Kostenloses eBook

LERNEN

Java Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#java

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit Java Language

Bemerkungen

Die Java-Programmiersprache ist ...

- **Allzweck** : Es ist für das Schreiben von Software in einer Vielzahl von Anwendungsdomänen vorgesehen und verfügt über keine speziellen Funktionen für eine bestimmte Domäne.
- **Klassenbasiert** : Seine Objektstruktur ist in Klassen definiert. Klasseninstanzen haben immer diese Felder und Methoden in ihren Klassendefinitionen angegeben (siehe Klassen und Objekte). Dies steht im Gegensatz zu nicht klassenbasierten Sprachen wie JavaScript.
- **Statisch typisiert** : Der Compiler prüft zur Kompilierzeit, ob Variablentypen beachtet werden. Wenn eine Methode beispielsweise ein Argument des Typs `String` erwartet, muss dieses Argument beim Aufruf der Methode tatsächlich eine Zeichenfolge sein.
- **Objektorientiert** : Die meisten Dinge in einem Java-Programm sind Klasseninstanzen, dh Zustandsbündel (Felder) und Verhalten (Methoden, die Daten verarbeiten und die *Schnittstelle* des Objekts zur Außenwelt bilden).
- **Portable** : Es kann auf jeder Plattform mit `javac` kompiliert werden, und die resultierenden Klassendateien können auf jeder Plattform mit JVM ausgeführt werden.

Java soll Anwendungsentwicklern das "einmalige Schreiben, überall ausführen" (WORA) ermöglichen. Das bedeutet, dass kompilierter Java-Code auf allen Plattformen, die Java unterstützen, ohne Neukompilierung ausgeführt werden kann.

Java-Code wird zu Bytecode (den `.class` Dateien) kompiliert, die wiederum von der Java Virtual Machine (JVM) interpretiert werden. In der Theorie sollte der von einem Java-Compiler erstellte Bytecode auf jeder JVM auf die gleiche Art und Weise laufen, selbst auf einem anderen Computer. Die JVM kann (und in realen Programmen) die häufig ausgeführten Teile des Bytecodes in native Maschinenbefehle kompilieren. Dies wird als "Just-in-Time-Kompilierung (JIT)" bezeichnet.

Java-Editionen und -Versionen

Es gibt drei "Ausgaben" von Java, die von Sun / Oracle definiert werden:

- *Java Standard Edition (SE)* ist die Edition, die für den allgemeinen Gebrauch bestimmt ist.
- *Java Enterprise Edition (EE)* fügt eine Reihe von Möglichkeiten hinzu, um "Enterprise-Grade" -Dienste in Java zu erstellen. Java EE wird separat behandelt .
- *Java Micro Edition (ME)* basiert auf einer Teilmenge von *Java SE* und ist für den Einsatz auf kleinen Geräten mit begrenzten Ressourcen vorgesehen.

Zu den Java SE / EE / ME-Editionen gibt es ein eigenes Thema.

Jede Ausgabe hat mehrere Versionen. Die Java SE-Versionen sind unten aufgeführt.

Java installieren

Es gibt ein eigenes Thema zum Installieren von Java (Standard Edition) .

Java-Programme kompilieren und ausführen

Es gibt separate Themen zu:

- Java-Quellcode kompilieren
- Java-Bereitstellung einschließlich der Erstellung von JAR-Dateien
- Java-Anwendungen ausführen
- Der Klassenpfad

Was kommt als nächstes?

Hier finden Sie Links zu Themen, um die Java-Programmiersprache weiter zu erlernen und zu verstehen. Diese Themen sind die Grundlagen der Java-Programmierung, um Ihnen den Einstieg zu erleichtern.

- Primitive Datentypen in Java
- Operatoren in Java
- Zeichenfolgen in Java
- Grundlegende Steuerungsstrukturen in Java
- Klassen und Objekte in Java
- Arrays in Java
- Java-Codestandards

Testen

Java bietet zwar keine Unterstützung für Tests in der Standardbibliothek, es gibt jedoch Bibliotheken von Drittanbietern, die Tests unterstützen. Die zwei beliebtesten Unit-Test-Bibliotheken sind:

- JUnit (Offizielle Seite)
- TestNG (Offizielle Seite)

Andere

- Entwurfsmuster für Java werden in Entwurfsmuster behandelt .
- Die Programmierung für Android ist in Android enthalten .
- Java Enterprise Edition-Technologien werden in Java EE behandelt .
- Die JavaFX-Technologien von Oracle werden in JavaFX behandelt .

1. Im Abschnitt " **Versionen** " *endet* das *Ende der Lebensdauer (kostenlos),* wenn Oracle keine weiteren Updates von Java SE auf seinen öffentlichen Download-Sites mehr veröffentlichen wird. Kunden, die weiterhin Zugriff auf wichtige Fehlerbehebungen und Sicherheitsupdates sowie allgemeine Wartungsarbeiten für Java SE benötigen, können über den Oracle Java SE-Support langfristig Unterstützung erhalten.

Versionen

Java SE-Version	Code Name	Ende des Lebens (kostenlos [1])	Veröffentlichungsdatum
Java SE 9 (früher Zugriff)	*Keiner*	Zukunft	2017-07-27
Java SE 8	Spinne	Zukunft	2014-03-18
Java SE 7	Delphin	2015-04-14	2011-07-28
Java SE 6	Mustang	2013-04-16	2006-12-23
Java SE 5	Tiger	2009-11-04	2004-10-04
Java SE 1.4	Merlin	vor dem 2009-11-04	2002-02-06
Java SE 1.3	Turmfalke	vor dem 2009-11-04	2000-05-08
Java SE 1.2	Spielplatz	vor dem 2009-11-04	1998-12-08
Java SE 1.1	*Keiner*	vor dem 2009-11-04	1997-02-19
Java SE 1.0	Eiche	vor dem 2009-11-04	1996-01-21

Examples

Erstellen Sie Ihr erstes Java-Programm

Erstellen Sie in Ihrem Texteditor oder der IDE eine neue Datei mit dem Namen `HelloWorld.java` . Fügen Sie dann diesen Codeblock in die Datei ein und speichern Sie:

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

Laufen Sie live auf Ideone

Hinweis: Für Java , dies zu erkennen als `public class` (und nicht werfen der Kompilierung Fehler), muss der Dateiname sein , der gleiche wie der Klassenname (`HelloWorld` in diesem Beispiel) mit einer `.java` - Erweiterung. Davor sollte auch ein Modifikator für den `public` Zugriff vorhanden sein.

Benennungskonventionen empfehlen, dass Java-Klassen mit Großbuchstaben beginnen und im Kamelformat sind (wobei der erste Buchstabe jedes Wortes groß geschrieben wird). Die Konventionen empfehlen gegen Unterstriche (`_`) und Dollarzeichen (`$`).

Öffnen Sie zum Kompilieren ein Terminalfenster und navigieren Sie zum Verzeichnis `HelloWorld.java` :

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

Hinweis: cd ist der Terminalbefehl zum Ändern des Verzeichnisses.

Geben Sie `javac` gefolgt von Dateiname und Erweiterung wie folgt ein:

```
$ javac HelloWorld.java
```

Es ist üblich, dass der Fehler `'javac' is not recognized as an internal or external command, operable program or batch file.` selbst wenn Sie das `JDK` installiert haben und das Programm von `IDE` ex ausführen können. `eclipse` usw. Da der Pfad standardmäßig nicht zur Umgebung hinzugefügt wird.

Falls Sie dies unter Windows erhalten, versuchen Sie zunächst, zu Ihrem `javac.exe` Pfad zu `javac.exe` . Er befindet sich höchstwahrscheinlich in `C:\Program Files\Java\jdk(version number)\bin` . Dann versuchen Sie es mit unten auszuführen.

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

Als wir `javac` , war es derselbe Befehl wie oben. Nur in diesem Fall wusste Ihr `OS` , wo sich `javac` . Sagen wir es jetzt, auf diese Weise müssen Sie nicht jedes Mal den gesamten Pfad eingeben. Wir müssen dies unserem `PATH` hinzufügen

So bearbeiten Sie die Umgebungsvariable `PATH` in Windows XP / Vista / 7/8/10:

- Systemsteuerung ⇒ System ⇒ Erweiterte Systemeinstellungen
- Wechseln Sie auf die Registerkarte "Erweitert" ⇒ Umgebungsvariablen
- Scrollen Sie unter "Systemvariablen" nach unten, um "PFAD" auszuwählen ⇒ Bearbeiten

Du kannst das nicht ungeschehen machen, also sei vorsichtig. Kopieren Sie zunächst Ihren vorhandenen Pfad in den Merkzettel. Um den genauen Pfad zu Ihrem `javac` navigieren Sie manuell zu dem Ordner, in dem sich `javac` befindet, klicken Sie in die Adressleiste und kopieren Sie ihn. Es sollte `c:\Program Files\Java\jdk1.8.0_xx\bin` wie `c:\Program Files\Java\jdk1.8.0_xx\bin`

Fügen Sie im Feld "Variablenwert" diese **IN FRONT** aller vorhandenen Verzeichnisse gefolgt von

einem Semikolon (;) ein. **LÖSCHEN SIE KEINE** vorhandenen Einträge.

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

Das sollte sich jetzt lösen.

Für Linux-basierte Systeme versuchen Sie es hier .

Anmerkung: *Der Befehl `javac` ruft den Java-Compiler auf.*

Der Compiler generiert dann eine Bytecode- Datei namens `HelloWorld.class` die in der Java Virtual Machine (JVM) ausgeführt werden kann . Der Java-Programmiersprachen-Compiler `javac` liest Quelldateien, die in der Java-Programmiersprache geschrieben sind, und kompiliert sie in `bytecode` Klassendateien. Optional kann der Compiler auch Anmerkungen in Quell- und Klassendateien mithilfe der `Pluggable` Annotation-Verarbeitungs-API verarbeiten. Der Compiler ist ein Befehlszeilentool, kann aber auch mit der Java Compiler-API aufgerufen werden.

Um Ihr Programm auszuführen, geben Sie `java` gefolgt vom Namen der Klasse ein, die die `main` enthält (in unserem Beispiel `HelloWorld`). Beachten Sie, wie die `.class` weggelassen wird:

```
$ java HelloWorld
```

Hinweis: *Der `java` Befehl führt eine Java-Anwendung aus.*

Dies wird auf Ihrer Konsole ausgegeben:

> Hallo Welt!

Sie haben Ihr erstes Java-Programm erfolgreich codiert und erstellt!

Hinweis: Damit Java-Befehle (`java` , `javac` usw.) erkannt werden, müssen Sie Folgendes sicherstellen:

- Ein JDK ist installiert (zB Oracle , OpenJDK und andere Quellen)
- Ihre Umgebungsvariablen sind ordnungsgemäß eingerichtet

Sie müssen einen Compiler (`javac`) und einen von Ihrer JVM bereitgestellten Executor (`java`) verwenden. Um herauszufinden, welche Versionen Sie installiert haben, geben `javac -version` in der Befehlszeile `java -version` und `javac -version` . Die Versionsnummer Ihres Programms wird im Terminal gedruckt (z. B. `1.8.0_73`).

Ein genauerer Blick auf das Hello World-Programm

Das „Hallo Welt“ Programm enthält eine einzelne Datei, die aus einer besteht `HelloWorld`

Klassendefinition, ein `main` und eine Erklärung innerhalb des `main`

```
public class HelloWorld {
```

Das `class` beginnt mit der Klassendefinition für eine Klasse mit dem Namen `HelloWorld` . Jede Java-Anwendung enthält mindestens eine Klassendefinition (Weitere Informationen zu Klassen).

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

Dies ist eine Einstiegspunktmethode (definiert durch ihren Namen und die Signatur von `public static void main(String[])`), von der aus die `JVM` Ihr Programm ausführen kann. Jedes Java-Programm sollte eine haben. Es ist:

- `public` : Bedeutet, dass die Methode auch außerhalb des Programms aufgerufen werden kann. Weitere Informationen hierzu finden Sie unter Sichtbarkeit .
- `static` : bedeutet, dass es existiert und von ihm selbst ausgeführt werden kann (auf Klassenebene, ohne ein Objekt zu erstellen).
- `void` : bedeutet, dass kein Wert zurückgegeben wird. ***Hinweis:*** *Dies unterscheidet sich von C und C ++, wo ein Rückgabewert wie* `int` *erwartet wird (Javas Weg ist* `System.exit()` *).*

Diese Hauptmethode akzeptiert:

- Ein Array (normalerweise als `args`) von `String` als Argument an die Hauptfunktion übergeben (z. B. von Befehlszeilenargumenten).

Fast alles ist für eine Java-Einstiegspunktmethode erforderlich.

Nicht benötigte Teile:

- Der Name `args` ist ein Variablenname, daher kann er beliebig genannt werden, obwohl er normalerweise `args` .
- Ob der Parametertyp ein Array (`String[] args`) oder Varargs (`String... args`) ist, spielt keine Rolle, da Arrays in Varargs übergeben werden können.

Hinweis: Eine einzelne Anwendung mehrere Klassen haben einen Eintrittspunkt (enthaltend `main`) -Methode. Der Einstiegspunkt der Anwendung wird durch den Klassennamen bestimmt, der als Argument an den `java` Befehl übergeben wird.

In der Hauptmethode sehen wir folgende Aussage:

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

Lassen Sie uns diese Anweisung Element für Element aufschlüsseln:

Element	Zweck
`System`	Dies bedeutet, dass der nachfolgende Ausdruck die Klasse `System` aus dem Paket `java.lang` aufrufen wird.

Element	Zweck
`.`	Dies ist ein "Punktoperator". Punktoperatoren bieten Ihnen Zugriff auf die Klassenmitglieder [1] ; dh seine Felder (Variablen) und ihre Methoden. In diesem Fall können Sie mit diesem Punktoperator auf das statische `out` Feld in der `System` verweisen.
`out`	Dies ist der Name des statischen Feldes von `PrintStream` innerhalb der `System` - Klasse , um die Standardausgabe - Funktionalität enthält.
`.`	Dies ist ein weiterer Punktoperator. Dieser Punktoperator ermöglicht den Zugriff auf die `println` Methode innerhalb der `out` Variablen.
`println`	Dies ist der Name einer Methode innerhalb der PrintStream-Klasse. Diese Methode druckt insbesondere den Inhalt der Parameter in die Konsole und fügt anschließend eine neue Zeile ein.
`(`	Diese Klammer zeigt an, dass auf eine Methode (und nicht auf ein Feld) zugegriffen wird, und beginnt mit der `println` der Parameter an die Methode `println` .
`"Hello, World!"`	Dies ist das String- Literal, das als Parameter an die Methode `println` wird. Die doppelten Anführungszeichen an jedem Ende begrenzen den Text als String.
`)`	Diese Klammer bedeutet, dass die Parameter geschlossen werden, die an die Methode `println` werden.
`;`	Dieses Semikolon markiert das Ende der Anweisung.

