.

```
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
    }
}
```

Ajouter deux tableaux `int` à l'aide d'un Threadpool

Un Threadpool a une file d'attente de tâches, dont chacune sera exécutée sur l'un de ces threads.

L'exemple suivant montre comment ajouter deux tableaux int aide d'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));
```

Remarques:

1. Cet exemple est purement illustratif. En pratique, il n'y aura pas d'accélération en utilisant des threads pour une tâche aussi petite. Un ralentissement est probable, car les frais généraux liés à la création et à la planification des tâches saturent le temps nécessaire à l'exécution d'une tâche.

2. Si vous utilisiez Java 7 et versions antérieures, vous utiliseriez des classes anonymes au lieu de lambdas pour implémenter les tâches.

Obtenir l'état de tous les threads démarrés par votre programme, à l'exception des threads système

Extrait de code:

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

Sortie:

```
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
```

Explication:

Thread.getAllStackTraces().keySet() renvoie tous les Thread y compris les threads d'application et les threads système. Si vous êtes uniquement intéressé par le statut des threads, démarré par votre application, effectuez une itération du jeu de Thread en vérifiant le groupe de threads d'un thread particulier par rapport au thread de votre programme principal.

En l'absence de la condition ThreadGroup ci-dessus, le programme renvoie le statut des threads système ci-dessous:

```
Reference Handler
Signal Dispatcher
Attach Listener
Finalizer
```

Callable et Future

Bien que Runnable fournisse un moyen d'emballer le code pour qu'il soit exécuté dans un thread différent, il présente une limitation en ce sens qu'il ne peut pas renvoyer un résultat de l'exécution. La seule façon d'obtenir une valeur de retour à partir de l'exécution d'un Runnable est d'affecter le résultat à une variable accessible dans une étendue en dehors de Runnable .

Callable été introduit dans Java 5 en tant que pair à Runnable . Callable est essentiellement le même sauf qu'il a une méthode d' call au lieu de run . La méthode call a la capacité supplémentaire de renvoyer un résultat et est également autorisée à lancer des exceptions vérifiées.

Le résultat d'une *soumission de tâche Callable* est disponible pour être exploité via un avenir

Future peut être considéré comme un conteneur contenant le résultat du calcul Callable . Le calcul du callable peut se poursuivre dans un autre thread, et toute tentative de toucher le résultat d'un Future bloquera et ne renverra le résultat qu'une fois disponible.

Interface appelable

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Avenir

```
interface Future<V> {
    V get();
    V get(long timeout, TimeUnit unit);
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
}
```

En utilisant l'exemple Callable et 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());
}
```

Notre callable qui fait un long calcul

```
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
    }
}
```

Sortie

```
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
```

Autres opérations autorisées sur Future

Alors que get() est la méthode pour extraire le résultat réel, Future a provision

- get(long timeout, TimeUnit unit) définit la durée maximale pendant laquelle le thread en cours attendra un résultat;
- Pour annuler l'appel de tâche, cancel(mayInterruptIfRunning) . L'indicateur mayInterrupt indique que la tâche doit être interrompue si elle a été démarrée et s'exécute maintenant.
- Pour vérifier si la tâche est terminée / terminée en appelant isDone() ;
- Pour vérifier si la longue tâche a été annulée isCancelled() .

Serrures comme aides à la synchronisation

Avant l'introduction du package simultané de Java 5, le thread était de niveau inférieur. L'introduction de ce package fournissait plusieurs aides / constructions de programmation concurrentes de niveau supérieur.

Les verrous sont des mécanismes de synchronisation de threads qui ont essentiellement le même objectif que les blocs synchronisés ou les mots-clés.

Verrouillage intrinsèque

```
int count = 0; // shared among multiple threads

public void doSomething() {
    synchronized(this) {
        ++count; // a non-atomic operation
    }
}
```

Synchronisation à l'aide de verrous

```
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
    }
}
```

Les verrous ont également des fonctionnalités disponibles que le verrouillage intrinsèque n'offre pas, telles que le verrouillage, mais en restant réactif aux interruptions, ou en essayant de se verrouiller, et non de bloquer le cas échéant.

Verrouillage, sensible aux interruptions

```
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
            }
        }
    }
}
```

Ne fais quelque chose que lorsque tu es capable de verrouiller

```
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
            }
        }
    }
}
```

Il existe plusieurs variantes de verrouillage disponibles .

Lire Programmation concurrente (threads) en ligne:
https://riptutorial.com/fr/java/topic/121/programmation-concurrente--threads-

Chapitre 148: Programmation parallèle avec framework Fork / Join

Examples

Tasks / Join Tasks in Java

Le framework fork / join en Java est idéal pour un problème qui peut être divisé en parties plus petites et résolu en parallèle. Les étapes fondamentales d'un problème fork / join sont les suivantes:

- Diviser le problème en plusieurs morceaux
- Résoudre chacune des pièces en parallèle
- Combinez chacune des sous-solutions en une solution globale

Une ForkJoinTask est l'interface qui définit un tel problème. On s'attend généralement à ce que vous sous-classiez l'une de ses implémentations abstraites (généralement la RecursiveTask) plutôt que d'implémenter l'interface directement.

Dans cet exemple, nous allons additionner une collection d'entiers, en divisant jusqu'à ce que nous atteignions des tailles de lots ne dépassant pas dix.

```
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;
        }
    }
}
```

Une instance de cette tâche peut maintenant être transmise à une instance de ForkJoinPool .

```
// Because I am not specifying the number of threads
// it will create a thread for each available processor
ForkJoinPool pool = new ForkJoinPool();

// Submit the task to the pool, and get what is effectively the Future
ForkJoinTask<Integer> task = pool.submit(new SummingTask(numbers));

// Wait for the result
int result = task.join();
```

Lire Programmation parallèle avec framework Fork / Join en ligne:
https://riptutorial.com/fr/java/topic/4245/programmation-parallele-avec-framework-fork---join

Chapitre 149: Récursivité

Introduction

La récursivité se produit lorsqu'une méthode s'appelle elle-même. Une telle méthode est appelée **récursive** . Une méthode récursive peut être plus concise qu'une approche non récursive équivalente. Cependant, pour une récursivité profonde, une solution itérative peut parfois consommer moins d'espace de pile fini d'un thread.

Cette rubrique comprend des exemples de récursivité en Java.

Remarques

Concevoir une méthode récursive

Lorsque vous concevez une méthode récursive, gardez à l'esprit que vous avez besoin de:

- **Cas de base.** Cela définira quand votre récursivité s'arrêtera et affichera le résultat. Le cas de base dans l'exemple factoriel est:

```
if (n <= 1) {
    return 1;
}
```

- **Appel récursif.** Dans cette instruction, vous appelez à nouveau la méthode avec un paramètre modifié. L'appel récursif dans l'exemple factoriel ci-dessus est:

```
else {
    return n * factorial(n - 1);
}
```

Sortie

Dans cet exemple, vous calculez le n-ième nombre factoriel. Les premières factorielles sont:

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

...

Élimination Java et Tail-Call

Les compilateurs Java actuels (jusqu'à Java 9 inclus) n'effectuent pas l'élimination des appels de queue. Cela peut avoir un impact sur les performances des algorithmes récursifs, et si la récursivité est suffisamment profonde, cela peut conduire à des StackOverflowError de StackOverflowError ; voir Récursivité profonde est problématique en Java

Examples

L'idée de base de la récursivité

Qu'est-ce que la récursivité:

En général, la récursivité se produit lorsqu'une fonction s'appelle directement ou indirectement. Par exemple:

```
// This method calls itself "infinitely"
public void useless() {
    useless();  // method calls itself (directly)
}
```

Conditions d'application de la récursivité à un problème:

Il existe deux conditions préalables à l'utilisation de fonctions récursives pour résoudre un problème spécifique:

1. Il doit y avoir une condition de base pour le problème, qui sera le point final de la récursivité. Lorsqu'une fonction récursive atteint la condition de base, il ne fait plus aucun appel récursif (plus profond).

2. Chaque niveau de récursivité devrait tenter un petit problème. La fonction récursive divise donc le problème en parties plus petites et plus petites. En supposant que le problème soit fini, cela garantira la fin de la récursivité.

À Java, il existe une troisième condition préalable: il ne faut pas avoir à se recentrer trop profondément pour résoudre le problème; voir Récursivité profonde est problématique en Java

Exemple

La fonction suivante calcule les factorielles en utilisant la récursivité. Notez que la méthode factorial s'appelle elle-même dans la fonction. Chaque fois qu'il appelle lui-même, il réduit le paramètre n de 1. Lorsque n atteint 1 (la condition de base), la fonction ne se déclenche plus.

```
public int factorial(int n) {
    if (n <= 1) { // the base condition
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Ce n'est pas un moyen pratique de calculer les factorielles en Java, car il ne prend pas en compte le débordement d'entier, ou le débordement de pile d'appel (c'est-à- `StackOverflowError` exceptions `StackOverflowError`) pour les grandes valeurs de `n` .

Calcul du nombre N ° Fibonacci

La méthode suivante calcule le Nième nombre de Fibonacci en utilisant la récursivité.

```
public int fib(final int n) {
    if (n > 2) {
        return fib(n - 2) + fib(n - 1);
    }
    return 1;
}
```

La méthode implémente un cas de base (n <= 2) et un cas récursif (n> 2). Ceci illustre l'utilisation de la récursivité pour calculer une relation récursive.

Cependant, bien que cet exemple soit illustratif, il est également inefficace: chaque instance de la méthode appelle la fonction elle-même deux fois, ce qui entraîne une croissance exponentielle du nombre de fois que la fonction est appelée lorsque N augmente. La fonction ci-dessus est O (2 ^{N}), mais une solution itérative équivalente a la complexité O (N). En outre, il

existe une expression "forme fermée" qui peut être évaluée en multiplications à virgule flottante O (N).

Calculer la somme des entiers de 1 à N

La méthode suivante calcule la somme des entiers de 0 à N en utilisant la récursivité.

```
public int sum(final int n) {
    if (n > 0) {
        return n + sum(n - 1);
    } else {
        return n;
    }
}
```

Cette méthode est O (N) et peut être réduite à une simple boucle en utilisant l'optimisation de l'appel. En fait, il existe une expression de *forme fermée* qui calcule la somme dans les opérations O(1) .

Calcul de la puissance N du nombre

La méthode suivante calcule la valeur de num élevée à la puissance de exp utilisant la récursivité:

```
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);
}
```

Ceci illustre les principes mentionnés ci-dessus: la méthode récursive implémente un cas de base (deux cas, n = 0 et n = 1) qui termine la récursivité et un cas récursif qui appelle à nouveau la méthode. Cette méthode est O (N) et peut être réduite à une simple boucle en utilisant l'optimisation de l'appel.

Inverser une chaîne en utilisant la récursivité

Voici un code récursif pour inverser une chaîne

```
/**
 * 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);
    }
}
```

Traverser une structure de données Tree avec récursivité

Considérons la classe Node ayant 3 données membres, le pointeur enfant gauche et le pointeur enfant droit comme ci-dessous.

```
public class Node {
    public int data;
    public Node left;
    public Node right;

    public Node(int data){
        this.data = data;
    }
}
```

Nous pouvons parcourir l'arbre construit en connectant plusieurs objets de la classe Node comme ci-dessous, la traversée est appelée traversée d'ordre de l'arbre.

```
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
        }
    }
```

Comme démontré ci-dessus, en utilisant la **récursivité**, nous pouvons traverser la **structure de données de l'arborescence** sans utiliser aucune autre structure de données qui n'est pas possible avec l'approche **itérative** .

Types de récursivité

La récursivité peut être classée comme **récursion de tête** ou **récursivité de queue** , selon l'endroit où l'appel de méthode récursif est placé.

Dans la **récursion principale** , l'appel récursif, lorsqu'il se produit, intervient avant tout autre traitement dans la fonction (pensez à ce qu'il se passe en haut ou en bas de la fonction).

Dans la **récursion de la queue** , c'est le contraire: le traitement a lieu avant l'appel récursif. Choisir entre les deux styles récursifs peut sembler arbitraire, mais le choix peut faire toute la différence.

Une fonction avec un chemin avec un seul appel récursif au début du chemin utilise ce qu'on appelle la récursion de la tête. La fonction factorielle d'une exposition précédente utilise la récursion de la tête. La première chose qu'il fait une fois qu'il détermine que la récursivité est nécessaire est de s'appeler avec le paramètre décrémenté. Une fonction avec un seul appel récursif à la fin d'un chemin utilise la récursion de queue.

```
public void tail(int n)
{
   if(n == 1)
      return;
   else
      System.out.println(n);
```

```
public void head(int n)
   {
     if(n == 0)
        return;
     else
        head(n-1);
```

```
    tail(n-1);                                  System.out.println(n);
}                                             }
```

Si l'appel récursif se produit à la fin d'une méthode, cela s'appelle une tail recursion . La récursion de la queue est similar to a loop . La method executes all the statements before jumping into the next recursive call .

Si l'appel récursif se produit au beginning of a method, it is called a head recursion . La method saves the state before jumping into the next recursive call .

Référence: La différence entre la récursion tête et queue

StackOverflowError & récursivité en boucle

Si un appel récursif devient "trop profond", cela se traduit par une StackOverflowError . Java alloue une nouvelle image pour chaque appel de méthode sur la pile de son thread. Cependant, l'espace de la pile de chaque thread est limité. Trop de cadres sur la pile mènent au dépassement de pile (SO).

Exemple

```
public static void recursion(int depth) {
    if (depth > 0) {
        recursion(depth-1);
    }
}
```

L'appel de cette méthode avec des paramètres importants (par exemple, la recursion(50000) entraînera probablement un débordement de pile. La valeur exacte dépend de la taille de la pile de threads, qui dépend à son tour de la structure du thread, des paramètres de ligne de commande tels que -Xss ou taille par défaut pour la JVM.

solution de contournement

Une récursivité peut être convertie en une boucle en stockant les données pour chaque appel récursif dans une structure de données. Cette structure de données peut être stockée sur le tas plutôt que sur la pile de threads.

En général, les données requises pour restaurer l'état d'une invocation de méthode peuvent être stockées dans une pile et une boucle while peut être utilisée pour "simuler" les appels récursifs. Les données pouvant être requises incluent:

- l'objet pour lequel la méthode a été appelée (méthodes d'instance uniquement)
- les paramètres de la méthode
- variables locales
- la position actuelle dans l'exécution ou la méthode

Exemple

La classe suivante permet de récurer une arborescence jusqu'à une profondeur spécifiée.

```
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(")");
        }
    }

}
```

par exemple

```
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();
```

Des tirages

```
(((...)20(...))10((...)30))
```

Cela pourrait être converti à la boucle suivante:

```
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();
```

Note: Ceci est juste un exemple de l'approche générale. Souvent, vous pouvez trouver une meilleure façon de représenter un cadre et / ou de stocker les données du cadre.

La récursivité profonde est problématique en Java

Considérons la méthode naïve suivante pour ajouter deux nombres positifs en utilisant la récursivité:

```
public static int add(int a, int b) {
    if (a == 0) {
        return b;
    } else {
        return add(a - 1, b + 1);  // TAIL CALL
    }
}
```

Ceci est algorithmiquement correct, mais il a un problème majeur. Si vous appelez add avec un grand a , il se bloquera avec une StackOverflowError , sur n'importe quelle version de Java

jusqu'à (au moins) Java 9.

Dans un langage de programmation fonctionnel typique (et dans de nombreux autres langages), le compilateur optimise la récursion de la queue . Le compilateur remarquerait que l'appel à add (à la ligne balisée) est un appel de queue , et réécrirait effectivement la récursivité en tant que boucle. Cette transformation s'appelle l'élimination par appel de queue.

Cependant, les compilateurs Java de la génération actuelle n'effectuent pas l'élimination des appels de queue. (Ce n'est pas un simple oubli. Il y a des raisons techniques substantielles à cela: voir ci-dessous.) Au lieu de cela, chaque appel récursif de add provoque l'allocation d'une nouvelle trame sur la pile du thread. Par exemple, si vous appelez add(1000, 1) , il faudra 1000 appels récursifs pour arriver à la réponse 1001 .

Le problème est que la taille de la pile de threads Java est fixe lorsque le thread est créé. (Cela inclut le thread "principal" dans un programme à thread unique.) Si trop de cadres de pile sont alloués, la pile débordera. La JVM détectera ceci et StackOverflowError une StackOverflowError .

Une approche pour gérer cela consiste simplement à utiliser une plus grande pile. Certaines options JVM contrôlent la taille par défaut d'une pile et vous pouvez également spécifier la taille de la pile en tant que paramètre du constructeur Thread . Malheureusement, cela ne fait que "retarder" le débordement de la pile. Si vous devez faire un calcul qui nécessite une pile encore plus grande, StackOverflowError revient.

La vraie solution consiste à identifier les algorithmes récursifs dans lesquels une récursivité profonde est probable, et *à* effectuer *manuellement* l'optimisation des appels de queue au niveau du code source. Par exemple, notre méthode d' add peut être réécrite comme suit:

```
public static int add(int a, int b) {
    while (a != 0) {
       a = a - 1;
       b = b + 1;
    }
    return b;
}
```

(De toute évidence, il existe de meilleures façons d'ajouter deux entiers. Ce qui précède sert simplement à illustrer l'effet de l'élimination manuelle de l'appel de la queue.)

Pourquoi l'élimination des appels de queue n'est pas encore implémentée en Java

Il y a un certain nombre de raisons pour lesquelles il n'est pas facile d'ajouter l'élimination des appels de queue à Java. Par exemple:

- Certains codes peuvent s'appuyer sur StackOverflowError pour (par exemple) placer une limite sur la taille d'un problème de calcul.
- Les responsables de la sécurité Sandbox s'appuient souvent sur l'analyse de la pile d'appels lorsqu'ils décident d'autoriser ou non le code non privilégié à effectuer une action privilégiée.

Comme l'explique John Rose dans "Tail calls in the VM" :

> *"Les effets de la suppression du cadre de pile de l'appelant sont visibles pour certaines API, notamment les contrôles de contrôle d'accès et le suivi de pile. C'est comme si l'appelant de l'appelant avait directement appelé l'appelé. Tous les privilèges Cependant, le couplage et l'accessibilité de la méthode appelée sont calculés avant le transfert du contrôle et prennent en compte l'appel appelant.*

En d'autres termes, l'élimination de l'appel de bout en bout pourrait amener une méthode de contrôle d'accès à penser à tort qu'une API sensible à la sécurité était appelée par du code de confiance.

Lire Récursivité en ligne: https://riptutorial.com/fr/java/topic/914/recursivite

Chapitre 150: Références d'objet

Remarques

Cela devrait vous aider à comprendre une "exception de pointeur nul" - on en obtient une car une référence d'objet est nulle, mais le code de programme s'attend à ce que le programme utilise quelque chose dans cette référence d'objet. Cependant, cela mérite son propre sujet ...

Examples

Références d'objet comme paramètres de méthode

Cette rubrique explique le concept d'une *référence d'objet* ; Il s'adresse aux personnes qui découvrent la programmation en Java. Vous devriez déjà être familiarisé avec certains termes et significations: définition de classe, méthode principale, instance d'objet, et appel de méthodes "sur" un objet, et transmission de paramètres aux méthodes.

```
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;
  }
}
```

Pour être pleinement compétent en programmation Java, vous devriez pouvoir expliquer cet exemple à quelqu'un d'autre part. Ses concepts sont fondamentaux pour comprendre le fonctionnement de Java.

Comme vous pouvez le voir, nous avons une main qui instancie un objet à la person variable et appelle une méthode pour définir le champ de name de cet objet sur "Bob" . Ensuite, il appelle une autre méthode et transmet la person comme l'un des deux paramètres; l'autre paramètre est une variable entière, définie sur 5.

La méthode appelée définit la valeur du name sur l'objet passé à "Linda" et définit la variable entière passée à 99, puis renvoie.

Alors qu'est-ce qui serait imprimé?

```
Linda 5
```

Alors, pourquoi le changement apporté à la person prend-il effet en main , mais le changement apporté à l'entier ne le fait pas?

Lorsque l'appel est effectué, la méthode principale transmet une *référence d'objet* à person à la méthode setPersonName ; Toute modification setAnotherName par setAnotherName à cet objet fait partie de cet objet. Par conséquent, ces modifications font toujours partie de cet objet lors du retour de la méthode.

Une autre façon de dire la même chose: la person pointe vers un objet (stocké sur le tas, si cela vous intéresse). Toute modification apportée par la méthode à cet objet est effectuée "sur cet objet" et n'est pas affectée par le fait que la méthode effectuant la modification est toujours active ou a été renvoyée. Lorsque la méthode retourne, toutes les modifications apportées à l'objet sont toujours stockées sur cet objet.

Contrastez ceci avec le nombre entier qui est passé. Comme il s'agit d'une *primitive* int (et non d'une instance d'objet Integer), elle est transmise "par valeur", ce qui signifie que sa valeur est fournie à la méthode et non par un entier. La méthode peut la changer pour la méthode. propres fins, mais cela n'affecte pas la variable utilisée lors de l'appel de la méthode.

En Java, toutes les primitives sont transmises par valeur. Les objets sont passés par référence, ce qui signifie qu'un pointeur sur l'objet est transmis en tant que paramètre à toutes les méthodes qui les prennent.

Une chose moins évidente: cela ne permet pas à une méthode appelée de créer un *nouvel* objet et de le retourner comme paramètre. La seule manière pour une méthode de renvoyer un objet créé, directement ou indirectement, par l'appel de méthode est la valeur renvoyée par la méthode. Voyons d'abord comment cela ne fonctionnerait pas, et ensuite comment cela fonctionnerait.

Ajoutons une autre méthode à notre petit exemple ici:

```
private static void getAnotherObjectNot(Person person) {
  person = new Person();
  person.setName("George");
}
```

Et, de retour dans la main , au- dessous de l'appel à setAnotherName , nous allons mettre un appel à cette méthode et un autre appel println:

```
getAnotherObjectNot(person);
System.out.println(person.getName());
```

Maintenant, le programme imprimera:

```
Linda 5
Linda
```

Qu'est-il arrivé à l'objet qui avait George? Eh bien, le paramètre qui a été transmis était un pointeur sur Linda; Lorsque la méthode getAnotherObjectNot créé un nouvel objet, elle a remplacé la référence à l'objet Linda par une référence à l'objet George. L'objet Linda existe toujours (sur le tas), la méthode main peut toujours y accéder, mais la méthode getAnotherObjectNot ne pourra rien faire après, car elle n'y fait aucune référence. Il semblerait que l'auteur du code ait voulu que la méthode crée un nouvel objet et le renvoie, mais si c'est le cas, cela n'a pas fonctionné.

Si c'est ce que l'auteur voulait faire, il devrait renvoyer l'objet nouvellement créé depuis la méthode, quelque chose comme ceci:

```
private static Person getAnotherObject() {
  Person person = new Person();
  person.setName("Mary");
  return person;
}
```

Alors appelez comme ceci:

```
Person mary;
mary = getAnotherObject();
System.out.println(mary.getName());
```

Et la totalité du programme serait désormais:

```
Linda 5
Linda
Mary
```

Voici le programme complet, avec les deux ajouts suivants:

```
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;
  }
}
```

Lire Références d'objet en ligne: https://riptutorial.com/fr/java/topic/5454/references-d-objet

Chapitre 151: Réglage des performances Java

Examples

Approche générale

Internet regorge de conseils pour améliorer les performances des programmes Java. Peut-être le conseil numéro un est la sensibilisation. Cela signifie:

- Identifier les problèmes de performance et les goulots d'étranglement possibles.
- Utilisez des outils d'analyse et de test.
- Connaître les bonnes pratiques et les mauvaises pratiques.

Le premier point devrait être fait lors de la phase de conception si vous parlez d'un nouveau système ou module. Si vous parlez de code hérité, des outils d'analyse et de test apparaissent. L'outil le plus fondamental pour analyser vos performances JVM est JVisualVM, qui est inclus dans le JDK.

Le troisième point concerne surtout l'expérience et les recherches approfondies, et bien sûr les astuces brutes qui apparaîtront sur cette page et d'autres, comme ceci .

Réduire la quantité de cordes

En Java, il est trop "facile" de créer de nombreuses instances de String qui ne sont pas nécessaires. Cela et d'autres raisons peuvent amener votre programme à avoir beaucoup de chaînes que le GC est en train de nettoyer.

Voici quelques façons de créer des instances de chaîne:

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

Ou pire, en boucle ou en récursion:

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

Le problème est que chaque + crée une nouvelle chaîne (généralement, puisque les nouveaux compilateurs optimisent certains cas). Une optimisation possible peut être faite en utilisant StringBuilder ou StringBuffer :

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

Si vous générez souvent de longues chaînes (SQL par exemple), utilisez une API de génération de chaînes.

Autres choses à considérer:

- Réduire l'utilisation de replace , substring - substring etc.
- Évitez String.toArray() , en particulier dans le code fréquemment String.toArray() .
- Les impressions de journal destinées à être filtrées (en raison du niveau de journalisation, par exemple) ne doivent pas être générées (le niveau de journalisation doit être vérifié au préalable).
- Utilisez des bibliothèques comme celle-ci si nécessaire.

- StringBuilder est préférable si la variable est utilisée d'une manière non partagée (à travers les threads).

Une approche basée sur des preuves pour l'optimisation des performances Java

Donald Knuth est souvent cité comme disant ceci:

> « Les programmeurs gaspillent des quantités énormes de temps à penser à, ou se soucier, la vitesse des parties non critiques de leurs programmes, et ces tentatives d'efficacité ont en fait un fort impact négatif lorsque sont considérés comme le débogage et la maintenance. *Il faut oublier* les *petits gains* d' *efficacité, disent 97% du temps* : l'optimisation prématurée est la racine de tous les maux. Pourtant, nous ne devrions pas laisser passer nos opportunités dans ces 3% critiques. "

la source

Compte tenu de ce conseil avisé, voici la procédure recommandée pour optimiser les programmes:

1. Tout d'abord, concevez et codez votre programme ou votre bibliothèque en mettant l'accent sur la simplicité et l'exactitude. Pour commencer, ne consacrez pas beaucoup d'efforts à la performance.
2. Mettez-le en état de marche et, idéalement, développez des tests unitaires pour les éléments clés du code.
3. Développer un benchmark de performance au niveau applicatif. Le test de performance doit couvrir les aspects critiques de votre application en termes de performances, et effectuer une série de tâches typiques de l'utilisation de l'application en production.
4. Mesurer la performance.
5. Comparez les performances mesurées à vos critères pour déterminer la rapidité avec laquelle l'application doit être. (Évitez les critères irréalistes, irréalisables ou non quantifiables tels que "aussi vite que possible".)
6. Si vous avez satisfait aux critères, STOP. Votre travail est terminé. (Tout effort supplémentaire est probablement une perte de temps.)
7. Profil de l'application pendant qu'elle exécute votre test de performances.
8. Examinez les résultats du profilage et sélectionnez les "zones sensibles de performance" les plus importantes (non optimisées). c'est-à-dire des sections du code où l'application semble passer le plus de temps.
9. Analysez la section de code du hotspot pour essayer de comprendre pourquoi il s'agit d'un goulot d'étranglement et réfléchissez à un moyen de le rendre plus rapide.
10. Implémentez cela comme un changement de code proposé, testez et déboguez.
11. Réexécutez le test pour voir si le changement de code a amélioré les performances:
    - Si oui, retournez à l'étape 4.
    - Si non, abandonner le changement et retourner à l'étape 9. Si vous ne faites aucun progrès, choisissez un autre point sensible pour votre attention.

Finalement, vous arriverez à un point où l'application est soit assez rapide, soit vous avez considéré tous les points chauds significatifs. À ce stade, vous devez arrêter cette approche. Si une section de code consomme (disons) 1% du temps total, alors même une amélioration de 50% ne fera que rendre l'application 0,5% plus rapide.

Clairement, il existe un point au-delà duquel l'optimisation des points d'accès est une perte de temps. Si vous arrivez à ce stade, vous devez adopter une approche plus radicale. Par exemple:

- Regardez la complexité algorithmique de vos algorithmes de base.

- Si l'application consacre beaucoup de temps à la récupération de place, recherchez des moyens de réduire le taux de création d'objets.
- Si des parties clés de l'application consomment beaucoup de ressources processeur et sont à thread unique, recherchez les opportunités de parallélisme.
- Si l'application est déjà multithread, recherchez les goulots d'étranglement de concurrence.

Mais dans la mesure du possible, utilisez des outils et des mesures plutôt que l'instinct pour diriger votre effort d'optimisation.

Lire Réglage des performances Java en ligne: https://riptutorial.com/fr/java/topic/4160/reglage-des-performances-java

Chapitre 152: Repères

Introduction

Ecrire des tests de performances en Java n'est pas aussi simple que d'obtenir System.currentTimeMillis() au début et à la fin et de calculer la différence. Pour écrire des tests de performances valides, il faut utiliser les outils appropriés.

Examples

Exemple JMH simple

JMH est l'un des outils pour écrire des tests de référence appropriés. Disons que nous voulons comparer les performances de recherche d'un élément dans HashSet et TreeSet .

Le moyen le plus simple d'intégrer JHM dans votre projet est d'utiliser les plugins maven et shade . Vous pouvez également voir pom.xml partir d' exemples 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>
```

Après cela, vous devez écrire la classe de référence elle-même:

```
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));
    }
}
```

N'oubliez pas ce blackhole.consume() , nous y reviendrons plus tard. Aussi, nous avons besoin de la classe principale pour exécuter le benchmark:

```
package benchmark;

import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

public class BenchmarkMain {
    public static void main(String[] args) throws RunnerException {
        final Options options = new OptionsBuilder()
                .include(CollectionFinderBenchmarkTest.class.getSimpleName())
                .forks(1)
                .build();

        new Runner(options).run();
    }
}
```

Et nous sommes tous prêts. Nous avons juste besoin de lancer le mvn package (il va créer benchmarks.jar dans votre dossier /target) et exécuter notre test de mvn package :

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

Et après quelques itérations d'échauffement et de calcul, nous aurons nos résultats:

```
# 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 propos de ce blackhole.consume() . Si vos calculs ne changent pas l'état de votre application, java le ignorera probablement. Donc, pour l'éviter, vous pouvez soit faire en sorte que vos méthodes de référence renvoient une valeur, soit utiliser l'objet Blackhole pour le consommer.

Vous pouvez trouver plus d'informations sur l'écriture de références dans le blog d'Aleksey Shipilëv , dans le blog de Jacob Jenkov et dans le blog java-performance: 1 , 2 .

Lire Repères en ligne: https://riptutorial.com/fr/java/topic/9514/reperes

Chapitre 153: Responsable de la sécurité

Examples

Activation du SecurityManager

Les machines virtuelles Java (JVM) peuvent être exécutées avec un SecurityManager installé. SecurityManager gouverne ce que le code exécuté dans la JVM est autorisé à faire, en fonction de facteurs tels que l'endroit où le code a été chargé et les certificats utilisés pour signer le code.

SecurityManager peut être installé en définissant la propriété système java.security.manager sur la ligne de commande lors du démarrage de la machine virtuelle Java:

```
java -Djava.security.manager <main class name>
```

ou par programme depuis le code Java:

```
System.setSecurityManager(new SecurityManager())
```

Java SecurityManager standard accorde des autorisations sur la base d'une stratégie, qui est définie dans un fichier de stratégie. Si aucun fichier de stratégie n'est spécifié, le fichier de stratégie par défaut sous $JAVA_HOME/lib/security/java.policy sera utilisé.

Classes de bac à sable chargées par un ClassLoader

ClassLoader doit fournir un ProtectionDomain identifiant la source du code:

```
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;
    }
}
```

En findClass lieu de loadClass le modèle de délégation est préservé et PluginClassLoader interroge d'abord le système et le classloader parent pour les définitions de classe.

Créer une politique:

```
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;
    }

}
```

Enfin, définissez la stratégie et un SecurityManager (l'implémentation par défaut est correcte):

```
    Policy.setPolicy(new PluginSecurityPolicy());
    System.setSecurityManager(new SecurityManager());
```

Implémentation de règles de refus de politique

Il est parfois souhaitable de *refuser* une certaine Permission à certains ProtectionDomain , *quelles que soient* les autres autorisations dont le domaine dispose. Cet exemple illustre l'une des approches possibles pour satisfaire ce type d'exigence. Il introduit une classe d'autorisation "négative", ainsi qu'un wrapper qui permet de réutiliser la Policy par défaut en tant que référentiel de ces autorisations.

Remarques:

- La syntaxe du fichier de stratégie standard et le mécanisme d'attribution des autorisations en général restent inchangés. Cela signifie que *les* règles de *refus* dans les fichiers de stratégie sont toujours exprimées sous forme d' *attributions* .
- L'encapsuleur de stratégie est conçu pour encapsuler spécifiquement la Policy par défaut sauvegardée par fichier (supposée être com.sun.security.provider.PolicyFile).
- Les autorisations refusées ne sont traitées qu'en tant que telles au niveau de la stratégie. Si elles sont assignées statiquement à un domaine, elles seront par défaut traitées par ce domaine comme des autorisations "positives" ordinaires.

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();
    }

}
```

Démo

```
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");
    }

}
```

Attribuez des autorisations:

```
grant codeBase "file:///path/to/classes/bin/-"
    permission java.util.PropertyPermission "*", "read,write";
    permission com.example.DeniedPermission "java.util.PropertyPermission:foo.bar:read";
};
```

Enfin, exécutez le Main et regardez-le échouer, en raison de la règle "deny" (la DeniedPermission) qui DeniedPermission le grant (son PropertyPermission). Notez qu'un setProperty("foo.baz", "xyz") aurait plutôt réussi, car l'autorisation refusée ne couvre que l'action "read", et uniquement pour la propriété "foo.bar".