Hinweis: *Jede Anweisung in Java muss mit einem Semikolon (;) enden.*

Der Methodenkörper und der Klassenkörper werden dann geschlossen.

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

Hier ist ein weiteres Beispiel, das das OO-Paradigma veranschaulicht. Lassen Sie uns eine Fußballmannschaft mit einem (ja, einem!) Mitglied modellieren. Es kann mehr geben, aber wir werden das besprechen, wenn wir zu Arrays kommen.

Zuerst definieren wir unsere `Team` :

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

Nun definieren wir unsere `Member` Klasse:

```
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;
    }
}
```

Warum benutzen wir hier `private` ? Wenn jemand Ihren Namen wissen wollte, sollte er Sie direkt fragen, anstatt in Ihre Tasche zu greifen und Ihre Sozialversicherungskarte herauszuziehen. Dieses `private` macht so etwas: Es verhindert, dass externe Entitäten auf Ihre Variablen zugreifen. Sie können `private` Mitglieder nur über Getter-Funktionen (siehe unten) zurückgeben.

Nachdem wir alles zusammengefügt und die Getter und die Hauptmethode wie zuvor besprochen haben, haben wir:

```
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
    }
}
```

Ausgabe:

```
Aurieel
light
10
1
```

Laufen Sie auf Ideone

Wieder einmal das `main` in der `Test` - ist - Klasse der Einstiegspunkt zu unserem Programm. Ohne die `main` Methode können wir sagen , der Java Virtual Machine nicht (JVM) , von wo aus der Ausführung des Programms zu beginnen.

---

1 - Da die `HelloWorld` Klasse wenig mit der `System` Klasse zusammenhängt, kann sie nur auf `public` Daten zugreifen.

Erste Schritte mit Java Language online lesen: https://riptutorial.com/de/java/topic/84/erste-schritte-mit-java-language

Kapitel 2: 2D-Grafiken in Java

Einführung

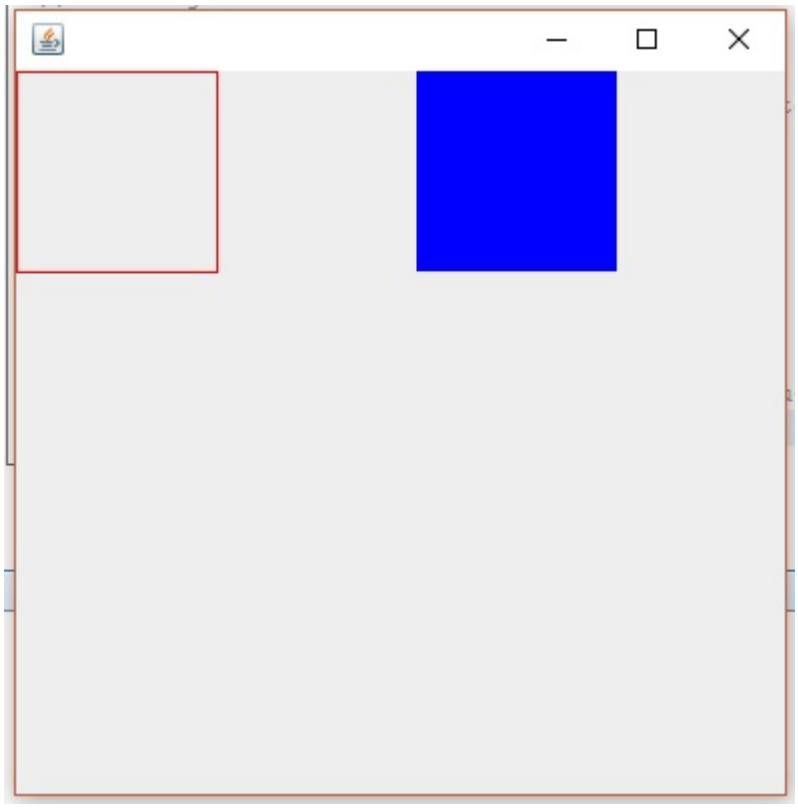
Grafiken sind visuelle Bilder oder Designs auf einer bestimmten Oberfläche, z. B. einer Wand, Leinwand, Leinwand, Papier oder Stein, zum Informieren, Illustrieren oder Unterhalten. Dazu gehören: bildliche Darstellung von Daten, wie beim computergestützten Entwurf und der Fertigung, beim Satz und in der Grafik sowie in Bildungs- und Freizeitsoftware. Bilder, die von einem Computer erzeugt werden, werden als Computergrafik bezeichnet.

Die Java 2D-API ist leistungsstark und komplex. Es gibt mehrere Möglichkeiten, 2D-Grafiken in Java zu erstellen.

Examples

Beispiel 1: Zeichnen und füllen Sie ein Rechteck mit Java

Dies ist ein Beispiel, bei dem Rechteck und Füllfarbe im Rechteck gedruckt werden.



https://i.stack.imgur.com/dlC5v.jpg

Die meisten Methoden der Graphics-Klasse können in zwei grundlegende Gruppen unterteilt werden:

1. Zeichnen und füllen Sie Methoden, um grundlegende Formen, Texte und Bilder zu rendern
2. Attribute Einstellungsmethoden, die beeinflussen, wie diese Zeichnung und Füllung angezeigt wird

Codebeispiel: Beginnen wir mit einem kleinen Beispiel, in dem Sie ein Rechteck zeichnen und die Füllfarbe einfügen. Dort deklarieren wir zwei Klassen, eine Klasse ist MyPanel und die andere Klasse ist Test. In der Klasse MyPanel verwenden wir drawRect () & fillRect () mathods, um ein Rechteck zu zeichnen und die Farbe darin zu füllen. Wir setzen die Farbe mit der setColor (Color.blue) -Methode. In der zweiten Klasse testen wir unsere Grafik, die Testklasse ist. Wir erstellen einen Frame und fügen MyPanel mit dem Objekt p = new MyPanel () in das Objekt ein. Bei der Testklasse sehen wir ein Rechteck und ein mit Farbe gefülltes Rechteck.

Erste Klasse: 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
    }
}
```

Zweite Klasse: 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();
    }
}
```

Weitere Erläuterungen zum Randlayout:
https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html

paintComponent ()

- Es ist eine Hauptmethode für das Malen
- Standardmäßig wird zuerst der Hintergrund gezeichnet
- Danach führt es benutzerdefinierte Bemalungen durch (Zeichnen von Kreisen, Rechtecken usw.).

Graphic2D bezieht sich auf die Graphic2D-Klasse

Hinweis: Mit der Java 2D-API können Sie die folgenden Aufgaben problemlos ausführen:

- Zeichnen Sie Linien, Rechtecke und andere geometrische Formen.
- Füllen Sie diese Formen mit Volltonfarben oder Farbverläufen und Texturen.
- Zeichnen Sie Text mit Optionen zur Feinsteuerung der Schriftart und des Renderprozesses.
- Zeichnen Sie Bilder und wenden Sie optional Filtervorgänge an.
- Wenden Sie Vorgänge an, z. B. Compositing und Transformation während einer der oben genannten Rendervorgänge.

Beispiel 2: Zeichnen und Oval füllen

```
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 Höhe, int Breite);
Diese Methode zeichnet ein Oval an der angegebenen x- und y-Position mit der angegebenen Höhe und Breite.

g2.fillOval (int x, int y, int Höhe, int Breite); Diese Methode füllt ein Oval an der angegebenen x- und y-Position mit der angegebenen Höhe und Breite.

2D-Grafiken in Java online lesen: https://riptutorial.com/de/java/topic/10127/2d-grafiken-in-java

Kapitel 3: Alternative Sammlungen

Bemerkungen

Dieses Thema behandelt Java-Sammlungen aus Guave, Apache, Eclipse: Multiset, Bag, Multimap, Utils-Funktion aus dieser Bibliothek usw.

Examples

Apache HashBag, Guava HashMultiset und Eclipse HashBag

Ein Bag / Ultiset speichert jedes Objekt in der Sammlung zusammen mit einer Anzahl von Vorkommen. Zusätzliche Methoden an der Schnittstelle ermöglichen das gleichzeitige Hinzufügen oder Entfernen mehrerer Kopien eines Objekts. JDK-Analog ist HashMap <T, Integer>, wenn Werte für die Anzahl dieser Kopien angegeben sind.

Art	Guave	Apache-Commons-Sammlungen	GS Sammlungen
Auftrag nicht definiert	HashMultiset	HashBag	HashBag
Sortiert	TreeMultiset	TreeBag	TreeBag
Einfügungsreihenfolge	LinkedHashMultiset	-	-
Gleichzeitige Variante	ConcurrentHashMultiset	SynchronizedBag	SynchronizedBag
Gleichzeitig und sortiert	-	SynchronizedSortedBag	SynchronizedSorted
Unveränderliche Sammlung	ImmutableMultiset	Unveränderbare Tasche	Unveränderbare Ta
Unveränderlich und sortiert	ImmutableSortedMultiset	UnveränderbareSortiertasche	UnveränderbareSor

Beispiele :

1. SynchronizedSortedBag von Apache verwenden :

```
    // 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. Verwenden von TreeBag von 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. Mit LinkedHashMultiset aus 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
```

Mehr Beispiele:

I. Apache-Sammlung:

1. HashBag - Reihenfolge nicht definiert
2. SynchronizedBag - Gleichlauf und Reihenfolge nicht definiert
3. SynchronizedSortedBag - - gleichzeitige und sortierte Reihenfolge
4. TreeBag - sortierte Reihenfolge

II. GS / Eclipse-Sammlung

5. MutableBag - Reihenfolge nicht definiert
6. MutableSortedBag - sortierte Reihenfolge

III. Guave

7. HashMultiset - Reihenfolge nicht definiert
8. TreeMultiset - sortierte Reihenfolge
9. LinkedHashMultiset - Reihenfolge beim Einfügen
10. ConcurrentHashMultiset - Gleichlauf und Reihenfolge nicht definiert

Multimap in Guave-, Apache- und Eclipse-Sammlungen

Diese Multimap ermöglicht das Duplizieren von Schlüssel-Wert-Paaren. JDK-Analoga sind HashMap <K, List>, HashMap <K, Set> und so weiter.

Reihenfolge des Schlüssels	Reihenfolge des Wertes	Duplikat	Analogschlüssel	Analogwert	Gua
nicht definiert	Einfügungsreihenfolge	Ja	HashMap	Anordnungsliste	Arra
nicht definiert	nicht definiert	Nein	HashMap	HashSet	Has

Reihenfolge des Schlüssels	Reihenfolge des Wertes	Duplikat	Analogschlüssel	Analogwert	Gua
nicht definiert	sortiert	Nein	HashMap	TreeSet	`Mult newM Hash <Tre`
Einfügungsreihenfolge	Einfügungsreihenfolge	Ja	LinkedHashMap	Anordnungsliste	Link
Einfügungsreihenfolge	Einfügungsreihenfolge	Nein	LinkedHashMap	LinkedHashSet	Link
sortiert	sortiert	Nein	TreeMap	TreeSet	Tree

Beispiele mit Multimap

Aufgabe : Parse "Hallo Welt! Hallo alle! Hallo Welt!" Zeichenfolge, um Wörter zu trennen und alle Indexe jedes Wortes mit MultiMap zu drucken (z. B. Hello = [0, 2], World! = [1, 5] usw.)

1. MultiValueMap von 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 aus der GS / Eclipse Collection

```
    String[] englishWords = {"one", "two", "three","ball","snow"};
    String[] russianWords = {"jeden", "dwa", "trzy", "kula", "snieg"};

    // Create Multiset
    MutableBiMap<String, String> biMap = new HashBiMap(englishWords.length);
    // Create English-Polish dictionary
    int i = 0;
    for(String englishWord: englishWords) {
        biMap.put(englishWord, russianWords[i]);
        i++;
    }

    // Print count words
    System.out.println(biMap); // print {two=dwa, ball=kula, one=jeden, snow=snieg,
three=trzy} - in random orders
    // Print all unique words
    System.out.println(biMap.keySet());    // print [snow, two, one, three, ball] - in random
orders
    System.out.println(biMap.values());    // print [dwa, kula, jeden, snieg, trzy] - in
random orders

    // Print translate by words
    System.out.println("one = " + biMap.get("one"));    // print one = jeden
    System.out.println("two = " + biMap.get("two"));    // print two = dwa
    System.out.println("kula = " + biMap.inverse().get("kula"));    // print kula = ball
    System.out.println("snieg = " + biMap.inverse().get("snieg"));    // print snieg = snow
    System.out.println("empty = " + biMap.get("empty"));    // print empty = null

    // Print count word's pair
    System.out.println(biMap.size());    //print 5
```

3. HashMultiMap von Guava

```
 String INPUT_TEXT = "Hello World! Hello All! Hi World!";
 // Parse text to words and index
 List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
 // Create Multimap
 Multimap<String, Integer> multiMap = HashMultimap.create();

 // Fill Multimap
 int i = 0;
 for(String word: words) {
     multiMap.put(word, i);
     i++;
 }

 // Print all words
 System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} -
keys and values in random orders
 // Print all unique words
 System.out.println(multiMap.keySet());    // print [Hi, Hello, World!, All!] - in random
```

```
orders

 // Print all indexes
 System.out.println("Hello = " + multiMap.get("Hello"));    // print [0, 2]
 System.out.println("World = " + multiMap.get("World!"));    // print [1, 5]
 System.out.println("All = " + multiMap.get("All!"));    // print [3]
 System.out.println("Hi = " + multiMap.get("Hi"));    // print [4]
 System.out.println("Empty = " + multiMap.get("Empty"));    // print []

 // Print count all words
 System.out.println(multiMap.size());    //print 6

 // Print count unique words
 System.out.println(multiMap.keySet().size());    //print 4
```

Weitere Beispiele:

I. Apache-Sammlung:

1. MultiValueMap
2. MultiValueMapLinked
3. MultiValueMapTree

II. GS / Eclipse-Sammlung

1. FastListMultimap
2. HashBagMultimap
3. TreeSortedSetMultimap
4. UnifiedSetMultimap

III. Guave

1. HashMultiMap
2. LinkedHashMultimap
3. LinkedListMultimap
4. TreeMultimap
5. ArrayListMultimap

Vorgang mit Sammlungen vergleichen - Sammlungen erstellen

Vorgang mit Sammlungen vergleichen - Sammlungen erstellen

1. Liste erstellen

Beschreibung	JDK	Guave	GS-Sammlungen
Leere Liste erstellen	`new ArrayList<> ()`	`Lists.newArrayList()`	`FastList.newList()`
Liste aus Werten	`Arrays.asList("1", "2", "3")`	`Lists.newArrayList("1", "2", "3")`	`FastList.newListWith("1", "2", "3")`

Beschreibung	JDK	Guave	GS-Sammlungen
erstellen			
Erstellen Sie eine Liste mit einer Kapazität von 100	`new ArrayList<>(100)`	`Lists.newArrayListWithCapacity(100)`	`FastList.newList(100)`
Erstellen Sie eine Liste aus einem beliebigen Collectin	`new ArrayList<>(collection)`	`Lists.newArrayList(collection)`	`FastList.newList(collec`
Erstellen Sie eine Liste aus einem beliebigen Iterable	-	`Lists.newArrayList(iterable)`	`FastList.newList(iterab`
Liste von Iterator erstellen	-	`Lists.newArrayList(iterator)`	-
Liste aus Array erstellen	`Arrays.asList(array)`	`Lists.newArrayList(array)`	`FastList.newListWith(ar`
Erstellen Sie eine Liste mit dem Werk	-	-	`FastList.newWithNValues` `() -> "1")`

Beispiele:

```
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 Set erstellen

Beschreibung	JDK	Guave	GS-Sammlungen
Leeres Set erstellen	`new HashSet<>()`	`Sets.newHashSet()`	`UnifiedSet.newSet()`
Erstelle aus Werten	`new HashSet<>(Arrays.asList("alpha", "beta", "gamma")`)	`Sets.newHashSet("alpha", "beta", "gamma")`	`UnifiedSet.newSetWith "beta", "gamma")`
Set aus beliebigen	`new HashSet<>(collection)`	`Sets.newHashSet(collection)`	`UnifiedSet.newSet(col`

Beschreibung	JDK	Guave	GS-Sammlungen
Kollektionen erstellen			
Set von einem beliebigen Iterable erstellen	-	`Sets.newHashSet(iterable)`	`UnifiedSet.newSet(ite`
Set von einem beliebigen Iterator erstellen	-	`Sets.newHashSet(iterator)`	-
Set aus Array erstellen	`new HashSet<>(Arrays.asList(array))`	`Sets.newHashSet(array)`	`UnifiedSet.newSetWith`

Beispiele:

```
    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 Karte erstellen

Beschreibung	JDK	Guave	GS-Sammlungen
Erstellen Sie eine leere Karte	`new HashMap<>()`	`Maps.newHashMap()`	`UnifiedMap.newMap()`
Erstellen Sie eine Karte mit einer Kapazität von 130	`new HashMap<>(130)`	`Maps.newHashMapWithExpectedSize(100)`	`UnifiedMap.newMap(130)`
Karte von einer anderen Karte erstellen	`new HashMap<>(map)`	`Maps.newHashMap(map)`	`UnifiedMap.newMap(map)`
Karte aus Schlüsseln erstellen	-	-	`UnifiedMap.newWithKeysValues("1 "a", "2", "b")`

Beispiele:

```
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");
```

Weitere Beispiele: CreateCollectionTest

1. CollectionCompare
2. CollectionSearch
3. JavaTransform

Alternative Sammlungen online lesen: https://riptutorial.com/de/java/topic/2958/alternative-sammlungen

Kapitel 4: Anmerkungen

Einführung

In Java ist eine Annotation eine Form von syntaktischen Metadaten, die dem Java-Quellcode hinzugefügt werden können. Sie liefert Daten zu einem Programm, das nicht Teil des Programms selbst ist. Anmerkungen haben keinen direkten Einfluss auf die Funktionsweise des Codes, den sie kommentieren. Klassen, Methoden, Variablen, Parameter und Pakete dürfen kommentiert werden.

Syntax

- @AnnotationsName // 'Markierungsanmerkung' (keine Parameter)
- @AnnotationName (someValue) // setzt Parameter mit dem Namen 'value'
- @Anmerkungsname (Parameter1 = Wert1) // benannter Parameter
- @Annotationsname (Parameter1 = Wert1, Parameter2 = Wert2) // mehrere benannte Parameter
- @Anmerkungsname (param1 = {1, 2, 3}) // benannter Array-Parameter
- @AnnotationsName ({value1}) // Array mit einem einzelnen Element als Parameter mit dem Namen 'value'

Bemerkungen

Parametertypen

Für Parameter sowie Arrays dieser Typen sind nur konstante Ausdrücke der folgenden Typen zulässig:

- `String`
- `Class`
- primitive Typen
- Aufzählungstypen
- Anmerkungsarten

Examples

Eingebaute Anmerkungen

Die Standard Edition von Java enthält einige vordefinierte Anmerkungen. Sie müssen sie nicht selbst definieren und können sie sofort verwenden. Sie ermöglichen dem Compiler, einige grundlegende Überprüfungen von Methoden, Klassen und Code zu ermöglichen.

@Überfahren

Diese Anmerkung gilt für eine Methode und besagt, dass diese Methode die Methode einer Superklasse überschreiben oder die Methodendefinition einer abstrakten Superklasse implementieren muss. Wenn diese Annotation mit einer anderen Methode verwendet wird, gibt der Compiler einen Fehler aus.

Konkrete Oberklasse

```
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");
    }
}
```

Abstrakte Klasse

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

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

Funktioniert nicht

```
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);
    }
}
```

Der Hauptzweck besteht darin, die falsche Schreibweise abzufangen, wenn Sie der Meinung sind, dass Sie eine Methode überschreiben, aber tatsächlich eine neue definieren.

```
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");
    }
}
```

Beachten Sie, dass sich die Bedeutung von `@Override` im Laufe der Zeit geändert hat:

- In Java 5 bedeutete dies, dass die annotierte Methode eine nicht abstrakte Methode überschreiben musste, die in der Superklassenkette deklariert wurde.
- Ab Java 6 ist es *auch* erfüllt, wenn die annotierte Methode eine abstrakte Methode implementiert, die in der Klassen-Superklasse / Schnittstellen-Hierarchie deklariert ist.

(Dies kann gelegentlich zu Problemen führen, wenn Code nach Java 5 zurück portiert wird.)

@Entschieden

Dies kennzeichnet die Methode als veraltet. Dafür kann es mehrere Gründe geben:

- Die API ist fehlerhaft und unpraktisch zu beheben.
- Die Verwendung der API führt wahrscheinlich zu Fehlern.
- Die API wurde durch eine andere API ersetzt.
- die API ist veraltet,
- Die API ist experimentell und unterliegt inkompatiblen Änderungen.
- oder eine Kombination der oben genannten.

Der spezifische Grund für die Abwertung ist in der Regel in der Dokumentation der API zu finden.

Die Anmerkung führt dazu, dass der Compiler einen Fehler ausgibt, wenn Sie ihn verwenden. IDEs können diese Methode auch als veraltet markieren

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

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

@SuppressWarnings

In fast allen Fällen ist es am besten, wenn der Compiler eine Warnung ausgibt, die Ursache zu beheben. In einigen Fällen (z. B. Generics-Code, der untypensicheren Prägenerics-Code verwendet) ist dies möglicherweise nicht möglich, und es ist besser, diese Warnungen zu unterdrücken, die Sie erwarten und nicht beheben können, sodass Sie unerwartete Warnungen deutlicher sehen können.

Diese Annotation kann auf eine ganze Klasse, Methode oder Zeile angewendet werden. Die Kategorie der Warnung wird als Parameter verwendet.

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

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

Es ist besser, den Umfang der Anmerkung so weit wie möglich einzuschränken, um zu verhindern, dass unerwartete Warnungen ebenfalls unterdrückt werden. B. den Umfang der Annotation auf eine einzelne Zeile beschränken:

```
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
```

Die von dieser Anmerkung unterstützten Warnungen können von Compiler zu Compiler variieren. In der JLS werden nur die `unchecked` und `deprecation` Warnungen ausdrücklich erwähnt. Nicht erkannte Warntypen werden ignoriert.

@SafeVarargs

Aufgrund der Typlöschung wird die `void method(T... t)` in die `void method(Object[] t)` konvertiert. Dies bedeutet, dass der Compiler nicht immer überprüfen kann, ob die Verwendung von varargs typsicher ist. Zum Beispiel:

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

Es gibt Fälle, in denen die Verwendung sicher ist. In diesem Fall können Sie die Methode mit der `SafeVarargs` Anmerkung versehen, um die Warnung zu unterdrücken. Dies verbirgt offensichtlich die Warnung, wenn Ihre Verwendung ebenfalls unsicher ist.

@Funktionsschnittstelle

Dies ist eine optionale Anmerkung zum Markieren eines FunctionalInterface. Der Compiler wird

beschwert, wenn er nicht mit der FunctionalInterface-Spezifikation übereinstimmt (eine einzige abstrakte Methode hat).

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

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

Laufzeitannotationsprüfungen über Reflection

Mit der Reflection-API von Java kann der Programmierer während der Laufzeit verschiedene Prüfungen und Vorgänge an Klassenfeldern, Methoden und Anmerkungen durchführen. Damit eine Anmerkung zur Laufzeit überhaupt sichtbar ist, muss die `RetentionPolicy` in `RUNTIME` geändert `RUNTIME` , wie im folgenden Beispiel gezeigt:

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

Anmerkungsarten definieren

Annotationstypen werden mit `@interface` definiert. Parameter werden ähnlich den Methoden einer regulären Schnittstelle definiert.

```
@interface MyAnnotation {
    String param1();
```

```
    boolean param2();
    int[] param3();  // array parameter
}
```

Standardwerte

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

Meta-Anmerkungen

Meta-Annotationen sind Annotationen, die auf Annotationstypen angewendet werden können. Spezielle vordefinierte Meta-Annotationen definieren, wie Annotationstypen verwendet werden können.

@Ziel

Die `@Target` - `@Target` beschränkt die Typen, auf die die Annotation angewendet werden kann.

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

Mehrere Werte können mithilfe der Array-Notation hinzugefügt werden, z. B.
`@Target({ElementType.FIELD, ElementType.TYPE})`

Verfügbare Werte

ElementType	Ziel	Beispiel für die Verwendung auf dem Zielelement
ANNOTATION_TYPE	Anmerkungsarten	`@Retention(RetentionPolicy.RUNTIME)` `@interface MyAnnotation`
KONSTRUKTEUR	Konstruktoren	`@MyAnnotation` `public MyClass() {}`
FELD	Felder, Enumenkonstanten	`@XmlAttribute` `private int count;`

ElementType	Ziel	Beispiel für die Verwendung auf dem Zielelement
LOKALE VARIABLE	Variablendeklarationen innerhalb von Methoden	`for (@LoopVariable int i = 0; i < 100; i++) { @Unused String resultVariable; }`
PAKET	Paket (in `package-info.java`)	`@Deprecated package very.old;`
METHODE	Methoden	`@XmlElement public int getCount() {...}`
PARAMETER	Methoden- / Konstruktorparameter	`public Rectangle( @NamedArg("width") double width, @NamedArg("height") double height) { ... }`
ART	Klassen, Schnittstellen, Enums	`@XmlRootElement public class Report {}`

Java SE 8

ElementType	Ziel	Beispiel für die Verwendung auf dem Zielelement
TYPE_PARAMETER	Geben Sie Parameterdeklarationen ein	`public <@MyAnnotation T> void f(T t) {}`
TYPE_USE	Verwendung eines Typs	`Object o = "42"; String s = (@MyAnnotation String) o;`

@Retention

Die Meta-Annotation `@Retention` definiert die Annotationssichtbarkeit während des

Kompilierungsprozesses oder der Ausführung von Anwendungen. Standardmäßig sind Annotationen in `.class` Dateien enthalten, zur Laufzeit jedoch nicht sichtbar. Um eine Annotation zur Laufzeit zugänglich zu machen, muss `RetentionPolicy.RUNTIME` für diese Annotation festgelegt werden.

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

Verfügbare Werte

Aufbewahrungsrichtlinie	Bewirken
KLASSE	Die Anmerkung ist in der `.class` Datei verfügbar, jedoch nicht zur Laufzeit
LAUFZEIT	Die Anmerkung ist zur Laufzeit verfügbar und kann über Reflection aufgerufen werden
QUELLE	Die Anmerkung ist zur Kompilierzeit verfügbar, wird jedoch nicht zu den `.class` Dateien hinzugefügt. Die Annotation kann zB von einem Annotationsprozessor verwendet werden.

@Dokumentiert

Mit der `@Documented` Meta-Annotation werden Annotationen markiert, deren Verwendung von API-Dokumentationsgeneratoren wie Javadoc dokumentiert werden soll. Es hat keine Werte. Bei `@Documented` werden alle Klassen, die die Annotation verwenden, auf ihrer generierten Dokumentationsseite `@Documented` . Ohne `@Documented` ist nicht `@Documented` , welche Klassen die Annotation in der Dokumentation verwenden.

@Vererbt

Die `@Inherited` ist für Annotationen relevant, die auf Klassen angewendet werden. Es hat keine Werte. Durch das Markieren einer Annotation als `@Inherited` die Funktionsweise der Annotationsabfrage `@Inherited` .

- Bei einer nicht geerbten Annotation untersucht die Abfrage nur die Klasse, die untersucht wird.
- Bei einer geerbten Annotation überprüft die Abfrage auch die übergeordnete Kette (rekursiv), bis eine Instanz der Annotation gefunden wird.

Beachten Sie, dass nur die Superklassen abgefragt werden: Anmerkungen, die an Schnittstellen in der Klassenhierarchie angefügt sind, werden ignoriert.

@Wiederholbar

Die `@Repeatable` Meta-Annotation wurde in Java 8 hinzugefügt. Sie zeigt an, dass mehrere Instanzen der Annotation dem Ziel der Annotation zugeordnet werden können. Diese Meta-Annotation hat keine Werte.

Annotationswerte zur Laufzeit abrufen

Sie können die aktuellen Eigenschaften der Annotation abrufen, indem Sie Reflection verwenden , um die Methode oder das Feld oder die Klasse mit einer Annotation abzurufen, und anschließend die gewünschten Eigenschaften abzurufen.

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

Die Ausgabe wird sein

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

Anmerkungen wiederholen

Bis Java 8 konnten zwei Instanzen derselben Anmerkung nicht auf ein einzelnes Element angewendet werden. Die standardmäßige Problemumgehung bestand in der Verwendung einer Container-Annotation, die ein Array mit einer anderen Annotation enthält:

```
// 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 bietet eine sauberere und transparentere Methode zur Verwendung von Containerkommentaren mithilfe der Annotation `@Repeatable` . Zuerst fügen wir dies der `Author` Klasse hinzu:

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

Dies weist Java an, mehrere `@Author` Anmerkungen so zu behandeln, als wären sie vom `@Authors` Container umgeben. Wir können `Class.getAnnotationsByType()` auch verwenden, um auf das `@Author` Array von seiner eigenen Klasse zuzugreifen, anstatt über seinen Container:

```
@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
        }
    }
}
```

Vererbte Anmerkungen

Standardmäßig gelten Klassenanmerkungen nicht für Typen, die sie erweitern. Dies kann durch Hinzufügen der Annotation `@Inherited` zur Annotationsdefinition geändert werden

Beispiel

Beachten Sie die folgenden 2 Anmerkungen:

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

und

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

Wenn drei Klassen so kommentiert werden:

```
@UninheritedAnnotationType
class A {
}

@InheritedAnnotationType
class B extends A {
}
```

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

diesen Code ausführen

```
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));
```

gibt ein ähnliches Ergebnis aus (abhängig von den Paketen der Anmerkung):

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

Beachten Sie, dass Anmerkungen nur von Klassen und nicht von Schnittstellen geerbt werden können.