Lire Responsable de la sécurité en ligne:
https://riptutorial.com/fr/java/topic/5712/responsable-de-la-securite

Chapitre 154: Ressources (sur classpath)

Introduction

Java permet la récupération des ressources basées sur des fichiers stockées dans un fichier JAR à côté des classes compilées. Cette rubrique se concentre sur le chargement de ces ressources et leur mise à disposition dans votre code.

Remarques

Une *ressource* est une donnée de type fichier avec un nom de chemin d'accès, qui réside dans le chemin de classe. L'utilisation la plus courante des ressources consiste à regrouper des images d'application, des sons et des données en lecture seule (telles que la configuration par défaut).

Les ressources sont accessibles avec les méthodes ClassLoader.getResource et ClassLoader.getResourceAsStream . Le cas d'utilisation le plus courant est d'avoir des ressources placées dans le même package que la classe qui les lit; les méthodes Class.getResource et Class.getResourceAsStream servent ce cas d'utilisation commun.

La seule différence entre une méthode getResource et la méthode getResourceAsStream est que la première renvoie une URL, tandis que la seconde ouvre cette URL et renvoie un InputStream.

Les méthodes de ClassLoader acceptent un nom de ressource de type chemin d'accès en tant qu'argument et effectuent une recherche dans chaque emplacement du chemin de classe du ClassLoader pour une entrée correspondant à ce nom.

- Si un emplacement de chemin de classe est un fichier .jar, une entrée jar avec le nom spécifié est considérée comme une correspondance.
- Si un emplacement de chemin de classe est un répertoire, un fichier relatif sous ce répertoire avec le nom spécifié est considéré comme une correspondance.

Le nom de la ressource est similaire à la portion de chemin d'une URL relative. Sur *toutes les plateformes,* il utilise des barres obliques (/) comme séparateurs de répertoires. Il ne doit pas commencer par une barre oblique.

Les méthodes correspondantes de la classe sont similaires, sauf:

- Le nom de la ressource peut commencer par une barre oblique, auquel cas cette barre oblique initiale est supprimée et le reste du nom est transmis à la méthode correspondante de ClassLoader.
- Si le nom de la ressource ne commence pas par une barre oblique, il est traité comme relatif à la classe dont la méthode getResource ou getResourceAsStream est appelée. Le nom de ressource réel devient *package* / *name* , où *package* est le nom du package auquel appartient la classe, chaque période étant remplacée par une barre oblique, et *name* est l'argument d'origine donné à la méthode.

Par exemple:

```
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);
    }
}
```

Les ressources doivent être placées dans des packages nommés, plutôt qu'à la racine d'un fichier .jar, pour la même raison, les classes sont placées dans des packages: pour empêcher les collisions entre plusieurs fournisseurs. Par exemple, si plusieurs chemins d'accès au fichier .jar se trouvent dans le chemin d'accès aux classes et que plusieurs d'entre eux contiennent une entrée config.properties dans sa racine, les appels aux méthodes getResource ou getResourceAsStream renvoient config.properties. le classpath. Ce comportement n'est pas prévisible dans les environnements où l'ordre de classpath n'est pas sous le contrôle direct de l'application, tel que Java EE.

Toutes les méthodes getResource et getResourceAsStream renvoient null si la ressource spécifiée n'existe pas. Comme les ressources doivent être ajoutées à l'application au moment de la construction, leurs emplacements doivent être connus lors de l'écriture du code; le fait de ne pas trouver une ressource à l'exécution est généralement le résultat d'une erreur du programmeur.

Les ressources sont en lecture seule. Il n'y a aucun moyen d'écrire sur une ressource. Les développeurs novices commettent souvent l'erreur de supposer que la ressource étant un fichier physique distinct lors du développement dans un IDE (comme Eclipse), il sera prudent de la traiter comme un fichier physique distinct dans le cas général. Cependant, ce n'est pas correct; les applications sont presque toujours distribuées sous forme d'archives telles que les fichiers .jar ou .war, et dans ce cas, une ressource ne sera pas un fichier distinct et ne sera pas accessible en écriture. (La méthode getFile de la classe d'URL n'est *pas* une solution de contournement pour cela; malgré son nom, elle renvoie simplement la portion de chemin d'une URL, ce qui n'est en aucun cas un nom de fichier valide.)

Il n'y a aucun moyen sûr de lister les ressources à l'exécution. Encore une fois, les développeurs étant responsables de l'ajout de fichiers de ressources à l'application au moment de la construction, les développeurs doivent déjà connaître leurs chemins. Bien qu'il existe des solutions de contournement, elles ne sont pas fiables et finiront par échouer.

Examples

Chargement d'une image à partir d'une ressource

Pour charger une image groupée:

```
package com.example;

public class ExampleApplication {
    private Image getIcon() throws IOException {
        URL imageURL = ExampleApplication.class.getResource("icon.png");
        return ImageIO.read(imageURL);
    }
}
```

Chargement de la configuration par défaut

Pour lire les propriétés de configuration par défaut:

```
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;
    }
}
```

Chargement d'une ressource de même nom à partir de plusieurs JAR

Une ressource ayant le même chemin et le même nom peut exister dans plusieurs fichiers JAR sur le chemin de classe. Les cas courants sont des ressources suivant une convention ou faisant partie d'une spécification d'emballage. Des exemples de telles ressources sont

- META-INF / MANIFEST.MF
- META-INF / beans.xml (Spéc. CDI)
- Propriétés ServiceLoader contenant des fournisseurs d'implémentation

Pour accéder à *toutes* ces ressources dans des fichiers JAR différents, il faut utiliser un ClassLoader, qui possède une méthode pour cela. L' Enumeration renvoyée peut être facilement convertie en une List utilisant une fonction Collections.

```
Enumeration<URL> resEnum = MyClass.class.getClassLoader().getResources("META-
INF/MANIFEST.MF");
ArrayList<URL> resources = Collections.list(resEnum);
```

Recherche et lecture de ressources à l'aide d'un chargeur de classe

Le chargement de ressources en Java comprend les étapes suivantes:

1. Recherche de la Class ou du ClassLoader qui trouvera la ressource.
2. Trouver la ressource.
3. Obtenir le flux d'octets pour la ressource.
4. Lecture et traitement du flux d'octets.
5. Fermer le flux d'octets.

Les trois dernières étapes sont généralement accomplies en passant l'URL à une méthode de bibliothèque ou à un constructeur pour charger la ressource. Vous utiliserez généralement une méthode getResource dans ce cas. Il est également possible de lire les données de la ressource dans le code de l'application. Vous utiliserez généralement getResourceAsStream dans ce cas.

Chemins de ressource absolus et relatifs

Les ressources pouvant être chargées à partir du *chemin de* classe sont désignées par un *chemin* . La syntaxe du chemin est similaire à un chemin de fichier UNIX / Linux. Il se compose de noms simples séparés par des barres obliques (/). Un *chemin relatif* commence par un nom et un *chemin absolu* commence par un séparateur.

Comme le décrivent les exemples de Classpath, le chemin de classe d'une machine virtuelle Java définit un espace de noms en superposant les espaces de noms des répertoires et des fichiers JAR ou ZIP dans le chemin de classes. Lorsqu'un chemin absolu est résolu, les chargeurs de classes interprètent l'initial / comme signifiant la racine de l'espace de noms. En revanche, un chemin relatif *peut être* résolu par rapport à n'importe quel "dossier" de l'espace de noms. Le dossier utilisé dépend de l'objet que vous utilisez pour résoudre le chemin.

Obtenir un cours ou un chargeur de classe

Une ressource peut être localisée à l'aide d'un objet Class ou d'un objet ClassLoader . Un objet Class peut résoudre des chemins relatifs, vous utiliserez donc généralement l'un d'entre eux si vous avez une ressource relative (de classe). Il existe plusieurs façons d'obtenir un objet Class . Par exemple:

- Un *littéral de classe* vous donnera l'objet Class pour toute classe que vous pouvez nommer dans le code source Java; Par exemple, String.class vous donne l'objet Class pour le type String .
- Object.getClass() vous donnera l'objet Class pour le type de n'importe quel objet; Par exemple, "hello".getClass() est un autre moyen d'obtenir la Class du type String .
- La Class.forName(String) chargera (si nécessaire) dynamiquement une classe et renverra son objet Class ; par exemple Class.forName("java.lang.String") .

Un objet ClassLoader est généralement obtenu en appelant getClassLoader() sur un objet Class . Il est également possible de récupérer le classloader par défaut de la machine ClassLoader.getSystemClassLoader() utilisant la méthode statique ClassLoader.getSystemClassLoader() .

Les méthodes get

Une fois que vous avez une instance Class ou ClassLoader , vous pouvez rechercher une ressource en utilisant l'une des méthodes suivantes:

Les méthodes	La description
`ClassLoader.getResource(path)` `ClassLoader.getResources(path)`	Retourne une URL qui représente l'emplacement de la ressource avec le chemin donné.
`ClassLoader.getResources(path)` `Class.getResources(path)`	Renvoie une `Enumeration<URL>` indiquant les URL pouvant être utilisées pour localiser la ressource `foo.bar` ; voir ci-dessous.
`ClassLoader.getResourceAsStream(path)` `Class.getResourceStream(path)`	Renvoie un `InputStream` partir duquel vous pouvez lire le contenu de la ressource `foo.bar` sous la forme d'une séquence d'octets.

Remarques:

- La principale différence entre les versions ClassLoader et Class des méthodes réside dans la manière dont les chemins relatifs sont interprétés.
    - Les méthodes Class résolvent un chemin d'accès relatif dans le "dossier" correspondant au package de classes.
    - Les méthodes ClassLoader traitent les chemins relatifs comme s'ils étaient absolus; c'est-à-dire les résoudre dans le "dossier racine" de l'espace de noms classpath.
- Si la ressource (ou les ressources) demandée (s) est introuvable, les methods return getResource et getResourceAsStream methods return null , and the methods return an empty getResources methods return an empty Enumeration` methods return an empty .
- Les URL renvoyées pourront être URL.toStream() aide d' URL.toStream() . Il peut s'agir d'URL de file: URL ou d'autres URL classiques, mais si la ressource réside dans un fichier JAR, il s'agira d'URL jar: identifiant le fichier JAR et une ressource spécifique.
- Si votre code utilise une méthode getResourceAsStream (ou URL.toStream()) pour obtenir un InputStream , il est responsable de la fermeture de l'objet de flux. Si vous ne fermez pas le flux, vous risquez de provoquer une fuite de ressources.

Lire Ressources (sur classpath) en ligne: https://riptutorial.com/fr/java/topic/2433/ressources--sur-classpath-

Chapitre 155: Ruisseaux

Introduction

Un Stream représente une séquence d'éléments et prend en charge différents types d'opérations pour effectuer des calculs sur ces éléments. Avec Java 8, l'interface de la Collection dispose de deux méthodes pour générer un Stream : stream() et parallelStream() . Stream opérations de Stream sont soit intermédiaires, soit terminales. Les opérations intermédiaires renvoient un Stream afin que plusieurs opérations intermédiaires puissent être chaînées avant la fermeture du Stream . Les opérations du terminal sont soit vides, soit renvoient un résultat non-flux.

Syntaxe

- collection.stream ()
- Arrays.stream (array)
- Stream.iterate (firstValue, currentValue -> nextValue)
- Stream.generate (() -> valeur)
- Stream.of (elementOfT [, elementOfT, ...])
- Stream.empty ()
- StreamSupport.stream (iterable.spliterator (), false)

Examples

Utiliser des flux

Un Stream est une séquence d'éléments sur laquelle des opérations d'agrégation séquentielles et parallèles peuvent être effectuées. Un Stream donné peut potentiellement contenir une quantité illimitée de données. En conséquence, les données reçues d'un Stream sont traitées individuellement à son arrivée, au lieu d'effectuer un traitement par lot sur les données. Lorsqu'elles sont combinées avec des expressions lambda, elles fournissent un moyen concis d'effectuer des opérations sur des séquences de données en utilisant une approche fonctionnelle.

Exemple: (voir ça marche sur 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);
```

Sortie:

```
POMME
BANANE
ORANGE
POIRE
```

Les opérations effectuées par le code ci-dessus peuvent être résumées comme suit:

1. Créez un Stream<String> contenant un Stream ordonné d'éléments de String de caractères fruit à l'aide de la méthode de fabrique statique Stream.of(values) .
2. L'opération filter() conserve uniquement les éléments correspondant à un prédicat donné (les éléments testés par le prédicat retournent true). Dans ce cas, il conserve les éléments contenant un "a" . Le prédicat est donné sous la forme d'une expression lambda .
3. L'opération map() transforme chaque élément en utilisant une fonction donnée, appelée mappeur. Dans ce cas, chaque String Fruit est mappée sur sa version String majuscule à

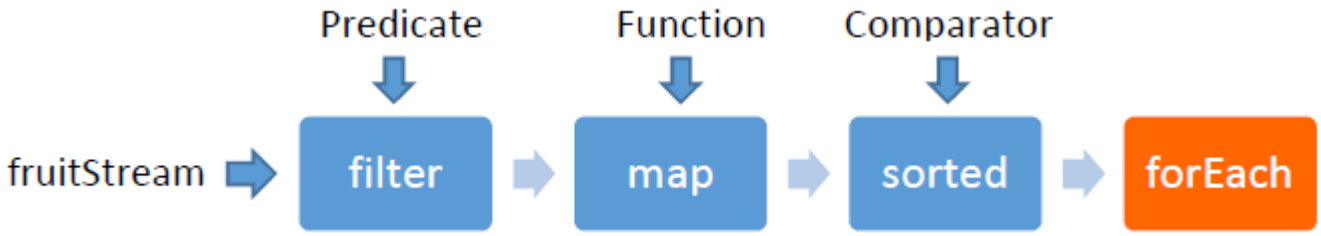
l'aide de la référence de méthode String::toUppercase .

> **Notez** que l'opération map() renverra un flux avec un type générique différent si la fonction de mappage renvoie un type différent de son paramètre d'entrée. Par exemple, sur un Stream<String> appel de .map(String::isEmpty) renvoie un Stream<Boolean>

4. L'opération sort sorted() trie les éléments du Stream fonction de leur ordre naturel (lexicographiquement, dans le cas de la String).

5. Enfin, l'opération forEach(action) exécute une action qui agit sur chaque élément du Stream , en le transmettant à un consommateur . Dans l'exemple, chaque élément est simplement imprimé sur la console. Cette opération est une opération de terminal, ce qui rend son fonctionnement impossible.

> **Notez** que les opérations définies sur le Stream sont effectuées en *raison* du fonctionnement du terminal. Sans opération de terminal, le flux n'est pas traité. Les flux ne peuvent pas être réutilisés. Une fois qu'une opération de terminal est appelée, l'objet Stream devient inutilisable.



Les opérations (comme vu ci-dessus) sont enchaînées pour former ce qui peut être vu comme une requête sur les données.

Fermeture des cours d'eau

> **Notez qu'un Stream n'a généralement pas besoin d'être fermé.** Il est seulement nécessaire de fermer les flux qui fonctionnent sur les canaux IO. La plupart des types de Stream ne fonctionnent pas sur les ressources et ne nécessitent donc pas de fermeture.

L'interface Stream étend AutoCloseable . Les flux peuvent être fermés en appelant la méthode close ou en utilisant des instructions try-with-resource.

Un exemple de cas d'utilisation où un Stream doit être fermé est lorsque vous créez un Stream de lignes à partir d'un fichier:

```
try (Stream<String> lines = Files.lines(Paths.get("somePath"))) {
    lines.forEach(System.out::println);
}
```

L'interface Stream déclare également la méthode Stream.onClose() qui vous permet d'enregistrer les gestionnaires Runnable qui seront appelés lorsque le flux sera fermé. Un exemple de cas d'utilisation est celui où le code qui produit un flux doit savoir quand il est utilisé pour effectuer un nettoyage.

```
public Stream<String>streamAndDelete(Path path) throws IOException {
    return Files.lines(path).onClose(() -> someClass.deletePath(path));
}
```

Le gestionnaire d'exécution ne s'exécutera que si la méthode close() est appelée, explicitement ou implicitement, par une instruction try-with-resources.

Commande en traitement

Le traitement d'un objet Stream peut être séquentiel ou parallèle .

En mode **séquentiel** , les éléments sont traités dans l'ordre de la source du Stream . Si le Stream est commandé (par exemple, une implémentation SortedMap ou une List), le traitement est garanti pour correspondre à l'ordre de la source. Dans d'autres cas, toutefois, il convient de ne pas dépendre de la commande (voir: l'ordre d'itération keySet() Java HashMap keySet() cohérent?).

Exemple:

```
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
```

Vivre sur Ideone

Le mode **parallèle** permet l'utilisation de plusieurs threads sur plusieurs cœurs, mais il n'y a aucune garantie sur l'ordre dans lequel les éléments sont traités.

Si plusieurs méthodes sont appelées sur un Stream séquentiel, toutes les méthodes ne doivent pas être appelées. Par exemple, si un Stream est filtré et que le nombre d'éléments est réduit à un, aucun appel ultérieur à une méthode telle que le sort ne se produira. Cela peut augmenter les performances d'un Stream séquentiel - une optimisation impossible avec un Stream parallèle.

Exemple:

```
// parallel
long howManyOddNumbersParallel = integerList.parallelStream()
                                            .filter(e -> (e % 2) == 1)
                                            .count();

System.out.println(howManyOddNumbersParallel); // Output: 2
```

Vivre sur Ideone

Différences par rapport aux conteneurs (ou aux collections)

Bien que certaines actions puissent être effectuées à la fois sur les conteneurs et les flux, elles servent en fin de compte à des objectifs différents et prennent en charge différentes opérations. Les conteneurs se concentrent davantage sur la manière dont les éléments sont stockés et sur la manière dont ces éléments peuvent être utilisés efficacement. Un Stream , d'autre part, ne fournit pas un accès et une manipulation directs à ses éléments; Il est plus dédié au groupe d'objets en tant qu'entité collective et effectue des opérations sur cette entité dans son ensemble. Stream et Collection sont des abstractions de haut niveau distinctes pour ces différents objectifs.

Collecte des éléments d'un flux dans une collection

Recueillir avec toList() et toSet()

Les éléments d'un Stream peuvent être facilement collectés dans un conteneur à l'aide de

l'opération 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]
```

D'autres instances de collection, telles qu'un Set , peuvent être créées à l'aide d'autres méthodes intégrées de Collectors . Par exemple, Collectors.toSet() collecte les éléments d'un Stream dans un Set .

Contrôle explicite de l'implémentation de List ou Set

Selon la documentation de Collectors#toList() et Collectors#toSet() , il n'ya aucune garantie sur le type, la mutabilité, la sérialisabilité ou la sécurité des threads de la List ou de l' Set renvoyés.

Pour que le contrôle explicite de l'implémentation soit renvoyé, Collectors#toCollection(Supplier) peut être utilisé à la place, où le fournisseur donné retourne une nouvelle collection vide.

```
// 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<>()))
);
```

Collecter des éléments en utilisant toMap

Le collecteur accumule des éléments dans une carte, où la clé est l'ID de l'étudiant et la valeur est la valeur de l'étudiant.

```
  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);
```

Sortie:

```
{1=test1, 2=test2, 3=test3}
```

Le Collectors.toMap a une autre implémentation Collector<T, ?, Map<K,U>> toMap(Function<? super

T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper, BinaryOperator<U> mergeFunction) .La fonction mergeFunction est principalement utilisée pour sélectionner une nouvelle valeur ou conserver l'ancienne valeur si la clé est répétée lors de l'ajout d'un nouveau membre dans la carte à partir d'une liste.

La fonction de fusion ressemble souvent à: (s1, s2) -> s1 pour conserver la valeur correspondant à la clé répétée ou (s1, s2) -> s2 pour mettre une nouvelle valeur pour la clé répétée.

Collecte des éléments à la carte des collections

Exemple: de ArrayList à mapper <String, List <>>

Souvent, il faut créer une carte de liste à partir d'une liste primaire. Exemple: À partir d'un élève de la liste, nous devons faire une carte de la liste des matières pour chaque élève.

```
    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);
```

Sortie:

```
{ Robert=[LITERATURE],
Sascha=[ENGLISH, MATH, SCIENCE, LITERATURE],
Davis=[MATH, SCIENCE, GEOGRAPHY] }
```

Exemple: de ArrayList à 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);
```

Sortie:

```
{ 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

Objectif	Code
Recueillir dans une `List`	`Collectors.toList()`
Recueillir dans une `ArrayList` avec une taille pré-allouée	`Collectors.toCollection(() -> new ArrayList<>(size))`
Recueillir à un `Set`	`Collectors.toSet()`
Recueillir dans un `Set` avec une meilleure performance d'itération	`Collectors.toCollection(() -> new LinkedHashSet<>())`
Recueillir dans un `Set<String>` insensible à la casse `Set<String>`	`Collectors.toCollection(() -> new TreeSet<>(String.CASE_INSENSITIVE_ORDER))`
Recueillir dans un `EnumSet<AnEnum>` (meilleures performances pour les énumérations)	`Collectors.toCollection(() -> EnumSet.noneOf(AnEnum.class))`
Recueillir sur une `Map<K,V>` avec des clés uniques	`Collectors.toMap(keyFunc,valFunc)`
Mappez MyObject.getter () sur un objet unique MyObject	`Collectors.toMap(MyObject::getter, Function.identity())`
Mappez MyObject.getter () sur plusieurs MyObjects	`Collectors.groupingBy(MyObject::getter)`

Streams infinis

Il est possible de générer un Stream qui ne se termine pas. L'appel d'une méthode de terminal sur un Stream infini entraîne l'entrée du Stream dans une boucle infinie. La méthode de limit d'un Stream peut être utilisée pour limiter le nombre de termes du Stream traité par Java.

Cet exemple génère un Stream de tous les nombres naturels, en commençant par le nombre 1. Chaque terme successif du Stream est supérieur à celui précédent. En appelant la méthode des limites de ce Stream , seuls les cinq premiers termes du Stream sont pris en compte et imprimés.

```
// 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);
```

Sortie:

```
1
2
3
4
5
```

Une autre façon de générer un flux infini consiste à utiliser la méthode Stream.generate . Cette méthode prend un lambda de type fournisseur .

```
// 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);
```

Consommer des flux

Un Stream ne sera parcouru que s'il y a une *opération de terminal* , comme count() , collect() ou forEach() . Sinon, aucune opération sur le Stream ne sera effectuée.

Dans l'exemple suivant, aucune opération de terminal n'est ajoutée au Stream . Par conséquent, l'opération filter() ne sera pas appelée et aucune sortie ne sera produite car peek() N'EST PAS une *opération de terminal* .

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).peek(System.out::println);
```

Vivre sur Ideone

Ceci est une séquence de Stream avec une *opération de terminal* valide, donc une sortie est produite.

Vous pouvez également utiliser forEach au lieu de peek :

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).forEach(System.out::println);
```

Vivre sur Ideone

Sortie:

```
2
4
6
8
```

Une fois l'opération du terminal effectuée, le Stream est consommé et ne peut pas être réutilisé.

Bien qu'un objet flux donné ne peut pas être réutilisé, il est facile de créer un réutilisable Iterable que les délégués à un pipeline de flux. Cela peut être utile pour renvoyer une vue modifiée d'un ensemble de données en direct sans avoir à collecter les résultats dans une

structure temporaire.

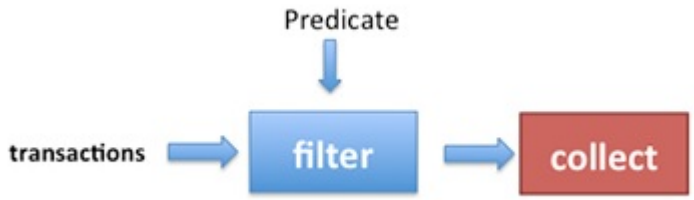
```
List<String> list = Arrays.asList("FOO", "BAR");
Iterable<String> iterable = () -> list.stream().map(String::toLowerCase).iterator();

for (String str : iterable) {
    System.out.println(str);
}
for (String str : iterable) {
    System.out.println(str);
}
```

Sortie:

```
foo
bar
foo
bar
```

Cela fonctionne car Iterable déclare une seule méthode abstraite Iterator<T> iterator() . Cela en fait une interface fonctionnelle, implémentée par un lambda qui crée un nouveau flux à chaque appel.

En général, un Stream fonctionne comme indiqué dans l'image suivante:



REMARQUE : Les vérifications d'argument sont toujours effectuées, même sans *opération de terminal* :

```
try {
    IntStream.range(1, 10).filter(null);
} catch (NullPointerException e) {
    System.out.println("We got a NullPointerException as null was passed as an argument to
filter()");
}
```

Vivre sur Ideone

Sortie:

```
Nous avons une exception NullPointerException car null a été passé en argument à
filter ()
```

Créer une carte de fréquence

Le collecteur groupingBy(classifier, downstream) permet la collecte d'éléments Stream dans une Map en classant chaque élément dans un groupe et en effectuant une opération en aval sur les éléments classés dans le même groupe.

Un exemple classique de ce principe consiste à utiliser une Map pour compter les occurrences d'éléments dans un Stream . Dans cet exemple, le classificateur est simplement la fonction d'identité, qui renvoie l'élément tel quel. L'opération en aval compte le nombre d'éléments

égaux, en utilisant le counting() .

```
Stream.of("apple", "orange", "banana", "apple")
      .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
      .entrySet()
      .forEach(System.out::println);
```

L'opération en aval est elle-même un collecteur (Collectors.counting()) qui opère sur des éléments de type String et produit un résultat de type Long . Le résultat de l'appel de la méthode de collect est un Map<String, Long> .

Cela produirait la sortie suivante:

```
banane = 1
orange = 1
apple = 2
```

Flux parallèle

Remarque: Avant de décider quel Stream utiliser, consultez le comportement ParallelStream vs Sequential Stream .

Lorsque vous souhaitez effectuer des opérations de Stream simultanément, vous pouvez utiliser l'une de ces méthodes.

```
List<String> data = Arrays.asList("One", "Two", "Three", "Four", "Five");
Stream<String> aParallelStream = data.stream().parallel();
```

Ou:

```
Stream<String> aParallelStream = data.parallelStream();
```

Pour exécuter les opérations définies pour le flux parallèle, appelez un opérateur de terminal:

```
aParallelStream.forEach(System.out::println);
```

(A possible) sortie du Stream parallèle:

```
Trois
Quatre
Un
Deux
Cinq
```

L'ordre peut changer car tous les éléments sont traités en parallèle (ce qui *peut le* rendre plus rapide). Utilisez parallelStream lorsque la commande n'a pas d'importance.

Impact sur la performance

Dans le cas où la mise en réseau est impliquée, les Stream parallèles peuvent dégrader les performances globales d'une application car tous les Stream parallèles utilisent un pool de threads de jointure commune pour le réseau.

D'autre part, les Stream parallèles peuvent améliorer considérablement les performances dans de nombreux autres cas, en fonction du nombre de cœurs disponibles dans le processeur en cours d'exécution.

Conversion d'un flux de données facultatif en un flux de valeurs

Vous devrez peut-être convertir un Stream émetteur Optional en un Stream de valeurs, en n'émettant que des valeurs de Optional existant. (c.-à-d. sans valeur null et ne pas traiter avec 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]
```

Créer un flux

Tous les java Collection<E> ont des méthodes stream() et parallelStream() partir desquelles un Stream<E> peut être construit:

```
Collection<String> stringList = new ArrayList<>();
Stream<String> stringStream = stringList.parallelStream();
```

Un Stream<E> peut être créé à partir d'un tableau en utilisant l'une des deux méthodes suivantes:

```
String[] values = { "aaa", "bbbb", "ddd", "cccc" };
Stream<String> stringStream = Arrays.stream(values);
Stream<String> stringStreamAlternative = Stream.of(values);
```

La différence entre Arrays.stream() et Stream.of() est que Stream.of() a un paramètre varargs, donc il peut être utilisé comme:

```
Stream<Integer> integerStream = Stream.of(1, 2, 3);
```

Il existe également des Stream primitifs que vous pouvez utiliser. Par exemple:

```
IntStream intStream = IntStream.of(1, 2, 3);
DoubleStream doubleStream = DoubleStream.of(1.0, 2.0, 3.0);
```

Ces flux primitifs peuvent également être construits à l'aide de la méthode Arrays.stream() :

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

Il est possible de créer un Stream partir d'un tableau avec une plage spécifiée.

```
int[] values= new int[]{1, 2, 3, 4, 5};
IntStream intStram = Arrays.stream(values, 1, 3);
```

Notez que tout flux primitif peut être converti en flux de type boîte en utilisant la méthode des boxed :

```
Stream<Integer> integerStream = intStream.boxed();
```

Cela peut être utile dans certains cas si vous souhaitez collecter les données, car le flux primitif ne possède aucune méthode de collect prenant un Collector comme argument.

Réutilisation des opérations intermédiaires d'une chaîne de flux

Le flux est fermé lorsque le terminal est appelé. Réutiliser le flux d'opérations intermédiaires, lorsque seul le fonctionnement du terminal ne fait que varier. nous pourrions créer un fournisseur de flux pour construire un nouveau flux avec toutes les opérations intermédiaires déjà configurées.

```
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[] tableaux int[] peuvent être convertis en List<Integer> aide de flux

```
int[] ints = {1,2,3};
List<Integer> list = IntStream.of(ints).boxed().collect(Collectors.toList());
```

Recherche de statistiques sur les flux numériques

Java 8 fournit des classes appelées IntSummaryStatistics , DoubleSummaryStatistics et LongSummaryStatistics qui fournissent un objet d'état pour collecter des statistiques telles que count , min , max , sum et 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);
```

Ce qui entraînera:

Java SE 8

```
IntSummaryStatistics{count=10, sum=55, min=1, max=10, average=5.500000}
```

Obtenir une tranche d'un flux

Exemple: Obtenez un Stream de 30 éléments, contenant du 21ème au 50ème élément (inclus) d'une collection.

```
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);
```

Remarques:

- IllegalArgumentException est IllegalArgumentException si n est négatif ou maxSize négatif
- skip(long) et limit(long) sont des opérations intermédiaires
- si un flux contient moins de n éléments, alors skip(n) renvoie un flux vide
- skip(long) et limit(long) sont des opérations peu coûteuses sur des pipelines de flux séquentiels, mais peuvent coûter très cher sur des pipelines parallèles ordonnés

Concatenate Streams

Déclaration de variable pour des exemples:

```
Collection<String> abc = Arrays.asList("a", "b", "c");
Collection<String> digits = Arrays.asList("1", "2", "3");
Collection<String> greekAbc = Arrays.asList("alpha", "beta", "gamma");
```

Exemple 1 - Concaténer deux Stream s

```
final Stream<String> concat1 = Stream.concat(abc.stream(), digits.stream());

concat1.forEach(System.out::print);
// prints: abc123
```

Exemple 2 - Concaténer plus de deux 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
```

Alternativement, pour simplifier la concat() imbriquée de concat() les Stream peuvent également être concaténés avec 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
```

Soyez prudent lors de la construction de Stream s à partir de concaténations répétées, car l'accès à un élément d'un Stream profondément concaténé peut entraîner des chaînes d'appel profondes ou même une StackOverflowException .

IntStream en chaîne

Java ne possède pas de *flux de caractères* , donc lorsque vous travaillez avec des String et que vous construisez un Stream de Character , une option consiste à obtenir un IntStream de points de code en utilisant la méthode String.codePoints() . Donc, IntStream peut être obtenu comme ci-dessous:

```
public IntStream stringToIntStream(String in) {
  return in.codePoints();
}
```

Il est un peu plus compliqué de faire la conversion autrement que IntStreamToString. Cela peut se faire comme suit:

```
public String intStreamToString(IntStream intStream) {
  return intStream.collect(StringBuilder::new, StringBuilder::appendCodePoint,
StringBuilder::append).toString();
}
```

Trier avec 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);
```

Sortie:

```
[Sydney, London, New York, Amsterdam, Mumbai, California]
[Amsterdam, California, London, Mumbai, New York, Sydney]
```

Il est également possible d'utiliser un mécanisme de comparaison différent car il existe une version sorted surchargée qui prend un comparateur comme argument.

En outre, vous pouvez utiliser une expression lambda pour le tri:

```
List<String> sortedData2 = data.stream().sorted((s1,s2) ->
s2.compareTo(s1)).collect(Collectors.toList());
```

Cela produirait [Sydney, New York, Mumbai, London, California, Amsterdam]

Vous pouvez utiliser Comparator.reverseOrder() pour avoir un comparateur qui impose l' reverse de l'ordre naturel.

```
List<String> reverseSortedData =
data.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
```

Flux de primitifs

Java fournit des Stream spécialisés pour trois types de primitives IntStream (for int s), LongStream (for s long) et DoubleStream (for double s). En plus d'être des implémentations optimisées pour leurs primitives respectives, elles fournissent également plusieurs méthodes de terminal spécifiques, généralement pour des opérations mathématiques. Par exemple:

```
IntStream is = IntStream.of(10, 20, 30);
double average = is.average().getAsDouble(); // average is 20.0
```

Recueillir les résultats d'un flux dans un tableau

Analogue pour obtenir une collection pour un Stream par collect() un tableau peut être obtenu par la méthode 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 est un type spécial de référence de méthode: une référence de constructeur.

Trouver le premier élément qui correspond à un prédicat

Il est possible de trouver le premier élément d'un Stream correspondant à une condition.

Pour cet exemple, nous trouverons le premier Integer dont le carré est supérieur à 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
```

Cette expression retournera un OptionalInt avec le résultat.

Notez qu'avec un Stream infini, Java continuera à vérifier chaque élément jusqu'à ce qu'il trouve un résultat. Avec un Stream fini, si Java est à court d'éléments mais ne peut toujours pas trouver de résultat, il retourne un OptionalInt vide.

Utiliser IntStream pour itérer sur les index

Stream d'éléments ne permettent généralement pas d'accéder à la valeur d'index de l'élément en cours. Pour parcourir un tableau ou ArrayList tout en ayant accès aux index, utilisez 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);
```

La méthode `range(start, endExclusive)` renvoie un autre `ÌntStream` et `mapToObj(mapper)` renvoie un flux de String .

Sortie:

```
# 1 Jon
# 2 Darin
# 3 Bauke
# 4 Hans
# 5 Marc
```

Ceci est très similaire à l' aide d' une normale for la boucle avec un compteur, mais avec l'avantage de pipelining et parallélisation:

```
for (int i = 0; i < names.length; i++) {
    String newName = String.format("#%d %s", i + 1, names[i]);
    System.out.println(newName);
```

```
}
```

Aplatir les Streams avec flatMap ()

Un Stream d'éléments pouvant être à leur tour diffusés peut être mis à plat en un seul Stream continu:

Un tableau de liste d'éléments peut être converti en une seule liste.

```
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]
```

Une carte contenant une liste d'éléments en tant que valeurs peut être aplatie en une liste combinée

```
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 de Map peut être mise à plat en un seul Stream continu

```
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]
```

Créer une carte basée sur un flux

Cas simple sans clés en double

```
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}
```

Pour rendre les choses plus déclaratives, nous pouvons utiliser la méthode statique dans Function interface de Function.identity() - Function.identity() . Nous pouvons remplacer cet element -> element lambda element -> element par Function.identity() .

Cas où il pourrait y avoir des clés en double

Le javadoc pour Collectors.toMap indique:

> Si les clés mis en correspondance contient des doublons (selon Object.equals(Object)), une IllegalStateException est levée lorsque l'opération de collecte est effectuée. Si les clés mappées peuvent avoir des doublons, utilisez 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}
```

Le BinaryOperator transmis à Collectors.toMap(...) génère la valeur à stocker en cas de collision. Ça peut:

- renvoyer l'ancienne valeur, de sorte que la première valeur du flux prenne le pas,
- renvoyer la nouvelle valeur, de sorte que la dernière valeur du flux soit prioritaire, ou
- combiner les anciennes et les nouvelles valeurs

Regroupement par valeur

Vous pouvez utiliser Collectors.groupingBy lorsque vous devez effectuer l'équivalent d'une opération "group by" en cascade dans une base de données. Pour illustrer cela, ce qui suit crée une carte dans laquelle les noms des personnes sont mappés aux noms de famille:

```
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]}
```

Vivre sur Ideone

Génération de chaînes aléatoires à l'aide de flux

Il est parfois utile de créer des Strings aléatoires, peut-être comme ID de session pour un service Web ou un mot de passe initial après l'inscription à une application. Cela peut être facilement réalisé en utilisant les Stream .

Nous devons d'abord initialiser un générateur de nombres aléatoires. Pour améliorer la sécurité des String générées, il est SecureRandom utiliser SecureRandom .

Remarque : la création d'un SecureRandom est assez onéreuse, il est donc SecureRandom de ne le faire qu'une seule fois et d'appeler de temps en temps l'une de ses méthodes setSeed() pour le réamorcer.

```
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
```

Lors de la création de String aléatoires, nous souhaitons généralement qu'elles utilisent uniquement certains caractères (par exemple, uniquement des lettres et des chiffres). Nous pouvons donc créer une méthode renvoyant un boolean qui pourra ensuite être utilisé pour filtrer le 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);
}
```

Ensuite, nous pouvons utiliser le RNG pour générer une chaîne aléatoire de longueur spécifique contenant le jeu de caractères qui passe notre vérification 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;
}
```

Utilisation de flux pour implémenter des fonctions mathématiques

Stream , et en particulier IntStream , constituent un moyen élégant d'implémenter des termes de sommation (∑). Les plages du Stream peuvent être utilisées comme limites de la somme.

Par exemple, l'approximation de Pi par Madhava est donnée par la formule (Source: wikipedia):

$$\pi = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-3)^{-k}}{2k+1} = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-\frac{1}{3})^{k}}{2k+1} = \sqrt{12}\left(\frac{1}{1\cdot 3^0} - \frac{1}{3\cdot 3^1} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots\right)$$

Cela peut être calculé avec une précision arbitraire. Par exemple, pour 101 termes:

```
double pi = Math.sqrt(12) *
            IntStream.rangeClosed(0, 100)
                     .mapToDouble(k -> Math.pow(-3, -1 * k) / (2 * k + 1))
                     .sum();
```

Note: Avec la précision du double , sélectionner une limite supérieure de 29 est suffisant pour obtenir un résultat indiscernable de Math.Pi

Utilisation de références de flux et de méthode pour écrire des processus auto-documentés

Les références de méthode constituent un excellent code auto-documenté, et l'utilisation de références de méthode avec Stream facilite la lisibilité et la compréhension des processus complexes. Considérez le code suivant:

```
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;
}
```

Cette dernière méthode réécrite à l'aide de Stream et de références de méthode est beaucoup plus lisible et chaque étape du processus est rapidement et facilement comprise - elle n'est pas seulement plus courte, elle montre également les interfaces et les classes responsables du code dans chaque étape:

```
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());
}
```

Utilisation de flux de Map.Entry pour conserver les valeurs initiales après le mappage

Lorsque vous devez mapper un Stream mais que vous souhaitez également conserver les valeurs initiales, vous pouvez mapper le Stream vers un Map.Entry<K,V> utilisant une méthode d'utilitaire comme celle-ci:

```
public static <K, V> Function<K, Map.Entry<K, V>> entryMapper(Function<K, V> mapper){
    return (k)->new AbstractMap.SimpleEntry<>(k, mapper.apply(k));
}
```

Vous pouvez ensuite utiliser votre convertisseur pour traiter les Stream ayant accès aux valeurs d'origine et mappées:

```
Set<K> mySet;
Function<K, V> transformer = SomeClass::transformerMethod;
Stream<Map.Entry<K, V>> entryStream = mySet.stream()
    .map(entryMapper(transformer));
```

Vous pouvez ensuite continuer à traiter ce Stream comme d'habitude. Cela évite de créer une collection intermédiaire.

Catégories d'opérations de flux

Les opérations de flux se répartissent en deux catégories principales, les opérations intermédiaires et terminales, et deux sous-catégories, sans état et avec état.

Opérations intermédiaires:

Une opération intermédiaire est toujours *paresseuse* , telle qu'une simple Stream.map . Il n'est pas appelé tant que le flux n'est pas réellement consommé. Cela peut être vérifié facilement:

```
Arrays.asList(1, 2 ,3).stream().map(i -> {
    throw new RuntimeException("not gonna happen");
    return i;
});
```

Les opérations intermédiaires sont les blocs de construction communs d'un flux, chaînés après la source et sont généralement suivis d'une opération de terminal déclenchant la chaîne de flux.

Opérations Terminal

Les opérations terminales sont ce qui déclenche la consommation d'un flux. Les plus courants sont Stream.forEach ou Stream.collect . Ils sont généralement placés après une chaîne d'opérations intermédiaires et sont presque toujours *impatients* .

Opérations apatrides

L'apatridie signifie que chaque élément est traité sans le contexte des autres éléments. Les opérations sans état permettent un traitement efficace des flux dans la mémoire. Les opérations telles que Stream.map et Stream.filter ne nécessitant pas d'informations sur d'autres éléments du flux sont considérées comme étant sans état.

Opérations avec état

Statefulness signifie que l'opération sur chaque élément dépend de (certains) autres éléments du flux. Cela nécessite un état à préserver. Les opérations d'état peuvent se rompre avec des flux longs ou infinis. Les opérations telles que Stream.sorted requièrent que l'intégralité du flux soit traitée avant que tout élément ne soit émis, ce qui entraînera un flux d'éléments suffisamment long. Cela peut être démontré par un long flux (**exécuté à vos risques et périls**):

```
// works - stateless stream
long BIG_ENOUGH_NUMBER = 999999999;
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).forEach(System.out::println);
```

Cela provoquera un manque de mémoire dû à l'état de Stream.sorted :

```
// Out of memory - stateful stream
IntStream.iterate(0, i -> i +
1).limit(BIG_ENOUGH_NUMBER).sorted().forEach(System.out::println);
```

Conversion d'un itérateur en flux

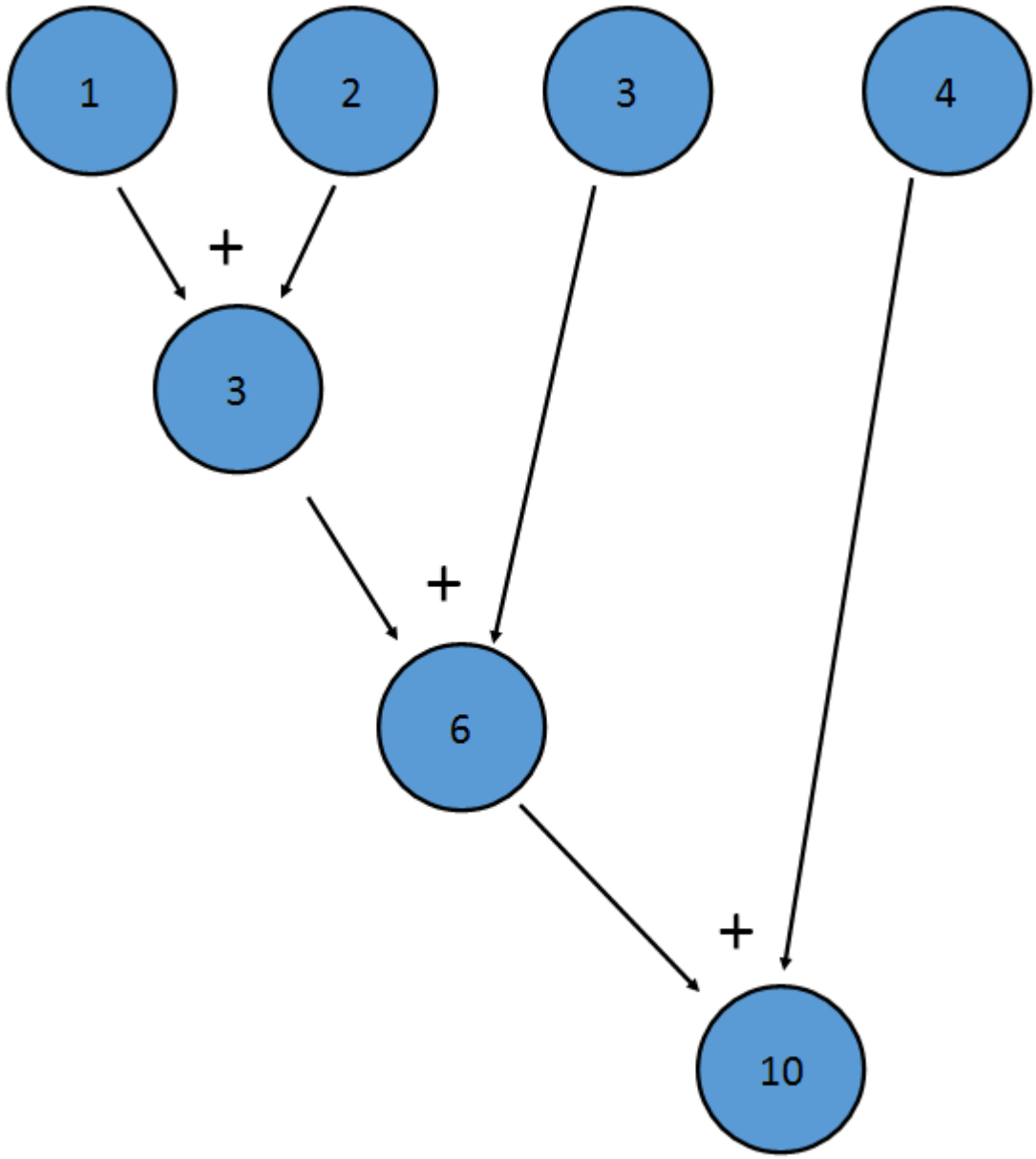
Utilisez Spliterators.spliterator() ou Spliterators.spliteratorUnknownSize() pour convertir un itérateur en flux:

```
Iterator<String> iterator = Arrays.asList("A", "B", "C").iterator();
Spliterator<String> spliterator = Spliterators.spliteratorUnknownSize(iterator, 0);
Stream<String> stream = StreamSupport.stream(spliterator, false);
```

Réduction avec des flux

La réduction est le processus consistant à appliquer un opérateur binaire à chaque élément d'un flux pour obtenir une valeur.

La méthode sum() d'un IntStream est un exemple de réduction; il applique une addition à chaque terme du Stream, résultant en une valeur finale:



Ceci est équivalent à (((1+2)+3)+4)

La méthode de reduce d'un flux permet de créer une réduction personnalisée. Il est possible d'utiliser la méthode reduce pour implémenter la méthode sum() :

```
IntStream istr;

//Initialize istr

OptionalInt istr.reduce((a,b)->a+b);
```

La version Optional est renvoyée afin que les flux vides puissent être traités de manière appropriée.

Un autre exemple de réduction consiste à combiner un Stream<LinkedList<T>> en un seul LinkedList<T> :

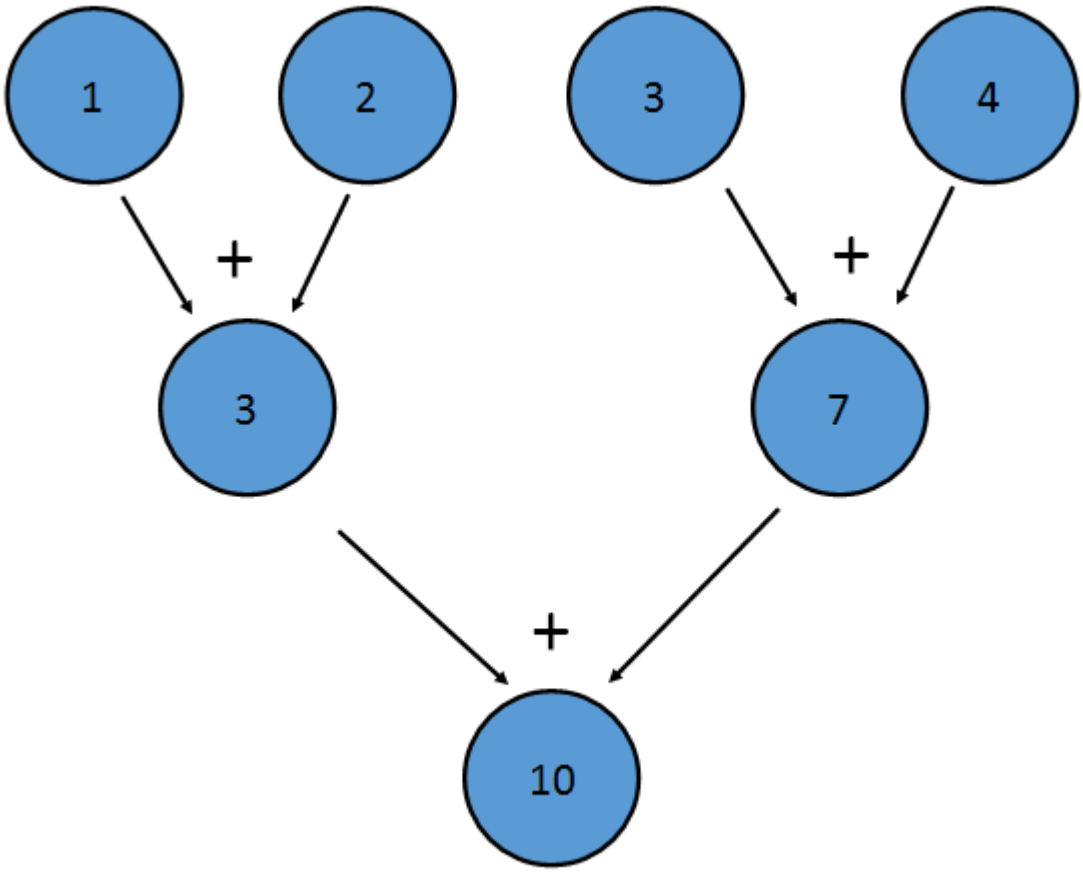
```
Stream<LinkedList<T>> listStream;

//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;
});
```

Vous pouvez également fournir un *élément d'identité* . Par exemple, l'élément d'identité pour l'addition est 0, comme x+0==x . Pour la multiplication, l'élément d'identité est 1, comme x*1==x . Dans le cas ci-dessus, l'élément identity est une LinkedList<T> vide LinkedList<T> , car si vous ajoutez une liste vide à une autre liste, la liste à laquelle vous "ajoutez" ne change pas:

```
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;
});
```

Notez que lorsqu'un élément d'identité est fourni, la valeur de retour n'est pas encapsulée dans un élément Optional -si appelé sur un flux vide, reduce() renvoie l'élément d'identité.

L'opérateur binaire doit également être *associatif* , ce qui signifie que (a+b)+c==a+(b+c) . C'est parce que les éléments peuvent être réduits dans n'importe quel ordre. Par exemple, la réduction d'addition ci-dessus peut être effectuée comme suit:



Cette réduction équivaut à écrire ((1+2)+(3+4)) . La propriété d'associativité permet également à Java de réduire le Stream en parallèle. Une partie du flux peut être réduite par chaque processeur, avec une réduction combinant le résultat de chaque processeur à la fin.

Joindre un flux à une seule chaîne

Un cas d'utilisation fréquemment rencontré consiste à créer une String partir d'un flux, où les éléments de flux sont séparés par un certain caractère. La méthode Collectors.joining() peut être utilisée pour cela, comme dans l'exemple suivant:

```
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);
```

Sortie:

```
     POMME, BANANE, ORANGE, POIRE
```

La méthode Collectors.joining() peut également prendre en charge les pré-et postfixes:

```
String result = fruitStream.filter(s -> s.contains("e"))
           .map(String::toUpperCase)
           .sorted()
```

```
            .collect(Collectors.joining(", ", "Fruits: ", "."));

System.out.println(result);
```

Sortie:

```
     Fruits: POMME, ORANGE, POIRE.
```

Vivre sur Ideone

Lire Ruisseaux en ligne: https://riptutorial.com/fr/java/topic/88/ruisseaux

Chapitre 156: Scanner

Syntaxe

- Scanner scanner = nouveau scanner (source source);
- Scanner scanner = nouveau scanner (System.in);

Paramètres

Paramètre	Détails
La source	La source peut être une chaîne, un fichier ou n'importe quel type de flux d'entrée.

Remarques

La classe Scanner été introduite dans Java 5. La méthode reset() été ajoutée à Java 6 et deux nouveaux constructeurs ont été ajoutés dans Java 7 pour assurer l'interopérabilité avec la nouvelle interface Path .

Examples

Lecture du système à l'aide du 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'objet scanner est initialisé pour lire les entrées du clavier. Donc, pour l'entrée ci-dessous de keyboar, il produira la sortie en Reading from keyboard

```
Reading
from
keyboard
  //space
```

Lecture de l'entrée de fichier à l'aide de 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();
}
```

Ici, un objet Scanner est créé en transmettant un objet File contenant le nom d'un fichier texte en entrée. Ce fichier texte sera ouvert par l'objet File et lu par l'objet scanner dans les lignes suivantes. scanner.hasNext() vérifiera s'il existe une ligne de données suivante dans le fichier texte. La combinaison de cela avec un while en boucle vous permettra de itérer chaque ligne de données dans le Names.txt fichier. Pour récupérer les données elles-mêmes, nous pouvons utiliser des méthodes telles que nextLine() , nextInt() , nextBoolean() , etc. Dans l'exemple ci-dessus, scanner.nextLine() est utilisé. nextLine() fait référence à la ligne suivante dans un fichier texte et sa combinaison avec un objet scanner vous permet d'imprimer le contenu de la ligne. Pour fermer un objet scanner, vous devez utiliser .close() .

En utilisant try with resources (à partir de Java 7), le code mentionné ci-dessus peut être écrit de manière élégante comme ci-dessous.

```
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!");
}
```

Lire l'intégralité de l'entrée sous forme de chaîne à l'aide de Scanner

Vous pouvez utiliser Scanner pour lire tout le texte dans l'entrée en tant que chaîne, en utilisant \Z (entrée entière) comme délimiteur. Par exemple, cela peut être utilisé pour lire tout le texte d'un fichier texte sur une seule ligne:

```
String content = new Scanner(new File("filename")).useDelimiter("\\Z").next();
System.out.println(content);
```

Rappelez-vous que vous devrez fermer le scanner, ainsi que le IoException cela peut IoException , comme décrit dans l'exemple Lecture de l'entrée de fichier à l'aide de Scanner .

Utilisation de délimiteurs personnalisés

Vous pouvez utiliser des délimiteurs personnalisés (expressions régulières) avec Scanner, avec .useDelimiter(",") , pour déterminer la manière dont l'entrée est lue. Cela fonctionne de la même manière que String.split(...) . Par exemple, vous pouvez utiliser Scanner pour lire une liste de valeurs séparées par des virgules dans une chaîne:

```
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();
}
```

Cela vous permettra de lire chaque élément de l'entrée individuellement. Notez que vous ne devez **pas l'** utiliser pour analyser les données CSV, utilisez plutôt une bibliothèque d'analyseur CSV

appropriée, voir l' analyseur CSV pour Java pour d'autres possibilités.

Modèle général qui pose le plus souvent des questions sur les tâches

La procédure suivante explique comment utiliser correctement la classe java.util.Scanner pour lire de manière interactive les entrées utilisateur de System.in (parfois appelées stdin , notamment en C, C ++ et autres langages ainsi que sous Unix et Linux). Il montre automatiquement les choses les plus courantes qui doivent être effectuées.

```
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);
    }
}
```

Lire un int depuis la ligne de commande

```
import java.util.Scanner;

Scanner s = new Scanner(System.in);
int number = s.nextInt();
```

Si vous souhaitez lire un int à partir de la ligne de commande, utilisez simplement cet extrait. Tout d'abord, vous devez créer un objet Scanner, qui écoute System.in, qui est par défaut la ligne de commande, lorsque vous démarrez le programme à partir de la ligne de commande. Après cela, à l'aide de l'objet Scanner, vous lisez le premier int que l'utilisateur passe dans la ligne de commande et le stockez dans le numéro de variable. Maintenant, vous pouvez faire ce que vous voulez avec ce stocké.

Fermer soigneusement un scanner

il peut arriver que vous utilisiez un scanner avec le paramètre System.in en tant que paramètre pour le constructeur, alors vous devez être conscient que la fermeture du scanner fermera le InputStream en donnant la prochaine fois que chaque tentative de lecture de l'entrée sur ce dernier (ou tout autre objet scanner) lancera une java.util.NoSuchElementException ou une java.lang.IllegalStateException

Exemple:

```
    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();
```

Lire Scanner en ligne: https://riptutorial.com/fr/java/topic/551/scanner

Chapitre 157: Sécurité et cryptographie

Examples

Calculer les hachages cryptographiques

Calculer les hachages de blocs de données relativement petits en utilisant différents algorithmes:

```
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)));
```

Produit cette sortie:

```
MD5    hash: E99E768582F6DD5A3BA2D9C849DF736E
SHA1   hash: 0135FAA6323685BA8A8FF8D3F955F0C36949D8FB
SHA256 hash: 8D35C97BCD902B96D1B551741BBE8A7F50BB5A690B4D0225482EAA63DBFB9DED
```

Des algorithmes supplémentaires peuvent être disponibles en fonction de votre implémentation de la plate-forme Java.

Générer des données aléatoires cryptographiques

Pour générer des échantillons de données cryptographiquement aléatoires:

```
final byte[] sample = new byte[16];

new SecureRandom().nextBytes(sample);

System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

Produit une sortie similaire à:

```
Sample: E4F14CEA2384F70B706B53A6DF8C5EFE
```

Notez que l'appel à nextBytes() peut bloquer pendant que l'entropie est recueillie en fonction de l'algorithme utilisé.

Pour spécifier l'algorithme et le fournisseur:

```
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));
```

Produit une sortie similaire à:

```
Provider: SUN version 1.8
Algorithm: SHA1PRNG
Sample: C80C44BAEB352FD29FBBE20489E4C0B9
```

Générer des paires de clés publiques / privées

Pour générer des paires de clés en utilisant différents algorithmes et tailles de clé:

```
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();
```

Des algorithmes et tailles de clé supplémentaires peuvent être disponibles sur votre implémentation de la plate-forme Java.

Pour spécifier une source de caractère aléatoire à utiliser lors de la génération des clés:

```
final KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");

generator.initialize(2048, SecureRandom.getInstance("SHA1PRNG", "SUN"));

final KeyPair pair = generator.generateKeyPair();
```

Calculer et vérifier les signatures numériques

Pour calculer une signature:

```
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();
```

Notez que l'algorithme de signature doit être compatible avec l'algorithme utilisé pour générer la paire de clés.

Pour vérifier une signature:

```
final PublicKey publicKey = keyPair.getPublic();
final Signature verifier = Signature.getInstance("SHA1withRSA");

verifier.initVerify(publicKey);
verifier.update(data);
```

```
System.out.println("Signature: " + verifier.verify(signature));
```

Produit cette sortie:

```
Signature: true
```

Crypter et déchiffrer des données avec des clés publiques / privées

Pour chiffrer des données avec une clé publique:

```
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));
```

Produit une sortie similaire à:

```
Message: Hello
Encrypted: 5641FBB9558ECFA9ED...
```

Notez que lors de la création de l'objet Cipher , vous devez spécifier une transformation compatible avec le type de clé utilisé. (Voir Noms des algorithmes standard JCA pour obtenir une liste des transformations prises en charge.). Pour les données de chiffrement RSA, la longueur du message.getBytes() doit être inférieure à la taille de la clé. Voir ce SO Réponse pour les détails.

Pour déchiffrer les données:

```
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);
```

Produit la sortie suivante:

```
Decrypted: Hello
```

Lire Sécurité et cryptographie en ligne: https://riptutorial.com/fr/java/topic/7529/securite-et-cryptographie

Chapitre 158: Sécurité et cryptographie

Introduction

Les pratiques de sécurité en Java peuvent être séparées en deux grandes catégories définies de manière vague. Sécurité de la plate-forme Java et programmation Java sécurisée.

Les pratiques de sécurité de la plate-forme Java traitent de la gestion de la sécurité et de l'intégrité de la JVM. Il comprend des sujets tels que la gestion des fournisseurs JCE et les stratégies de sécurité.

Les pratiques de programmation Java sécurisées concernent les meilleurs moyens d'écrire des programmes Java sécurisés. Il inclut des sujets tels que l'utilisation de nombres aléatoires et la cryptographie, ainsi que la prévention des vulnérabilités.

Remarques

Bien que des exemples doivent être clairement établis, certains sujets doivent être couverts:

1. Le concept / la structure du fournisseur JCE
2. Élément de la liste

Examples

Le JCE

JCE (Java Cryptography Extension) est une infrastructure intégrée à la JVM qui permet aux développeurs d'utiliser facilement et en toute sécurité la cryptographie dans leurs programmes. Pour ce faire, il fournit une interface simple et portable aux programmeurs, tout en utilisant un système de fournisseurs JCE pour implémenter de manière sécurisée les opérations cryptographiques sous-jacentes.

Gestion des clés et des clés

Alors que JCE sécurise les opérations cryptographiques et la génération de clés, il appartient au développeur de gérer ses clés. Plus d'informations doivent être fournies ici.

L'une des meilleures pratiques couramment acceptées pour gérer les clés à l'exécution consiste à les stocker uniquement sous byte tableaux d' byte et jamais en tant que chaînes. Ceci est dû au fait que les chaînes Java sont immuables et ne peuvent pas être "effacées" ou "mises à zéro" manuellement en mémoire. Bien qu'une référence à une chaîne puisse être supprimée, la chaîne exacte restera en mémoire jusqu'à ce que son segment de mémoire soit récupéré et réutilisé. Un attaquant aurait une grande fenêtre dans laquelle ils pourraient vider la mémoire du programme et trouver facilement la clé. Au contraire, les tableaux d' byte sont mutables et leur contenu peut être écrasé en place; C'est une bonne idée de "mettre à zéro" vos clés dès que vous n'en avez plus besoin.

Vulnérabilités Java courantes

Besoin de contenu

Problèmes de réseautage

Besoin de contenu

Aléatoire et vous

Besoin de contenu

Pour la plupart des applications, la classe java.utils.Random est une source parfaitement fine

de données "aléatoires". Si vous devez choisir un élément aléatoire dans un tableau ou générer une chaîne aléatoire ou créer un identifiant "unique" temporaire, vous devez probablement utiliser Random .

Cependant, de nombreux systèmes cryptographiques reposent sur le caractère aléatoire pour leur sécurité, et le caractère aléatoire fourni par Random n'est tout simplement pas de qualité suffisante. Pour toute opération cryptographique nécessitant une entrée aléatoire, utilisez plutôt SecureRandom .

Hachage et validation

Plus d'informations nécessaires

Une fonction de hachage cryptographique fait partie d'une classe de fonctions possédant trois propriétés essentielles. cohérence, unicité et irréversibilité.

Consistance: Étant donné les mêmes données, une fonction de hachage renverra toujours la même valeur. C'est-à-dire que si X = Y, f (x) sera toujours égal à f (y) pour la fonction de hachage f.

Unicité: Deux entrées dans une fonction de hachage ne produiront jamais la même sortie. C'est-à-dire que si X! = Y, f (x)! = F (y), pour toute valeur de X et Y.

Irréversibilité: Il est difficile, voire impossible, d'inverser une fonction de hachage. C'est-à-dire que, étant donné seulement f (X), il ne devrait y avoir aucun moyen de trouver le X originel sans mettre toutes les valeurs possibles de X à travers la fonction f (force brute). Il ne devrait y avoir aucune fonction f1 telle que f1 (f (X)) = X.

De nombreuses fonctions ne possèdent pas au moins un de ces attributs. Par exemple, MD5 et SHA1 sont connus pour avoir des collisions, c'est-à-dire deux entrées ayant la même sortie, elles manquent donc d'unicité. Certaines fonctions actuellement considérées comme sécurisées sont SHA-256 et SHA-512.

Lire Sécurité et cryptographie en ligne: https://riptutorial.com/fr/java/topic/9371/securite-et-cryptographie

Chapitre 159: Service d'impression Java

Introduction

L' API Java Print Service fournit des fonctionnalités permettant de découvrir les services d'impression et d'envoyer des demandes d'impression pour ces services.

Il inclut des attributs d'impression extensibles basés sur les attributs standard spécifiés dans le protocole IPP (Internet Printing Protocol) 1.1 de la spécification IETF, RFC 2911 .

Examples

Découvrir les services d'impression disponibles

Pour découvrir tous les services d'impression disponibles, nous pouvons utiliser la classe PrintServiceLookup . Voyons comment:

```
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());
        }
    }

}
```

Ce programme, lorsqu'il est exécuté dans un environnement Windows, imprimera quelque chose comme ceci:

```
Print service name: Fax
Print service name: Microsoft Print to PDF
Print service name: Microsoft XPS Document Viewer
```

Découverte du service d'impression par défaut

Pour découvrir le service d'impression par défaut, nous pouvons utiliser la classe PrintServiceLookup . Voyons comment ::

```
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());
    }

}
```

Création d'un travail d'impression à partir d'un service d'impression

Un travail d'impression est une demande d'impression d'un élément dans un service d'impression spécifique. Il consiste essentiellement en:

- les données qui seront imprimées (voir Construire le document à imprimer)
- un ensemble d'attributs

Après avoir sélectionné l'instance de service d'impression appropriée, nous pouvons demander la création d'un travail d'impression:

```
DocPrintJob printJob = printService.createPrintJob();
```

L'interface DocPrintJob nous fournit la méthode d' print :

```
printJob.print(doc, pras);
```

L'argument doc est un Doc : les données qui seront imprimées.

Et l'argument pras est une interface PrintRequestAttributeSet : un ensemble de PrintRequestAttribute . Sont des exemples d'attributs de demande d'impression:

- quantité de copies (1, 2, etc.),
- orientation (portrait ou paysage)
- chromacité (monochrome, couleur)
- qualité (brouillon, normal, élevé)
- côtés (recto, recto verso, etc.)
- etc...

La méthode print peut lancer une PrintException .

Construire le Doc qui sera imprimé

Doc est une interface et l'API Java Print Service fournit une implémentation simple appelée SimpleDoc .

Chaque instance de Doc est essentiellement composée de deux aspects:

- le contenu des données d'impression lui-même (un courrier électronique, une image, un document, etc.)
- le format de données d'impression, appelé DocFlavor (type MIME + classe de représentation).

Avant de créer l'objet Doc , nous devons charger notre document à partir de quelque part. Dans l'exemple, nous allons charger un fichier spécifique à partir du disque:

```
FileInputStream pdfFileInputStream = new FileInputStream("something.pdf");
```

Alors maintenant, nous devons choisir un DocFlavor qui correspond à notre contenu. La classe DocFlavor a un tas de constantes pour représenter les types de données les plus courants. Prenons un INPUT_STREAM.PDF :

```
DocFlavor pdfDocFlavor = DocFlavor.INPUT_STREAM.PDF;
```

Maintenant, nous pouvons créer une nouvelle instance de SimpleDoc :

```
Doc doc = new SimpleDoc(pdfFileInputStream, pdfDocFlavor , null);
```

L'objet doc peut maintenant être envoyé à la demande de travail d'impression (voir Création d'un travail d'impression à partir d'un service d'impression).

Définition d'attributs de demande d'impression

Parfois, nous devons déterminer certains aspects de la demande d'impression. Nous les appellerons *attribut* .

Sont des exemples d'attributs de demande d'impression:

- quantité de copies (1, 2, etc.),
- orientation (portrait ou paysage)
- chromacité (monochrome, couleur)
- qualité (brouillon, normal, élevé)
- côtés (recto, recto verso, etc.)
- etc...