Compilierzeitverarbeitung mit Anmerkungsprozessor

In diesem Beispiel wird veranschaulicht, wie die Kompilierzeit eines kommentierten Elements überprüft wird.

Die Anmerkung

Die `@Setter` Annotation ist eine Markierung, die auf Methoden angewendet werden kann. Die Annotation wird während des Kompilierens nicht mehr verfügbar sein.

```
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 {
}
```

Der Anmerkungsprozessor

Die `SetterProcessor` Klasse wird vom Compiler zur Verarbeitung der Anmerkungen verwendet. Es prüft, ob die mit der `@Setter` Annotation annotierten Methoden `public` , nicht `static` Methoden mit einem Namen, der mit `set` beginnt und einen Großbuchstaben als vierten Buchstaben hat. Wenn eine dieser Bedingungen nicht erfüllt ist, wird ein Fehler in den `Messager` . Der Compiler schreibt dies in stderr, aber andere Tools könnten diese Informationen anders verwenden. In der NetBeans-IDE kann der Benutzer beispielsweise Anmerkungsprozessoren angeben, die zum Anzeigen von Fehlernachrichten im Editor verwendet werden.

```
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();
    }

}
```

Verpackung

Um vom Compiler angewendet zu werden, muss der Anmerkungsprozessor der SPI zur Verfügung gestellt werden (siehe ServiceLoader).

Dazu muss eine Textdatei `META-INF/services/javax.annotation.processing.Processor` zur JAR-Datei hinzugefügt werden, die den Anmerkungsprozessor und die Anmerkung zusätzlich zu den anderen Dateien enthält. Die Datei muss den vollständig qualifizierten Namen des Anmerkungsprozessors enthalten, dh es sollte so aussehen

```
annotation.processor.SetterProcessor
```

Wir nehmen an, dass die JAR-Datei unten als `AnnotationProcessor.jar` wird.

Beispiel kommentierte Klasse

Die folgende Klasse ist eine Beispielklasse im Standardpaket, wobei die Anmerkungen gemäß der Aufbewahrungsrichtlinie auf die richtigen Elemente angewendet werden. Nur der Anmerkungsprozessor betrachtet jedoch nur die zweite Methode als gültiges Anmerkungsziel.

```
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) {}

}
```

Verwenden des Anmerkungsprozessors mit Javac

Wenn der Anmerkungsprozessor mithilfe der SPI erkannt wird, wird er automatisch zur Verarbeitung von Anmerkungselementen verwendet. ZB Kompilieren der `AnnotationProcessorTest` Klasse mit

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

ergibt die folgende Ausgabe

```
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
```

anstatt normal zu kompilieren. Es wird keine `.class` Datei erstellt.

Dies könnte durch die Angabe der verhindert werden `-proc:none` Option für `javac` . Sie können auch auf die übliche Kompilierung verzichten, indem `-proc:only` stattdessen `-proc:only` .

IDE-Integration

Netbeans

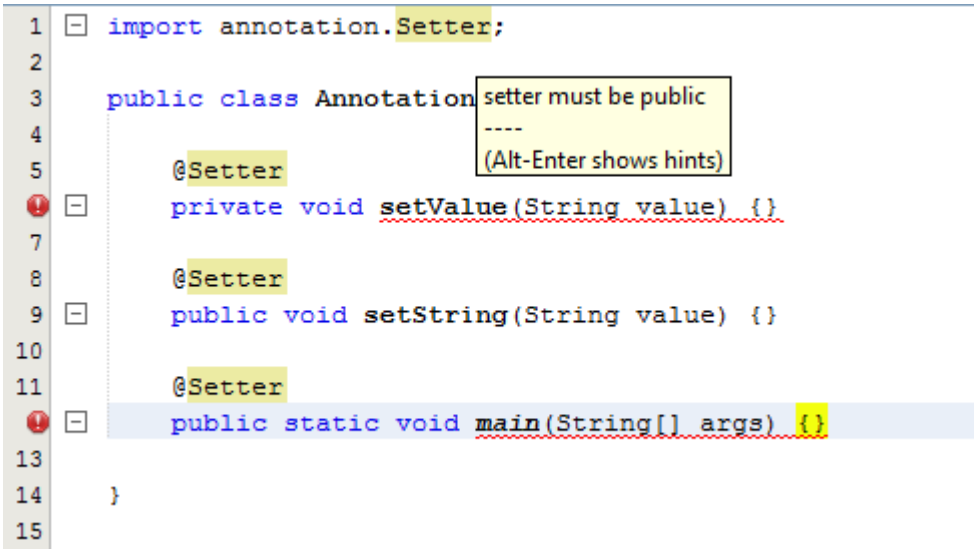
Anmerkungsprozessoren können im NetBeans-Editor verwendet werden. Dazu muss der Anmerkungsprozessor in den Projekteinstellungen angegeben werden:

1. Gehen Sie zu `Project Properties` > `Build` > `Compiling`
2. Hinzufügen von Häkchen für `Enable Annotation Processing` und `Enable Annotation Processing in Editor`

3. Klicken `Add` neben der Anmerkungsprozessorliste auf `Add`

4. Geben Sie im angezeigten Popup den vollständig qualifizierten Klassennamen des Anmerkungsprozessors ein und klicken Sie auf `Ok` .

Ergebnis



Die Idee hinter den Anmerkungen

In der Java-Sprachspezifikation werden Annotationen wie folgt beschrieben:

> Eine Annotation ist eine Markierung, die Informationen mit einem Programmkonstrukt verknüpft, jedoch zur Laufzeit keine Auswirkungen hat.

Anmerkungen können vor Typen oder Deklarationen erscheinen. Es ist möglich, dass sie an einem Ort erscheinen, an dem sie sich sowohl auf einen Typ als auch auf eine Deklaration beziehen können.
Für was genau eine Anmerkung gilt, wird von `@Target` "meta-annotation" `@Target` . Weitere Informationen finden Sie unter "Annotationstypen definieren" .

Anmerkungen werden für eine Vielzahl von Zwecken verwendet. Frameworks wie Spring und Spring-MVC verwenden Anmerkungen, um zu definieren, wo Abhängigkeiten eingefügt werden sollen oder wohin Anforderungen geleitet werden sollen.

Andere Frameworks verwenden Annotationen für die Codegenerierung. Lombok und JPA sind erstklassige Beispiele, die Annotationen verwenden, um Java-Code (und SQL-Code) zu erstellen.

Dieses Thema soll einen umfassenden Überblick über Folgendes vermitteln:

- Wie definiere ich eigene Anmerkungen?
- Welche Anmerkungen bietet die Java-Sprache?
- Wie werden Annotationen in der Praxis verwendet?

Anmerkungen für 'this' und Empfängerparameter

Bei der Einführung von Java-Annotationen war es nicht vorgesehen, das Ziel einer Instanzmethode oder den Parameter für einen ausgeblendeten Konstruktor für einen Konstruktor für innere Klassen zu kommentieren. Dies wurde in Java 8 durch Hinzufügen von *Empfängerparameterdeklarationen* behoben. siehe JLS 8.4.1 .

> Der Empfängerparameter ist ein optionales syntaktisches Gerät für eine Instanzmethode oder den Konstruktor einer inneren Klasse. Bei einer Instanzmethode stellt der Empfängerparameter das Objekt dar, für das die Methode aufgerufen wird. Für den Konstruktor einer inneren Klasse stellt der Empfängerparameter die unmittelbar einschließende Instanz des neu erstellten Objekts dar. In beiden Fällen ist der Empfängerparameter nur vorhanden, um zu ermöglichen, dass der Typ des dargestellten Objekts im Quellcode angegeben wird, sodass der Typ mit Anmerkungen versehen werden kann. Der Empfängerparameter ist kein formaler Parameter. Genauer gesagt, es handelt sich nicht um eine Deklaration irgendeiner Art von Variable (§4.12.3), sie ist niemals an einen Wert gebunden, der als Argument in einem Methodenaufrufausdruck oder einem qualifizierten Klasseninstanzerstellungsausdruck übergeben wird Laufzeit

Das folgende Beispiel veranschaulicht die Syntax für beide Arten von Empfängerparametern:

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

Der einzige Zweck der Empfängerparameter besteht darin, dass Sie Anmerkungen hinzufügen können. Beispielsweise haben Sie möglicherweise eine benutzerdefinierte Annotation `@IsOpen` deren Zweck darin besteht, zu bestätigen, dass ein `Closeable` Objekt beim `Closeable` einer Methode nicht geschlossen wurde. Zum Beispiel:

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

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

Auf einer Ebene könnte die `@IsOpen` Anmerkung `this` einfach als Dokumentation dienen. Wir könnten jedoch möglicherweise mehr tun. Zum Beispiel:

- Ein Anmerkungsprozessor könnte eine Laufzeitüberprüfung einfügen, die besagt, dass sich `this` nicht im geschlossenen Zustand befindet, wenn das `update` aufgerufen wird.
- Ein Code-Checker könnte eine statische Code-Analyse durchführen, um Fälle zu finden, in denen `this` beim Aufruf der `update` geschlossen werden *könnte* .

Fügen Sie mehrere Anmerkungswerte hinzu

Ein Annotation-Parameter kann mehrere Werte akzeptieren, wenn er als Array definiert ist. Zum Beispiel ist die Standardannotation `@SuppressWarnings` folgendermaßen definiert:

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

Der `value` Parameter ist ein Array von Strings. Sie können mehrere Werte festlegen, indem Sie eine Notation verwenden, die den Array-Initialisierern ähnelt:

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

Wenn Sie nur einen einzelnen Wert festlegen müssen, können die Klammern weggelassen werden:

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

Anmerkungen online lesen: https://riptutorial.com/de/java/topic/157/anmerkungen

Kapitel 5: Apache Commons Lang

Examples

Implementieren Sie die Methode equals ()

Um die `equals` Methode eines Objekts einfach zu implementieren, können Sie die `EqualsBuilder` Klasse verwenden.

Auswahl der Felder:

```
@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();
}
```

Reflexion verwenden:

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

Der boolesche Parameter gibt an, ob Gleiche transiente Felder prüfen sollen.

Reflexion verwenden, um einige Felder zu vermeiden:

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

Implementieren Sie die hashCode () -Methode

Um die Umsetzung `hashCode` Methode eines Objekts leicht könnte man die Verwendung `HashCodeBuilder` Klasse.

Auswahl der Felder:

```
@Override
public int hashCode() {

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

    return builder.hashCode();
}
```

Reflexion verwenden:

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

Der boolesche Parameter gibt an, ob transiente Felder verwendet werden sollen.

Reflexion verwenden, um einige Felder zu vermeiden:

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

Implementieren Sie die toString () -Methode

Um die `toString` Methode eines Objekts einfach zu implementieren, können Sie die `ToStringBuilder` Klasse verwenden.

Auswahl der Felder:

```
@Override
public String toString() {

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

    return builder.toString();
}
```

Beispielergebnis:

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

Explizite Benennung der Felder:

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

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

    return builder.toString();
}
```

Beispielergebnis:

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

Sie können den Stil über Parameter ändern:

```
@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();
}
```

Beispielergebnis:

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

Es gibt einige Stile, z. B. JSON, keinen Klassennamen, Kurzzeichen usw.

Über Reflexion:

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

Sie können auch den Stil angeben:

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

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

Kapitel 6: AppDynamics und TIBCO BusinessWorks Instrumentation für einfache Integration

Einführung

Da AppDynamics eine Möglichkeit bietet, die Anwendungsleistung zu messen, ist die Geschwindigkeit der Entwicklung, Bereitstellung (Bereitstellung) von Anwendungen ein wesentlicher Faktor, um die Bemühungen von DevOps zu einem echten Erfolg zu machen. Das Überwachen einer TIBCO BW-Anwendung mit AppD ist im Allgemeinen einfach und nicht zeitaufwändig. Bei der Bereitstellung großer Anwendungsgruppen ist jedoch eine schnelle Instrumentierung von entscheidender Bedeutung. In diesem Handbuch wird gezeigt, wie Sie alle Ihre BW-Anwendungen in einem einzigen Schritt instrumentieren können, ohne die einzelnen Anwendungen vor der Bereitstellung zu ändern.

Examples

Beispiel für die Instrumentierung aller BW-Anwendungen in einem Schritt für Appdynamics

1. Suchen und öffnen Sie Ihre TIBCO BW bwengine.tra-Datei unter TIBCO_HOME / bw / 5.12 / bin / bwengine.tra (Linux-Umgebung).

2. Suchen Sie nach der Zeile, die besagt:

*** Gemeinsame Variablen. Ändern Sie nur diese. ***

3. Fügen Sie die folgende Zeile direkt nach diesem Abschnitt ein. Tibco.deployment =% tibco.deployment%

4. Gehen Sie an das Ende der Datei und fügen Sie hinzu (ersetzen Sie? Durch Ihre eigenen Werte oder entfernen Sie das nicht zutreffende Flag). 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. Speichern Sie die Datei und stellen Sie sie erneut bereit. Alle Ihre Anwendungen sollten jetzt

automatisch zum Zeitpunkt der Bereitstellung instrumentiert werden.

AppDynamics und TIBCO BusinessWorks Instrumentation für einfache Integration online lesen: https://riptutorial.com/de/java/topic/10602/appdynamics-und-tibco-businessworks-instrumentation-fur-einfache-integration

Kapitel 7: Applets

Einführung

Applets sind seit der offiziellen Veröffentlichung Teil von Java und werden seit einigen Jahren zum Unterrichten von Java und zum Programmieren verwendet.

In den letzten Jahren gab es einen aktiven Anstoß, sich von Applets und anderen Browser-Plugins zu entfernen. Einige Browser blockierten sie oder unterstützten sie nicht aktiv.

Im Jahr 2016 kündigte Oracle an, das Plugin "In ein Plugin-Free-Web wechseln" abzulehnen

Neuere und bessere APIs sind jetzt verfügbar

Bemerkungen

Ein Applet ist eine Java-Anwendung, die normalerweise in einem Webbrowser ausgeführt wird. Die Grundidee besteht darin, mit dem Benutzer zu interagieren, ohne mit dem Server interagieren und Informationen übertragen zu müssen. Dieses Konzept war um das Jahr 2000 sehr erfolgreich, als die Internetkommunikation langsam und teuer war.

Ein Applet bietet fünf Methoden zur Steuerung des Lebenszyklus.

Methodenname	Beschreibung
`init()`	wird einmal aufgerufen, wenn das Applet geladen wird
`destroy()`	wird einmal aufgerufen, wenn das Applet aus dem Speicher entfernt wird
`start()`	wird aufgerufen, wenn das Applet sichtbar wird
`stop()`	wird aufgerufen, wenn das Applet von anderen Fenstern überlappt wird
`paint()`	wird bei Bedarf aufgerufen oder manuell durch Aufruf von `repaint()` ausgelöst

Examples

Minimales Applet

Ein sehr einfaches Applet zeichnet ein Rechteck und druckt eine Zeichenfolge auf dem Bildschirm.