Avant de choisir l'un d'eux et la valeur que chacun aura, nous devons d'abord créer un ensemble d'attributs:

```
PrintRequestAttributeSet pras = new HashPrintRequestAttributeSet();
```

Maintenant, nous pouvons les ajouter. Certains exemples sont:

```
pras.add(new Copies(5));
pras.add(MediaSize.ISO_A4);
pras.add(OrientationRequested.PORTRAIT);
pras.add(PrintQuality.NORMAL);
```

L'objet pras peut maintenant être envoyé à la demande de travail d'impression (voir Création d'un travail d'impression à partir d'un service d'impression).

Modification du statut de demande d'impression d'un travail d'écoute

Pour les clients d'impression les plus nombreux, il est extrêmement utile de savoir si un travail d'impression est terminé ou a échoué.

L'API Java Print Service fournit des fonctionnalités pour s'informer sur ces scénarios. Tout ce que nous avons à faire est:

- fournir une implémentation pour l'interface PrintJobListener et
- enregistrer cette implémentation sur le travail d'impression.

Lorsque l'état du travail d'impression change, nous en serons informés. Nous pouvons faire tout ce qui est nécessaire, par exemple:

- mettre à jour une interface utilisateur,
- lancer un autre processus métier,
- enregistrer quelque chose dans la base de données,
- ou simplement le connecter.

Dans l'exemple ci-dessous, nous enregistrerons chaque changement d'état de travail d'impression:

```
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 ");
    }
}
```

Enfin, nous pouvons ajouter l'implémentation de l'auditeur de travaux d'impression sur le travail d'impression avant la demande d'impression elle-même, comme suit:

```
DocPrintJob printJob = printService.createPrintJob();

printJob.addPrintJobListener(new LoggerPrintJobListener());

printJob.print(doc, pras);
```

L'argument *PrintJobEvent pje*

Notez que chaque méthode a un argument PrintJobEvent pje . Nous ne l'utilisons pas dans cet exemple à des fins de simplicité, mais vous pouvez l'utiliser pour explorer le statut. Par exemple:

```
pje.getPrintJob().getAttributes();
```

Renvoie une instance d'objet PrintJobAttributeSet et vous pouvez les exécuter d'une manière pour-chaque.

Une autre façon d'atteindre le même objectif

Une autre option pour atteindre le même objectif est d'étendre la classe PrintJobAdapter , comme son nom l'indique, est un adaptateur pour PrintJobListener . En mettant en œuvre l'interface, nous devons obligatoirement les implémenter tous. L'avantage de cette façon, c'est que nous devons remplacer uniquement les méthodes que nous voulons. Voyons voir comment ça fonctionne:

```
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 =( ");
    }
}
```

Notez que nous ne remplaçons que certaines méthodes spécifiques.

De la même manière que dans l'exemple implémentant l'interface PrintJobListener , nous ajoutons l'écouteur au travail d'impression avant de l'envoyer à l'impression:

```
printJob.addPrintJobListener(new LoggerPrintJobAdapter());

printJob.print(doc, pras);
```

Lire Service d'impression Java en ligne: https://riptutorial.com/fr/java/topic/10178/service-d-impression-java

Chapitre 160: ServiceLoader

Remarques

ServiceLoader peut être utilisé pour obtenir des instances de classes étendant un type donné (= service) spécifié dans un fichier contenu dans un fichier .jar . Le service étendu / implémenté est souvent une interface, mais ce n'est pas obligatoire.

Les classes d'extension / implémentation doivent fournir un constructeur à argument nul pour que ServiceLoader les instancie.

Pour être découvert par ServiceLoader un fichier texte portant le nom complet du type de nom du service implémenté doit être stocké dans le META-INF/services du fichier jar. Ce fichier contient un nom qualifié complet d'une classe implémentant le service par ligne.

Examples

Service d'enregistrement

L'exemple suivant montre comment instancier une classe pour la journalisation via ServiceLoader .

Un service

```
package servicetest;

import java.io.IOException;

public interface Logger extends AutoCloseable {

    void log(String message) throws IOException;
}
```

Implémentations du service

L'implémentation suivante écrit simplement le message sur 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() {
    }

}
```

L'implémentation suivante écrit les messages dans un fichier texte:

```
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

Le fichier META-INF/services/servicetest.Logger répertorie les noms des implémentations Logger .

```
servicetest.logger.ConsoleLogger
servicetest.logger.FileLogger
```

Usage

La méthode main suivante écrit un message sur tous les enregistreurs disponibles. Les enregistreurs sont instanciés à l'aide de 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);
        }
    }
}
```

Exemple simple de ServiceLoader

Le ServiceLoader est un mécanisme intégré simple et facile à utiliser pour le chargement

dynamique des implémentations d'interface. Avec le chargeur de service - fournissant des moyens pour l'instauration (mais pas le câblage) - un simple mécanisme d'injection de dépendance peut être intégré à Java SE. Avec l'interface ServiceLoader, la séparation de l'implémentation devient naturelle et les programmes peuvent être facilement étendus. En fait, de nombreuses API Java sont implémentées sur la base du ServiceLoader

Les concepts de base sont

- Fonctionnant sur des *interfaces* de services
- Obtention des implémentations du service via ServiceLoader
- Mise en place de services

Commençons par l'interface et placez-la dans un pot nommé par exemple accounting-api.jar

```
package example;

public interface AccountingService {

  long getBalance();
}
```

Maintenant, nous fournissons une implémentation de ce service dans un jar nommé accounting-impl.jar , contenant une implémentation du service

```
package example.impl;
import example.AccountingService;

public interface DefaultAccountingService implements AccouningService {

  public long getBalance() {
    return balanceFromDB();
  }

  private long balanceFromDB(){
    ...
  }
}
```

De plus, le fichier accounting-impl.jar contient un fichier déclarant que ce fichier jar fournit une implémentation de AccountingService . Le fichier doit avoir un chemin commençant par META-INF/services/ et doit avoir le même nom que le nom *complet* de l'interface:

- META-INF/services/example.AccountingService

Le contenu du fichier est le nom *entièrement qualifié* de l'implémentation:

```
example.impl.DefaultAccountingService
```

Étant donné que les deux jars se trouvent dans le chemin de classe du programme, qui utilise AccountingService , une instance du service peut être obtenue à l'aide de ServiceLauncher.

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
AccountingService service = loader.next();
long balance = service.getBalance();
```

Comme ServiceLoader est une Iterable , il prend en charge plusieurs fournisseurs d'implémentation, parmi lesquels le programme peut choisir:

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
for(AccountingService service : loader) {
   //...
}
```

Notez que lors de l'appel de next() une nouvelle instance sera toujours créée. Si vous souhaitez réutiliser une instance, vous devez utiliser la méthode iterator() du ServiceLoader ou la boucle for-each, comme indiqué ci-dessus.

Lire ServiceLoader en ligne: https://riptutorial.com/fr/java/topic/5433/serviceloader

Chapitre 161: Singletons

Introduction

Un singleton est une classe qui ne possède qu'une seule instance. Pour plus d'informations sur le *modèle de conception* Singleton, reportez-vous à la rubrique Singleton dans la balise Design Patterns .

Examples

Enum Singleton

Java SE 5

```
public enum Singleton {
    INSTANCE;

    public void execute (String arg) {
        // Perform operation here
    }
}
```

Les énumérations ont des constructeurs privés, sont finales et fournissent des machines de sérialisation appropriées. Ils sont également très concis et paresseusement initialisés de manière sécurisée.

La JVM garantit que les valeurs enum ne seront pas instanciées plus d'une fois chacune, ce qui confère au modèle enum singleton une défense très forte contre les attaques par réflexion.

Ce que le modèle enum *ne* protège *pas, ce* sont les autres développeurs qui ajoutent physiquement plus d'éléments au code source. Par conséquent, si vous choisissez ce style d'implémentation pour vos singletons, il est impératif que vous décriviez très clairement qu'aucune nouvelle valeur ne devrait être ajoutée à ces énumérations.

C'est la manière recommandée d'implémenter le modèle singleton, comme expliqué par Joshua Bloch dans Effective Java.

Filetage sûr Singleton avec double verrouillage de vérification

Ce type de Singleton est thread-safe et empêche le verrouillage inutile après la création de l'instance 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;
    }
}
```

Il faut le souligner - dans les versions antérieures à Java SE 5, l'implémentation ci-dessus est incorrecte et doit être évitée. Il est impossible d'implémenter correctement le verrouillage à double vérification dans Java avant Java 5.

Singleton sans utilisation d'Enum (initialisation enthousiaste)

```
public class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

On peut soutenir que cet exemple est *une* initialisation paresseuse. La section 12.4.1 de la spécification du langage Java indique:

> Une classe ou un type d'interface T sera initialisé immédiatement avant la première occurrence de l'un des éléments suivants:
>
> - T est une classe et une instance de T est créée
> - T est une classe et une méthode statique déclarée par T est invoquée
> - Un champ statique déclaré par T est affecté
> - Un champ statique déclaré par T est utilisé et le champ n'est pas une variable constante
> - T est une classe de niveau supérieur et une instruction assert lexicale imbriquée dans T est exécutée.

Par conséquent, tant qu'il n'y a pas d'autres champs statiques ou méthodes statiques dans la classe, l'instance Singleton ne sera pas initialisée tant que la méthode getInstance() n'est pas appelée pour la première fois.

Initialisation par défaut sécurisée des threads à l'aide de la classe holder | Mise en œuvre de Bill Pugh Singleton

```
public class Singleton {
    private static class InstanceHolder {
        static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return InstanceHolder.INSTANCE;
    }

    private Singleton() {}
```

```
}
```

Cela initialise la variable INSTANCE lors du premier appel à Singleton.getInstance() , en tirant parti des garanties de sécurité des threads de la langue pour l'initialisation statique sans nécessiter de synchronisation supplémentaire.

Cette implémentation est également connue sous le nom de singleton de Bill Pugh. [Wiki]

Extension de singleton (héritage singleton)

Dans cet exemple, la classe de base Singleton fournit la méthode getMessage() qui renvoie "Hello world!" message.

Il est sous - classes UppercaseSingleton et LowercaseSingleton remplacer méthode getMessage () pour fournir une représentation appropriée du message.

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

Lire Singletons en ligne: https://riptutorial.com/fr/java/topic/130/singletons

Chapitre 162: Sockets Java

Introduction

Les sockets sont une interface réseau de bas niveau qui aide à créer une connexion entre deux programmes, principalement des clients qui peuvent ou non s'exécuter sur le même ordinateur.

Socket Programming est l'un des concepts de réseautage les plus utilisés.

Remarques

Il existe deux types de trafic Internet:
1. TCP - Transmission Control Protocol 2. UDP - Protocole de datagramme utilisateur

TCP est un protocole orienté connexion.
UDP est un protocole sans connexion.

Le protocole TCP convient aux applications nécessitant une grande fiabilité et le temps de transmission est relativement moins critique.

UDP convient aux applications nécessitant une transmission rapide et efficace, comme les jeux. La nature sans état d'UDP est également utile pour les serveurs qui répondent à de petites requêtes provenant d'un grand nombre de clients.

En termes plus simples -
Utilisez le protocole TCP lorsque vous ne pouvez pas vous permettre de perdre des données et que le temps nécessaire pour envoyer et recevoir des données importe peu. Utilisez UDP lorsque vous ne pouvez pas vous permettre de perdre du temps et que la perte de données n'a pas d'importance.

Il existe une garantie absolue que les données transférées restent intactes et arrivent dans l'ordre dans lequel elles ont été envoyées en cas de TCP.
alors qu'il n'y a aucune garantie que les messages ou les paquets envoyés atteindraient du tout dans UDP.

Examples

Un serveur de retour d'écho TCP simple

Notre serveur TCP echo back sera un thread séparé. C'est simple comme c'est un début. Il ne fera que répéter ce que vous envoyez, mais sous forme de majuscule.

```
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);
        }
    }

}
```

Acceptez maintenant les connexions. Mettons à jour la méthode d'exécution.

```
@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);
    }
}
```

Maintenant, si vous pouvez ouvrir telnet et essayer de vous connecter, vous verrez un message de bienvenue.

Vous devez vous connecter au port que vous avez spécifié et à l'adresse IP.

Vous devriez voir un résultat similaire à celui-ci:

```
Welcome!

Connection to host lost.
```

Eh bien, la connexion a été perdue parce que nous l'avons terminée. Parfois, nous devrions programmer notre propre client TCP. Dans ce cas, nous avons besoin d'un client pour demander une entrée de l'utilisateur et l'envoyer sur le réseau, recevoir la saisie en majuscule.

Si le serveur envoie d'abord des données, le client doit d'abord lire les données.

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

Maintenant, mettez à jour le serveur

```
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();
```

Et maintenant, exécutez le serveur et le client, vous devriez avoir une sortie similaire à celle-ci

```
Welcome!
>
```

Lire Sockets Java en ligne: https://riptutorial.com/fr/java/topic/9923/sockets-java

Chapitre 163: SortedMap

Introduction

Introduction à la carte triée.

Examples

Introduction à la carte triée.

Point clé :-

- L'interface SortedMap étend la carte.
- les entrées sont conservées dans un ordre croissant de clés.

Méthodes de la carte triée:

- Comparateur comparateur ().
- Object firstKey ().
- SortedMap headMap (fin de l'objet).
- Object lastKey ().
- Carte secondaire SortedMap (début d'objet, fin d'objet).
- SortedMap tailMap (démarrage d'objet).

Exemple

```
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"));
   }
```

Lire SortedMap en ligne: https://riptutorial.com/fr/java/topic/10748/sortedmap

Chapitre 164: StringBuffer

Introduction

Introduction à la classe Java StringBuffer.

Examples

Classe de tampon de chaîne

Points clés :-

- utilisé pour créer une chaîne modifiable (modifiable).
- **Mutable** : - Qui peut être changé.
- est thread-safe, c'est-à-dire que plusieurs threads ne peuvent pas y accéder simultanément.

Méthodes: -

- ajout de StringBuffer public synchronisé (String s)
- insert public StringBuffer synchronisé (int offset, String s)
- remplacement public de StringBuffer synchronisé (int startIndex, int endIndex, String str)
- suppression publique de StringBuffer synchronisée (int startIndex, int endIndex)
- StringBuffer inversé public reverse ()
- public int capacity ()
- public void EnsureCapacity (int minimumCapacity)
- char charAt public (index int)
- public int length ()
- sous-chaîne de chaîne publique (int beginIndex)
- sous-chaîne de chaînes publique (int beginIndex, int endIndex)

Exemple montrant la différence entre l'implémentation de String et de 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
 }
}
```

Lire StringBuffer en ligne: https://riptutorial.com/fr/java/topic/10757/stringbuffer

Chapitre 165: StringBuilder

Introduction

La classe Java StringBuilder est utilisée pour créer une chaîne modifiable (modifiable). La classe Java StringBuilder est identique à la classe StringBuffer, sauf qu'elle n'est pas synchronisée. Il est disponible depuis JDK 1.5.

Syntaxe

- nouveau StringBuilder ()
- nouveau StringBuilder (capacité int)
- new StringBuilder (CharSequence seq)
- new StringBuilder (générateur StringBuilder)
- new StringBuilder (chaîne de caractères)
- new StringJoiner (délimiteur CharSequence)
- new StringJoiner (délimiteur CharSequence, préfixe CharSequence, suffixe CharSequence)

Remarques

La création d'un nouveau StringBuilder avec le type char tant que paramètre entraînerait l'appel du constructeur avec l'argument int capacity et non celui avec l'argument 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

Répéter une chaîne n fois

Problème: Créez une String contenant n répétitions d'une String s .

L'approche triviale serait la concaténation répétée de la String

```
final int n = ...
final String s = ...
String result = "";

for (int i = 0; i < n; i++) {
    result += s;
}
```

Cela crée n nouvelles instances de chaîne contenant 1 à n répétitions de s résultant en un temps d'exécution de O(s.length() * n²) = O(s.length() * (1+2+...+(n-1)+n)) .

Pour éviter cela, StringBuilder devrait être utilisé, ce qui permet de créer la String dans O(s.length() * n) place:

```
final int n = ...
final String s = ...

StringBuilder builder = new StringBuilder();

for (int i = 0; i < n; i++) {
    builder.append(s);
}

String result = builder.toString();
```

Comparaison de StringBuffer, StringBuilder, Formatter et StringJoiner

Les classes StringBuffer , StringBuilder , Formatter et StringJoiner sont des classes d'utilitaires Java SE principalement utilisées pour assembler des chaînes à partir d'autres informations:

- La classe StringBuffer est présente depuis Java 1.0 et fournit diverses méthodes pour créer et modifier un "tampon" contenant une séquence de caractères.
- La classe StringBuilder été ajoutée à Java 5 pour résoudre les problèmes de performances liés à la classe StringBuffer origine. Les API pour les deux classes sont essentiellement les mêmes. La principale différence entre StringBuffer et StringBuilder est que le premier est thread-safe et synchronisé et que le second ne l'est pas.

Cet exemple montre comment StringBuilder peut être utilisé:

```
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 est utilisée de la même manière: changez simplement StringBuilder en StringBuffer dans ce qui précède)

Les classes StringBuffer et StringBuilder conviennent pour l'assemblage et la modification de chaînes. c'est-à-dire qu'ils fournissent des méthodes pour remplacer et supprimer des caractères, ainsi que pour les ajouter à divers. Le remining deux classes sont spécifiques à la tâche d'assemblage des chaînes.

- La classe Formatter été ajoutée à Java 5 et est modélisée librement sur la fonction sprintf dans la bibliothèque standard C. Il prend une chaîne de *format* avec des *spécificateurs de format* incorporés et une séquence d'autres arguments, et génère une chaîne en convertissant les arguments en texte et en les substituant aux spécificateurs de format. Les détails des spécificateurs de format indiquent comment les arguments sont convertis en texte.
- La classe StringJoiner été ajoutée à Java 8. C'est un formateur spécial qui formate succinctement une séquence de chaînes avec des séparateurs entre elles. Il est conçu avec une API *fluide* et peut être utilisé avec les flux Java 8.

Voici quelques exemples typiques d'utilisation de 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 n'est pas idéale pour la tâche ci-dessus, voici donc un exemple de formatage d'un tableau de chaînes.

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
for (String s : new String[]{"A", "B", "C"}) {
    sj.add(s);
}
System.out.println(sj);
// Prints "[A, B, C]"
```

Les cas d'utilisation des 4 classes peuvent être résumés:

- StringBuilder convient à toute tâche de modification de chaîne OU d'assemblage de chaînes.
- StringBuffer utilise (uniquement) lorsque vous avez besoin d'une version thread-safe de StringBuilder .
- Formatter fournit des fonctionnalités de formatage de chaînes beaucoup plus riches, mais n'est pas aussi efficace que StringBuilder . C'est parce que chaque appel à Formatter.format(...) implique:
    - analyser la chaîne de format ,
    - créer et remplir un tableau *varargs* , et
    - Autoboxing des arguments de type primitif.
- StringJoiner fournit un formatage succinct et efficace d'une séquence de chaînes avec des séparateurs, mais ne convient pas aux autres tâches de formatage.

Lire StringBuilder en ligne: https://riptutorial.com/fr/java/topic/1037/stringbuilder

Chapitre 166: Structures de contrôle de base

Remarques

Toutes les structures de contrôle, sauf indication contraire, utilisent des instructions de bloc . Celles-ci sont indiquées par des accolades {} .

Cela diffère des instructions normales , qui ne nécessitent *pas* d'accolades, mais comportent également une réserve stricte: seule la ligne *qui suit immédiatement* l'instruction précédente est prise en compte.

Ainsi, il est parfaitement valable d'écrire n'importe laquelle de ces structures de contrôle sans accolades, à condition qu'une *seule* instruction suive le début, mais qu'elle soit ***fortement déconseillée*** , car elle peut entraîner des mises en œuvre erronées ou du code cassé.

Exemple:

```
// valid, but discouraged
Scanner scan = new Scanner(System.in);
int val = scan.nextInt();
if(val % 2 == 0)
    System.out.println("Val was even!");


// invalid; will not compile
// note the misleading indentation here
for(int i = 0; i < 10; i++)
    System.out.println(i);
    System.out.println("i is currently: " + i);
```

Examples

Si / Sinon Si / Contrôle Else

```
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");
}
```

Le bloc if ne fonctionnera que lorsque i est inférieur ou égal à 1.

La condition else if est vérifiée uniquement si toutes les conditions avant (dans les constructions précédentes else if et les constructions parent if) ont été testées sur false . Dans cet exemple, la condition else if ne sera vérifiée que si i est supérieur ou égal à 2.

Si son résultat est true , son bloc est exécuté, et tout else if et else construit après son saut.

Si aucune des if et else if n'a été testée sur true , le bloc else à la fin sera exécuté.

Pour les boucles

```
for (int i = 0; i < 100; i++) {
    System.out.println(i);
```

```
}
```

Les trois composants de la boucle for (séparés par ;) sont la déclaration / initialisation des variables (ici int i = 0), la condition (ici i < 100) et l'instruction d'incrémentation (ici i++). La déclaration de variable est effectuée une fois comme si elle était placée juste à l'intérieur du { lors de la première exécution. Ensuite, la condition est vérifiée, s'il est true le corps de la boucle s'exécutera, s'il est false la boucle s'arrêtera. En supposant que la boucle se poursuive, le corps s'exécutera et finalement lorsque l'instruction } sera atteinte, l'instruction d'incrémentation s'exécutera juste avant que la condition ne soit à nouveau vérifiée.

Les accolades sont facultatives (vous pouvez une ligne avec un point-virgule) si la boucle ne contient qu'une seule instruction. Mais, il est toujours recommandé d'utiliser des accolades pour éviter les malentendus et les bugs.

Les composants for loop sont facultatifs. Si votre logique métier contient l'une de ces parties, vous pouvez omettre le composant correspondant de votre boucle 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 structure for (;;) { function-body } est égale à une boucle while (true) .

Nested For Loops

Toute instruction de boucle ayant une autre instruction de boucle à l'intérieur appelée boucle imbriquée. La même façon de boucler avec plus de boucle interne est appelée «imbriquée pour la boucle».

```
    for(;;){
        //Outer Loop Statements
        for(;;){
            //Inner Loop Statements
        }
        //Outer Loop Statements
    }
```

Il est possible de démontrer que les imbriqués pour la boucle impriment des nombres en forme de triangle.

```
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);
    }
 }
```

Pendant que les boucles

```
int i = 0;
while (i < 100) { // condition gets checked BEFORE the loop body executes
    System.out.println(i);
    i++;
```

```
}
```

A while boucle court aussi longtemps que la condition entre parenthèses est true . Cela s'appelle aussi la structure "boucle de pré-test" car l'instruction conditionnelle doit être satisfaite avant que le corps de la boucle principale ne soit exécuté à chaque fois.

Les accolades sont facultatives si la boucle ne contient qu'une seule instruction, mais certaines conventions de style de codage préfèrent les accolades.

do ... tandis que la boucle

La boucle do...while while diffère des autres boucles en ce sens qu'elle est garantie **au moins une fois** . On l'appelle aussi la structure "post-test loop" car l'instruction conditionnelle est exécutée après le corps de la boucle principale.

```
int i = 0;
do {
    i++;
    System.out.println(i);
} while (i < 100); // Condition gets checked AFTER the content of the loop executes.
```

Dans cet exemple, la boucle s'exécutera jusqu'à ce que le nombre 100 soit imprimé (même si la condition est i < 100 et non i <= 100), car la condition de la boucle est évaluée *après* l'exécution de la boucle.

Avec la garantie d'au moins une exécution, il est possible de déclarer des variables en dehors de la boucle et de les initialiser à l'intérieur.

```
String theWord;
Scanner scan = new Scanner(System.in);
do {
    theWord = scan.nextLine();
} while (!theWord.equals("Bird"));

System.out.println(theWord);
```

Dans ce contexte, le theWord est défini en dehors de la boucle, mais comme il est garanti d'avoir une valeur basée sur son flux naturel, le theWord sera initialisé.

Pour chaque

Java SE 5

Avec Java 5 et plus, on peut utiliser des boucles for-each, également appelées boucles for-loops:

```
List strings = new ArrayList();

strings.add("This");
strings.add("is");
strings.add("a for-each loop");

for (String string : strings) {
    System.out.println(string);
}
```

Pour chaque boucle peut être utilisée pour itérer sur Arrays et les implémentations de l'interface Iterable , le dernier inclut les classes Collections , telles que List ou Set .

La variable de boucle peut être de tout type pouvant être assigné à partir du type de source.

La variable de boucle pour une boucle améliorée pour Iterable<T> ou T[] peut être de type S , si

- T extends S
- les deux T et S sont des types primitifs et assignables sans fonte
- S est un type primitif et T peut être converti en un type assignable à S après la conversion unboxing.
- T est un type primitif et peut être converti en S par conversion automatique.

Exemples:

```
T elements = ...
for (S s : elements) {
}
```

T	S	Compile
int []	longue	Oui
longue[]	int	non
Iterable<Byte>	longue	Oui
Iterable<String>	CharSequence	Oui
Iterable<CharSequence>	Chaîne	non
int []	Longue	non
int []	Entier	Oui

Sinon

```
int i = 2;
if (i < 2) {
  System.out.println("i is less than 2");
} else {
  System.out.println("i is greater than 2");
}
```

Une instruction if exécute le code de manière conditionnelle en fonction du résultat de la condition entre parenthèses. Lorsque la condition entre parenthèses est vraie, elle entrera dans le bloc de l'instruction if qui est défini par des accolades comme { et } . parenthèse ouvrante jusqu'à la parenthèse fermante est la portée de l'instruction if.

Le bloc else est facultatif et peut être omis. Il s'exécute si l'instruction if est false et ne s'exécute pas si l'instruction if est vraie. Dans ce cas, if instruction est exécutée.

Voir aussi: Si ternaire

Déclaration de changement

L'instruction switch est l'instruction de branche multi-voies de Java. Il est utilisé pour remplacer les chaînes long if - else if - else et les rendre plus lisibles. Cependant, à la différence if les déclarations, on ne peut pas utiliser les inégalités; chaque valeur doit être définie concrètement.

L'instruction switch comporte trois composants essentiels:

- case : Valeur évaluée pour l'équivalence avec l'argument de l'instruction switch .
- default : il s'agit d'une expression facultative, catch-all, si aucune des instructions de case n'est évaluée à true .
- Achèvement complet de la déclaration de case ; En général, break : Ceci est nécessaire pour empêcher l'évaluation indésirable d'autres déclarations de case .

À l'exception de continue , il est possible d'utiliser n'importe quelle instruction qui entraînerait l' achèvement brutal d'une déclaration . Ceci comprend:

- break
  - return
  - throw

Dans l'exemple ci-dessous, une instruction de switch typique est écrite avec quatre cas possibles, y compris la 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");
}
```

En omettant les break ou toute déclaration qui aboutirait brusquement, nous pouvons tirer parti de ce que nous appelons des cas «tombants», qui s'évaluent en fonction de plusieurs valeurs. Cela peut être utilisé pour créer des plages pour une valeur réussie, mais n'est pas aussi flexible que les inégalités.

```
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");
}
```

Dans le cas de foo == 1 le résultat sera:

```
I'm equal or greater than one
I'm one, two, or three
```

Dans le cas de foo == 3 le résultat sera:

```
I'm one, two, or three
```

Java SE 5

L'instruction switch peut également être utilisée avec 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'instruction switch peut également être utilisée avec 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");
    }
}
```

Opérateur ternaire

Parfois, vous devez vérifier une condition et définir la valeur d'une variable.

Pour ex.

```
String name;

if (A > B) {
    name = "Billy";
} else {
    name = "Jimmy";
}
```

Cela peut être facilement écrit dans une ligne comme

```
String name = A > B ? "Billy" : "Jimmy";
```

La valeur de la variable est définie sur la valeur immédiatement après la condition, si la condition est vraie. Si la condition est fausse, la deuxième valeur sera donnée à la variable.

Pause

L'instruction break termine une boucle (comme for , while) ou l'évaluation d'une instruction switch .

Boucle:

```
while(true) {
   if(someCondition == 5) {
       break;
   }
}
```

La boucle dans l'exemple fonctionnerait pour toujours. Mais quand une someCondition est égale à 5 à un moment donné, la boucle se termine.

Si plusieurs boucles sont en cascade, seule la boucle la plus interne se termine par un break .

Essayez ... Catch ... Enfin

La structure de contrôle try { ... } catch (...) { ... } est utilisée pour gérer les exceptions .

```
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.");
}
```

Cela imprimerait:

```
Entrée invalide. 'abc' n'est pas un entier valide.
```

Une clause finally peut être ajoutée après le catch . La clause finally serait toujours exécutée, qu'une exception ait été levée ou non.

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");
}
```

Cela imprimerait:

```
Entrée invalide. 'abc' n'est pas un entier valide.
Ce code sera toujours exécuté, même si une exception est levée
```

Nested break / continue

Il est possible de break / continue vers une boucle externe en utilisant des instructions d'étiquette:

```
outerloop:
for(...) {
    innerloop:
    for(...) {
        if(condition1)
            break outerloop;

        if(condition2)
            continue innerloop; // equivalent to: continue;
    }
}
```

Il n'y a pas d'autre utilisation pour les étiquettes en Java.

Continuer la déclaration en Java

L'instruction continue permet d'ignorer les étapes restantes de l'itération en cours et de commencer avec l'itération de boucle suivante. Le contrôle passe de l'instruction continue à la valeur d'étape (incrémentation ou décrémentation), le cas échéant.

```
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'instruction continue peut également faire passer le contrôle du programme à la valeur d'étape (le cas échéant) d'une boucle nommée:

```
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;
    }
}
```

Lire Structures de contrôle de base en ligne:
https://riptutorial.com/fr/java/topic/118/structures-de-controle-de-base

Chapitre 167: sun.misc.Unsafe

Remarques

La classe Unsafe permet à un programme de faire des choses qui ne sont pas autorisées par le compilateur Java. Les programmes normaux devraient éviter d'utiliser Unsafe .

AVERTISSEMENTS

1. Si vous commettez une erreur en utilisant les API Unsafe , vos applications risquent de provoquer le blocage de la JVM et / ou de présenter des symptômes difficiles à diagnostiquer.
2. L'API Unsafe est sujet à modification sans préavis. Si vous l'utilisez dans votre code, vous devrez peut-être réécrire le code lors de la modification des versions de Java.

Examples

Instanciation de sun.misc.Unsafe via la réflexion

```
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 possède un constructeur Private et la getUnsafe() statique getUnsafe() est protégée par une vérification du chargeur de classe pour garantir que le code a été chargé avec le chargeur de classe principal. Par conséquent, une méthode de chargement de l'instance consiste à utiliser la réflexion pour obtenir le champ statique.

Instanciation de sun.misc.Unsafe via bootclasspath

```
public class UnsafeLoader {
    public static Unsafe loadUnsafe() {
        return Unsafe.getUnsafe();
    }
}
```

Bien que cet exemple compile, il est probable qu'il échoue à l'exécution à moins que la classe Unsafe ne soit chargée avec le chargeur de classe principal. Pour que cela se produise, la JVM doit être chargée avec les arguments appropriés, tels que:

```
java -Xbootclasspath:$JAVA_HOME/jre/lib/rt.jar:./UnsafeLoader.jar foo.bar.MyApp
```

La classe foo.bar.MyApp peut alors utiliser UnsafeLoader.loadUnsafe() .

Obtenir l'instance de Unsafe

Unsafe est stocké dans un champ privé auquel on ne peut pas accéder directement. Le constructeur est privé et la seule méthode pour accéder à public static Unsafe getUnsafe() a un accès privilégié. En utilisant la réflexion, il existe un moyen de contourner les domaines privés:

```
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;
}
```

Utilisations de Unsafe

Voici quelques utilisations de unsafe:

Utilisation	API
Allocation de mémoire / mémoire directe, réaffectation et désallocation	`allocateMemory(bytes)` , `reallocateMemory(address, bytes)` et `freeMemory(address)`
Clôtures à mémoire	`loadFence()` , `storeFence()` , `fullFence()`
Fil de stationnement actuel	`park(isAbsolute, time)` , `unpark(thread)`
Accès direct au champ et / ou à la mémoire	`get*` et `put*` famille de méthodes
Lancer des exceptions non vérifiées	`throwException(e)`
CAS et opérations atomiques	Famille de méthodes `compareAndSwap*`
Mise en mémoire	`setMemory`
Opérations volatiles ou simultanées	`get*Volatile` , `put*Volatile` , `putOrdered*`

La famille de méthodes get et put est relative à un objet donné. Si l'objet est nul, il est traité comme une adresse absolue.

```
// 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);
```

Certaines méthodes ne sont définies que pour int et longs. Vous pouvez utiliser ces méthodes sur floats et doubles en utilisant floatToRawIntBits , intBitsToFloat, doubleToRawLongBits , longBitsToDouble`

Lire sun.misc.Unsafe en ligne: https://riptutorial.com/fr/java/topic/6771/sun-misc-unsafe

Chapitre 168: super mot clé

Examples

Super mot-clé avec des exemples

super mot-clé joue un rôle important dans trois endroits

1. Niveau constructeur
2. Niveau de méthode
3. Niveau variable

Niveau constructeur

super mot clé super est utilisé pour appeler le constructeur de la classe parent. Ce constructeur peut être constructeur par défaut ou constructeur paramétré.

- Constructeur par défaut: super();
- Constructeur paramétré: 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();
  }
}
```

Note : super() doit être la première instruction du constructeur sinon nous obtiendrons le message d'erreur de compilation.

Niveau de méthode

super mot super clé peut également être utilisé en cas de substitution de méthode. super mot clé super peut être utilisé pour appeler ou appeler la méthode de la classe parente.

```
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();
   }
}
```

Remarque : S'il n'y a pas de substitution de méthode, il n'est pas nécessaire d'utiliser le mot super clé super pour appeler la méthode de la classe parente.

Niveau variable

super est utilisé pour désigner la variable d'instance de la classe parente immédiate. En cas d'héritage, la classe de base et la classe dérivée peuvent avoir des membres de données similaires. Afin de différencier le membre de données de la classe de base / parent et de la classe dérivée / enfant, dans le contexte de la classe dérivée les membres doivent être précédés du mot super clé super .

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

Remarque : Si vous n'écrivez pas de mot super clé super avant le nom du membre de données de la classe de base, il sera alors appelé membre de données de classe en cours et le membre de données de classe de base est masqué dans le contexte de la classe dérivée.

Lire super mot clé en ligne: https://riptutorial.com/fr/java/topic/5764/super-mot-cle

Chapitre 169: Tableaux

Introduction

Les tableaux permettent le stockage et la récupération d'une quantité arbitraire de valeurs. Ils sont analogues aux vecteurs en mathématiques. Les tableaux de tableaux sont analogues aux matrices et agissent comme des tableaux multidimensionnels. Les tableaux peuvent stocker toutes les données de tout type: les primitives telles que les types int ou de référence tels que Object .

Syntaxe

- `ArrayType[] myArray; // Déclaration des tableaux`
- `ArrayType myArray[]; // Une autre syntaxe valide (moins utilisée et déconseillée)`
- `ArrayType[][][] myArray; // Déclarer des tableaux déchiquetés multidimensionnels (repeter [] s)`
- `ArrayType myVar = myArray[index]; // Accès à l'élément (lecture) à l'index`
- `myArray[index] = value; // Attribuer une valeur à l' index de position du tableau`
- `ArrayType[] myArray = new ArrayType[arrayLength]; // Syntaxe d'initialisation de tableau`
- `int[] ints = {1, 2, 3}; // Syntaxe d'initialisation du tableau avec les valeurs fournies, longueur déduite du nombre de valeurs fournies: {[valeur1 [, valeur2] *]}`
- `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}; // Initialisation du tableau multidimensionnel. int [] étend Object (tout comme AnyType []), donc null est une valeur valide.`

Paramètres

Paramètre	Détails
`ArrayType`	Type du tableau. Cela peut être primitif (`int` , `long` , `byte`) ou Objects (`String` , `MyObject` , etc.).
indice	Index fait référence à la position d'un certain objet dans un tableau.
longueur	Chaque tableau, lors de sa création, nécessite une longueur définie. Ceci est soit fait lors de la création d'un tableau vide (`new int[3]`) ou implicite lors de la spécification des valeurs (`{1, 2, 3}`).

Examples

Création et initialisation de tableaux

Cas de 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
```

Les tableaux peuvent être créés en utilisant n'importe quel type de primitive ou de référence.

```
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).
```

Pour le dernier exemple, notez que les sous-types du type de tableau déclaré sont autorisés dans le tableau.

Les tableaux pour les types définis par l'utilisateur peuvent également être construits de manière similaire aux types primitifs

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

Tableaux, collections et flux

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 );
```

Introduction

Un *tableau* est une structure de données contenant un nombre fixe de valeurs primitives **ou des** références à des instances d'objets.

Chaque élément d'un tableau est appelé un élément et chaque élément est accessible par son index numérique. La longueur d'un tableau est établie lorsque le tableau est créé:

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

La taille d'un tableau est fixée à l'exécution lors de l'initialisation. Il ne peut pas être modifié après l'initialisation. Si la taille doit être mutable à l'exécution, une classe Collection telle ArrayList doit être utilisée à la place. ArrayList stocke des éléments dans un tableau et prend en charge le redimensionnement en allouant un nouveau tableau et en copiant des éléments de l'ancien tableau.

Si le tableau est de type primitif, c'est-à-dire

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

les valeurs sont stockées dans le tableau lui-même. En l'absence d'initialiseur (comme dans array2 ci-dessus), la valeur par défaut attribuée à chaque élément est 0 (zéro).

Si le type de tableau est une référence d'objet, comme dans

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

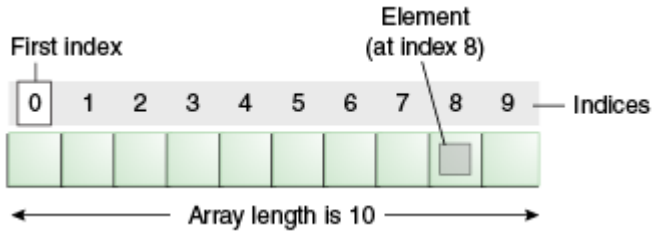
le tableau contient alors des *références* à des objets de type SomeClassOrInterface . Ces références peuvent faire référence à une instance de SomeClassOrInterface *ou de toute sous-classe (pour les classes) ou à la classe d'implémentation (pour les interfaces) de SomeClassOrInterface* . Si la déclaration de tableau n'a pas d'initialiseur, la valeur par défaut de null est affectée à chaque élément.

Comme tous les tableaux sont int -indexed, la taille d'un tableau doit être spécifiée par un int . La taille du tableau ne peut pas être spécifiée comme un long :

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

Les tableaux utilisent un système d' **index basé sur zéro** , ce qui signifie que l'indexation commence à 0 et se termine à la length - 1 .

Par exemple, l'image suivante représente un tableau de taille 10 . Ici, le premier élément est à l'indice 0 et le dernier élément à l'indice 9 , au lieu que le premier élément soit à l'indice 1 et le dernier élément à l'index 10 (voir la figure ci-dessous).



Les accès aux éléments des tableaux se font à **temps constant** . Cela signifie que l'accès au premier élément du tableau a le même coût (en temps) d'accès au deuxième élément, au troisième élément, etc.

Java offre plusieurs façons de définir et d'initialiser des tableaux, notamment **des** notations **littérales** et des **constructeurs** . Lors de la déclaration de tableaux utilisant le new Type[length] , chaque élément sera initialisé avec les valeurs par défaut suivantes:

- 0 pour les types numériques primitifs : byte , short , int , long , float et double .
- '\u0000' (caractère nul) pour le char de type.
- false pour le type boolean .
- null pour les types de référence .

Création et initialisation de tableaux de type primitif

```
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.
```

Lors de la déclaration d'un tableau, [] apparaîtra dans le type au début de la déclaration (après le nom du type) ou dans le déclarateur d'une variable particulière (après le nom de la variable), ou les deux:

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

Dans l'exemple suivant, les deux déclarations sont correctes et peuvent être compilées et exécutées sans aucun problème. Cependant, la convention de codage Java et le guide de style Java de Google découragent tous les deux le formulaire entre parenthèses après le nom de la variable. Les crochets identifient le type de tableau et doivent apparaître avec la désignation du type . La même chose devrait être utilisée pour les signatures de retour de méthode.

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

Le type déconseillé est destiné à accueillir les utilisateurs de la transition C , qui sont familiarisés avec la syntaxe de C qui contient les crochets après le nom de la variable.

En Java, il est possible d'avoir des tableaux de taille 0 :

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

Cependant, comme il s'agit d'un tableau vide, aucun élément ne peut être lu ou assigné:

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

De tels tableaux vides sont généralement utiles en tant que valeurs de retour, de sorte que le code appelant ne doit se préoccuper que du traitement d'un tableau, plutôt que d'une valeur null potentielle pouvant mener à une NullPointerException .

La longueur d'un tableau doit être un entier non négatif:

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

La taille du tableau peut être déterminée en utilisant un champ final public appelé length :

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

Remarque : array.length renvoie la taille réelle du tableau et non le nombre d'éléments de tableau auxquels une valeur a été attribuée, contrairement à ArrayList.size() qui renvoie le nombre d'éléments de tableau auxquels une valeur a été attribuée.

Création et initialisation de tableaux multidimensionnels

La manière la plus simple de créer un tableau multidimensionnel est la suivante:

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

Il créera deux tableaux int trois longueurs: a[0] et a[1] . Ceci est très similaire à l'initialisation classique de style C des tableaux multidimensionnels rectangulaires.

Vous pouvez créer et initialiser en même temps:

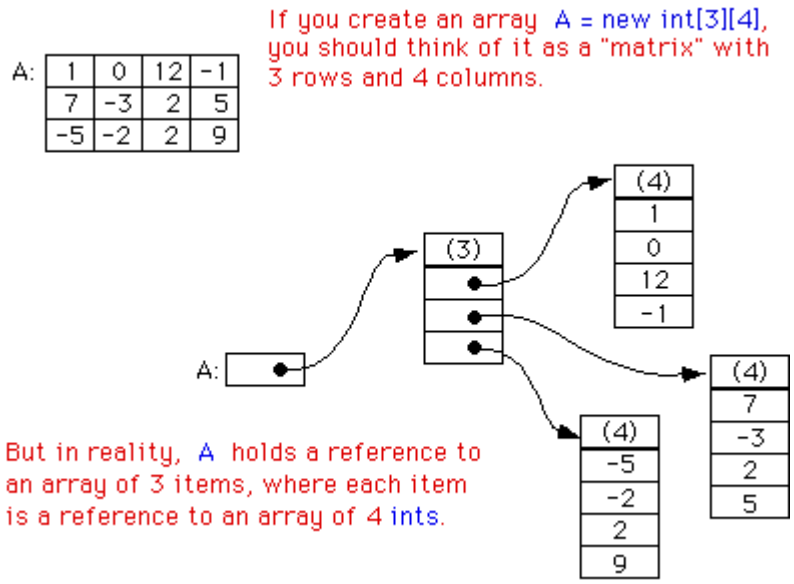
```
int[][] a = { {1, 2}, {3, 4}, {5, 6} };
```

Contrairement à C , où seuls les tableaux rectangulaires multidimensionnels sont pris en charge, les tableaux internes ne doivent pas nécessairement être de même longueur, ni même définis:

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

Ici, a[0] est un tableau int longueur, tandis a[1] est un tableau int deux longueurs et a[2] est null . Les tableaux comme celui-ci sont appelés tableaux irréguliers ou tableaux déchiquetés , c'est-à-dire qu'ils sont des tableaux de tableaux. Les tableaux multidimensionnels en Java sont implémentés en tant que tableaux de tableaux, c'est-à-dire que le array[i][j][k] est équivalent à ((array[i])[j])[k] . Contrairement à C # , le array[i,j] syntaxe array[i,j] n'est pas pris en charge en Java.

Représentation multidimensionnelle des tableaux en Java



Source - Live on Ideone

Création et initialisation de tableaux de type référence

```
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
```

Vivre sur Ideone

En plus des littéraux et des primitives de String présentés ci-dessus, la syntaxe de raccourci pour l'initialisation du tableau fonctionne également avec les types d' Object canoniques:

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

Les tableaux étant covariants, un tableau de type référence peut être initialisé en tant que tableau d'une sous-classe, bien qu'une ArrayStoreException soit émise si vous essayez de définir un élément sur autre chose qu'une String :

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

La syntaxe de raccourci ne peut pas être utilisée pour cela car la syntaxe de raccourci aurait un type implicite d' Object[] .

Un tableau peut être initialisé avec des éléments nuls en utilisant String[] emptyArray = new String[0] . Par exemple, un tableau comme celui-ci est utilisé pour créer un Array partir d'une Collection lorsque la méthode nécessite le type d'exécution d'un objet.

Dans les deux types de primitive et de référence, une initialisation de tableau vide (par exemple, String[] array8 = new String[3]) initialisera le tableau avec la valeur par défaut pour chaque type de données .

Création et initialisation de tableaux de type générique

Dans les classes génériques, les tableaux de types génériques **ne peuvent pas** être initialisés comme ceci en raison de l' effacement de type :

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

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

Au lieu de cela, ils peuvent être créés en utilisant l'une des méthodes suivantes: (notez que cela générera des avertissements non vérifiés)

1. En créant un tableau Object et en le convertissant en type générique:

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

   C'est la méthode la plus simple, mais comme le tableau sous-jacent est toujours de type Object[] , cette méthode ne fournit pas de sécurité de type. Par conséquent, cette méthode de création d'un tableau est mieux utilisée que dans la classe générique - non exposée publiquement.

2. En utilisant Array.newInstance avec un paramètre de classe:

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

   Ici, la classe de T doit être explicitement passée au constructeur. Le type de retour de Array.newInstance est toujours Object . Cependant, cette méthode est plus sûre car le tableau nouvellement créé est toujours de type T[] et peut donc être externalisé en toute

sécurité.

Remplissage d'un tableau après l'initialisation

Java SE 1.2

Arrays.fill() peut être utilisé pour remplir un tableau avec **la même valeur** après l'initialisation:

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

Vivre sur Ideone

fill() peut également attribuer une valeur à chaque élément de la plage spécifiée du tableau:

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

Vivre sur Ideone

Java SE 8

Depuis la version 8 de Java, la méthode setAll et son équivalent Concurrent parallelSetAll peuvent être utilisés pour définir chaque élément d'un tableau sur des valeurs générées. Ces méthodes sont passées à une fonction de générateur qui accepte un index et renvoie la valeur souhaitée pour cette position.

L'exemple suivant crée un tableau d'entiers et définit tous ses éléments sur leur valeur d'index respective:

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

Vivre sur Ideone

Déclaration séparée et initialisation des tableaux

La valeur d'un index pour un élément de tableau doit être un nombre entier (0, 1, 2, 3, 4, ...) et inférieur à la longueur du tableau (les index sont basés sur zéro). Sinon, une exception ArrayIndexOutOfBoundsException sera lancée:

```
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 }
```

Les tableaux ne peuvent pas être réinitialisés avec la syntaxe de raccourci de l'initialiseur de tableau

Il n'est pas possible de réinitialiser un tableau via une syntaxe de raccourci avec un initialiseur de tableau, car un initialiseur de tableau ne peut être spécifié que dans une déclaration de champ ou une déclaration de variable locale ou dans une expression de création de tableau.

Cependant, il est possible de créer un nouveau tableau et de l'assigner à la variable utilisée pour référencer l'ancien tableau. Bien que cela entraîne la ré-initialisation du tableau référencé par cette variable, le contenu de la variable est un tableau complètement nouveau. Pour ce faire, le new opérateur peut être utilisé avec un initialiseur de tableau et affecté à la variable de tableau:

```
// 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.
```

Vivre sur Ideone

Création d'un tableau à partir d'une collection

Deux méthodes dans java.util.Collection créent un tableau à partir d'une collection:

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

Object[] toArray() peut être utilisé comme suit:

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) peut être utilisé comme suit:

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 différence entre eux est plus que juste avoir des résultats non typés vs typés. Leurs

performances peuvent également différer (pour plus de détails, veuillez lire cette section d'analyse des performances):

- Object[] toArray() utilise une arraycopy vectorisée, qui est beaucoup plus rapide que la arraycopy vérifiée par arraycopy utilisée dans T[] toArray(T[] a) .
- T[] toArray(new T[non-zero-size]) doit mettre à zéro le tableau à l'exécution, alors que T[] toArray(new T[0]) ne le fait pas. Un tel évitement rend le dernier appel plus rapide que le premier. Analyse détaillée ici: tableaux de sagesse des anciens .

Java SE 8

À partir de Java SE 8+, où le concept de Stream a été introduit, il est possible d'utiliser le Stream produit par la collection pour créer un nouveau tableau à l'aide de la méthode Stream.toArray .

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

Exemples pris de deux réponses (1 , 2) à Convertir 'ArrayList en' String [] 'en Java sur Stack Overflow.

Tableaux à une chaîne

Java SE 5

Depuis Java 1.5, vous pouvez obtenir une représentation String du contenu du tableau spécifié sans itérer chaque élément. Utilisez simplement Arrays.toString(Object[]) ou Arrays.deepToString(Object[]) pour les tableaux multidimentionnels:

```
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() méthode Arrays.toString() utilise la méthode Object.toString() pour produire les valeurs String de chaque élément du tableau. En plus du tableau de type primitif, il peut être utilisé pour tous les types de tableaux. Par exemple:

```
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!]
```

S'il n'y a pas de substitution à toString() pour la classe, alors le toString() hérité de Object sera utilisé. Généralement, la sortie n'est pas très utile, par exemple:

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

Dog[] arr = { new Dog() };
```

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

Créer une liste à partir d'un tableau

La méthode Arrays.asList() peut être utilisée pour renvoyer une List taille fixe contenant les éléments du tableau donné. La List résultante sera du même type de paramètre que le type de base du tableau.

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

Remarque : cette liste est soutenue par (*une vue* de) le tableau d'origine, ce qui signifie que toute modification apportée à la liste modifiera le tableau et inversement. Cependant, les modifications de la liste qui changeraient sa taille (et donc la longueur du tableau) déclencheront une exception.

Pour créer une copie de la liste, utilisez le constructeur de java.util.ArrayList prenant une Collection en argument:

Java SE 5

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

Java SE 7

Dans Java SE 7 et versions ultérieures, une paire de crochets <> (un ensemble vide d'arguments de type) peut être utilisée, appelée Diamond . Le compilateur peut déterminer les arguments de type à partir du contexte. Cela signifie que les informations de type peuvent être omises lors de l'appel du constructeur de ArrayList et qu'elles seront automatiquement déduites lors de la compilation. Cela s'appelle Type Inference, qui fait partie de Java Generics .

```
// Using Arrays.asList()

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

// Using ArrayList.addAll()

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

// Using Collections.addAll()

String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, stringArray);
```

Un point à noter à propos du diamant est qu'il ne peut pas être utilisé avec les classes anonymes .

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());
```

Remarques importantes relatives à l'utilisation de la méthode Arrays.asList ()

- Cette méthode renvoie List , qui est une instance de Arrays$ArrayList (classe interne statique de Arrays) et non java.util.ArrayList . La List résultante est de taille fixe. Cela signifie que l'ajout ou la suppression d'éléments n'est pas pris en charge et lancera une UnsupportedOperationException :

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

- Une nouvelle List peut être créée par le passage d' un tableau soutenu par List au constructeur d'une nouvelle List . Cela crée une nouvelle copie des données, qui a une taille modifiable et qui n'est pas soutenue par le tableau d'origine:

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

- L'appel de <T> List<T> asList(T... a) sur un tableau primitif, tel qu'un int[] , produira une List<int[]> dont le seul élément est le tableau de primitives source au lieu des éléments réels du tableau source.

  La raison de ce comportement est que les types primitifs ne peuvent pas être utilisés à la place des paramètres de type générique, de sorte que le tableau primitif tout entier remplace le paramètre de type générique dans ce cas. Pour convertir un tableau primitif en List , tout d'abord, convertissez le tableau primitif en un tableau du type wrapper correspondant (c'est-à-dire appelez Arrays.asList sur un Integer[] au lieu d'un int[]).

  Par conséquent, cela va imprimer false :

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

  Voir la démo

  D'un autre côté, cela sera true :

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

  Voir la démo

  Cela imprimera également true , car le tableau sera interprété comme un Integer[]):

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

  Voir la démo

Tableaux multidimensionnels et dentelés

Il est possible de définir un tableau avec plusieurs dimensions. Au lieu d'être accessible en fournissant un index unique, un tableau multidimensionnel est accessible en spécifiant un index pour chaque dimension.

La déclaration du tableau multidimensionnel peut être effectuée en ajoutant [] pour chaque dimension à une valeur de décomposition de tableau régulière. Par exemple, pour créer un tableau

int dimensions, ajoutez un autre ensemble de crochets à la déclaration, par exemple int[][] . Cela continue pour les tableaux à trois dimensions (int[][][]) et ainsi de suite.

Pour définir un tableau à deux dimensions avec trois lignes et trois colonnes:

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

Le tableau peut être indexé et lui attribuer des valeurs avec cette construction. Notez que les valeurs non attribuées sont les valeurs par défaut pour le type d'un tableau, dans ce cas 0 pour int .

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

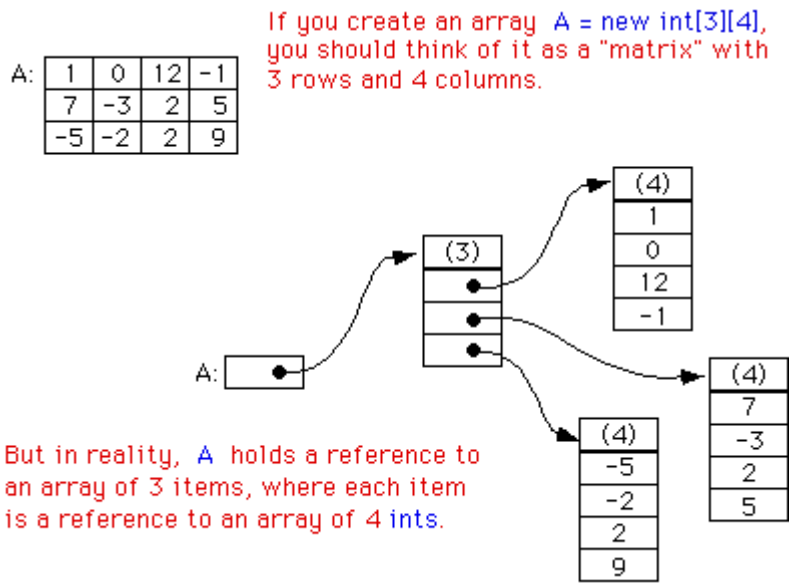
Il est également possible d'instancier une dimension à la fois et même de créer des tableaux non rectangulaires. Celles-ci sont plus communément appelées **tableaux déchiquetés** .

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

Il est important de noter que même s'il est possible de définir une dimension de tableau irrégulier, son niveau précédent ***doit*** être défini.

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

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

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

Comment les tableaux multidimensionnels sont représentés en Java



Source de l'image: http://math.hws.edu/eck/cs124/javanotes3/c8/s5.html

---

Intialisation littérale du tableau dentelé

Les tableaux multidimensionnels et les tableaux irréguliers peuvent également être initialisés avec une expression littérale. Les éléments suivants déclarent et remplissent un tableau int 2x3:

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

Remarque : Les sous-réseaux déchiquetés peuvent également être null . Par exemple, le code suivant déclare et renseigne un tableau int deux dimensions dont le premier sous-tableau est null , le second sous-tableau est de longueur nulle, le troisième sous-tableau a une longueur et le dernier sous-tableau est un tableau à deux longueurs:

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

Pour les tableaux multidimensionnels, il est possible d'extraire des tableaux de dimension inférieure par leurs indices:

```
int[][][] arr = new int[3][3][3];
int[][] arr1 = arr[0]; // get first 3x3-dimensional array from arr
int[] arr2 = arr1[0]; // get first 3-dimensional array from arr1
int[] arr3 = arr[0]; // error: cannot convert from int[][] to int[]
```

ArrayIndexOutOfBoundsException

L' ArrayIndexOutOfBoundsException est ArrayIndexOutOfBoundsException lorsqu'un index non existant d'un tableau est en cours d'accès.

Les tableaux sont indexés sur zéro, donc l'index du premier élément est 0 et l'index du dernier élément est la capacité du tableau moins 1 (ie array.length - 1).

Par conséquent, toute demande d'un élément de tableau par l'indice i doit satisfaire à la condition 0 <= i < array.length , sinon le ArrayIndexOutOfBoundsException sera jeté.

Le code suivant est un exemple simple où une ArrayIndexOutOfBoundsException est levée.

```
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.
```

Sortie:

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

Notez que l'index illégal auquel vous accédez est également inclus dans l'exception (2 dans l'exemple); cette information pourrait être utile pour trouver la cause de l'exception.

Pour éviter cela, vérifiez simplement que l'index est dans les limites du tableau:

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

Obtenir la longueur d'un tableau

Les tableaux sont des objets qui fournissent de l'espace pour stocker jusqu'à sa taille des éléments de type spécifié. La taille d'un tableau ne peut pas être modifiée après la création du tableau.

```
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
```

Le champ length d'un tableau stocke la taille d'un tableau. C'est un final champ et ne peut pas être modifié.

Ce code indique la différence entre la length d'un tableau et la quantité d'objets stockés par un tableau.

```
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");
}
```

Résultat:

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

Comparer les tableaux pour l'égalité

Types Array héritent leurs equals() (et hashCode()) implémentations de java.lang.Object , donc equals() à equals() ne retourne vrai lorsque l'on compare contre exactement le même objet tableau. Pour comparer les tableaux en fonction de leurs valeurs, utilisez java.util.Arrays.equals , qui est surchargé pour tous les types de tableau.

```
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
```

Lorsque le type d'élément est un type de référence, Arrays.equals() appelle equals() sur les éléments du tableau pour déterminer l'égalité. En particulier, si le type d'élément est lui-même un type de tableau, la comparaison d'identité sera utilisée. Pour comparer les tableaux multidimensionnels pour l'égalité, utilisez Arrays.deepEquals() comme ci-dessous:

```
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
```

Étant donné que les ensembles et les cartes utilisent equals() et hashCode() , les tableaux ne sont généralement pas utiles en tant qu'éléments de définition ou clés de carte. Soit les envelopper dans une classe d'assistance qui implémente equals() et hashCode() en fonction des éléments du tableau, soit les convertir en instances List et stocker les listes.

Tableaux à diffuser

Java SE 8

Conversion d'un tableau d'objets en Stream :

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

La conversion d'un tableau de primitives en Stream aide d' Arrays.stream() transformera le tableau en une spécialisation primitive de Stream:

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

Vous pouvez également limiter le Stream à une série d'éléments du tableau. L'index de démarrage est inclusif et l'index de fin est exclusif:

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

Une méthode similaire à Arrays.stream() apparaît dans la classe Stream : Stream.of() . La différence est que Stream.of() utilise un paramètre varargs, vous pouvez donc écrire quelque chose comme:

```
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});
```

Itération sur les tableaux

Vous pouvez effectuer une itération sur les tableaux en utilisant des améliorations pour les boucles (aka foreach) ou en utilisant des index de tableaux:

```
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);
}
```

Il convient de noter qu'il n'ya pas de moyen direct d'utiliser un Iterator sur un tableau, mais qu'il peut être facilement converti en une liste pour obtenir un objet Iterable via la bibliothèque Arrays.

Pour les tableaux en boîte, utilisez `Arrays.asList` :

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

Pour les tableaux primitifs (utilisant java 8), utilisez des flux (en particulier dans cet exemple - `Arrays.stream` -> `IntStream`):

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

Si vous ne pouvez pas utiliser les flux (pas de Java 8), vous pouvez choisir d'utiliser la bibliothèque de goyave de Google:

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

Dans les tableaux à deux dimensions ou plus, les deux techniques peuvent être utilisées de manière un peu plus complexe.

Exemple:

```
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);
    }
}
```

Il est impossible de définir un tableau sur une valeur non uniforme sans utiliser une boucle basée sur un index.

Bien sûr, vous pouvez également utiliser des boucles while ou do-while lors d'une itération à l'aide d'indices.

Une note de prudence: lorsque vous utilisez des indices de tableau, assurez-vous que l'index est compris entre 0 et array.length - 1 (tous deux inclus). Ne faites pas d'hypothèses codées sur la longueur du tableau, sinon vous risquez de casser votre code si la longueur du tableau change, mais pas vos valeurs codées en dur.

Exemple:

```
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;
    }
}
```

Il est également préférable que vous n'utilisiez pas de calculs fantaisistes pour obtenir l'index mais que vous utilisiez l'index pour effectuer des itérations et que si vous avez besoin de valeurs différentes, calculez-les.

Exemple:

```
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;
    }
}
```

Accéder aux tableaux dans l'ordre inverse

```
int[] array = {0, 1, 1, 2, 3, 5, 8, 13};
for (int i = array.length - 1; i >= 0; i--) {
   System.out.println(array[i]);
```

```
}
```

Utilisation de tableaux temporaires pour réduire la répétition du code

Itérer sur un tableau temporaire au lieu de répéter le code peut rendre votre code plus propre. Il peut être utilisé lorsque la même opération est effectuée sur plusieurs variables.

```
// we want to print out all of these
String name = "Margaret";
int eyeCount = 16;
double height = 50.2;
int legs = 9;
int arms = 5;


// copy-paste approach:
System.out.println(name);
System.out.println(eyeCount);
System.out.println(height);
System.out.println(legs);
System.out.println(arms);


// temporary array approach:
for(Object attribute : new Object[]{name, eyeCount, height, legs, arms})
    System.out.println(attribute);

// using only numbers
for(double number : new double[]{eyeCount, legs, arms, height})
    System.out.println(Math.sqrt(number));
```

Gardez à l'esprit que ce code ne doit pas être utilisé dans les sections critiques, car un tableau est créé chaque fois que la boucle est entrée et que les variables primitives sont copiées dans le tableau et ne peuvent donc pas être modifiées.

Copier des tableaux

Java propose plusieurs méthodes pour copier un tableau.

pour la boucle

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

Notez que l'utilisation de cette option avec un tableau Object au lieu d'un tableau primitif remplira la copie en référence au contenu d'origine plutôt qu'à sa copie.

Object.clone ()

Puisque les tableaux sont des Object en Java, vous pouvez utiliser `Object.clone()` .

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

Notez que la méthode Object.clone pour un tableau effectue une **copie superficielle** , c.-à-d. Object.clone renvoie une référence à un nouveau tableau qui référence les **mêmes** éléments que le tableau source.

Arrays.copyOf ()

java.util.Arrays fournit un moyen simple d'effectuer la copie d'un tableau sur un autre. Voici l'utilisation de base:

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

Notez que Arrays.copyOf fournit également une surcharge qui vous permet de changer le type du tableau:

```
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) Copie un tableau du tableau source spécifié, en commençant à la position spécifiée, à la position spécifiée du tableau de destination.

Ci-dessous un exemple d'utilisation

```
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 ()

Principalement utilisé pour copier une partie d'un tableau, vous pouvez également l'utiliser pour copier un tableau entier vers un autre, comme ci-dessous:

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

Bâtis de coulée

Les tableaux sont des objets, mais leur type est défini par le type des objets contenus. Par conséquent, on ne peut pas simplement convertir A[] en T[] , mais chaque membre du A[] spécifique doit être converti en un objet T Exemple générique:

```
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;
}
```

Ainsi, étant donné un tableau A[] :

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

Java SE fournit la méthode Arrays.copyOf(original, newLength, newType) à cet effet:

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

Supprimer un élément d'un tableau

Java ne fournit pas de méthode directe dans java.util.Arrays pour supprimer un élément d'un tableau. Pour l'exécuter, vous pouvez soit copier le tableau d'origine sur un nouveau sans l'élément pour supprimer ou convertir votre tableau dans une autre structure permettant la suppression.

Utiliser ArrayList

Vous pouvez convertir le tableau en java.util.List , supprimer l'élément et convertir la liste en un tableau comme suit:

```
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]
```

Utiliser System.arraycopy

System.arraycopy() peut être utilisé pour faire une copie du tableau d'origine et supprimer l'élément souhaité. Ci-dessous un exemple:

```
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]
```

Utiliser Apache Commons Lang

Pour supprimer facilement un élément, vous pouvez utiliser la bibliothèque Apache Commons Lang et en particulier la méthode statique removeElement() de la classe ArrayUtils . Ci-dessous un exemple:

```
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]
```

Covariance des tableaux

Les tableaux d'objets sont covariants, ce qui signifie que, tout comme Integer est une sous-classe de Number , Integer[] est une sous-classe de Number[] . Cela peut sembler intuitif, mais peut entraîner un comportement surprenant:

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

Bien que Integer[] soit une sous-classe de Number[] , il ne peut contenir que des Integer s, et essayer d'attribuer un élément Long génère une exception d'exécution.

Notez que ce comportement est unique pour les tableaux et peut être évité en utilisant une List générique à la place:

```
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
```

Il n'est pas nécessaire que tous les éléments du tableau partagent le même type, tant qu'ils constituent une sous-classe du type de tableau:

```
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.
```

Comment changez-vous la taille d'un tableau?

La réponse simple est que vous ne pouvez pas faire cela. Une fois qu'un tableau a été créé, sa taille ne peut plus être modifiée. Au lieu de cela, un tableau ne peut être "redimensionné" qu'en créant un nouveau tableau avec la taille appropriée et en copiant les éléments du tableau existant vers le nouveau.

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

Supposons (par exemple) qu'un nouvel élément listOfCities être ajouté au tableau listOfCities défini ci-dessus. Pour ce faire, vous devrez:

1. créer un nouveau tableau de taille 4,
2. copier les 3 éléments existants de l'ancien tableau dans le nouveau tableau aux décalages 0, 1 et 2, et
3. ajouter le nouvel élément au nouveau tableau à l'offset 3.

Il y a plusieurs façons de faire ce qui précède. Avant Java 6, le moyen le plus concis était le suivant:

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

A partir de Java 6, les méthodes Arrays.copyOf et Arrays.copyOfRange peuvent le faire plus simplement:

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

Pour d'autres moyens de copier un tableau, reportez-vous à l'exemple suivant. Gardez à l'esprit que vous devez disposer d'une copie de tableau de longueur différente de l'original lors du redimensionnement.

- Copier des tableaux

Une meilleure alternative au redimensionnement de tableau

Il y a deux inconvénients majeurs à redimensionner un tableau comme décrit ci-dessus:

- C'est inefficace. Pour agrandir un tableau (ou le rendre plus petit), copiez tous les éléments de tableau existants ou tous les allouez et allouez un nouvel objet de tableau. Plus la matrice est grande, plus elle est chère.
- Vous devez pouvoir mettre à jour toutes les variables "live" contenant des références à l'ancien tableau.

Une alternative consiste à créer le tableau avec une taille suffisante pour commencer. Ceci n'est viable que si vous pouvez déterminer cette taille avec précision *avant d'allouer le tableau* . Si vous ne pouvez pas faire cela, le problème de redimensionnement du tableau se pose à nouveau.

L'autre solution consiste à utiliser une classe de structure de données fournie par la bibliothèque de classes Java SE ou une bibliothèque tierce. Par exemple, la structure «collections» de Java SE fournit un certain nombre d'implémentations des API List , Set et Map avec différentes propriétés d'exécution. La classe ArrayList se rapproche le plus des caractéristiques de performance d'un tableau brut (par exemple, recherche O (N), obtention et définition de O (1), insertion et suppression aléatoires O (N)) tout en offrant un redimensionnement plus efficace.

(L'efficacité du redimensionnement pour ArrayList provient de sa stratégie consistant à doubler la taille de la matrice de sauvegarde sur chaque redimensionnement. Dans le cas d'un cas typique, cela signifie que vous ne redimensionnez que occasionnellement. par insert est O(1) . Il peut être possible d'utiliser la même stratégie lors du redimensionnement d'un tableau brut.)

Trouver un élément dans un tableau

Il existe plusieurs façons de trouver l'emplacement d'une valeur dans un tableau. Les exemples de code suivants supposent tous que le tableau est l'un des suivants:

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

De plus, chacun définit l' index ou l' index2 sur l'index de l'élément requis ou sur -1 si l'élément n'est pas présent.

Utiliser Arrays.binarySearch (pour les tableaux triés uniquement)

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

Utiliser un Arrays.asList (pour les tableaux non primitifs uniquement)

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

Utiliser 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
```

Recherche linéaire en boucle

```
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
```

Recherche linéaire en utilisant des bibliothèques tierces telles que org.apache.commons

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

Remarque: L'utilisation d'une recherche linéaire directe est plus efficace que le regroupement dans une liste.