```
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);
    }
}
```

Die Hauptklasse eines Applets reicht von `javax.swing.JApplet` .

Java SE 1.2

Vor Java 1.2 und der Einführung der Swing-API-Applets hatte sich `java.applet.Applet` .

Applets erfordern keine Hauptmethode. Der Einstiegspunkt wird vom Lebenszyklus gesteuert. Um sie verwenden zu können, müssen sie in ein HTML-Dokument eingebettet sein. Dies ist auch der Punkt, an dem ihre Größe definiert wird.

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

Erstellen einer GUI

Applets können leicht zum Erstellen einer GUI verwendet werden. Sie `awt` wie ein `Container` und haben eine `add()` Methode, die beliebige `awt` oder `swing` Komponenten übernimmt.

```
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);
    }
}
```

Öffnen Sie Links innerhalb des Applets

Sie können die Methode `getAppletContext()` , um ein `AppletContext` Objekt zu erhalten, mit dem Sie den Browser zum Öffnen eines Links `AppletContext` können. Dazu verwenden Sie die Methode `showDocument()` . Sein zweiter Parameter weist den Browser an, ein neues Fenster `_blank` oder das Fenster, in dem das 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);
    }
}
```

Laden von Bildern, Audio und anderen Ressourcen

Java-Applets können verschiedene Ressourcen laden. Da sie jedoch im Webbrowser des Clients ausgeführt werden, müssen Sie sicherstellen, dass auf diese Ressourcen zugegriffen werden kann. Applets können nicht als lokales Dateisystem auf Clientressourcen zugreifen.

Wenn Sie Ressourcen von derselben URL laden möchten, in der das Applet gespeichert ist, können Sie mit der Methode `getCodeBase()` die Basis-URL abrufen. Zum Laden von Ressourcen bieten Applets die Methoden `getImage()` und `getAudioClip()` zum Laden von Bildern oder Audiodateien.

Laden und zeigen Sie ein Bild

```
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);
        }
}
```

Laden und spielen Sie eine Audiodatei

```
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();
    }
}
```

Laden Sie eine Textdatei und zeigen Sie sie an

```
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 */ }
    }
}
```

Applets online lesen: https://riptutorial.com/de/java/topic/5503/applets

Kapitel 8: Arrays

Einführung

Arrays ermöglichen das Speichern und Abrufen einer beliebigen Menge von Werten. Sie sind analog zu Vektoren in der Mathematik. Arrays von Arrays sind analog zu Matrizen und dienen als mehrdimensionale Arrays. Arrays können beliebige Daten eines beliebigen Typs speichern: Grundelemente wie `int` oder Referenztypen wie `Object` .

Syntax

- `ArrayType[] myArray;` // Arrays deklarieren
- `ArrayType myArray[];` // Eine andere gültige Syntax (weniger häufig verwendet und nicht empfohlen)
- `ArrayType[][][] myArray;` // Multi-dimensionale gezackte Arrays deklarieren (repeat [] s)
- `ArrayType myVar = myArray[index];` // Zugriff auf (Lese-) Element am Index
- `myArray[index] = value;` // Assign Wert `index` des Arrays
- `ArrayType[] myArray = new ArrayType[arrayLength];` // Array-Initialisierungssyntax
- `int[] ints = {1, 2, 3};` // Array-Initialisierungssyntax mit angegebenen Werten, Länge wird aus der Anzahl der angegebenen Werte abgeleitet: {[value1 [, value2] *]}
- `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};` // Mehrdimensionale Array-Initialisierung. int [] erweitert Object (und damit auch anyType []), sodass null einen gültigen Wert darstellt.

Parameter

Parameter	Einzelheiten
`ArrayType`	Typ des Arrays Dies kann primitiv (`int` , `long` , `byte`) oder Objekte (`String` , `MyObject` usw.) sein.
Index	Index bezieht sich auf die Position eines bestimmten Objekts in einem Array.
Länge	Jedes Array benötigt beim Erstellen eine festgelegte Länge. Dies geschieht entweder beim Erstellen eines leeren Arrays (`new int[3]`) oder beim Angeben von Werten (`{1, 2, 3}`).

Examples

Arrays erstellen und initialisieren

Grundlegende Fälle

```
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
```

Arrays können mit einem beliebigen Grundelement oder Referenztyp erstellt werden.

```
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).
```

Beachten Sie für das letzte Beispiel, dass Untertypen des deklarierten Array-Typs im Array zulässig sind.

Arrays für benutzerdefinierte Typen können auch ähnlich zu primitiven Typen erstellt werden

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

Arrays, Sammlungen und Streams

Java SE 1.2

```
// Parameters require objects, not primitives

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

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

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

Java SE 8

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

Intro

Ein *Array* ist eine Datenstruktur, die eine feste Anzahl von Grundwerten **oder** Referenzen auf Objektinstanzen enthält.

Jedes Element in einem Array wird als Element bezeichnet, und auf jedes Element wird über seinen numerischen Index zugegriffen. Die Länge eines Arrays wird festgelegt, wenn das Array erstellt wird:

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

Die Größe eines Arrays wird bei der Initialisierung zur Laufzeit festgelegt. Sie kann nach der Initialisierung nicht geändert werden. Wenn die Größe zur Laufzeit veränderbar sein muss, sollte stattdessen eine `Collection` Klasse wie `ArrayList` verwendet werden. `ArrayList` speichert Elemente in einem Array und unterstützt die Größenänderung, indem ein neues Array zugewiesen und Elemente aus dem alten Array kopiert werden.

Wenn das Array von einem primitiven Typ ist, d

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

Die Werte werden im Array selbst gespeichert. In Abwesenheit eines Initialisierers (wie in `array2` oben) ist der jedem Element zugewiesene Standardwert `0` (Null).

Wenn der Array-Typ eine Objektreferenz ist, wie in

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

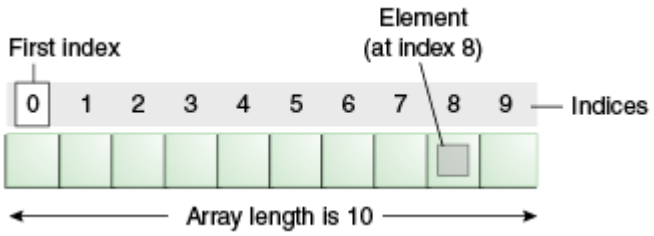
dann enthält das Array *Verweise* auf Objekte vom Typ `SomeClassOrInterface` . Diese Verweise können sich auf eine Instanz von `SomeClassOrInterface` *oder eine beliebige Unterklasse (für Klassen) oder die implementierende Klasse (für Schnittstellen) von* `SomeClassOrInterface` . Wenn die Array-Deklaration keinen Initialisierer hat, wird jedem Element der Standardwert `null` zugewiesen.

Da alle Arrays `int` -indexiert sind, muss die Größe eines Arrays durch ein `int` angegeben werden. Die Größe des Arrays kann nicht als `long` :

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

Arrays verwenden ein auf **Null basierendes Indexsystem** , das heißt, die Indizierung beginnt bei `0` und endet bei `length - 1 0` .

Das folgende Bild stellt beispielsweise ein Array mit der Größe `10` . Hier befindet sich das erste Element am Index `0` und das letzte Element am Index `9` , anstelle des ersten Elements am Index `1` und des letzten Elements am Index `10` (siehe Abbildung unten).



Zugriffe auf Elemente von Arrays erfolgen in **konstanter Zeit** . Das bedeutet, dass der Zugriff auf das erste Element des Arrays (zeitlich) die gleichen Kosten für den Zugriff auf das zweite Element, das dritte Element usw. verursacht.

Java bietet verschiedene Möglichkeiten zum Definieren und Initialisieren von Arrays, einschließlich **Literal-** und **Konstruktornotationen** . Bei der Deklaration von Arrays mit dem `new Type[length]` - Konstruktor wird jedes Element mit den folgenden Standardwerten initialisiert:

- `0` für primitive numerische Typen : `byte` , `short` , `int` , `long` , `float` und `double` .
- `'\u0000'` (null character) für den `char` - Typen.
- `false` für den `boolean` Typ.
- `null` für Referenztypen .

Erstellen und Initialisieren von primitiven Arrays

```
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.
```

Bei der Deklaration eines Arrays erscheint `[]` als Teil des Typs am Anfang der Deklaration (nach dem Typnamen) oder als Teil des Deklarators für eine bestimmte Variable (nach Variablenname) oder beides:

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

Im folgenden Beispiel sind beide Deklarationen korrekt und können ohne Probleme kompiliert und ausgeführt werden. Sowohl die Java Coding Convention als auch der Google Java Style Guide raten jedoch davon ab, dass das Formular mit Klammern hinter dem Variablennamen steht. Die Klammern geben den Array-Typ an und sollten mit der Typbezeichnung angezeigt werden . Dasselbe sollte für Methodenrücksignaturen verwendet werden.

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

Der entmutigte Typ ist für übergebende C-Benutzer gedacht , die mit der Syntax für C vertraut sind, das nach dem Variablennamen die Klammern hat.

In Java können Arrays der Größe `0` :

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

Da es sich jedoch um ein leeres Array handelt, können keine Elemente daraus gelesen oder zugewiesen werden:

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

Solche leeren Arrays sind normalerweise als Rückgabewerte nützlich, so dass sich der aufrufende Code nur um den Umgang mit einem Array kümmern muss, anstatt einen potenziellen `null` , der zu einer `NullPointerException` führen `NullPointerException` .

Die Länge eines Arrays muss eine nicht negative ganze Zahl sein:

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

Die Arraygröße kann mithilfe eines öffentlichen letzten Felds namens `length` :

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

Anmerkung : `array.length` gibt die tatsächliche Größe des Arrays und nicht die Anzahl der `array.length` zurück, denen ein Wert zugewiesen wurde, im Gegensatz zu `ArrayList.size()` das die Anzahl der `array.length` zurückgibt, denen ein Wert zugewiesen wurde.

Erstellen und Initialisieren von mehrdimensionalen Arrays

Die einfachste Methode zum Erstellen eines mehrdimensionalen Arrays ist wie folgt:

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

Es werden zwei `int` Arrays mit drei Längen erstellt - `a[0]` und `a[1]` . Dies ist der klassischen Initialisierung im C-Stil von rechteckigen mehrdimensionalen Arrays sehr ähnlich.

Sie können gleichzeitig erstellen und initialisieren:

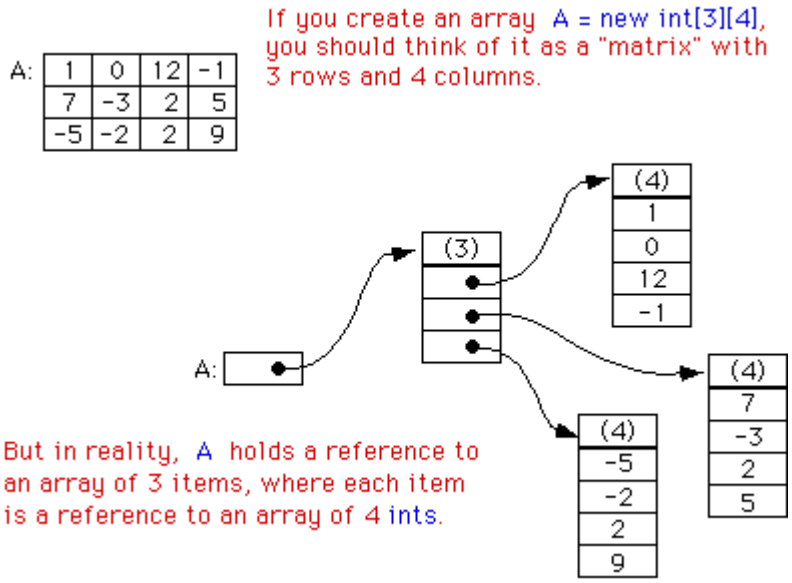
```
int[][] a = { {1, 2}, {3, 4}, {5, 6} };
```

Im Gegensatz zu C müssen , wenn nur rechteckige mehrdimensionale Arrays unterstützt werden, innere Arrays nicht die gleiche Länge haben oder sogar definiert sein:

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

Hier ist `a[0]` ein `int` Array mit einer Länge, wohingegen `a[1]` ein `int` Array mit zwei Längen und `a[2] null` . Arrays wie dieses werden gezackte Arrays oder unregelmäßige Arrays genannt , das heißt Arrays von Arrays. Mehrdimensionale Arrays in Java werden als Arrays von Arrays implementiert, dh `array[i][j][k]` entspricht `((array[i])[j])[k]` . Im Gegensatz zu C # wird das Syntax- `array[i,j]` in Java nicht unterstützt.

Mehrdimensionale Array-Darstellung in Java



Source - Live auf Ideone

Erstellen und Initialisieren von Referenztyp-Arrays