Tester si un tableau contient un élément

Les exemples ci-dessus peuvent être adaptés pour tester si le tableau contient un élément en

testant simplement pour voir si l'indice calculé est supérieur ou égal à zéro.

Il existe également des variantes plus concises:

```
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);
```

Tri des tableaux

Le tri des tableaux peut être facilement effectué avec l'API des tableaux .

```
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));
```

Tri des tableaux de chaînes:

String n'est pas une donnée numérique, il définit son propre ordre appelé ordre lexicographique, également appelé ordre alphabétique. Lorsque vous triez un tableau de String en utilisant la méthode sort() , il trie le tableau dans l'ordre naturel défini par l'interface Comparable, comme indiqué ci-dessous:

Ordre croissant

```
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));
```

Sortie:

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

Ordre décroissant

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

Sortie:

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

Tri d'un tableau d'objets

Afin de trier un tableau d'objets, tous les éléments doivent implémenter l'interface Comparable ou Comparator pour définir l'ordre du tri.

Nous pouvons utiliser l'une des méthodes sort(Object[]) pour trier un tableau d'objets dans son ordre naturel, mais vous devez vous assurer que tous les éléments du tableau doivent implémenter Comparable .

De plus, ils doivent également être mutuellement comparables, par exemple e1.compareTo(e2) ne doit pas lancer une *ClassCastException* pour les éléments e1 et e2 du tableau. Vous pouvez également trier un tableau d'objets sur un ordre personnalisé en utilisant la méthode de sort(T[], Comparator) , comme illustré dans l'exemple suivant.

```
// 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));
```

Sortie:

```
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 ]
```

Conversion de tableaux entre les primitives et les types en boîte

Parfois, la conversion de types primitifs en types encadrés est nécessaire.

Pour convertir le tableau, il est possible d'utiliser des flux (en Java 8 et plus):

Java SE 8

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

Avec des versions inférieures, cela peut être en itérant le tableau primitif et en le copiant explicitement dans le tableau encadré:

Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};
Integer[] boxedArray = new Integer[primitiveArray.length];
for (int i = 0; i < primitiveArray.length; ++i) {
    boxedArray[i] = primitiveArray[i]; // Each element is autoboxed here
}
```

De même, un tableau en boîte peut être converti en un tableau de son homologue primitif:

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
}
```

Lire Tableaux en ligne: https://riptutorial.com/fr/java/topic/99/tableaux

Chapitre 170: Test d'unité

Introduction

Les tests unitaires font partie intégrante du développement piloté par les tests et constituent une fonctionnalité importante pour la création de toute application robuste. En Java, les tests unitaires sont presque exclusivement réalisés à l'aide de bibliothèques et de frameworks externes, dont la plupart ont leur propre balise de documentation. Ce module sert à présenter au lecteur les outils disponibles et leur documentation respective.

Remarques

Cadre de test d'unité

De nombreux frameworks sont disponibles pour les tests unitaires au sein de Java. L'option la plus populaire est de loin JUnit. Il est documenté sous les points suivants:

JUnit

JUnit4 - Balise *proposée pour les fonctionnalités JUnit4; pas encore implémenté* .

D'autres frameworks de test unitaires existent et disposent de documentation:

TestNG

Outils de test unitaires

Il existe plusieurs autres outils utilisés pour les tests unitaires:

Mockito - Cadre moqueur ; permet aux objets d'être imités. Utile pour imiter le comportement **attendu** d'une unité externe dans le test d'une unité donnée, afin de ne pas lier le comportement de l'unité externe aux tests de l'unité donnée.

JBehave - Framework BDD . Permet de relier les tests aux comportements des utilisateurs (permettant la validation des exigences / scénarios). *Aucun tag de document disponible au moment de l'écriture; voici un lien externe* .

Examples

Qu'est-ce que le test d'unité?

C'est un peu une amorce. C'est surtout parce que la documentation est forcée d'avoir un exemple, même si elle est conçue comme un article de remplacement. Si vous connaissez déjà les principes de base des tests unitaires, n'hésitez pas à passer aux remarques où des cadres spécifiques sont mentionnés.

Les tests unitaires garantissent qu'un module donné se comporte comme prévu. Dans les applications à grande échelle, assurer la bonne exécution des modules dans le vide fait partie intégrante de la fidélité des applications.

Considérons le pseudo-exemple suivant (trivial):

```
public class Example {
  public static void main (String args[]) {
    new Example();
  }

  // Application-level test.
  public Example() {
    Consumer c = new Consumer();
```

```
    System.out.println("VALUE = " + c.getVal());
  }

  // Your Module.
  class Consumer {
    private Capitalizer c;

    public Consumer() {
      c = new Capitalizer();
    }

    public String getVal() {
      return c.getVal();
    }
  }

  // Another team's module.
  class Capitalizer {
    private DataReader dr;

    public Capitalizer() {
      dr = new DataReader();
    }

    public String getVal() {
      return dr.readVal().toUpperCase();
    }
  }

  // Another team's module.
  class DataReader {
    public String readVal() {
      // Refers to a file somewhere in your application deployment, or
      // perhaps retrieved over a deployment-specific network.
      File f;
      String s = "data";
      // ... Read data from f into s ...
      return s;
    }
  }
}
```

Donc, cet exemple est trivial; DataReader obtient les données d'un fichier, les transmet au Capitalizer , qui convertit tous les caractères en majuscules, qui sont ensuite transmis au Consumer . Mais le DataReader est fortement lié à notre environnement d’application, donc nous reportons les tests de cette chaîne jusqu’à ce que nous soyons prêts à déployer une version de test.

Maintenant, supposons, quelque part dans une version, pour des raisons inconnues, la méthode getVal() de Capitalizer remplacée par le retour d'une toUpperCase() à une chaîne toLowerCase() :

```
  // Another team's module.
  class Capitalizer {
    ...

    public String getVal() {
      return dr.readVal().toLowerCase();
    }
  }
```

Clairement, cela brise le comportement attendu. Mais, en raison des processus ardus impliqués

dans l'exécution du DataReader , nous ne le remarquerons pas avant notre prochain déploiement de test. Ainsi, les jours / semaines / mois passent avec ce bogue dans notre système, puis le chef de produit le constate et se tourne instantanément vers vous, le chef d'équipe associé au Consumer . "Pourquoi ça se passe? Qu'est-ce que vous avez changé?" De toute évidence, vous êtes naïf. Vous n'avez aucune idée de ce qui se passe. Vous n'avez modifié aucun code qui devrait toucher à cela; pourquoi est-il brisé soudainement?

Finalement, après discussion entre les équipes et la collaboration, le problème est tracé et le problème résolu. Mais, cela pose la question; Comment cela aurait-il pu être évité?

Il y a deux choses évidentes:

Les tests doivent être automatisés

Notre confiance dans les tests manuels a permis à ce bug de passer inaperçu beaucoup trop longtemps. Nous avons besoin d'un moyen d'automatiser le processus d'introduction **instantanée des** bogues. Pas 5 semaines à partir de maintenant. Pas 5 jours à partir de maintenant. Pas 5 minutes à partir de maintenant. Maintenant.

Vous devez comprendre que, dans cet exemple, j'ai présenté un bug **très trivial** qui a été présenté et qui est passé inaperçu. Dans une application industrielle, avec des dizaines de modules constamment mis à jour, ceux-ci peuvent s'introduire partout. Vous corrigez quelque chose avec un module, seulement pour vous rendre compte que le comportement même que vous "corrigiez" était utilisé d'une manière ou d'une autre (en interne ou en externe).

Sans validation rigoureuse, les choses vont se glisser dans le système. Il est possible que, si on les néglige assez, cela se traduira par beaucoup de travail supplémentaire en essayant de corriger les changements (et la fixation de ces corrections, etc.), qu'un produit **augmentera** effectivement dans le travail restant que l' effort est mis. Vous ne voulez pas être dans cette situation.

Les tests doivent être précis

Le deuxième problème noté dans notre exemple ci-dessus est le temps nécessaire pour tracer le bogue. Le responsable de produit vous a envoyé une requête lorsque les testeurs l'ont remarqué, vous avez enquêté et constaté que le Capitalizer renvoyait des données apparemment mauvaises, vous avez envoyé un message à l'équipe Capitalizer avec vos conclusions, etc.

Le même point que j'ai fait ci-dessus à propos de la quantité et de la difficulté de cet exemple trivial existe ici. De toute évidence, toute personne raisonnablement bien familiarisée avec Java pourrait trouver le problème introduit rapidement. Mais il est souvent beaucoup plus difficile de retracer et de communiquer les problèmes. Peut-être que l'équipe Capitalizer vous a fourni un JAR sans source. Peut-être sont-ils situés à l'autre bout du monde, et les heures de communication sont très limitées (peut-être pour les e-mails envoyés une fois par jour). Cela peut entraîner des bogues prenant des semaines ou plus à tracer (et, encore une fois, il pourrait y en avoir plusieurs pour une version donnée).

Afin d'atténuer ce problème, nous souhaitons des tests rigoureux à un niveau aussi **fin** que possible (vous souhaitez également des tests grossiers pour vous assurer que les modules interagissent correctement, mais ce n'est pas notre objectif principal). Nous souhaitons spécifier de manière rigoureuse le fonctionnement de toutes les fonctionnalités tournées vers l'extérieur (au minimum) et tester cette fonctionnalité.

Entrer les tests unitaires

Imaginez si nous avions un test, en nous assurant spécifiquement que la méthode getVal() de Capitalizer getVal() une chaîne en majuscule pour une chaîne d'entrée donnée. De plus, imaginez que ce test ait été exécuté avant même que nous ayons commis un code. Le bogue introduit dans le système (qui est, toUpperCase() étant remplacé par toLowerCase()) causerait aucun problème parce que le bug ne serait jamais introduit dans le système. Nous le prendrions dans un test, le développeur réaliserait (espérons-le) son erreur, et une solution alternative serait trouvée quant à la manière d'introduire l'effet souhaité.

Il y a des omissions ici concernant la **manière** de mettre en œuvre ces tests, mais celles-ci sont couvertes dans la documentation spécifique au framework (liée dans les remarques). J'espère que cela sert d'exemple pour **expliquer pourquoi les** tests unitaires sont importants.

Lire Test d'unité en ligne: https://riptutorial.com/fr/java/topic/8155/test-d-unite

Chapitre 171: Tokenizer de chaîne

Introduction

La classe java.util.StringTokenizer vous permet de diviser une chaîne en jetons. C'est un moyen simple de casser la chaîne.

L'ensemble des délimiteurs (les caractères qui séparent les jetons) peut être spécifié au moment de la création ou sur une base par jeton.

Examples

StringTokenizer Split par espace

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

Sortie:

Pomme

ballon

chat

chien

StringTokenizer Split par une virgule ','

```
public static void main(String args[]) {
        StringTokenizer st = new StringTokenizer("apple,ball cat,dog", ",");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
```

Sortie:

Pomme

Balle à chat

chien

Lire Tokenizer de chaîne en ligne: https://riptutorial.com/fr/java/topic/10563/tokenizer-de-chaine

Chapitre 172: Traitement des arguments en ligne de commande

Syntaxe

- `main statique vide statique (String [] args)`

Paramètres

Paramètre	Détails
args	Les arguments de la ligne de commande. En supposant que la méthode `main` soit appelée par le lanceur Java, `args` sera non nul et n'aura aucun élément `null` .

Remarques

Lorsqu'une application Java standard est lancée à l'aide de la commande java (ou équivalente), une méthode main sera appelée, transmettant les arguments de la ligne de commande dans le tableau args .

Malheureusement, les bibliothèques de classes Java SE ne fournissent aucun support direct pour le traitement des arguments de commande. Cela vous laisse deux alternatives:

- Implémenter le traitement des arguments à la main en Java.
- Utilisez une bibliothèque tierce.

Ce sujet tentera de couvrir certaines des bibliothèques tierces les plus populaires. Pour une liste complète des alternatives, consultez cette réponse à la question StackOverflow "Comment analyser les arguments de ligne de commande en Java?" .

Examples

Traitement des arguments à l'aide de GWT ToolBase

Si vous voulez analyser des arguments de ligne de commande plus complexes, par exemple avec des paramètres facultatifs, il est préférable d'utiliser l'approche GWT de Google. Toutes les classes sont publiques à:

https://gwt.googlesource.com/gwt/+/2.8.0-beta1/dev/core/src/com/google/gwt/util/tools/ToolBase.java

Un exemple de traitement de la ligne de commande myprogram -dir "~/Documents" -port 8888 est:

```
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 également une méthode isRequired() qui peut être écrasée pour indiquer que l'argument de ligne de commande est requis (le retour par défaut est false pour que l'argument soit facultatif).

Traitement des arguments à la main

Lorsque la syntaxe de ligne de commande d'une application est simple, il est raisonnable de traiter l'argument de commande entièrement en code personnalisé.

Dans cet exemple, nous présenterons une série d'études de cas simples. Dans chaque cas, le code produira des messages d'erreur si les arguments sont inacceptables, puis appelez System.exit(1) pour indiquer au shell que la commande a échoué. (Nous supposerons dans chaque cas que le code Java est appelé à l'aide d'un wrapper dont le nom est "myapp".)

Une commande sans arguments

Dans cette étude de cas, la commande ne nécessite aucun argument. Le code montre que args.length nous donne le nombre d'arguments de la ligne de commande.

```
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");
    }
}
```

Une commande avec deux arguments

Dans cette étude de cas, la commande nécessite précisément deux arguments.

```
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]);
    }
}
```

Notez que si nous args.length vérifier args.length , la commande planterait si l'utilisateur l' args.length avec trop peu d'arguments en ligne de commande.

Une commande avec les options "flag" et au moins un argument

Dans cette étude de cas, la commande dispose de deux options d'indicateur (facultatif) et nécessite au moins un argument après les options.

```
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);
   }
}
```

Comme vous pouvez le constater, le traitement des arguments et des options devient assez compliqué si la syntaxe de commande est compliquée. Il est conseillé d'utiliser une bibliothèque "d'analyse de ligne de commande"; voir les autres exemples.

Lire Traitement des arguments en ligne de commande en ligne:

https://riptutorial.com/fr/java/topic/4775/traitement-des-arguments-en-ligne-de-commande

Chapitre 173: TreeMap et TreeSet

Introduction

TreeMap et TreeSet sont des collections Java de base ajoutées à Java 1.2. TreeMap est une implémentation de Map **mutable** , **ordonnée** . De même, TreeSet est une implémentation d' Set **ordonnée** et **mutable** .

TreeMap est implémenté comme un arbre rouge-noir, qui fournit des temps d'accès O(log n) . TreeSet est implémenté en utilisant un TreeMap avec des valeurs factices.

Les deux collections **ne** sont **pas** compatibles avec les threads.

Examples

TreeMap d'un type Java simple

Tout d'abord, nous créons une carte vide et y insérons des éléments:

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");
```

Une fois que nous avons quelques éléments dans la carte, nous pouvons effectuer certaines opérations:

```
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
```

Nous pouvons également parcourir les éléments de la carte en utilisant soit un itérateur, soit une boucle foreach. Notez que les entrées sont imprimées en fonction de leur ordre naturel et non de l'ordre d'insertion:

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 d'un type Java simple

Tout d'abord, nous créons un ensemble vide et y insérons des éléments:

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);
```

Une fois que nous avons quelques éléments dans l'ensemble, nous pouvons effectuer certaines opérations:

```
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
```

Nous pouvons également parcourir les éléments de la carte en utilisant soit un itérateur, soit une boucle foreach. Notez que les entrées sont imprimées en fonction de leur ordre naturel et non de l'ordre d'insertion:

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 d'un type Java personnalisé

Depuis que TreeMap s et TreeSet conservent les clés / éléments en fonction de leur ordre naturel . Les clés TreeMap et les éléments TreeSet doivent donc être comparables.

Disons que nous avons une classe Person :

```
public class Person  {

    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constuctors, getters, setters and various methods
}
```

Si nous le stockons tel quel dans un TreeSet (ou une clé dans un TreeMap):

```
TreeSet<Person2> set = ...
set.add(new Person(1,"first","last",Date.from(Instant.now())));
```

Ensuite, nous avons rencontré une exception telle que celle-ci:

```
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)
```

Pour corriger cela, supposons que nous voulions ordonner les instances de Person fonction de l'ordre de leurs identifiants (private int id). Nous pourrions le faire de deux manières:

1. Une solution consiste à modifier la Person afin qu'elle implémente l' interface 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. Une autre solution consiste à fournir à TreeSet un comparateur :

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());
    }
});
```

Cependant, il y a deux mises en garde pour les deux approches:

1. Il est **très important de** ne modifier aucun champ utilisé pour la commande une fois qu'une instance a été insérée dans un TreeSet / TreeMap . Dans l'exemple ci-dessus, si nous modifions l' id d'une personne déjà insérée dans la collection, nous pourrions rencontrer un comportement inattendu.

2. Il est important de mettre en œuvre la comparaison correctement et de manière cohérente. Selon le Javadoc :

   Le réalisateur doit assurer sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) pour tous les x et y. (Cela implique que x.compareTo(y) doit lancer une exception ssi

y.compareTo(x) lance une exception)

L'implémenteur doit également s'assurer que la relation est transitive:
(x.compareTo(y)>0 && y.compareTo(z)>0) implique x.compareTo(z)>0 .

Enfin, l'implémenteur doit s'assurer que x.compareTo(y)==0 implique que
sgn(x.compareTo(z)) == sgn(y.compareTo(z)) , pour tout z.

TreeMap et TreeSet Thread Safety

TreeMap et TreeSet **ne** sont **pas** des collections thread-safe, il convient donc de veiller à ce qu'ils soient utilisés dans des programmes multithread.

TreeMap et TreeSet sont tous deux sûrs lorsqu'ils sont lus, même simultanément, par plusieurs threads. Donc, si elles ont été créées et remplies par un seul thread (par exemple, au début du programme), et seulement après avoir lu, mais pas modifié par plusieurs threads, il n'y a aucune raison de synchroniser ou de verrouiller.

Toutefois, si elle est lue et modifiée simultanément ou modifiée simultanément par plusieurs threads, la collection peut générer une exception ConcurrentModificationException ou se comporter de manière inattendue. Dans ces cas, il est impératif de synchroniser / verrouiller l'accès à la collection en utilisant l'une des méthodes suivantes:

1. Utilisation de Collections.synchronizedSorted.. :

```
SortedSet<Integer> set = Collections.synchronizedSortedSet(new TreeSet<Integer>());
SortedMap<Integer,String> map = Collections.synchronizedSortedMap(new
TreeMap<Integer,String>());
```

   Cela fournira une SortedSet / SortedMap mise en œuvre soutenue par la collecte effective et synchronisée sur un objet mutex. Notez que cela synchronisera tous les accès en lecture et en écriture à la collection sur un seul verrou, de sorte que même les lectures simultanées ne seraient pas possibles.

2. En synchronisant manuellement certains objets, comme la collection elle-même:

```
 TreeSet<Integer> set = new TreeSet<>();
```

   ...

```
//Thread 1
synchronized (set) {
    set.add(4);
}
```

   ...

```
//Thread 2
synchronized (set) {
    set.remove(5);
}
```

3. En utilisant un verrou, tel qu'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();
```

Contrairement aux méthodes de synchronisation précédentes, l'utilisation d'un ReadWriteLock permet à plusieurs threads de lire simultanément sur la carte.

Lire TreeMap et TreeSet en ligne: https://riptutorial.com/fr/java/topic/9905/treemap-et-treeset

Chapitre 174: Types atomiques

Introduction

Les types atomiques Java sont de simples types mutables qui fournissent des opérations de base qui sont sûres et atomiques sans avoir recours au verrouillage. Ils sont destinés à être utilisés dans des cas où le verrouillage constituerait un goulot d'étranglement ou un risque de blocage ou de blocage.

Paramètres

Paramètre	La description
ensemble	Ensemble volatil du champ
obtenir	Lecture volatile du champ
lazySet	Ceci est un magasin commandé opération du champ
compareAndSet	Si la valeur est la valeur d'expiration, elle est envoyée à la nouvelle valeur
getAndSet	obtenir la valeur actuelle et mettre à jour

Remarques

Beaucoup sur essentiellement des combinaisons de lectures ou écritures volatiles et des opérations CAS . La meilleure façon de comprendre cela est d'examiner directement le code source. Par exemple , AtomicInteger , Unsafe.getAndSet

Examples

Création de types atomiques

Pour un code multithread simple, l'utilisation de la synchronisation est acceptable. Cependant, l'utilisation de la synchronisation a un impact sur la vie, et comme une base de code devient plus complexe, il est probable que vous vous retrouviez avec un blocage , une famine ou un blocage .

Dans le cas d'une concurrence plus complexe, l'utilisation de variables atomiques constitue souvent une meilleure alternative, car elle permet d'accéder à une variable individuelle de manière sécurisée sans utiliser de méthodes synchronisées ou de blocs de code.

Créer un type AtomicInteger :

```
AtomicInteger aInt = new AtomicInteger() // Create with default value 0

AtomicInteger aInt = new AtomicInteger(1) // Create with initial value 1
```

De même pour d'autres types d'instance.

```
AtomicIntegerArray aIntArray = new AtomicIntegerArray(10) // Create array of specific length
AtomicIntegerArray aIntArray = new AtomicIntegerArray(new int[] {1, 2, 3}) // Initialize array
with another array
```

De même pour d'autres types atomiques.

Il y a une exception notable: il n'y a pas de type float et double . Celles-ci peuvent être simulées en utilisant Float.floatToIntBits(float) et Float.intBitsToFloat(int) pour float ainsi que Double.doubleToLongBits(double) et Double.longBitsToDouble(long) pour les doubles.

Si vous souhaitez utiliser sun.misc.Unsafe vous pouvez utiliser n'importe quelle variable primitive comme atomique en utilisant l'opération atomique dans sun.misc.Unsafe . Tous les types primitifs doivent être convertis ou codés en int ou en longueurs pour l'utiliser de cette manière. Pour plus d'informations, voir: sun.misc.Unsafe .

Motivation pour les types atomiques

Le moyen le plus simple d'implémenter des applications multithread consiste à utiliser les primitives de synchronisation et de verrouillage intégrées à Java; Par exemple, le mot clé synchronized . L'exemple suivant montre comment nous pourrions utiliser synchronized comptes synchronized pour accumuler des comptes.

```
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;
    }
}
```

Cette implémentation fonctionnera correctement. Toutefois, si un grand nombre de threads effectuent de nombreux appels simultanés sur le même objet Counters , la synchronisation risque d'être un goulot d'étranglement. Plus précisément:

1. Chaque appel de méthode synchronized commencera par le thread en cours qui acquiert le verrou de l'instance Counters .
2. Le thread tiendra le verrou pendant qu'il vérifie la valeur du number et met à jour le compteur.
3. Enfin, il libère le verrou, autorisant l'accès à d'autres threads.

Si un thread essaie d'acquérir le verrou alors qu'un autre le maintient, le thread qui tente de le bloquer sera bloqué à l'étape 1 jusqu'à ce que le verrou soit libéré. Si plusieurs threads sont en attente, l'un d'entre eux l'obtiendra et les autres continueront à être bloqués.

Cela peut entraîner quelques problèmes:

- S'il y a beaucoup de *conflits* pour le verrou (c'est-à-dire que beaucoup de threads essaient de l'acquérir), alors certains threads peuvent être bloqués pendant longtemps.
- Lorsqu'un thread est bloqué dans l'attente du verrou, le système d'exploitation essaie généralement de passer l'exécution à un autre thread. Ce *changement de contexte* entraîne un

impact relativement important sur les performances du processeur.

- Lorsqu'il y a plusieurs threads bloqués sur le même verrou, il n'y a aucune garantie que l'un d'entre eux soit traité "équitablement" (c'est-à-dire que l'exécution de chaque thread est garantie). Cela peut conduire à la *famine du fil* .

Comment met-on en œuvre les types atomiques?

Commençons par réécrire l'exemple ci-dessus en utilisant les compteurs 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;
    }
}
```

Nous avons remplacé le int[] par un AtomicInteger[] et l'initialisé avec une instance dans chaque élément. Nous avons également ajouté des appels à incrementAndGet() et get() à la place des opérations sur les valeurs int .

Mais le plus important est de pouvoir supprimer le mot-clé synchronized car le verrouillage n'est plus requis. Cela fonctionne parce que les opérations incrementAndGet() et get() sont *atomiques* et *thread-safe* . Dans ce contexte, cela signifie que:

- Chaque compteur dans le tableau ne sera *observable que* dans l'état "avant" pour une opération (comme un "incrément") ou dans l'état "après".

- En supposant que l'opération a lieu à l'instant T , aucun thread ne pourra voir l'état "avant" après l'heure T

De plus, bien que deux threads puissent réellement tenter de mettre à jour la même instance AtomicInteger en même temps, les implémentations des opérations garantissent qu'un seul incrément se produit à la fois sur l'instance donnée. Cela se fait sans verrouillage, ce qui entraîne souvent de meilleures performances.

Comment fonctionnent les types atomiques?

Les types atomiques s'appuient généralement sur des instructions matérielles spécialisées dans

le jeu d'instructions de la machine cible. Par exemple, les jeux d'instructions basés sur Intel fournissent une instruction CAS (Compare and Swap) qui effectuera une séquence spécifique d'opérations de mémoire de manière atomique.

Ces instructions de bas niveau sont utilisées pour implémenter des opérations de niveau supérieur dans les API des classes AtomicXxx respectives. Par exemple, (encore une fois, en pseudocode 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;
    }
  }
}
```

S'il n'y a pas de conflit sur AtomicXxxx , le test if réussira et la boucle se terminera immédiatement. S'il y a contention, alors le if échouera pour tous les threads sauf un, et ils "tourneront" dans la boucle pour un petit nombre de cycles de la boucle. En pratique, la rotation est de l'ordre de grandeur plus rapide (sauf à *des* niveaux de conflit *irréalistes* , où les performances synchronisées sont meilleures que celles des classes atomiques car lorsque l'opération CAS échoue, alors la tentative ne fera qu'ajouter à la suspension) un.

Incidemment, les instructions CAS sont généralement utilisées par la JVM pour mettre en œuvre *un verrouillage non réglementé* . Si la JVM peut voir qu'un verrou n'est pas actuellement verrouillé, il tentera d'utiliser un CAS pour acquérir le verrou. Si le CAS réussit, il n'est alors pas nécessaire de procéder à la planification de thread, au changement de contexte, etc. Pour plus d'informations sur les techniques utilisées, voir Verrouillage biaisé dans HotSpot .

Lire Types atomiques en ligne: https://riptutorial.com/fr/java/topic/5963/types-atomiques

Chapitre 175: Types de données de référence

Examples

Instancier un type de référence

```
Object obj = new Object(); // Note the 'new' keyword
```

Où:

- Object est un type de référence.
- obj est la variable dans laquelle stocker la nouvelle référence.
- Object() est l'appel à un constructeur d' Object .

Ce qui se produit:

- L'espace en mémoire est alloué à l'objet.
- Le constructeur Object() est appelé pour initialiser cet espace mémoire.
- L'adresse mémoire est stockée dans obj , de sorte qu'elle *référence* l'objet nouvellement créé.

Ceci est différent des primitives:

```
int i = 10;
```

Où la valeur réelle 10 est stockée dans i .

Déréférencer

Déréférencer se passe avec le . opérateur:

```
Object obj = new Object();
String text = obj.toString(); // 'obj' is dereferenced.
```

Le déréférencement *suit* l'adresse de mémoire stockée dans une référence, à l'endroit en mémoire où réside l'objet réel. Lorsqu'un objet a été trouvé, la méthode demandée est appelée (toString dans ce cas).

Lorsqu'une référence a la valeur null , le déréférencement entraîne une exception NullPointerException :

```
Object obj = null;
obj.toString(); // Throws a NullpointerException when this statement is executed.
```

null indique l'absence d'une valeur, c'est- *à*- dire que *suivre* l'adresse de la mémoire ne mène nulle part. Il n'y a donc aucun objet sur lequel la méthode demandée peut être appelée.

Lire Types de données de référence en ligne: https://riptutorial.com/fr/java/topic/1046/types-de-donnees-de-reference

Chapitre 176: Types de données primitifs

Introduction

Les 8 types de données primitifs byte , short , int , long , char , boolean , float et double sont les types qui stockent la plupart des données numériques brutes dans les programmes Java.

Syntaxe

- int aInt = 8; // La partie définissant (nombre) de cette déclaration int est appelée un littéral.
- int hexInt = 0x1a; // = 26; Vous pouvez définir des littéraux avec des valeurs hexadécimales préfixées par *0x* .
- int binInt = 0b11010; // = 26; Vous pouvez également définir des littéraux binaires; préfixé avec *0b* .
- long goodLong = 10000000000L; // Par défaut, les littéraux entiers sont de type int. En ajoutant le L à la fin du littéral, vous indiquez au compilateur que le littéral est long. Sans cela, le compilateur émettrait une erreur "nombre entier trop grand".
- double aDouble = 3,14; // Les littéraux à virgule flottante sont de type double par défaut.
- float aFloat = 3.14F; // Par défaut, ce littéral aurait été un double (et provoqué une erreur "Types incompatibles"), mais en ajoutant un F, nous indiquons au compilateur qu'il s'agit d'un flottant.

Remarques

Java a 8 *types de données primitifs* , à savoir boolean , byte , caractères short , char , int , long , float et double . (Tous les autres types sont *des* types de *référence* . Cela inclut tous les types de tableau et les types / classes d'objets intégrés qui ont une signification particulière dans le langage Java; par exemple, String , Class et Throwable et ses sous-classes.)

Le résultat de l' ensemble des opérations (addition, soustraction, multiplication, etc) sur un type primitif est au moins un int , ajoutant ainsi un short à un short produit un int , tout comme l' ajout d' un byte d'un byte , ou une char à une char . Si vous souhaitez affecter le résultat à une valeur du même type, vous devez le lancer. par exemple

```
byte a = 1;
byte b = 2;
byte c = (byte) (a + b);
```

Ne pas lancer l'opération entraînera une erreur de compilation.

Cela est dû à la partie suivante de Java Language Spec, §2.11.1 :

> Un compilateur encode des charges de valeurs littérales de types byte et short utilisant des instructions Java Virtual Machine qui étendent ces valeurs aux valeurs de type int à la compilation ou à l'exécution. Les charges de valeurs littérales des types boolean et char sont codées en utilisant des instructions qui étendent le littéral à zéro à une valeur de type int à la compilation ou à l'exécution. [..]. Ainsi, la plupart des opérations sur les valeurs des types réels boolean , byte , char et short sont effectuées correctement par des instructions fonctionnant sur des valeurs de type computational int .

La raison derrière cela est également spécifiée dans cette section:

> Étant donné la **taille de l'opcode un octet de** la machine virtuelle Java, les types d'encodage en opcodes exercent une pression sur la conception de son jeu d'instructions. Si chaque instruction typée prenait en charge tous les types de données d'exécution de la machine virtuelle Java, il y aurait plus d'instructions que celles pouvant être représentées dans un byte . [...] Des instructions distinctes peuvent être utilisées pour convertir entre des types de données non pris en charge et pris en charge, selon les besoins.

Examples

La primitive int

Un type de données primitif tel que int contient des valeurs directement dans la variable qui l'utilise, tandis qu'une variable déclarée à l'aide d' Integer contient une référence à la valeur.

Selon l' API java : "La classe Integer encapsule une valeur du type primitif int dans un objet. Un objet de type Integer contient un seul champ dont le type est int."

Par défaut, int est un entier signé 32 bits. Il peut stocker une valeur minimale de -2^{31} et une valeur maximale de 2^{31-1} .