```
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
```

Live auf Ideone

Zusätzlich zu den oben gezeigten `String` Literalen und Primitiven funktioniert die Abkürzungssyntax für die Array-Initialisierung auch mit kanonischen `Object` :

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

Da Arrays kovariant sind, kann ein Referenztyparray als Array einer Unterklasse `ArrayStoreException` werden. Eine `ArrayStoreException` wird jedoch ausgelöst, wenn Sie versuchen, ein Element auf einen anderen `String` als einen `String` :

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

Die Shortcut-Syntax kann dafür nicht verwendet werden, da die Shortcut-Syntax einen impliziten Typ von `Object[]` .

Ein Array kann mit null Elementen unter Verwendung von `String[] emptyArray = new String[0]` initialisiert werden. Ein Array mit der Länge Null wie diesem wird beispielsweise zum Erstellen eines `Array` aus einer `Collection` wenn die Methode den Laufzeittyp eines Objekts benötigt.

Sowohl bei primitiven als auch bei Referenztypen initialisiert eine leere Array-Initialisierung (z. B. `String[] array8 = new String[3]`) das Array mit dem Standardwert für jeden Datentyp .

Generische Typ- Arrays erstellen und initialisieren

In generischen Klassen können Arrays generischer Typen aufgrund von Typlöschung **nicht** wie folgt initialisiert werden:

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

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

Sie können stattdessen mit einer der folgenden Methoden erstellt werden: (Beachten Sie, dass diese ungeprüfte Warnungen erzeugen.)

1. Erstellen Sie ein `Object` Array und geben Sie es in einen generischen Typ um:

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

   Dies ist die einfachste Methode. Da das zugrunde liegende Array jedoch immer noch vom Typ `Object[]` bietet diese Methode keine Typsicherheit. Daher wird diese Methode zum Erstellen eines Arrays am besten nur innerhalb der generischen Klasse verwendet - nicht öffentlich verfügbar gemacht.

2. Verwenden Sie `Array.newInstance` mit einem Klassenparameter:

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

Hier muss die Klasse von `T` explizit an den Konstruktor übergeben werden. Der Rückgabetyp von `Array.newInstance` ist immer `Object` . Diese Methode ist jedoch sicherer, da das neu erstellte Array immer vom Typ `T[]` ist und daher sicher externalisiert werden kann.

Array nach der Initialisierung füllen

Java SE 1.2

`Arrays.fill()` kann verwendet werden, um ein Array nach der Initialisierung mit **demselben Wert** zu füllen:

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

Live auf Ideone

`fill()` kann auch jedem Element des angegebenen Bereichs des Arrays einen Wert zuweisen:

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

Live auf Ideone

Java SE 8

Seit Java Version 8 können mit der Methode `setAll` und ihrem `Concurrent` Äquivalent `parallelSetAll` jedes Element eines Arrays auf generierte Werte gesetzt werden. Diesen Methoden wird eine Generatorfunktion übergeben, die einen Index akzeptiert und den gewünschten Wert für diese Position zurückgibt.

Im folgenden Beispiel wird ein Integer-Array erstellt und alle Elemente auf ihren jeweiligen Indexwert festgelegt:

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

Live auf Ideone

Separate Deklaration und Initialisierung von Arrays

Der Wert eines Index für ein Array-Element muss eine ganze Zahl (0, 1, 2, 3, 4, ...) und weniger als die Länge des Arrays sein (Indizes basieren auf Null). Andernfalls wird eine

ArrayIndexOutOfBoundsException ausgelöst:

```
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 }
```

Arrays können mit der Array-Initialisierungs-Verknüpfungssyntax nicht erneut initialisiert werden

Es ist nicht möglich, ein Array über eine Verknüpfungssyntax mit einem Array-Initialisierer erneut zu initialisieren, da ein Array-Initialisierer nur in einer Felddeklaration oder einer Deklaration lokaler Variablen oder als Teil eines Ausdrucks zur Array-Erstellung angegeben werden kann.

Es ist jedoch möglich, ein neues Array zu erstellen und es der Variablen zuzuweisen, die zum Referenzieren des alten Arrays verwendet wird. Während dies dazu führt, dass das Array, auf das diese Variable verweist, neu initialisiert wird, ist der Inhalt der Variablen ein völlig neues Array. Zu diesem Zweck kann der `new` Operator mit einem Array-Initialisierer verwendet und der Array-Variablen zugewiesen werden:

```
// 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.
```

Live auf Ideone

Erstellen eines Arrays aus einer Sammlung

Zwei Methoden in `java.util.Collection` erstellen ein Array aus einer Auflistung:

- `Object[] toArray()`

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

`Object[] toArray()` kann wie folgt verwendet werden:

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)` kann wie folgt verwendet werden:

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()]);
```

Der Unterschied zwischen ihnen ist mehr als nur die Eingabe von untypisierten und nicht typisierten Ergebnissen. Ihre Leistung kann auch unterschiedlich sein (für Details lesen Sie bitte diesen Abschnitt zur Leistungsanalyse):

- `Object[] toArray()` verwendet `arraycopy` , was viel schneller ist als die `arraycopy` die in `T[] toArray(T[] a)` .
- `T[] toArray(new T[non-zero-size])` muss das Array zur Laufzeit auf Null setzen, während `T[] toArray(new T[0])` dies nicht tut. Eine solche Vermeidung macht den letzteren schneller als den ersten. Detaillierte Analyse hier: Arrays der Weisheit der Ahnen .

Java SE 8

Ausgehend von Java SE 8+, wo das Konzept des `Stream` eingeführt wurde, ist es möglich, den von der Auflistung erzeugten `Stream` zu verwenden, um ein neues Array mit der `Stream.toArray` Methode zu erstellen.

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

Beispiele aus zwei Antworten (1 , 2) bis Konvertierung von 'ArrayList in' String [] 'in Java bei Stapelüberlauf.

Arrays zu einem String

Java SE 5

Seit Java 1.5 können Sie eine `String` Darstellung des Inhalts des angegebenen Arrays erhalten, ohne jedes einzelne Element zu durchlaufen. Verwenden `Arrays.toString(Object[])` einfach `Arrays.toString(Object[])` oder `Arrays.deepToString(Object[])` für multidimentale Arrays:

```
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()` -Methode verwendet die `Object.toString()` -Methode, um `String` Werte für jedes Element im Array zu erzeugen. Neben dem primitiven `Object.toString()` kann er für alle Array-Typen verwendet werden. Zum Beispiel:

```
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!]
```

Wenn für die Klasse kein überschriebenes `toString()` vorhanden ist, wird das von `Object` geerbte `toString()` verwendet. Normalerweise ist die Ausgabe dann nicht sehr nützlich, zum Beispiel:

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

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

Erstellen einer Liste aus einem Array

Die `Arrays.asList()` -Methode kann verwendet werden, um eine `List` mit fester Größe zurückzugeben, die die Elemente des angegebenen Arrays enthält. Die resultierende `List` den gleichen Parametertyp wie der Basistyp des Arrays.

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

Hinweis : Diese Liste wird durch das ursprüngliche Array (*eine Ansicht* des ursprünglichen

Arrays) unterstützt, was bedeutet, dass Änderungen an der Liste das Array ändern und umgekehrt. Änderungen an der Liste, die die Größe (und damit die Arraylänge) ändern würden, lösen jedoch eine Ausnahme aus.

Verwenden Sie zum Erstellen einer Kopie der Liste den Konstruktor von `java.util.ArrayList` wobei eine `Collection` als Argument verwendet wird:

Java SE 5

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

Java SE 7

In Java SE 7 und höher kann ein Paar spitze Klammern `<>` (leere Menge von Typargumenten) verwendet werden, das als Diamant bezeichnet wird . Der Compiler kann die Typargumente aus dem Kontext ermitteln. Dies bedeutet, dass die `ArrayList` beim Aufruf des Konstruktors von `ArrayList` ausgelassen werden können und beim Kompilieren automatisch abgeleitet werden. Dies wird als Type Inference bezeichnet, einem Bestandteil von 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);
```

Ein bemerkenswerter Punkt bezüglich des Diamanten ist, dass er nicht mit anonymen Klassen verwendet werden kann .

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());
```

Wichtige Hinweise zur Verwendung der Arrays.asList () -

Methode

- Diese Methode gibt `List` , eine Instanz von `Arrays$ArrayList` (statische innere Klasse von `Arrays`) und nicht `java.util.ArrayList` . Die resultierende `List` hat eine feste Größe. Das bedeutet, dass das Hinzufügen oder Entfernen von Elementen nicht unterstützt wird und eine `UnsupportedOperationException` :

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

- Eine neue `List` kann erstellt werden, indem eine Array-unterstützte `List` an den Konstruktor einer neuen `List` . Dadurch wird eine neue Kopie der Daten erstellt, deren Größe veränderbar ist und die nicht vom ursprünglichen Array unterstützt wird:

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

- Durch Aufrufen von `<T> List<T> asList(T... a)` auf einem primitiven Array, beispielsweise einem `int[]` , wird ein `List<int[]>` dessen einziges Element das primitive Quellarray anstelle der eigentlichen Elemente ist des Quellarrays.

  Der Grund für dieses Verhalten ist, dass primitive Typen nicht anstelle generischer Typparameter verwendet werden können. In diesem Fall ersetzt das gesamte primitive Array den generischen Typparameter. Um ein primitives Array in eine `List` zu konvertieren, konvertieren Sie zuerst das primitive Array in ein Array des entsprechenden Wrapper-Typs (dh rufen Sie `Arrays.asList` für eine `Integer[]` statt eines `int[]`) auf.

  Daher wird dies `false` gedruckt:

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

  Demo anzeigen

  Auf der anderen Seite wird dies `true` gedruckt:

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

  Demo anzeigen

  Dies wird auch als `true` , da das Array als `Integer[]` interpretiert wird:

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

  Demo anzeigen

Mehrdimensionale und gezackte Arrays

Es ist möglich, ein Array mit mehr als einer Dimension zu definieren. Anstatt auf einen einzelnen Index zuzugreifen, wird auf ein mehrdimensionales Array durch Angabe eines Index für jede Dimension zugegriffen.

Die Deklaration eines mehrdimensionalen Arrays kann durch Hinzufügen von `[]` für jede Dimension zu einer regulären Array-Deklaration erfolgen. Um beispielsweise ein 2-dimensionales `int` Array zu `int` , fügen Sie der Deklaration einen weiteren Satz von Klammern hinzu, beispielsweise `int[][]` . Dies gilt für 3-dimensionale Arrays (`int[][][]`) und so weiter.

So definieren Sie ein zweidimensionales Array mit drei Zeilen und drei Spalten:

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

Das Array kann mit diesem Konstrukt indiziert werden und ihm Werte zuweisen. Beachten Sie, dass die nicht zugewiesenen Werte die Standardwerte für den Typ eines Arrays sind, in diesem Fall `0` für `int` .

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

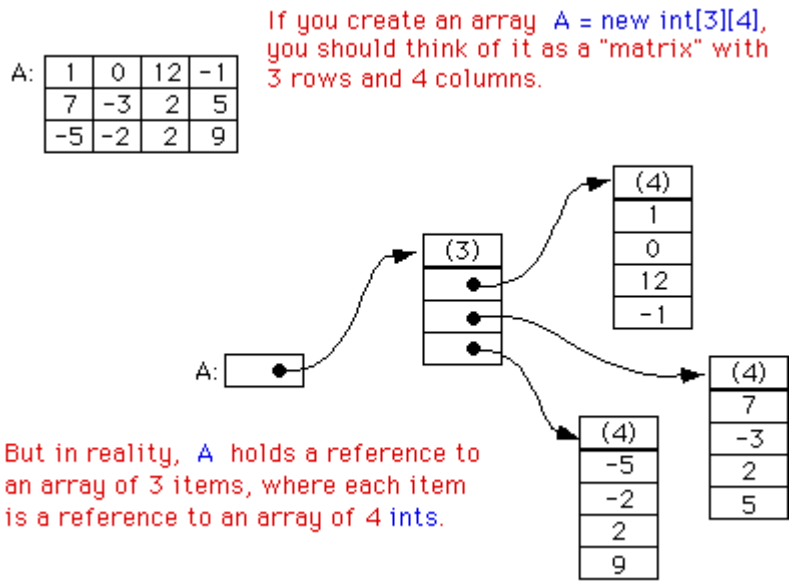
Es ist auch möglich, eine Dimension auf einmal zu instanziieren und sogar nicht rechteckige Arrays zu erstellen. Diese werden häufiger als **gezackte Arrays bezeichnet** .

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

Es ist wichtig zu beachten, dass es zwar möglich ist, eine beliebige Dimension eines gezackten Arrays zu definieren, die vorherige Ebene ***muss*** jedoch definiert werden.

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

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

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

Wie multidimensionale Arrays in Java dargestellt werden



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

---

Gezackte Array-Intialisierung

Mehrdimensionale Arrays und gezackte Arrays können auch mit einem Literalausdruck initialisiert werden. Im Folgenden wird ein 2x3 `int` Array deklariert und aufgefüllt:

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

Hinweis : Gezackte Unterfelder können auch `null` . Mit dem folgenden Code wird beispielsweise ein zweidimensionales `int` Array deklariert und aufgefüllt, dessen erstes Unterfeld `null` , das zweite Unterfeld die Länge Null hat, das dritte Unterfeld eine Länge hat und das letzte Unterfeld ein Array mit zwei Längen ist:

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

Bei mehrdimensionalen Arrays können Arrays mit untergeordneten Dimensionen anhand ihrer Indizes extrahiert werden:

```
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

Die `ArrayIndexOutOfBoundsException` wird ausgelöst, wenn auf einen nicht vorhandenen Index eines Arrays zugegriffen wird.

Arrays sind nullbasiert, daher ist der Index des ersten Elements `0` und der Index des letzten Elements ist die `array.length - 1` minus `1` (dh `array.length - 1`).

Daher muss jede Anforderung eines Arrayelements durch den Index `i` die Bedingung `0 <= i < array.length` , andernfalls wird die `ArrayIndexOutOfBoundsException` ausgelöst.

Der folgende Code ist ein einfaches Beispiel, in dem eine `ArrayIndexOutOfBoundsException` ausgelöst wird.

```
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.
```

Ausgabe:

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

Beachten Sie, dass der illegale Index, auf den zugegriffen wird, auch in der Ausnahme enthalten ist (`2` im Beispiel). Diese Informationen können hilfreich sein, um die Ursache der Ausnahme zu ermitteln.

Um dies zu vermeiden, überprüfen Sie einfach, ob sich der Index innerhalb der Grenzen des Arrays befindet:

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

Länge eines Arrays ermitteln

Arrays sind Objekte, die Platz für die Speicherung von Elementen des angegebenen Typs bieten. Die Größe eines Arrays kann nicht geändert werden, nachdem das Array erstellt wurde.

```
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
```

Das `length` in einem Array speichert die Größe eines Arrays. Es ist ein `final` Feld und kann nicht geändert werden.

Dieser Code zeigt den Unterschied zwischen der `length` eines Arrays und der Anzahl der Objekte, die ein Array speichert.

```
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");
}
```

Ergebnis:

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

Vergleich von Arrays auf Gleichheit

Array-Typen erben ihre Implementierungen von `equals()` (und `hashCode()`) von java.lang.Object. Daher gibt `equals()` nur beim Vergleich mit demselben Array-Objekt true zurück. Verwenden Sie `java.util.Arrays.equals` , um Arrays auf Gleichheit basierend auf ihren Werten zu vergleichen, die für alle Array-Typen überlastet ist.

```
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
```

Wenn der Elementtyp ein Referenztyp ist, ruft `Arrays.equals()` `equals()` für die Array-Elemente auf, um die Gleichheit zu bestimmen. Wenn der Elementtyp selbst ein Array-Typ ist, wird der Identitätsvergleich verwendet. Um multidimensionale Arrays auf Gleichheit zu vergleichen, verwenden `Arrays.deepEquals()` stattdessen `Arrays.deepEquals()`

```
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
```

Da Sets und Maps `equals()` und `hashCode()` , sind Arrays im Allgemeinen nicht als Setelemente oder Map-Schlüssel nützlich. Entweder wickeln sie in eine Hilfsklasse , die implementiert `equals()` und `hashCode()` in Bezug auf die Array - Elemente, oder wandeln sie in die `List` Instanzen und die Listen zu speichern.