```
int example = -42;
int myInt = 284;
int anotherInt = 73;

int addedInts = myInt + anotherInt; // 284 + 73 = 357
int subtractedInts = myInt - anotherInt; // 284 - 73 = 211
```

Si vous devez stocker un nombre en dehors de cette plage, utilisez plutôt long . Le dépassement de la plage de valeurs de int conduit à un dépassement d'entier, ce qui entraîne l'ajout de la valeur supérieure à la plage sur le site opposé de la plage (le positif devient négatif et vice versa). La valeur est ((value - MIN_VALUE) % RANGE) + MIN_VALUE ou ((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
```

Les valeurs maximales et minimales de int peuvent être trouvées à:

```
int high = Integer.MAX_VALUE;    // high == 2147483647
int low = Integer.MIN_VALUE;     // low == -2147483648
```

La valeur par défaut d'un int est *0*

```
int defaultInt;    // defaultInt == 0
```

La courte primitive

Un short - short est un entier signé de 16 bits. Il a une valeur minimale de -2^{15} (-32 768), et une valeur maximale de $2^{15}-1$ (32767)

```
short example = -48;
short myShort = 987;
short anotherShort = 17;
```

```
short addedShorts = (short) (myShort + anotherShort); // 1,004
short subtractedShorts = (short) (myShort - anotherShort); // 970
```

Les valeurs maximales et minimales de short peuvent être trouvées à:

```
short high = Short.MAX_VALUE;        // high == 32767
short low = Short.MIN_VALUE;         // low == -32768
```

La valeur par défaut d'un short est *0*

```
short defaultShort;    // defaultShort == 0
```

La longue primitive

Par défaut, long est un entier signé de 64 bits (dans Java 8, il peut être signé ou non signé). Signé, il peut stocker une valeur minimale de -2^{63} , et une valeur maximale de $2^{63} - 1$, et non signé il peut stocker une valeur minimale de 0 et une valeur maximale de $2^{64} - 1$

```
long example = -42;
long myLong = 284;
long anotherLong = 73;

//an "L" must be appended to the end of the number, because by default,
//numbers are assumed to be the int type. Appending an "L" makes it a long
//as 549755813888 (2 ^ 39) is larger than the maximum value of an int (2^31 - 1),
//"L" must be appended
long bigNumber = 549755813888L;

long addedLongs = myLong + anotherLong; // 284 + 73 = 357
long subtractedLongs = myLong - anotherLong; // 284 - 73 = 211
```

Les valeurs maximales et minimales de long peuvent être trouvées à:

```
long high = Long.MAX_VALUE;    // high == 9223372036854775807L
long low = Long.MIN_VALUE;     // low == -9223372036854775808L
```

La valeur par défaut d'un long est *0L*

```
long defaultLong;    // defaultLong == 0L
```

Note: la lettre "L" ajoutée à la fin du littéral long est insensible à la casse, mais il est conseillé d'utiliser le majuscule car il est plus facile de le distinguer du chiffre 1:

```
2L == 2l;            // true
```

Avertissement: Java met en cache les instances d'objets entiers de la plage allant de -128 à 127. Le raisonnement est expliqué ici:
https://blogs.oracle.com/darcy/entry/boxing_and_caches_integer_valueof

Les résultats suivants peuvent être trouvés:

```
Long val1 = 127L;
Long val2 = 127L;
```

```
System.out.println(val1 == val2); // true

Long val3 = 128L;
Long val4 = 128L;

System.out.println(val3 == val4); // false
```

Pour comparer correctement 2 valeurs longues d'objet, utilisez le code suivant (à partir de Java 1.7):

```
Long val3 = 128L;
Long val4 = 128L;

System.out.println(Objects.equal(val3, val4)); // true
```

Comparer une longueur primitive à un objet long n'entraînera pas de faux négatif, par exemple en comparant 2 objets à ==.

La primitive booléenne

Un boolean peut stocker une des deux valeurs, true ou 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
```

La valeur par défaut d'un boolean est *false*

```
boolean defaultBoolean;    // defaultBoolean == false
```

La primitive d'octet

Un byte est un entier signé de 8 bits. Il peut stocker une valeur minimale de -2^7 (-128) et une valeur maximale de $2^7 - 1$ (127)

```
byte example = -36;
byte myByte = 96;
byte anotherByte = 7;

byte addedBytes = (byte) (myByte + anotherByte); // 103
byte subtractedBytes = (byte) (myBytes - anotherByte); // 89
```

Les valeurs maximales et minimales de l' byte sont disponibles à l'adresse suivante:

```
byte high = Byte.MAX_VALUE;        // high == 127
byte low = Byte.MIN_VALUE;         // low == -128
```

La valeur par défaut d'un byte est *0*

```
byte defaultByte;    // defaultByte == 0
```

La primitive float

Un float est un nombre à virgule flottante IEEE 754 32 bits à simple précision. Par défaut, les décimales sont interprétées comme des doubles. Pour créer un float , ajoutez simplement un f au littéral décimal.

```
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
```

Les flotteurs gèrent les cinq opérations arithmétiques courantes: addition, soustraction, multiplication, division et module.

Remarque: Les erreurs suivantes peuvent légèrement varier. Certains résultats ont été arrondis pour des raisons de clarté et de lisibilité (le résultat imprimé de l'exemple d'addition était en fait 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
```

En raison de la façon dont les nombres à virgule flottante sont stockés (c'est-à-dire sous forme binaire), de nombreux nombres n'ont pas de représentation exacte.

```
float notExact = 3.1415926f;
System.out.println(notExact); // 3.1415925
```

Bien que l'utilisation de float soit appropriée pour la plupart des applications, ni float ni double ne doivent être utilisés pour stocker des représentations exactes de nombres décimaux (comme des montants monétaires) ou des nombres nécessitant une plus grande précision. Au lieu de cela, la classe BigDecimal doit être utilisée.

La valeur par défaut d'un float est *0.0f* .

```
float defaultFloat;    // defaultFloat == 0.0f
```

Un float est précis à environ une erreur de 1 sur 10 millions.

Remarque: Float.POSITIVE_INFINITY , Float.NEGATIVE_INFINITY , Float.NaN sont des valeurs float . NaN signifie les résultats des opérations qui ne peuvent pas être déterminés, tels que la division de 2 valeurs infinies. De plus, 0f et -0f sont différents, mais == donne une valeur vraie:

```
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
```

La double primitive

Un double est un nombre à virgule flottante IEEE 754 64 bits à double précision.

```
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
```

En raison de la façon dont les nombres à virgule flottante sont stockés, beaucoup de nombres n'ont pas de représentation exacte.

```
double notExact = 1.32 - 0.42; // result should be 0.9
System.out.println(notExact); // 0.9000000000000001
```

Bien que l'utilisation du double soit appropriée pour la plupart des applications, ni float ni double ne doivent être utilisés pour stocker des chiffres précis tels que la devise. Au lieu de cela, la classe BigDecimal doit être utilisée

La valeur par défaut d'un double est *0.0d*

```
public double defaultDouble;    // defaultDouble == 0.0
```

Remarque: Double.POSITIVE_INFINITY , Double.NEGATIVE_INFINITY , Double.NaN sont double valeurs double . NaN signifie les résultats des opérations qui ne peuvent pas être déterminés, tels que la division de 2 valeurs infinies. De plus, 0d et -0d sont différents, mais == donne une valeur vraie:

```
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
```

La primitive char

Un char peut stocker un seul caractère Unicode 16 bits. Un littéral de caractère est placé entre guillemets simples

```
char myChar = 'u';
char myChar2 = '5';
char myChar3 = 65; // myChar3 == 'A'
```

Il a une valeur minimale de \u0000 (0 dans la représentation décimale, également appelée le *caractère nul*) et une valeur maximale de \uffff (\uffff).

La valeur par défaut d'un char est \u0000 .

```
char defaultChar;    // defaultChar == \u0000
```

Afin de définir un produit de carbonisation de ' valeur d' une séquence d'échappement (caractère précédé d'une barre oblique inverse) doit être utilisée:

```
char singleQuote = '\'';
```

Il y a aussi d'autres séquences d'échappement:

```
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
```

Vous pouvez déclarer une char de caractère Unicode.

```
char heart = '\u2764';
System.out.println(Character.toString(heart)); // Prints a line containing "❤".
```

Il est également possible d'ajouter à un char . Par exemple, pour parcourir chaque lettre minuscule, vous pouvez effectuer les opérations suivantes:

```
for (int i = 0; i <= 26; i++) {
    char letter = (char) ('a' + i);
    System.out.println(letter);
}
```

Représentation de valeur négative

Java et la plupart des autres langages stockent des nombres intégraux négatifs dans une représentation appelée notation *du complément 2* .

Pour une représentation binaire unique d'un type de données utilisant n bits, les valeurs sont codées comme suit:

Les n-1 bits les moins significatifs stockent un nombre entier positif x dans la représentation intégrale. La valeur la plus significative stocke un bit avec la valeur s . La valeur

représentée par ces bits est

$x - s * 2^{n-1}$

c'est-à-dire que si le bit le plus significatif est 1, alors une valeur juste supérieure de 1 au nombre que vous pourriez représenter avec les autres bits ($2^{n-2} + 2^{n-3} + ... + 2^1 + 2^0 = 2^{n-1} - 1$) est soustrait permettant une représentation binaire unique pour chaque valeur de -2^{n-1} ($s = 1; x = 0$) à $2^{n-1} - 1$ ($s = 0; x = 2^{n-1} - 1$).

Cela a aussi l'effet secondaire agréable, que vous pouvez ajouter les représentations binaires comme s'il s'agissait de nombres binaires positifs:

$$v1 = x1 - s1 * 2^{n-1}$$
$$v2 = x2 - s2 * 2^{n-1}$$

s1	s2	débordement x1 + x2	résultat supplémentaire
0	0	Non	x1 + x2 = v1 + v2
0	0	Oui	trop grand pour être représenté avec le type de données (débordement)
0	1	Non	$x1 + x2 - 2^{n-1} = x1 + x2 - s2 * 2^{n-1} = v1 + v2$
0	1	Oui	$(x1 + x2) \bmod 2^{n-1} = x1 + x2 - 2^{n-1} = v1 + v2$
1	0	*	voir ci-dessus (sommets de swap)
1	1	Non	trop petit pour être représenté avec le type de données (x1 + x2 - 2^n <-2^{n-1} ; sous - dépassement)
1	1	Oui	$(x1 + x2) \bmod 2^{n-1} - 2^{n-1} = (x1 + x2 - 2^{n-1}) - 2^{n-1} = (x1 - s1 * 2^{n-1}) + (x2 - s2 * 2^{n-1}) = v1 + v2$

Notez que ce fait facilite la recherche de représentation binaire de l'inverse additif (c'est-à-dire la valeur négative):

Observez que l'ajout du complément binaire au nombre a pour résultat que tous les bits sont 1. Maintenant, ajoutez 1 pour que la valeur soit dépassée et vous obtenez l'élément neutre 0 (tous les bits 0).

Donc, la valeur négative d'un nombre i peut être calculée en utilisant (en ignorant la promotion possible à int ici)

```
(~i) + 1
```

Exemple: en prenant la valeur négative de 0 (byte):

Le résultat de la négation 0 est 11111111 . L'ajout de 1 donne une valeur de 100000000 (9 bits). Un byte ne pouvant stocker que 8 bits, la valeur la plus à gauche est tronquée et le résultat est 00000000

Original	Processus	Résultat
0 (00000000)	Nier	-0 (11111111)
11111111	Ajouter 1 au binaire	100000000
100000000	Tronquer à 8 bits	00000000 (-0 est égal à 0)

Consommation de mémoire des primitives vs primitives en boîte

Primitif	Type en boîte	Taille de la mémoire de la primitive / en boîte
booléen	Booléen	1 octet / 16 octets
octet	Octet	1 octet / 16 octets
court	Court	2 octets / 16 octets
carboniser	Carboniser	2 octets / 16 octets
int	Entier	4 octets / 16 octets
longue	Longue	8 octets / 16 octets
flotte	Flotte	4 octets / 16 octets
double	Double	8 octets / 16 octets

Les objets en boîte nécessitent toujours 8 octets pour la gestion des types et de la mémoire, et comme la taille des objets est toujours un multiple de 8, les types en boîte *nécessitent un total de 16 octets* . En *outre* , chaque utilisation d'un objet en boîte implique le stockage d'une référence qui représente 4 ou 8 octets *supplémentaires* , en fonction des options de la machine virtuelle Java et de la machine virtuelle Java.

Dans les opérations gourmandes en données, la consommation de mémoire peut avoir un impact majeur sur les performances. La consommation de mémoire augmente encore plus lorsque vous utilisez des tableaux: un tableau float[5] ne nécessite que 32 octets; alors qu'un Float[5] stockant 5 valeurs non nulles distinctes nécessitera un total de 112 octets (sur 64 bits sans pointeurs compressés, cela augmente à 152 octets).

Caches de valeur en boîte

Les frais généraux d'espace des types encadrés peuvent être atténués dans une certaine mesure par les caches de valeur encadrés. Certains types encadrés implémentent un cache d'instances. Par exemple, par défaut, la classe Integer mettra en cache les instances pour représenter des nombres compris entre −128 et +127 . Cela ne réduit cependant pas le coût supplémentaire lié à l'indirection supplémentaire de la mémoire.

Si vous créez une instance d'un type en boîte soit par la mise en file d'attente automatique, soit en appelant la méthode statique valueOf(primitive) , le système d'exécution tente d'utiliser une valeur mise en cache. Si votre application utilise beaucoup de valeurs dans la plage mise en cache, cela peut réduire considérablement la pénalité de mémoire liée à l'utilisation de types encadrés. Certes, si vous créez des instances de valeurs encadrées "à la main", il vaut mieux utiliser valueOf plutôt que new . (La new opération crée toujours une nouvelle instance.) Si, toutefois, la majorité de vos valeurs *ne se* trouvent *pas* dans la plage mise en cache, il peut être plus rapide d'appeler new et d'enregistrer la recherche de cache.

Conversion de primitifs

En Java, nous pouvons convertir entre des valeurs entières et des valeurs à virgule flottante. En outre, étant donné que chaque caractère correspond à un numéro dans le codage Unicode, char types peuvent être convertis et des entiers et des types à virgule flottante. boolean est le seul type de données primitif qui ne peut être converti en aucun autre type de données primitif.

Il existe deux types de conversions: l' *élargissement de la conversion* et la *réduction de la conversion* .

Une *conversion élargie* se produit lorsqu'une valeur d'un type de données est convertie en une valeur d'un autre type de données qui occupe plus de bits que le premier. Il n'y a pas de problème de perte de données dans ce cas.

De manière correspondante, une *conversion restreinte* se produit lorsqu'une valeur d'un type de données est convertie en une valeur d'un autre type de données qui occupe moins de bits que le premier. La perte de données peut se produire dans ce cas.

Java effectue automatiquement des *conversions* élargies. Mais si vous souhaitez effectuer une *conversion restreinte* (si vous êtes certain qu'aucune perte de données ne se produira), vous pouvez forcer Java à effectuer la conversion en utilisant une structure de langage appelée cast .

Conversion d'élargissement:

```
int a = 1;
double d = a;    // valid conversion to double, no cast needed (widening)
```

Conversion rétrécie:

```
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
```

Types de primitives

Tableau indiquant la taille et la plage de valeurs de tous les types primitifs:

Type de données	représentation numérique	gamme de valeurs	valeur par défaut
booléen	n / a	faux et vrai	faux
octet	8 bits signés	-2^7 à $2^7 - 1$	0

Type de données	représentation numérique	gamme de valeurs	valeur par défaut
		-128 à +127	
court	16 bits signé	-2 ^{15} à 2 ^{15} - 1	0
		-32 768 à +32 767	
int	32 bits signés	-2 ^{31} à 2 ^{31} - 1	0
		-2 147 483 648 à +2 147 483 647	
longue	64 bits signés	-2 ^{63} à 2 ^{63} - 1	0L
		-9.223.372.036.854.775.808 à 9.223.372.036.854.775.807	
flotte	Virgule flottante 32 bits	1.401298464e-45 à 3.402823466e + 38 (positif ou négatif)	0,0F
double	Virgule flottante 64 bits	4.94065645841246544e-324d à 1.79769313486231570e + 308d (positif ou négatif)	0.0D
carboniser	16 bits non signés	0 à 2 ^{16} - 1	0
		0 à 65,535	

Remarques:

1. La spécification de langage Java stipule que les types intégraux signés (byte sur long) utilisent la représentation binaire à double complément, et que les types à virgule flottante utilisent des représentations à virgule flottante binaire standard IEE 754.
2. Java 8 et versions ultérieures fournissent des méthodes pour effectuer des opérations arithmétiques non signées sur int et long . Bien que ces méthodes permettent à un programme de *traiter les* valeurs des types respectifs comme non signés, les types restent des types signés.
3. Le plus petit point flottant montré ci-dessus est *sous - normal* ; c'est-à-dire qu'ils ont moins de précision qu'une valeur *normale* . Les plus petits nombres normaux sont 1.175494351e - 38 et 2.2250738585072014e - 308
4. Une char représente classiquement une *unité de code* Unicode / UTF-16.
5. Bien qu'un boolean contienne qu'un seul bit d'information, sa taille en mémoire varie en fonction de l'implémentation de Java Virtual Machine (voir le type booléen).

Lire Types de données primitifs en ligne: https://riptutorial.com/fr/java/topic/148/types-de-donnees-primitifs

Chapitre 177: Types de référence

Examples

Différents types de référence

java.lang.ref package java.lang.ref fournit des classes d'objets de référence, qui prennent en charge un degré d'interaction limité avec le garbage collector.

Java possède quatre principaux types de référence. Elles sont:

- Référence forte
- Référence faible
- Référence souple
- Référence fantôme

1. référence forte

C'est la forme habituelle de création d'objets.

```
MyObject myObject = new MyObject();
```

Le détenteur de la variable contient une référence forte à l'objet créé. Tant que cette variable est MyObject et contient cette valeur, l'instance MyObject ne sera pas collectée par le garbage collector.

2. Référence faible

Lorsque vous ne souhaitez pas conserver un objet plus longtemps et que vous devez effacer / libérer la mémoire allouée à un objet dès que possible, c'est la manière de procéder.

```
WeakReference myObjectRef = new WeakReference(MyObject);
```

Simplement, une référence faible est une référence qui n'est pas assez forte pour forcer un objet à rester en mémoire. Les références faibles vous permettent d'exploiter la capacité du ramasse-miettes à déterminer l'accessibilité pour vous, vous n'avez donc pas à le faire vous-même.

Lorsque vous avez besoin de l'objet que vous avez créé, utilisez .get() méthode .get() :

```
myObjectRef.get();
```

Le code suivant illustrera ceci:

```
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. Référence souple

Les références molles sont légèrement plus fortes que les références faibles. Vous pouvez créer un objet référencé comme suit:

```
SoftReference myObjectRef = new SoftReference(MyObject);
```

Ils peuvent retenir la mémoire plus fortement que la référence faible. Si vous avez suffisamment de mémoire / ressources en mémoire, le garbage collector ne nettoiera pas les références logicielles avec autant d'enthousiasme que les références faibles.

Les références logicielles sont pratiques à utiliser dans la mise en cache. Vous pouvez créer des objets référencés en tant que cache, où ils sont conservés jusqu'à épuisement de la mémoire. Lorsque votre mémoire ne peut pas fournir suffisamment de ressources, le ramasse-miettes supprime les références logicielles.

```
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. Référence fantôme

C'est le type de référence le plus faible. Si vous avez créé une référence d'objet à l'aide de Phantom Reference, la méthode get() renverra toujours null!

L'utilisation de ce référencement est la suivante: "Les objets de référence Phantom, qui sont mis en file d'attente après le collecteur, déterminent que leurs référents peuvent être récupérés. Les références fantômes sont le plus souvent utilisées pour planifier des actions de nettoyage pré-mortem Mécanisme de finalisation Java. " - De la référence fantôme Javadoc d'Oracle.

Vous pouvez créer un objet de référence fantôme comme suit:

```
PhantomReference myObjectRef = new PhantomReference(MyObject);
```

Lire Types de référence en ligne: https://riptutorial.com/fr/java/topic/4017/types-de-reference

Chapitre 178: Utilisation de ThreadPoolExecutor dans les applications MultiThreaded.

Introduction

Lors de la création d'une application performante et pilotée par des données, il peut être très utile d'effectuer des tâches chronophages de manière asynchrone et d'exécuter simultanément plusieurs tâches. Cette rubrique présente le concept d'utilisation de ThreadPoolExecutors pour effectuer plusieurs tâches Ansynchrones simultanément.

Examples

Exécution de tâches asynchrones pour lesquelles aucune valeur de retour n'est requise à l'aide d'un instance de classe runnable

Certaines applications peuvent vouloir créer des tâches dites "Fire & Forget" qui peuvent être déclenchées périodiquement et n'ont pas besoin de renvoyer un type de valeur renvoyé à la fin de la tâche assignée (par exemple, purge d'anciens fichiers temporaires, Etat).

Dans cet exemple, nous allons créer deux classes: une qui implémente l'interface Runnable et l'autre qui contient une méthode 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'exécution de AsyncExample1.main () a généré la sortie suivante:

```
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
```

Observations de Note: Il y a plusieurs choses à noter dans le résultat ci-dessus,

1. Les tâches ne s'exécutaient pas dans un ordre prévisible.
2. Étant donné que chaque tâche dormait pendant une durée (pseudo) aléatoire, elle ne s'est pas nécessairement terminée dans l'ordre dans lequel elle a été appelée.

Exécution de tâches asynchrones lorsqu'une valeur de retour est nécessaire à l'aide d'un instance de classe appelable

Il est souvent nécessaire d'exécuter une tâche longue et d'utiliser le résultat de cette tâche une fois celle-ci terminée.

Dans cet exemple, nous allons créer deux classes: une qui implémente l'interface Callable <T> (où T est le type que nous souhaitons renvoyer) et l'autre qui contient une méthode 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'exécution d'AsyncExample2.main () a généré la sortie suivante:

```
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
```

Observations de note:

Il y a plusieurs choses à noter dans la sortie ci-dessus,

1. Chaque appel à ExecutorService.submit () a renvoyé une instance de Future, qui était stockée dans une liste pour une utilisation ultérieure.
2. Future contient une méthode appelée isDone () qui peut être utilisée pour vérifier si notre tâche est terminée avant d'essayer de vérifier sa valeur de retour. L'appel de la méthode Future.get () sur un Future qui n'est pas encore effectué bloque le thread en cours jusqu'à la fin de la tâche, annulant potentiellement les nombreux avantages de l'exécution asynchrone de la tâche.
3. La méthode executorService.shutdown () a été appelée avant de vérifier les valeurs de retour des objets Future. Ce n'est pas obligatoire, mais a été fait de cette manière pour montrer que c'est possible. La méthode executorService.shutdown () n'empêche pas l'achèvement des tâches qui ont déjà été soumises à ExecutorService, mais empêche plutôt l'ajout de nouvelles tâches à la file d'attente.

Définition de tâches asynchrones en ligne à l'aide de Lambdas

Bien qu'une bonne conception logicielle maximise souvent la réutilisabilité du code, il peut parfois être utile de définir des tâches asynchrones en ligne dans votre code via des expressions Lambda pour optimiser la lisibilité du code.

Dans cet exemple, nous allons créer une classe unique contenant une méthode main (). Dans cette méthode, nous utiliserons des expressions Lambda pour créer et exécuter des instances de Callable et 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);
    }
}
```

Observations de note:

Il y a plusieurs choses à noter dans la sortie ci-dessus,

1. Les expressions lambda ont accès aux variables et aux méthodes disponibles pour l'étendue dans laquelle elles sont définies, mais toutes les variables doivent être finales (ou effectivement finales) pour être utilisées dans une expression lambda.
2. Nous n'avons pas à spécifier si notre expression Lambda est explicitement un Callable ou un Runnable <T>, le type de retour est automatiquement inféré par le type de retour.

Lire Utilisation de ThreadPoolExecutor dans les applications MultiThreaded. en ligne: https://riptutorial.com/fr/java/topic/8646/utilisation-de-threadpoolexecutor-dans-les-applications-multithreaded-

Chapitre 179: Utilisation du mot clé static

Syntaxe

- public static int myVariable; // Déclaration d'une variable statique
- public static myMethod () {} // Déclaration d'une méthode statique
- double final statique public MY_CONSTANT; // Déclaration d'une variable constante partagée par toutes les instances de la classe
- double final public MY_CONSTANT; // Déclaration d'une variable constante spécifique à cette instance de la classe (mieux utilisée dans un constructeur qui génère une constante différente pour chaque instance)

Examples

Utilisation de static pour déclarer des constantes

Comme le mot-clé static est utilisé pour accéder aux champs et aux méthodes sans classe instanciée, il peut être utilisé pour déclarer des constantes à utiliser dans d'autres classes. Ces variables resteront constantes dans toutes les instanciations de la classe. Par convention, static variables static sont toujours ALL_CAPS et utilisent des ALL_CAPS soulignement plutôt que des cas camel. ex:

```
 static E STATIC_VARIABLE_NAME
```

Comme les constantes ne peuvent pas changer, static peut également être utilisé avec le modificateur final :

Par exemple, pour définir la constante mathématique de pi:

```
public class MathUtilities {

    static final double PI = 3.14159265358

}
```

Qui peut être utilisé dans n'importe quelle classe en tant que constante, par exemple:

```
public class MathCalculations {

    //Calculates the circumference of a circle
    public double calculateCircumference(double radius) {
        return (2 * radius * MathUtilities.PI);
    }

}
```

Utiliser statique avec cette

Statique fournit une méthode ou une variable de stockage qui n'est *pas* allouée pour chaque instance de la classe. La variable statique est plutôt partagée entre tous les membres de la classe. Incidemment, en essayant de traiter la variable statique comme un membre de l'instance de la classe, vous obtenez un avertissement:

```
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
```

Les méthodes déclarées statiques se comportent de la même manière, mais avec une restriction supplémentaire:

Vous ne pouvez pas utiliser this mot this clé en eux!

```
public class Pineapple {

    private static int numberOfSpikes;
    private int age;

    public static getNumberOfSpikes() {
        return this.numberOfSpikes; // This doesn't compile
    }


    public static getNumberOfSpikes() {
        return numberOfSpikes; // This compiles
    }

}
```

En général, il est préférable de déclarer les méthodes génériques qui s'appliquent à différentes instances d'une classe (telles que les méthodes clones) static , tout en conservant les méthodes telles que equals() comme non statiques. La méthode main d'un programme Java est toujours statique, ce qui signifie que le mot this clé this ne peut pas être utilisé dans main() .

Référence à un membre non statique à partir d'un contexte statique

Les variables et méthodes statiques ne font pas partie d'une instance. Il y aura toujours une seule copie de cette variable, quel que soit le nombre d'objets que vous créez pour une classe particulière.

Par exemple, vous pourriez vouloir avoir une liste de constantes immuable, ce serait une bonne idée de la garder statique et de l'initialiser une seule fois dans une méthode statique. Cela vous donnerait un gain de performance significatif si vous créez régulièrement plusieurs instances d'une classe particulière.

De plus, vous pouvez également avoir un bloc statique dans une classe. Vous pouvez l'utiliser pour affecter une valeur par défaut à une variable statique. Ils ne sont exécutés qu'une fois lorsque la classe est chargée en mémoire.

Les variables d'instance, comme le nom l'indique, dépendent d'une instance d'un objet particulier, et vivent pour en satisfaire les caprices. Vous pouvez jouer avec eux pendant un cycle de vie particulier d'un objet.

Tous les champs et méthodes d'une classe utilisée dans une méthode statique de cette classe doivent être statiques ou locaux. Si vous essayez d'utiliser des variables ou des méthodes d'instance (non statiques), votre code ne sera pas compilé.

```
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
    }

}
```

Lire Utilisation du mot clé static en ligne:
https://riptutorial.com/fr/java/topic/2253/utilisation-du-mot-cle-static

Chapitre 180: Utiliser d'autres langages de script en Java

Introduction

Java en soi est un langage extrêmement puissant, mais sa puissance peut encore être étendue Grâce à JSR223 (Java Specification Request 223), qui introduit un moteur de script

Remarques

L'API de script Java permet aux scripts externes d'interagir avec Java

L'API de script peut permettre une interaction entre le script et Java. Les langages de script doivent avoir une implémentation de Script Engine sur le classpath.

Par défaut, JavaScript (également appelé ECMAScript) est fourni par nashorn par défaut. Chaque moteur de script a un contexte de script où toutes les variables, fonctions et méthodes sont stockées dans des liaisons. Parfois, vous pouvez utiliser plusieurs contextes car ils prennent en charge la redirection de la sortie vers un enregistreur en mémoire tampon et une erreur vers une autre.

Il existe de nombreuses autres bibliothèques de moteur de script comme Jython et JRuby. Tant qu'ils sont sur le classpath, vous pouvez évaluer le code.

Nous pouvons utiliser des liaisons pour exposer des variables dans le script. Dans certains cas, nous avons besoin de plusieurs liaisons, car l'exposition des variables au moteur consiste essentiellement à exposer les variables uniquement à ce moteur. Parfois, nous devons exposer certaines variables comme l'environnement système et le chemin d'accès pour tous les moteurs du même type. Dans ce cas, nous avons besoin d'une liaison de portée globale. Exposition de variables à ceux qui l'exposent à tous les moteurs de script créés par le même EngineFactory

Examples

Évaluation d'un fichier javascript en mode script de nashorn

```
public class JSEngine {

    /*
    * Note Nashorn is only available for Java-8 onwards
    * You can use rhino from ScriptEngineManager.getEngineByName("js");
    */

    ScriptEngine engine;
    ScriptContext context;
    public Bindings scope;

    // Initialize the Engine from its factory in scripting mode
    public JSEngine(){
        engine = new NashornScriptEngineFactory().getScriptEngine("-scripting");
        // Script context is an interface so we need an implementation of it
        context = new SimpleScriptContext();
        // Create bindings to expose variables into
        scope = engine.createBindings();
    }

    // Clear the bindings to remove the previous variables
    public void newBatch(){
        scope.clear();
    }

    public void execute(String file){
```

```
        try {
            // Get a buffered reader for input
            BufferedReader br = new BufferedReader(new FileReader(file));
            // Evaluate code, with input as bufferdReader
            engine.eval(br);
        } catch (FileNotFoundException ex) {
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        } catch (ScriptException ex) {
            // Script Exception is basically when there is an error in script
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    public void eval(String code){
        try {
            // Engine.eval basically treats any string as a line of code and evaluates it,
executes it
            engine.eval(code);
        } catch (ScriptException ex) {
            // Script Exception is basically when there is an error in script
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    // Apply the bindings to the context and set the engine's default context
    public void startBatch(int SCP){
        context.setBindings(scope, SCP);
        engine.setContext(context);
    }

    // We use the invocable interface to access methods from the script
    // Invocable is an optional interface, please check if your engine implements it
    public Invocable invocable(){
        return (Invocable)engine;
    }

}
```

Maintenant la méthode 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);");
}
```

Votre sortie devrait être similaire à celle-ci
hello world

Comme vous pouvez le voir, la variable exposée x a été imprimée. Maintenant, testez avec un fichier.

Ici nous avons test.js

```
print(x);
function test(){
    print("hello test.js:test");
}
test();
```

Et la méthode principale mise à jour

```
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");
}
```

En supposant que test.js se trouve dans le même répertoire que votre application Vous devriez avoir une sortie similaire à celle-ci

```
hello world
hello test.js:test
```

Lire Utiliser d'autres langages de script en Java en ligne:
https://riptutorial.com/fr/java/topic/9926/utiliser-d-autres-langages-de-script-en-java

Chapitre 181: Varargs (argument variable)

Remarques

Un argument de méthode «varargs» permet aux appelants de cette méthode de spécifier plusieurs arguments du type désigné, chacun étant un argument distinct. Il est spécifié dans la déclaration de méthode par trois points ASCII (...) après le type de base.

La méthode elle-même reçoit ces arguments sous la forme d'un tableau unique, dont le type d'élément est le type de l'argument varargs. Le tableau est créé automatiquement (bien que les appelants soient toujours autorisés à passer un tableau explicite au lieu de transmettre plusieurs valeurs en tant qu'arguments de méthode distincts).

Règles pour varargs:

1. Varargs doit être le dernier argument.
2. Il ne peut y avoir qu'un seul Varargs dans la méthode.

Vous devez suivre les règles ci-dessus sinon le programme donnera une erreur de compilation.

Examples

Spécifier un paramètre varargs

```
void doSomething(String... strings) {
    for (String s : strings) {
        System.out.println(s);
    }
}
```

Les trois périodes après le type de paramètre final indiquent que l'argument final peut être passé en tant que tableau ou en tant que séquence d'arguments. Varargs ne peut être utilisé que dans la position d'argument final.

Travailler avec les paramètres de Varargs

En utilisant varargs comme paramètre pour une définition de méthode, il est possible de passer un tableau ou une séquence d'arguments. Si une séquence d'arguments est passée, ils sont automatiquement convertis en tableau.

Cet exemple montre à la fois un tableau et une séquence d'arguments passés dans la méthode printVarArgArray() et leur traitement identique dans le code de la méthode:

```
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);
    }
}
```

Sortie:

```
10,20,
5,6,5,8,6,31
```

Si vous définissez la méthode comme celle-ci, cela donnera des erreurs de compilation.

```
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
```

Lire Varargs (argument variable) en ligne: https://riptutorial.com/fr/java/topic/1948/varargs--argument-variable-

Chapitre 182: Visibilité (contrôle de l'accès aux membres d'une classe)

Syntaxe

- nom de type public [= valeur];
- nom de type privé [= valeur];
- nom de type protégé [= valeur];
- tapez nom [= valeur];
- nom de classe publique {
- nom du cours{

Remarques

A partir du tutoriel Java :

Les modificateurs de niveau d'accès déterminent si d'autres classes peuvent utiliser un champ particulier ou invoquer une méthode particulière. Il y a deux niveaux de contrôle d'accès:

- Au niveau supérieur - public ou *package-private* (pas de modificateur explicite).
- Au niveau des membres: public , private , protected ou *package-privé* (aucun modificateur explicite).

Une classe peut être déclarée avec le modificateur public , auquel cas cette classe est visible par toutes les classes partout. Si une classe n'a pas de modificateur (valeur par défaut, également appelée *package-private*), elle n'est visible que dans son propre package.

Au niveau des membres, vous pouvez également utiliser le modificateur public ou aucun modificateur (*package-private*) de la même manière qu'avec les classes de niveau supérieur et avec la même signification. Pour les membres, il existe deux modificateurs d'accès supplémentaires: private et protected . Le modificateur private spécifie que le membre ne peut être accédé que dans sa propre classe. Le modificateur protected spécifie que le membre ne peut être accédé que dans son propre package (comme avec *package-private*) et, en outre, par une sous-classe de sa classe dans un autre package.

Le tableau suivant montre l'accès aux membres permis par chaque modificateur.