Arrays zum Streamen

Java SE 8

Konvertieren eines Arrays von Objekten in `Stream` :

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

`Arrays.stream()` ein Array von Primitiven mithilfe von Arrays in `Stream` `Arrays.stream()` wird das Array in eine primitive Spezialisierung von Stream umgewandelt:

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

Sie können den `Stream` auf eine Reihe von Elementen im Array beschränken. Der Startindex ist inklusive und der Endindex ist exklusiv:

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

Eine ähnliche Methode wie `Arrays.stream()` in der `Stream` Klasse `Stream.of()` : `Stream.of()` . Der Unterschied besteht darin, dass `Stream.of()` einen varargs-Parameter verwendet, sodass Sie Folgendes schreiben können:

```
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});
```

Iteration über Arrays

Sie können über Arrays iterieren, indem Sie entweder for for (erweitert) oder Array-Indizes verwenden:

```
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);
}
```

Es ist erwähnenswert, dass es keinen direkten Weg gibt, einen Iterator für ein Array zu verwenden, er kann jedoch durch die Arrays-Bibliothek leicht in eine Liste konvertiert werden, um ein `Iterable` Objekt zu erhalten.

Verwenden Sie für Box-Arrays Arrays.asList :

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

Verwenden Sie für primitive Arrays (mit Java 8) Streams (speziell in diesem Beispiel - Arrays.stream -> IntStream):

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

Wenn Sie keine Streams verwenden können (keine Java 8), können Sie die Guava- Bibliothek von Google verwenden:

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

In zweidimensionalen Arrays oder mehr können beide Techniken auf etwas komplexere Weise verwendet werden.

Beispiel:

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

for (int indexOuter = 0; indexOuter < array.length; indexOuter++) {
    for (int indexInner = 0; indexInner < array[indexOuter].length; indexInner++) {
        array[indexOuter][indexInner] = indexOuter + indexInner;
    }
}
```

Java SE 5

```
for (int[] numbers : array) {
    for (int value : numbers) {
        System.out.println(value);
    }
}
```

Es ist nicht möglich, ein Array auf einen nicht einheitlichen Wert zu setzen, ohne eine indexbasierte Schleife zu verwenden.

Natürlich können Sie auch `while` oder `do-while` Schleifen verwenden, wenn Sie mit Indizes iterieren.

Ein Hinweis zur Vorsicht: `array.length - 1` bei der Verwendung von `array.length - 1` sicher, dass der Index zwischen `0` und `array.length - 1` (beide inklusive) liegt. Nehmen Sie keine fest codierten Annahmen für die Arraylänge vor. Andernfalls könnte Ihr Code beschädigt werden, wenn sich die Arraylänge ändert, Ihre hartcodierten Werte jedoch nicht.

Beispiel:

```
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;
    }
}
```

Es ist auch am besten, wenn Sie keine ausgefallenen Berechnungen verwenden, um den Index abzurufen, sondern den Index zur Iteration verwenden und wenn Sie andere Werte benötigen, diese berechnen.

Beispiel:

```
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;
    }
}
```

Zugreifen auf Arrays in umgekehrter Reihenfolge

```
int[] array = {0, 1, 1, 2, 3, 5, 8, 13};
for (int i = array.length - 1; i >= 0; i--) {
   System.out.println(array[i]);
}
```

Verwenden temporärer Arrays, um die Codewiederholung zu reduzieren

Wenn Sie ein temporäres Array durchlaufen, anstatt Code zu wiederholen, kann der Code sauberer werden. Es kann verwendet werden, wenn dieselbe Operation für mehrere Variablen ausgeführt wird.

```
// 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));
```

Beachten Sie, dass dieser Code nicht in leistungskritischen Abschnitten verwendet werden sollte, da bei jeder Eingabe der Schleife ein Array erstellt wird und dass primitive Variablen in das Array kopiert werden und daher nicht geändert werden können.

Arrays kopieren

Java bietet mehrere Möglichkeiten, ein Array zu kopieren.

für Schleife

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

Beachten Sie, dass bei Verwendung dieser Option mit einem Object-Array anstelle eines primitiven Arrays die Kopie mit einem Verweis auf den ursprünglichen Inhalt anstatt mit einer Kopie davon gefüllt wird.

Object.clone ()

Da Arrays `Object` s in Java, können Sie `Object.clone()` .

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

Beachten Sie, dass die `Object.clone` Methode für ein Array eine **flache Kopie** ausführt, dh einen Verweis auf ein neues Array zurückgibt, das auf **dieselben** Elemente verweist wie das Quellarray.

Arrays.copyOf ()

`java.util.Arrays` einfache Weise die Kopie eines Arrays auf ein anderes kopieren. Hier ist die grundlegende Verwendung:

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

Beachten Sie, dass `Arrays.copyOf` auch eine Überladung bereitstellt, mit der Sie den Typ des Arrays ändern können:

```
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)` Kopiert ein Array aus dem angegebenen `public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)` , beginnend an der angegebenen Position, an die angegebene Position des Zielarrays.

Unten ein Anwendungsbeispiel

```
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 ()

Hauptsächlich zum Kopieren eines Teils eines Arrays verwendet, können Sie es auch verwenden, um das gesamte Array in ein anderes zu kopieren:

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

Casting-Arrays

Arrays sind Objekte, ihr Typ wird jedoch durch den Typ der enthaltenen Objekte definiert. Daher kann man `A[]` nicht einfach in `T[]` , sondern jedes A-Mitglied des spezifischen `A[]` muss in ein `T` Objekt umgewandelt werden. Allgemeines Beispiel:

```
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;
}
```

Daher ein `A[]` -Array gegeben:

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

Java SE stellt für diesen Zweck die Methode `Arrays.copyOf(original, newLength, newType)` zur Verfügung:

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

Entfernen Sie ein Element aus einem Array

Java bietet keine direkte Methode in `java.util.Arrays` , um ein Element aus einem Array zu entfernen. Um dies durchzuführen, können Sie entweder das ursprüngliche Array in ein neues kopieren, ohne das Element zu entfernen, oder das Array in eine andere Struktur konvertieren, die das Entfernen zulässt.

ArrayList verwenden

Sie können das Array in eine `java.util.List` konvertieren, das Element entfernen und die Liste wie folgt in ein Array zurückwandeln:

```
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]
```

System.arraycopy verwenden

`System.arraycopy()` kann verwendet werden, um eine Kopie des ursprünglichen Arrays zu erstellen und das gewünschte Element zu entfernen. Unten ein Beispiel:

```
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]
```

Apache Commons verwenden Lang

Um ein Element leicht zu entfernen, können Sie die Apache Commons Lang- Bibliothek und insbesondere die statische Methode `removeElement()` der Klasse `ArrayUtils` . Unten ein Beispiel:

```
int[] array = new int[]{1,2,3,4};
array = ArrayUtils.removeElement(array, 2); //remove first occurrence of 2
System.out.println(Arrays.toString(array)); //[1, 3, 4]
```

Array-Kovarianz

Objektarrays sind kovariant, was bedeutet, dass `Integer` eine Unterklasse von `Number` und `Integer[]` eine Unterklasse von `Number[]` . Dies kann intuitiv erscheinen, kann aber zu überraschendem Verhalten führen:

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

Obwohl `Integer[]` eine Unterklasse von `Number[]` , kann es nur `Integer` Werte enthalten, und der Versuch, ein `Long` Element zuzuweisen, löst eine Laufzeitausnahme aus.

Beachten Sie, dass dieses Verhalten nur für Arrays gilt und stattdessen eine generische `List` verwendet werden kann:

```
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
```

Es ist nicht notwendig, dass alle Array-Elemente denselben Typ verwenden, sofern sie eine

Unterklasse des Array-Typs sind:

```
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.
```

Wie ändern Sie die Größe eines Arrays?

Die einfache Antwort ist, dass Sie das nicht tun können. Nachdem ein Array erstellt wurde, kann seine Größe nicht mehr geändert werden. Stattdessen kann ein Array nur "verkleinert" werden, indem ein neues Array mit der entsprechenden Größe erstellt und die Elemente aus dem vorhandenen Array in das neue Array kopiert werden.

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

Nehmen Sie beispielsweise an, dass ein neues Element zu dem wie oben definierten `listOfCities` Array hinzugefügt werden `listOfCities` . Dazu müssen Sie:

1. ein neues Array mit Größe 4 erstellen,
2. Kopieren Sie die vorhandenen 3 Elemente des alten Arrays in das neue Array bei den Offsets 0, 1 und 2 und
3. Fügen Sie das neue Element im Offset 3 dem neuen Array hinzu.

Es gibt verschiedene Möglichkeiten, dies zu tun. Vor Java 6 war der prägnanteste Weg:

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

Ab Java 6 können die Methoden `Arrays.copyOf` und `Arrays.copyOfRange` dies einfacher tun:

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

```
newArray[listOfCities.length] = "Sydney";
```

Weitere Möglichkeiten zum Kopieren eines Arrays finden Sie im folgenden Beispiel. Beachten Sie, dass Sie bei der Größenänderung eine Array-Kopie mit einer anderen Länge als das Original benötigen.

- Arrays kopieren

Eine bessere Alternative zur Größenänderung von Arrays

Es gibt zwei Hauptnachteile bei der Größenänderung eines Arrays wie oben beschrieben:

- Es ist ineffizient. Um ein Array größer (oder kleiner) zu machen, müssen Sie viele oder alle vorhandenen Array-Elemente kopieren und ein neues Array-Objekt zuordnen. Je größer das Array, desto teurer wird es.
- Sie müssen in der Lage sein, alle "Live" -Variablen zu aktualisieren, die Verweise auf das alte Array enthalten.

Eine Alternative besteht darin, das Array mit einer ausreichend großen Größe zu erstellen, um damit zu beginnen. Dies ist nur möglich, wenn Sie diese Größe genau bestimmen können, *bevor Sie das Array zuweisen* . Wenn Sie dies nicht tun können, tritt das Problem der Größenänderung des Arrays erneut auf.

Die andere Alternative ist die Verwendung einer Datenstrukturklasse, die von der Java SE-Klassenbibliothek oder einer Bibliothek eines Drittanbieters bereitgestellt wird. Beispielsweise bietet das Java SE-Framework "Collections" eine Reihe von Implementierungen der `List` , `Set` und `Map` APIs mit unterschiedlichen Laufzeiteigenschaften. Die `ArrayList` Klasse ist den Leistungsmerkmalen eines einfachen Arrays am nächsten (z. B. O (N) -Suchen, O (1) Holen und Festlegen, O (N) zufälliges Einfügen und Löschen), bietet jedoch eine effizientere Größenänderung ohne das Problem der Referenzaktualisierung.

(Die Größenänderungseffizienz für `ArrayList` auf der Strategie, die Größe des Backing-Arrays bei jeder Größenänderung zu verdoppeln. Für einen typischen Anwendungsfall bedeutet dies, dass Sie die Größe nur gelegentlich ändern. Wenn Sie sich über die Lebensdauer der Liste amortisieren, kostet die Größenänderung Kosten Jede Einfügung ist `O(1)` . Bei der Größenänderung eines einfachen Arrays kann möglicherweise dieselbe Strategie verwendet werden.)

Ein Element in einem Array finden

Es gibt viele Möglichkeiten, die Position eines Werts in einem Array zu ermitteln. In den folgenden Beispielausschnitten wird davon ausgegangen, dass das Array eines der folgenden ist:

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

Außerdem setzt jeder `index` oder `index2` entweder auf den Index des erforderlichen Elements oder

auf `-1` wenn das Element nicht vorhanden ist.

`Arrays.binarySearch` (nur für sortierte Arrays)

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

Verwenden einer `Arrays.asList` (nur für nicht primitive Arrays)

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

Verwenden eines `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
```

Lineare Suche über eine Schleife

```
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
```

Lineare Suche mithilfe von Drittanbieter-Bibliotheken wie z. B. org.apache.commons

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

Hinweis: Die Verwendung einer direkten linearen Suche ist effizienter als das Umbrechen einer Liste.

Testen, ob ein Array ein Element enthält

Die obigen Beispiele können angepasst werden, um zu testen, ob das Array ein Element enthält,

indem einfach getestet wird, ob der berechnete Index größer oder gleich Null ist.

Alternativ gibt es auch einige prägnantere Varianten:

```
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);
```

Arrays sortieren

Mit dem Arrays- API können Arrays einfach sortiert werden.

```
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));
```

String-Arrays sortieren:

`String` ist kein numerisches Datum, sondern definiert seine eigene Reihenfolge, die als lexikographische Reihenfolge oder auch alphabetische Reihenfolge bezeichnet wird. Wenn Sie ein Array von String mithilfe der `sort()` -Methode `sort()` , wird das Array in die natürliche Reihenfolge sortiert, die von der Comparable-Schnittstelle definiert wird (siehe unten):

Zunehmende Reihenfolge

```
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));
```

Ausgabe:

```
String array before sorting : [John, Steve, Shane, Adam, Ben]
```

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

Absteigende Reihenfolge

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

Ausgabe:

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

Objektarray sortieren

Um ein Objektarray zu sortieren, müssen alle Elemente entweder die `Comparable` oder `Comparator` Schnittstelle implementieren, um die Sortierreihenfolge zu definieren.

Wir können entweder `sort(Object[])` Methode ein Objekt - Array auf seiner natürlichen Reihenfolge zu sortieren, aber Sie müssen sicherstellen , dass alle Elemente im Array implementieren müssen `Comparable` .

Darüber hinaus müssen sie auch miteinander vergleichbar sein, z. B. `e1.compareTo(e2)` keine *`ClassCastException`* für die Elemente e1 und e2 im Array *`ClassCastException`* . Alternativ können Sie ein Object-Array nach der benutzerdefinierten Reihenfolge `sort(T[], Comparator)` , indem Sie die `sort(T[], Comparator)` Methode `sort(T[], Comparator)` verwenden, wie im folgenden Beispiel gezeigt.

```
// 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));
```

Ausgabe:

```
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 ]
```

Konvertieren von Arrays zwischen Grundelementen und geschachtelten Typen

Manchmal ist eine Konvertierung von primitiven Typen in geschachtelte Typen erforderlich.