Niveaux d'accès:

Modificateur	Classe	Paquet	Sous classe	Monde
public	Y	Y	Y	Y
protected	Y	Y	Y	N
pas de modificateur	Y	Y	N	N
private	Y	N	N	N

Examples

Membres d'interface

```
public interface MyInterface {
    public void foo();
    int bar();
```

```
    public String TEXT = "Hello";
    int ANSWER = 42;

    public class X {
    }

    class Y {
    }
}
```

Les membres d'interface ont toujours une visibilité publique, même si le mot clé public est omis. Ainsi, foo() , bar() , TEXT , ANSWER , X et Y ont une visibilité publique. Cependant, l'accès peut encore être limité par l'interface contenant - MyInterface ayant une visibilité publique, ses membres peuvent être accédés de n'importe où, mais si MyInterface avait eu la visibilité du package, ses membres n'auraient été accessibles qu'à partir du même package.

Visibilité publique

Visible à la classe, au package et à la sous-classe.

Voyons un exemple avec la classe Test.

```
public class Test{
    public int number = 2;

    public Test(){

    }
}
```

Essayons maintenant de créer une instance de la classe. Dans cet exemple, **nous pouvons** accéder au number car il est public .

```
public class Other{

    public static void main(String[] args){
        Test t = new Test();
        System.out.println(t.number);
    }

}
```

Visibilité privée

private visibilité private permet à une variable d'être uniquement accessible par sa classe. Ils sont souvent utilisés en **conjonction** avec public getters et les installateurs public .

```
class SomeClass {
    private int variable;

    public int getVariable() {
        return variable;
    }

    public void setVariable(int variable) {
        this.variable = variable;
    }
```

```
}

public class SomeOtherClass {
    public static void main(String[] args) {
        SomeClass sc = new SomeClass();

        // These statement won't compile because SomeClass#variable is private:
        sc.variable = 7;
        System.out.println(sc.variable);

        // Instead, you should use the public getter and setter:
        sc.setVariable(7);
        System.out.println(sc.getVariable());
    }
}
```

Visibilité du package

Sans **modificateur** , la valeur par défaut est la visibilité du package. *Dans la documentation Java,* "[visibilité du package] indique si les classes du même package que la classe (indépendamment de leur parenté) ont accès au membre." Dans cet exemple de javax.swing ,

```
package javax.swing;
public abstract class JComponent extends Container … {
    …
    static boolean DEBUG_GRAPHICS_LOADED;
    …
}
```

DebugGraphics est dans le même package, de sorte que DEBUG_GRAPHICS_LOADED est accessible.

```
package javax.swing;
public class DebugGraphics extends Graphics {
    …
    static {
        JComponent.DEBUG_GRAPHICS_LOADED = true;
    }
    …
}
```

Cet article donne quelques informations sur le sujet.

Visibilité protégée

La visibilité protégée signifie que ce membre est visible pour son package, ainsi que ses sous-classes.

Par exemple:

```
package com.stackexchange.docs;
public class MyClass{
    protected int variable; //This is the variable that we are trying to access
    public MyClass(){
        variable = 2;
    };
}
```

Nous allons maintenant étendre cette classe et essayer d'accéder à l'un de ses membres protected .

```
package some.other.pack;
import com.stackexchange.docs.MyClass;
public class SubClass extends MyClass{
    public SubClass(){
        super();
        System.out.println(super.variable);
    }
}
```

Vous pourrez également accéder à un membre protected sans le prolonger si vous y accédez à partir du même package.

Notez que ce modificateur ne fonctionne que sur les membres d'une classe, pas sur la classe elle-même.

Résumé des modificateurs d'accès des membres du groupe

Modificateur d'accès	Visibilité	Héritage
Privé	Classe seulement	Ne peut pas être hérité
Aucun modificateur / paquet	En paquet	Disponible si la sous-classe dans le package
Protégé	En paquet	Disponible dans la sous-classe
Publique	Partout	Disponible dans la sous-classe

Il était une fois un modificateur private protected (les deux mots-clés à la fois) qui pouvait être appliqué aux méthodes ou aux variables pour les rendre accessibles depuis une sous-classe en dehors du paquet, mais les rendre privées aux classes de ce paquet. Cependant, cela a été supprimé dans la version de Java 1.0 .

Lire Visibilité (contrôle de l'accès aux membres d'une classe) en ligne: https://riptutorial.com/fr/java/topic/134/visibilite--controle-de-l-acces-aux-membres-d-une-classe-

Chapitre 183: WeakHashMap

Introduction

Concepts de Hashmap faible

Examples

Concepts de WeakHashmap

Points clés:-

- Mise en œuvre de la carte.
- ne stocke que des références faibles à ses clés.

Références faibles : Les objets référencés uniquement par des références faibles sont récupérés avec empressement; le GC n'attendra pas d'avoir besoin de mémoire dans ce cas.

Difficulté entre Hashmap et WeakHashMap: -

Si le gestionnaire de mémoire Java n'a plus de référence forte à l'objet spécifié en tant que clé, alors l'entrée dans la carte sera supprimée dans WeakHashMap.

Exemple :-

```
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"));
    }
```

Différences de taille (HashMap vs WeakHashMap):

L'appel de la méthode size () sur l'objet HashMap renverra le même nombre de paires clé-valeur. La taille diminuera uniquement si la méthode remove () est appelée explicitement sur l'objet HashMap.

Comme le ramasse-miettes peut ignorer les clés à tout moment, WeakHashMap peut se comporter comme si un thread inconnu supprimait silencieusement des entrées. Il est donc possible que la méthode size retourne des valeurs plus petites avec le temps. Ainsi, dans **WeakHashMap, la diminution de la taille se produit automatiquement** .

Lire WeakHashMap en ligne: https://riptutorial.com/fr/java/topic/10749/weakhashmap

Chapitre 184: XJC

Introduction

XJC est un outil Java SE qui compile un fichier de schéma XML en classes Java entièrement annotées.

Il est distribué dans le package JDK et se trouve dans le chemin /bin/xjc .

Syntaxe

- xjc [options] fichier de schéma / URL / dir / jar ... [-b bindinfo] ...

Paramètres

Paramètre	Détails
fichier de schéma	Le fichier de schéma xsd à convertir en Java

Remarques

L'outil XJC est disponible dans le cadre du JDK. Il permet de créer du code Java annoté avec des annotations JAXB adaptées à la (dés) marshallation.

Examples

Générer du code Java à partir d'un simple fichier XSD

Schéma XSD (schema.xsd)

Le schéma xml suivant (xsd) définit une liste d'utilisateurs avec le name et la reputation attributs.

```
<?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>
```

Utiliser xjc

Cela nécessite que le chemin d'accès à l'outil xjc (fichiers binaires JDK) se trouve dans la variable de chemin du système d'exploitation.

La génération de code peut être démarrée en utilisant

```
xjc schema.xsd
```

Cela générera des fichiers Java dans le répertoire de travail.

Fichiers de résultats

Il y aura des commentaires supplémentaires, mais les fichiers Java générés ressemblent à ceci:

```
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;
```

Lire XJC en ligne: https://riptutorial.com/fr/java/topic/4538/xjc

Chapitre 185: XOM - Modèle d'objet XML

Examples

Lecture d'un fichier XML

Pour charger les données XML avec XOM, vous devez créer un Builder partir duquel vous pouvez le créer dans un Document .

```
Builder builder = new Builder();
Document doc = builder.build(file);
```

Pour obtenir l'élément racine, le parent le plus élevé du fichier xml, vous devez utiliser la getRootElement() sur l'instance du Document .

```
Element root = doc.getRootElement();
```

Maintenant, la classe Element a beaucoup de méthodes pratiques qui facilitent la lecture de XML. Certains des plus utiles sont énumérés ci-dessous:

- getChildElements(String name) - Retourne une instance Elements qui agit comme un tableau d'éléments
- getFirstChildElement(String name) - renvoie le premier élément enfant avec cette balise.
- getValue() - Retourne la valeur dans l'élément.
- getAttributeValue(String name) - Retourne la valeur d'un attribut avec le nom spécifié.

Lorsque vous appelez la getChildElements() vous obtenez une instance Elements . De là, vous pouvez parcourir et appeler la méthode get(int index) pour récupérer tous les éléments à l'intérieur.

```
Elements colors = root.getChildElements("color");
for (int q = 0; q < colors.size(); q++){
    Element color = colors.get(q);
}
```

Exemple: Voici un exemple de lecture d'un fichier XML:

Fichier 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>
```

Code pour le lire et l'imprimer:

```
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();
            }
        }
    }

}
```

Cela va imprimer dans la console:

```
Name: Smith, Dan
Age: 23 (years)
Favorite Color: green
----------------
Name: Autry, Bob
Age: 3 (months)
Favorite Color: N/A
----------------
```

Écrire dans un fichier XML

L'écriture dans un fichier XML à l'aide de XOM est très similaire à la lecture, sauf dans ce cas, nous créons les instances au lieu de les extraire de la racine.

Pour créer un nouvel élément, utilisez l' Element(String name) constructeur Element(String name) . Vous voudrez créer un élément racine pour pouvoir l'ajouter facilement à un Document .

```
Element root = new Element("root");
```

La classe Element contient des méthodes pratiques pour éditer les éléments. Ils sont énumérés ci-dessous:

- appendChild(String name) - cela définira essentiellement la valeur de l'élément à nommer.
- appendChild(Node node) - cela fera du node le parent des éléments. (Les éléments sont des nœuds pour que vous puissiez analyser les éléments).
- addAttribute(Attribute attribute) - ajoutera un attribut à l'élément.

La classe d' Attribute a deux constructeurs différents. Le plus simple est Attribute(String

name, String value) .

Une fois que tous vos éléments sont ajoutés à votre élément racine, vous pouvez le transformer en Document . Document prendra un Element comme argument dans son constructeur.

Vous pouvez utiliser un Serializer pour écrire votre XML dans un fichier. Vous devrez créer un nouveau flux de sortie à analyser dans le constructeur de Serializer .

```
FileOutputStream fileOutputStream = new FileOutputStream(file);
Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
serializer.setIndent(4);
serializer.write(doc);
```

Exemple

Code:

```
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; }
```

```
    }

}
```

Ce sera le contenu de "out.xml":

```xml
<?xml version="1.0" encoding="UTF-8"?>
<example>
    <person>
        <name>
            <first>Dan</first>
            <last>Smith</last>
        </name>
        <age unit="years">23</age>
        <fav_color>green</fav_color>
    </person>
    <person>
        <name>
            <first>Bob</first>
            <last>Autry</last>
        </name>
        <age unit="months">3</age>
        <fav_color>N/A</fav_color>
    </person>
</example>
```

Lire XOM - Modèle d'objet XML en ligne: https://riptutorial.com/fr/java/topic/5091/xom---modele-d-objet-xml

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec le langage Java	aa_oo, Aaqib Akhtar, abhinav, Abhishek Jain, Abob, acdcjunior, Adeel Ansari, adsalpha, AER, akhilsk, Akshit Soota, Alex A, alphaloop, altomnr, Amani Kilumanga, AndroidMechanic, Ani Menon, ankit dassor, Ankur Anand, antonio, Arkadiy, Ashish Ahuja, Ben Page, Blachshma, bpoiss, Burkhard, Carlton, Charlie H, Coffeehouse Coder, c□ LDS□ EED, Community, 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, ıɯɐƃoʇ 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 , Sнаđомƒа□, 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	Affirmer	Jonathan, Makoto, rajah9, RamenChef, The Guy with The Hat, Uri Agassi
3	Agents Java	Display Name, mnoronha
4	Analyse XML à l'aide des API JAXP	GPI
5	Annotations	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, ФХосę □ Перeúра ツ
6	Apache Commons Lang	Jonathan Barbero
7	API de réflexion	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, ФХосę Перeúра ツ
8	API Stack-Walking	manouti
9	AppDynamics et TIBCO BusinessWorks Instrumentation pour une intégration facile	Alexandre Grimaud
10	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
11	BigDecimal	alain.janinm, Christian, Dth, Enigo, ggolding, Harish Gyanani, John Nash, Loris Securo, Łukasz Piaszczyk, Manish Kothari, mszymborski, RamenChef, sudo, xwoker
12	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
13	Bit Manipulation	Aimee Borda, Blubberguy22, dosdebug, esin88, Gerald Mücke, Jorn Vernee, Kineolyan, mnoronha, Nayuki, Rednivrug, Ryan Hilbert, Stephen C, thatguy
14	BufferedWriter	Andrii Abramov, fabian, Jorn Vernee, Robin, VatsalSura
15	ByteBuffer	Community, Jon Ericson, Jorn Vernee, Tarık Yılmaz, Tomasz Bawor, victorantunes, Vogel612
16	Bytecode Modification	bloo, Display Name, rakwaht, Squidward
17	Calendrier et ses sous-classes	Bob Rivers, cdm, kann, Makoto, mnoronha, ppeterka, Ram, VGR
18	Chargeurs de Classes	FFY00, Flow, Holger, Makoto, Stephen C
19	Chiffrement RSA	Dennis Kriechel, Drunix, iqbal_cs, Maarten Bodewes, Nicktar, Shog9
20	Choisir des collections	John DiFini
21	Classe - Réflexion Java	gobes, KIRAN KUMAR MATAM
22	Classe de propriétés	17slim, Arthur, J Atkin, Jabir, KIRAN KUMAR MATAM, Marvin, peterh, Stephen C, VGR, vorburger
23	Classe EnumSet	KIRAN KUMAR MATAM
24	Classe immuable	Mykola Yashchenko
25	Classe locale	KIRAN KUMAR MATAM
26	Classes et Objets	Community, Confiqure, Daniel LIn, Dave Ranjan, EJP, eveysky,

		fabian, Jens Schauder, Kevin Johnson, KIRAN KUMAR MATAM, MasterBlaster, Mureinik, Rakitić, Ram, RamenChef, Ryan Cocuzzo, Salman Kazmi, Tyler Zika
27	Classes imbriquées et internes	ChemicalFlash, DimaSan, fgb, hd84335, Mshnik, RamenChef, Sandesh , sargue, Slava Babin, Stephen C, tynn
28	Clonage d'objets	Ayush Bansal, Christophe Weis, Jonathan
29	Code officiel Oracle 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
30	Collections	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
31	Collections alternatives	mnoronha, ppeterka, Viacheslav Vedenin
32	Collections concurrentes	GPI, Kenster, Powerlord, user2296600
33	Commandes d'exécution	RamenChef
34	Comparable et Comparateur	Andrii Abramov, Conrad.Dean, Daniel Nugent, fabian, GPI, Hazem Farahat, JAVAC, Mshnik, Nolequen, Petter Friberg, Prateek Agarwal, sebkur, Stephen C
35	Comparaison C ++	John DiFini
36	Compilateur Java - 'javac'	CraftedCart, Jatin Balodhi, Mark Stewart, nishizawa23, Stephen C , Snađomfa, Tom Gijselinck
37	Compilateur Just in Time (JIT)	Liju Thomas, Stephen C
38	CompletableFuture	Adowrath, Kishore Tulsiani, WillShackleford
39	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
40	Constructeurs	Andrii Abramov, Asiat, BrunoDM, ced-b, Codebender, Dylan, George Bailey, Jeremy, Ralf Kleberhoff, RamenChef, Thomas Gerot, tynn, Vogel612

41	Conversion de type	4castle, Filip Smola, Joshua Carmody, Nick Donnelly, RamenChef, Squidward
42	Conversion vers et à partir de chaînes	Chirag Parmar, DarkV1, Gihan Chathuranga, Jabir, JonasCz, Kaushal28, Lachlan Dowding, Laurel, Maarten Bodewes, Matt Clark, PSo, RamenChef, Shaan, Stephen C, still_learning
43	Cordes	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, S нађошfa□, 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 □ Пepeúpa ツ
44	Créer des images par programme	alain.janinm, Dariusz, kajacx, Kenster, mnoronha
45	Date classe	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
46	Dates et heure (java.time. *)	Bilbo Baggins, bowmore, Michael Piefel, Miles, mnoronha, Simon, Squidward, Tarun Maganti, Vogel612, ΦXocę □ Пepeúpa ツ
47	Démonter et décompiler	ipsi, mnoronha
48	Déploiement Java	garg10may, nishizawa23, Pseudonym Patel, RamenChef, Smit, Stephen C
49	Des applets	ArcticLord, Enigo, MadProgrammer, ppeterka

50	Des listes	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
51	Documentation du code 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
52	Douilles	Ordiel
53	Editions Java, versions, versions et distributions	Gal Dreiman, screab, Stephen C
54	Encapsulation	Adam Ratzman, Adil, Daniel M., Drayke, VISHWANATH N P
55	Encodage de caractère	Ilya
56	Ensembles	A_Arnold, atom, ced-b, Chirag Parmar, Daniel Stradowski, demongolem, DimaSan, fabian, Kaushal28, Kenster
57	Enum Carte	KIRAN KUMAR MATAM
58	Enum commençant par numéro	Sugan
59	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, Φ Xocę Пepeúpa ツ
60	Envoi de méthode dynamique	Jeet
61	Évaluation XPath XML	17slim, manouti
62	Exceptions et gestion des exceptions	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
63	Expressions	1d0m3n30, Andreas, EJP, Li357, RamenChef, shmosel, Stephen C, Stephen Leppik
64	Expressions lambda	Abhishek Jain, Ad Infinitum, Adam, aioobe, Amit Gupta, Andrei Maieras, Andrew Tobilko, Andrii Abramov, Ankit Katiyar, Anony-Mousse, assylias, Brian Goetz, Burkhard, Conrad.Dean, cringe, Daniel M., David Soroko, dimitrisli, Draken, DVarga, Emre Bolat, enrico.bacis, fabian, fgb, Gal Dreiman, gar, GPI, Hank D, hexafraction, Ivan Vergiliev, J Atkin, Jean-François Savard, Jeroen Vandevelde, John Slegers, JonasCz, Jorn Vernee, Jude Niroshan, JudgingNotJudging, Kevin Raoofi, Malt, Mark Green, Matt, Matthew Trout, Matthias Braun, ncmathsadist, nobeh, Ortomala Lokni, Paŭlo Ebermann, Paweł Albecki, Petter Friberg, philnate, Pujan Srivastava, Radouane ROUFID, RamenChef, rolve, Saclyr Barlonium, Sergii Bishyr, Skylar Sutton, solomonope, Stephen C, Stephen Leppik, timbooo, Tunaki, Unihedron, vincentvanjoe, Vlasec, Vogel612, webo80, William Ritson, Wolfgang, Xaerxess, xploreraj, Yogi, Ze Rubeus
65	Expressions régulières	Amani Kilumanga, Andy Thomas, Asaph, ced-b, Daniel M., fabian, hd84335, intboolstring, kaotikmynd, Laurel, Makoto, nhahtdh, ppeterka, Ram, RamenChef, Saif, Tot Zam, Unihedron, Vogel612
66	Fichier 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
67	Fichiers JAR multi-versions	manouti
68	Files d'attente et deques	Ad Infinitum, Alek Mieczkowski, Androbin, DimaSan, engineercoding, ppeterka, RamenChef, rd22, Samk, Stephen C
69	FileUpload vers AWS	Amit Gujarathi
70	FilLocal	Dariusz, Liju Thomas, Manish Kothari, Nithanim, taer
71	Fonctionnalités de Java SE 8	compuhosny, RamenChef, sun-solar-arrow
72	Fonctionnalités Java SE 7	compuhosny, RamenChef
73	Format de nombre	arpit pandey, John Nash, RamenChef, ΦXocę Пepeúpa ע
74	Fractionner une chaîne en parties de longueur fixe	Bohemian
75	FTP (protocole de transfert de fichiers)	Kelvin Kellner

76	Génération de code Java	Tony
77	Génération de nombres aléatoires	Arthur, David Grant, David Soroko, dorukayhan, F. Stephen Q, Kichiin, MasterBlaster, michaelbahr, rokonoid, Stephen C, Thodgnir
78	Génériques	1d0m3n30, 4444, Aaron Digulla, Abhishek Jain, Alex Meiburg, alex s, Andrei Maieras, Andrii Abramov, Anony-Mousse, Bart Enkelaar, bitek, Blubberguy22, Bob Brinks, Burkhard, Cache Staheli, Cannon , Ce7, Chriss, code11, Codebender, Daniel Figueroa, daphshez, DVarga, Emil Sierżęga, enrico.bacis, Eran, faraa, hd84335, hexafraction, Jan Vladimir Mostert, Jens Schauder, Jorn Vernee, Jude Niroshan, kcoppock, Kevin Montrose, Lahiru Ashan, Lii, manfcas, Mani Muthusamy, Marc, Matt, Mistalis, Mshnik, mvd, Mzzzzzz, NatNgs, nishizawa23, Oleg Sklyar, Onur, Ortomala Lokni, paisanco, Paul Bellora, Paweł Albecki, PcAF, Petter Friberg, phant0m, philnate, Radouane ROUFID, RamenChef, rap-2-h, rd22, Rogério, rolve, RutledgePaulV, S.K. Venkat, Siguza, Stephen C, Stephen Leppik, suj1th, tainy, ThePhantomGamer, Thomas, TNT, ʇolɐəz əɥʇ qoq, Unihedron, Vlad-HC, Wesley, Wilson, yiwei, Yury Fedorov
79	Gestion de la mémoire Java	Daniel M., engineercoding, fgb, John Nash, jwd630, mnoronha, OverCoder, padippist, RamenChef, Squidward, Stephen C
80	Getters et Setters	Fildor, Ironcache, Kröw, martin, Petter Friberg, Stephen C, Sujith Niraikulathan, Thisaru Guruge, uzaif
81	Graphiques 2D en Java	17slim, ABDUL KHALIQ
82	Hashtable	KIRAN KUMAR MATAM
83	Héritage	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
84	Heure 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
85	HttpURLConnection	Community, Datagrammar, EJP, Inzimam Tariq IT, JonasCz, kiedysktos, Mureinik, NageN, Stephen C, still_learning
86	Implémentations du système de plug-in Java	Alexiy
87	InputStreams et OutputStreams	akgren_soar, EJP, Gubbel, J Atkin, Jens Schauder, John Nash, Kip , KIRAN KUMAR MATAM, Matt Clark, Michael, RamenChef, Stephen C, Vogel612

88	Installation de 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, Sнаđоɯfaᗡ, tonirush, Walery Strauch, Ze Rubeus
89	Interface de l'outil JVM	desilijic
90	Interface Dequeue	Suketu Patel
91	Interface Fluent	bn., noscreenname, P.J.Meisch, RamenChef, TuringTux
92	Interface native Java	Coffee Ninja, Fjoni Yzeiri, Jorn Vernee, RamenChef, Stephen C, user1803551
93	Interfaces	100rabh, A Boschman, Abhishek Jain, Adowrath, Alex Shesterov, Andrew Tobilko, Andrii Abramov, Cà phê đen, Chirag Parmar, Conrad.Dean, Daniel Käfer, devguy, DVarga, Hilikus, inovaovao, intboolstring, James Oswald, Jan Vladimir Mostert, JavaHopper, Johannes, Jojodmo, Jonathan, Jorn Vernee, Kai, kstandell, Laurel , Marvin, MikeW, Paul Nelson Baker, Peter Rader, ppovoski, Prateek Agarwal, Radouane ROUFID, RamenChef, Robin, Simulant, someoneigna, Stephen C, Stephen Leppik, Sujith Niraikulathan, Thomas Gerot, user187470, Vasiliy Vlasov, Vince Emigh, xwoker, Zircon
94	Interfaces fonctionnelles	Andreas
95	Invocation de méthode distante (RMI)	RamenChef, smichel, Stephen C, user1803551, Vasiliy Vlasov
96	Itérateur et Iterable	Abubakkar, Comic Sans, Dariusz, Hulk, Lukas Knuth, RamenChef, Stephen C, user1121883, WillShackleford
97	Java Native Access	Ezekiel Baniaga, Stephan, Stephen C
98	Java Virtual Machine (JVM)	Dushman, RamenChef, Rory McCrossan, Stephen Leppik
99	JavaBean	foxt7ot, J. Pichardo, James Fry, SaWo, Stephen C
100	JAXB	Dariusz, Drunix, fabian, hd84335, Jabir, ppeterka, Ram, Stephan, Thomas Fritsch, vallismortis, Walery Strauch
101	JAX-WS	ext1812, Jonathan Barbero, Stephen Leppik
102	JMX	esin88
103	JNDI	EJP, neohope, RamenChef
104	Journalisation (java.util.logging)	bn., Christophe Weis, Emil Sierżęga, P.J.Meisch, vallismortis
105	JShell	ostrichofevil, Sudip Bhandari
106	JSON en Java	Asaph, Bogdan Korinnyi, Burkhard, Cache Staheli, hd84335, ipsi, Jared Hooper, Kurzalead, MikaelF, Mrunal Pagnis, Nicholas J Panella, Nikita Kurtin, ppeterka, Prem Singh Bist, RamenChef,

		Ray Kiddy, SirKometa, still_learning, Stoyan Dekov, systemfreund, Tim, Vikas Gupta, vsminkov, Yury Fedorov
107	JVM Flags	Confiqure, RamenChef
108	l'audio	Dac Saunders, Petter Friberg, RamenChef, TNT, tonirush, Tot Zam, Vogel612
109	La classe java.util.Objects	mnoronha, RamenChef, Stephen C
110	La commande Java - "java" et "javaw"	4444, Ben, mnoronha, Stephen C, Vogel612
111	La mise en réseau	Arthur, Burkhard, devnull69, DonyorM, glee8e, Grayson Croom, Ilya, Malt, Matej Kormuth, Matthieu, Mine_Stone, ppeterka, RamenChef, Stephen C, Tot Zam, vsav
112	La sérialisation	akhilsk, Batty, Bilesh Ganguly, Burkhard, EJP, emotionlessbananas, faraa, GradAsso, KIRAN KUMAR MATAM, noscreenname, Onur, rokonoid, Siva Sainath Reddy Bandi, Vasilis Vasilatos, Vasiliy Vlasov
113	Le Classpath	Aaron Digulla, GPI, K'', Kenster, Ruslan Ulanov, Stephen C, trashgod
114	Lecteurs et écrivains	JD9999, KIRAN KUMAR MATAM, Mureinik, Stephen C, VatsalSura
115	Les opérateurs	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
116	LinkedHashMap	Amit Gujarathi, KIRAN KUMAR MATAM
117	Liste vs SET	KIRAN KUMAR MATAM
118	Littéraux	1d0m3n30, EJP, ParkerHalo, Stephen C, ThePhantomGamer
119	Localisation et internationalisation	Code.IT, dimo414, Eduard Wirch, emotionlessbananas, Squidward, sun-solar-arrow
120	log4j / log4j2	Daniel Wild, Fildor, HCarrasko, hd84335, Mrunal Pagnis, Rens van der Heijden
121	Méthodes de classe d'objet et constructeur	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
122	Méthodes de collecte d'usine	Jacob G.
123	Méthodes par défaut	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
124	Modèle de mémoire Java	Shree, Stephen C, Suminda Sirinath S. Dharmasena
125	Modificateurs de non-accès	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
126	Modules	Jonathan, user140547
127	Monnaie et argent	Alexey Lagunov
128	Moteur JavaScript Nashorn	ben75, ekaerovets, Francesco Menzani, hd84335, Ilya, InitializeSahib, kasperjj, VatsalSura
129	NIO - Mise en réseau	Matthieu, mnoronha
130	Nouveau fichier E / S	dorukayhan, niheno, TuringTux
131	Objets immuables	1d0m3n30, Bohemian, Holger, Idcmp, Jon Ericson, kristyna, Michael Piefel, Stephen C, Vogel612
132	Objets sécurisés	Ankit Katiyar
133	Opérations en virgule flottante Java	Dariusz, hd84335, HTNW, Ilya, Mr. P, Petter Friberg, ravthiru, Stephen C, Stephen Leppik, Vogel612
134	Optionnel	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
135	Paquets	JamesENL, KIRAN KUMAR MATAM
136	Pièges de Java - Nulls et NullPointerException	17slim, Andrii Abramov, Daniel Nugent, dorukayhan, fabian, François Cassin, Miles, Stephen C, Zircon
137	Pièges Java - Problèmes de performances	Dorian, GPI, John Starich, Jorn Vernee, Michał Rybak, mnoronha, ppeterka, Sharon Rozinsky, steffen, Stephen C, xTrollxDudex
138	Pièges Java - Syntaxe du langage	Alex T., Cody Gray, Enwired, Friederike, Gal Dreiman, hd84335, Hiren, Peter Rader, piyush_baderia, RamenChef, Ravindra HV, RudolphEst, Stephen C, Todd Sewell, user3105453

139	Pièges Java - Threads et accès concurrents	dorukayhan, james large, Stephen C
140	Pièges Java - Utilisation des exceptions	Bhoomika, bruno, dimo414, Gal Dreiman, hd84335, SachinSarawgi, scorpp, Stephen C, Stephen Leppik, user3105453
141	Pièges Java communs	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
142	Plans	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
143	Polymorphisme	Adrian Krebs, Amani Kilumanga, Daniel LIn, Dushman, Kakarot, Lernkurve, Markus L, NageN, Pawan, Ravindra babu, Saiful Azad, Stephen C
144	Pools d'executor, d'executorService et de threads	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
145	Préférences	RAnders00
146	Processus	Andy Thomas, Bob Rivers, ppeterka, vorburger, yitzih
147	Programmation concurrente (threads)	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
148	Programmation parallèle avec framework Fork / Join	Community, Joe C
149	Récursivité	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
150	Références d'objet	Andrii Abramov, arcy, Vasiliy Vlasov

151	Réglage des performances Java	Gene Marin, jatanp, Stephen C, Vogel612
152	Repères	esin88
153	Responsable de la sécurité	alphaloop, hexafraction, Uux
154	Ressources (sur classpath)	Androbin, Christian, Emily Mabrey, Enwired, fabian, Gerald Mücke , Jesse van Bekkum, Kenster, Stephen C, timbooo, VGR, vorburger
155	Ruisseaux	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
156	Scanner	Alek Mieczkowski, Chirag Parmar, Community, Jon Ericson, JonasCz , Ram, RamenChef, Redterd, Stephen C, sun-solar-arrow, ΦXocę □ Π epeúpa ツ
157	Sécurité et cryptographie	John Nash, shibli049
158	Service d'impression Java	Danilo Guimaraes, Leonardo Pina
159	ServiceLoader	fabian, Florian Genser, Gerald Mücke
160	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
161	Sockets Java	Nikhil R
162	SortedMap	Amit Gujarathi

163	StringBuffer	Amit Gujarathi
164	StringBuilder	Andrii Abramov, Cache Staheli, David Soroko, Enigo, fabian, fgb, JudgingNotJudging, KIRAN KUMAR MATAM, Nicktar, P.J.Meisch, Stephen C
165	Structures de contrôle de 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
166	sun.misc.Unsafe	4444, Daniel Nugent, Grexis, Stephen C, Suminda Sirinath S. Dharmasena
167	super mot clé	Abhijeet
168	Tableaux	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, noʏʜʎdʎzɐɔ, Nuri Tasdemir, Ocracoke, OldMcDonald, Onur, orccrusher99, Ortomala Lokni, Panda, Paolo Forgia, Paul Bellora, Paweł Albecki, PeerNet, Peter Gordon, phatfingers, Pimgd, Piyush, ppeterka, Přemysl Šťastný, PSN, Pujan Srivastava, QoP, Radiodef, Radouane ROUFID, Raidri, Rajesh , Rakitić, Ram, RamenChef, Ravi Chandra, René Link, Reut Sharabani, Richard Hamilton, Robert Columbia, rolfedh, rolve, Roman Cherepanov, roottraveller, Ross, Ryan Hilbert, Sam

		Hazleton, sandbo00, Saurabh, Sayakiss, sebkur, Sergii Bishyr, sevenforce, shmosel, Shoe, Siguza, Simulant, Slayther, Smi, solidcell, Spencer Wieczorek, Squidward, stackptr, stark, Stephen C, Stephen Leppik, Sualeh Fatehi, sudo, Sumurai8, Sunnyok, syb0rg, tbodt, tdelev, tharkay, Thomas, ThunderStruct, Tol182, ʇolɐəz əɥʇ qoq, tpunt, Travis J, Tunaki, Un3qual, Unihedron, user6653173, uzaif, vasili111, VedX, Ven, Victor G., Vikas Gupta, vincentvanjoe, Vogel612, Wilson, Winter, X.lophix, YCF_L, Yohanes Khosiawan □□□, yuku, Yury Fedorov, zamonier, Ф Xocę □ Пepeúpa ツ
169	Test d'unité	Ironcache
170	Tokenizer de chaîne	M M
171	Traitement des arguments en ligne de commande	Burkhard, Michael von Wenckstern, Stephen C
172	TreeMap et TreeSet	Malt, Stephen C
173	Types atomiques	Daniel Nugent, Stephen C, Suminda Sirinath S. Dharmasena, xTrollxDudex
174	Types de données de référence	Do Nhu Vy, giucal, Jorn Vernee, Lord Farquaad, Yohanes Khosiawan □□□
175	Types de données primitifs	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
176	Types de référence	EJP, NageN, Thisaru Guruge
177	Utilisation de ThreadPoolExecutor dans les applications MultiThreaded.	Brendon Dugan
178	Utilisation du mot clé static	17slim, Amir Rachum, Andrew Brooke, Arthur, ben75, CarManuel, Daniel Nugent, EJP, Hi I'm Frogatto, Mark Yisri, Sadiq Ali, Skepter, Squidward
179	Utiliser d'autres langages de script en Java	Nikhil R
180	Varargs (argument variable)	Daniel Nugent, Dushman, Omar Ayala, Rafael Pacheco, RamenChef, VGR, xsami
181	Visibilité (contrôle de l'accès aux membres d'une classe)	Aasmund Eldhuset, Abhishek Balaji R, Catalina Island, Daniel M., intboolstring, Jonathan, Mark Yisri, Mureinik, NageN, ParkerHalo , Stephen C, Vogel612

182	WeakHashMap	Amit Gujarathi, KIRAN KUMAR MATAM
183	XJC	Danilo Guimaraes, fabian
184	XOM - Modèle d'objet XML	Arthur, Makoto