Zur Konvertierung des Arrays können Streams verwendet werden (in Java 8 und höher):

Java SE 8

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

Bei niedrigeren Versionen kann das primitive Array durchlaufen und explizit in das geschachtelte Array kopiert werden:

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
}
```

Ebenso kann ein Boxed Array in ein Array seines primitiven Gegenstücks konvertiert werden:

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
}
```

Arrays online lesen: https://riptutorial.com/de/java/topic/99/arrays

Kapitel 9: Atomtypen

Einführung

Java Atomic Types sind einfache veränderliche Typen, die grundlegende Vorgänge ermöglichen, die Thread-sicher und atomar sind, ohne auf Sperren zurückzugreifen. Sie sind für den Einsatz in Fällen vorgesehen, in denen das Sperren ein Engpass bei gleichzeitiger Verwendung wäre oder die Gefahr eines Deadlocks oder Livelock besteht.

Parameter

Parameter	Beschreibung
einstellen	Flüchtiger Satz des Feldes
erhalten	Flüchtiges Lesen des Feldes
LazySet	Dies ist eine vom Geschäft geordnete Operation des Feldes
compareAndSet	Wenn der Wert der Expeed-Wert ist, wird er an den neuen Wert gesendet
getAndSet	Holen Sie sich den aktuellen Wert und aktualisieren Sie

Bemerkungen

Viele im Wesentlichen Kombinationen flüchtiger Lese- oder Schreibvorgänge und CAS-Operationen. Der beste Weg, dies zu verstehen, ist der direkte Blick auf den Quellcode. ZB AtomicInteger , Unsafe.getAndSet

Examples

Atomtypen erstellen

Bei einfachem Multithreadcode ist die Verwendung der Synchronisierung akzeptabel. Die Verwendung der Synchronisierung hat jedoch Auswirkungen auf die Lebendigkeit. Wenn eine Codebase komplexer wird, steigt die Wahrscheinlichkeit, dass Sie mit Deadlock , Starvation oder Livelock enden.

In Fällen komplexer Parallelität ist die Verwendung von Atomic Variables oft die bessere Alternative, da auf eine einzelne Variable Thread-sicher zugegriffen werden kann, ohne synchronisierte Methoden oder Codeblöcke verwenden zu müssen.

Einen `AtomicInteger` Typ `AtomicInteger` :

```
AtomicInteger aInt = new AtomicInteger() // Create with default value 0

AtomicInteger aInt = new AtomicInteger(1) // Create with initial value 1
```

Ähnlich für andere Instanztypen.

```
AtomicIntegerArray aIntArray = new AtomicIntegerArray(10) // Create array of specific length
AtomicIntegerArray aIntArray = new AtomicIntegerArray(new int[] {1, 2, 3}) // Initialize array
with another array
```

Ähnlich für andere Atomtypen.

Es gibt eine bemerkenswerte Ausnahme, dass es keine `float` und `double` gibt. Diese können durch Verwendung von `Float.floatToIntBits(float)` und `Float.intBitsToFloat(int)` für `float` sowie `Double.doubleToLongBits(double)` und `Double.longBitsToDouble(long)` für Doppel `Double.longBitsToDouble(long)` .

Wenn Sie `sun.misc.Unsafe` verwenden `sun.misc.Unsafe` , können Sie jede primitive Variable als atomar verwenden, indem Sie die atomare Operation in `sun.misc.Unsafe` . Alle primitiven Typen sollten in int oder long konvertiert oder codiert werden, um sie auf diese Weise zu verwenden. Weitere Informationen hierzu finden Sie unter: sun.misc.Unsafe .

Motivation für Atomtypen

Der einfachste Weg, Multithread-Anwendungen zu implementieren, ist die Verwendung der integrierten Synchronisations- und Sperr-Grundelemente von Java. zB das `synchronized` Schlüsselwort. Das folgende Beispiel zeigt, wie wir `synchronized` , um Zählwerte zu sammeln.

```
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;
    }
}
```

Diese Implementierung wird korrekt funktionieren. Wenn jedoch eine große Anzahl von Threads viele gleichzeitige Aufrufe für dasselbe `Counters` Objekt durchführt, kann die Synchronisierung einen Engpass darstellen. Speziell:

1. Jeder `synchronized` Methodenaufruf beginnt mit dem aktuellen Thread, der die Sperre für die `Counters` Instanz erhält.
2. Der Thread hält die Sperre, während er den `number` überprüft und den Zähler aktualisiert.
3. Schließlich gibt er die Sperre frei und ermöglicht anderen Threads den Zugriff.

Wenn ein Thread versucht, die Sperre abzurufen, während ein anderer die Sperre aufrechterhält, wird der versuchte Thread in Schritt 1 blockiert (angehalten), bis die Sperre freigegeben wird. Wenn mehrere Threads warten, wird einer von ihnen darauf zugreifen und die anderen werden weiterhin blockiert.

Dies kann zu einigen Problemen führen:

- Wenn es viele *Konflikte* um die Sperre gibt (dh viele Threads versuchen, sie zu erhalten), können einige Threads für lange Zeit blockiert werden.
- Wenn ein Thread blockiert ist und auf die Sperre wartet, versucht das Betriebssystem normalerweise, die Ausführung auf einen anderen Thread umzustellen. Diese *Kontextumschaltung* verursacht eine relativ große Auswirkung auf die Leistung auf den Prozessor.
- Wenn mehrere Threads für dieselbe Sperre blockiert sind, kann nicht garantiert werden, dass einer von ihnen "fair" behandelt wird (dh, dass jeder Thread garantiert zur Ausführung geplant ist). Dies kann zu einem *Fadenknappheit führen* .

Wie implementiert man Atomtypen?

Beginnen wir mit dem `AtomicInteger` des obigen Beispiels mit `AtomicInteger` Zählern:

```
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;
    }
}
```

Wir haben das `int[]` durch ein `AtomicInteger[]` und es mit einer Instanz in jedem Element initialisiert. Wir haben auch Aufrufe von `incrementAndGet()` und `get()` anstelle von Operationen für `int` Werte hinzugefügt.

Das Wichtigste ist jedoch, dass wir das `synchronized` Schlüsselwort entfernen können, da das Sperren nicht mehr erforderlich ist. Dies funktioniert, weil die Operationen `incrementAndGet()` und `get()` *atomar* und *threadsicher sind* . In diesem Zusammenhang bedeutet dies:

- Jeder Zähler in dem Feld wird nur in entweder den „vor dem “ Zustand für eine Operation (wie ein „Inkrement“) oder in dem „nach“ Zustand *beobachtbar* sein.
- Unter der Annahme, dass die Operation zum Zeitpunkt `T` auftritt, kann kein Thread den Zustand "vor" nach dem Zeitpunkt `T` .

Während zwei Threads tatsächlich versuchen können, dieselbe `AtomicInteger` Instanz gleichzeitig zu aktualisieren, stellen die Implementierungen der Vorgänge außerdem sicher, dass jeweils nur ein Inkrement für die angegebene Instanz erfolgt. Dies erfolgt ohne Verriegelung, was häufig zu einer besseren Leistung führt.

Wie funktionieren Atomtypen?

Atomtypen basieren normalerweise auf speziellen Hardwareanweisungen im Befehlssatz des Zielcomputers. Beispielsweise bieten Intel-basierte Befehlssätze einen `CAS` -Befehl (Compare and Swap), der eine bestimmte Folge von Speicheroperationen atomar ausführt.

Diese Anweisungen auf niedriger Ebene werden verwendet, um Vorgänge höherer Ebene in den APIs der jeweiligen `AtomicXxx` Klassen zu `AtomicXxx` . Zum Beispiel (wieder in C-artigem Pseudocode):

```
private volatile num;

int increment() {
  while (TRUE) {
    int old = num;
    int new = old + 1;
    if (old == compare_and_swap(&num, old, new)) {
      return new;
    }
  }
}
```

Wenn auf dem `AtomicXxxx` kein Konflikt `AtomicXxxx` , ist der `if` Test erfolgreich, und die Schleife

endet sofort. Wenn es Konflikte gibt, wird das `if` für alle außer einem der Threads fehlschlagen, und sie werden sich in der Schleife für eine kleine Anzahl von Zyklen der Schleife "drehen". In der Praxis ist das Drehen um Größenordnungen schneller (außer bei *unrealistisch hohen* Konflikten, bei denen synchronisierte besser als Atomklassen abschneidet, da bei einem CAS-Vorgang der Versuch nur mehr Konflikte hinzufügt), als den Thread anzuhalten und zu einem anderen zu wechseln ein.

Im Übrigen werden CAS-Anweisungen normalerweise von der JVM verwendet, um ein *unkontrolliertes Sperren* zu implementieren. Wenn die JVM erkennt, dass eine Sperre derzeit nicht gesperrt ist, versucht sie, ein CAS zum Abrufen der Sperre zu verwenden. Wenn der CAS erfolgreich ist, müssen keine teuren Thread-Zeitpläne, Kontextwechsel usw. durchgeführt werden. Weitere Informationen zu den verwendeten Techniken finden Sie unter Verzerrte Sperren in HotSpot .

Atomtypen online lesen: https://riptutorial.com/de/java/topic/5963/atomtypen

Kapitel 10: Audio

Bemerkungen

Anstelle des `Clip` javax.sound.sampled können Sie auch den `AudioClip` der Applet-API verwenden. Es wird jedoch empfohlen, `Clip` zu verwenden, da `AudioClip` nur älter ist und nur eingeschränkte Funktionen bietet.

Examples

Spielen Sie eine Audiodatei Looped

Notwendige Importe:

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

Dieser Code erstellt einen Clip und spielt ihn nach dem Start fortlaufend ab:

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

Holen Sie sich ein Array mit allen unterstützten Dateitypen:

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

Spielen Sie eine MIDI-Datei

MIDI-Dateien können mit mehreren Klassen aus dem Paket `javax.sound.midi` werden. Ein `Sequencer` führt die Wiedergabe der MIDI-Datei aus, und viele seiner Methoden können verwendet werden, um Wiedergabesteuerelemente wie Loop-Count, Tempo, Spur-Muting und andere einzustellen.

Die allgemeine Wiedergabe von MIDI-Daten kann auf folgende Weise erfolgen:

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

Um die Wiedergabe zu stoppen, verwenden Sie:

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

Ein Sequenzer kann so eingestellt werden, dass eine oder mehrere Spuren der Sequenz während der Wiedergabe stummgeschaltet werden, sodass keines der Instrumente der angegebenen Musik wiedergegeben wird. Im folgenden Beispiel wird der erste Track in der Sequenz stummgeschaltet:

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

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

Ein Sequenzer kann eine Sequenz wiederholt abspielen, wenn die Anzahl der Schleifen angegeben ist. Im Folgenden wird der Sequenzer so eingestellt, dass eine Sequenz viermal und unbegrenzt abgespielt wird:

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

Der Sequenzer muss die Sequenz nicht immer von Anfang an spielen, noch muss er die Sequenz bis zum Ende spielen. Sie kann an einem beliebigen Punkt beginnen und enden, indem Sie das *Häkchen* in der Sequenz *festlegen,* mit der begonnen und beendet wird. Es ist auch möglich, manuell festzulegen, von welchem Tick in der Sequenz der Sequenzer abgespielt werden soll:

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

Sequenzer können auch eine MIDI-Datei in einem bestimmten Tempo abspielen, das durch Angabe des Tempos in Beats pro Minute (BPM) oder Mikrosekunden pro Viertelnote (MPQ) gesteuert werden kann. Der Faktor, bei dem die Sequenz gespielt wird, kann ebenfalls eingestellt werden.

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

Wenn Sie mit dem `Sequencer` fertig sind, schließen Sie ihn bitte

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

Bare-Metal-Sound

Wenn Sie mit Java Sound erzeugen, können Sie auch fast nackt sein. Dieser Code schreibt rohe Binärdaten in den OS-Audiopuffer, um Ton zu erzeugen. Es ist äußerst wichtig, die Einschränkungen und notwendigen Berechnungen zur Erzeugung eines solchen Sounds zu verstehen. Da die Wiedergabe grundsätzlich sofort erfolgt, müssen Berechnungen nahezu in Echtzeit durchgeführt werden.

Daher ist dieses Verfahren für eine kompliziertere Tonabtastung unbrauchbar. Für solche Zwecke ist die Verwendung von Spezialwerkzeugen der bessere Ansatz.

Das folgende Verfahren erzeugt eine Rechteckwelle einer gegebenen Frequenz in einem gegebenen Volumen für eine gegebene Dauer und gibt diese direkt aus.

```
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
}
```

Für eine differenziertere Art, verschiedene Schallwellen zu erzeugen, sind Sinusberechnungen und möglicherweise größere Stichprobengrößen erforderlich. Dies führt zu wesentlich

komplexerem Code und wird dementsprechend hier weggelassen.

Grundlegende Audioausgabe

Das Hallo Audio! von Java, das eine Audiodatei aus lokalem oder Internetspeicher abspielt, sieht wie folgt aus. Es funktioniert für unkomprimierte WAV-Dateien und sollte nicht für die Wiedergabe von MP3- oder komprimierten Dateien verwendet werden.

```
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();
   }
}
```

Audio online lesen: https://riptutorial.com/de/java/topic/160/audio

Kapitel 11: Aufteilen einer Schnur in Teile mit fester Länge

Bemerkungen

Das Ziel hier ist, keinen Inhalt zu verlieren, so dass der Regex keine Eingaben konsumieren muss. Vielmehr muss es *zwischen* dem letzten Zeichen der vorherigen Zieleingabe und dem ersten Zeichen der nächsten Zieleingabe übereinstimmen. Für 8-stellige Teilzeichenfolgen müssen wir die Eingabe an den unten markierten Stellen aufteilen (dh übereinstimmen):

```
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
              ^               ^               ^
```

Ignorieren Sie die Leerzeichen in der Eingabe, die *zwischen den* Zeichenpositionen angezeigt werden sollen.

Examples

Zerlegen Sie einen String in alle bekannten Strings

Der Trick ist, einen Rückblick mit dem Regex `\G` , was "Ende des vorherigen Matches" bedeutet:

```
String[] parts = str.split("(?<=\\G.{8})");
```

Der Regex entspricht 8 Zeichen nach dem Ende des letzten Matches. Da in diesem Fall die Übereinstimmung null ist, können wir einfacher "8 Zeichen nach der letzten Übereinstimmung" sagen.

Praktischerweise wird `\G` mit dem Start der Eingabe initialisiert, so dass es auch für den ersten Teil der Eingabe funktioniert.

Unterteilen Sie einen String in Teilstrings mit variabler Länge

Wie das bekannte Längenbeispiel, aber fügen Sie die Länge in Regex ein:

```
int length = 5;
String[] parts = str.split("(?<=\\G.{" + length + "})");
```

Aufteilen einer Schnur in Teile mit fester Länge online lesen:
https://riptutorial.com/de/java/topic/5613/aufteilen-einer-schnur-in-teile-mit-fester-lange

Kapitel 12: Aufzählung beginnend mit der Nummer

Einführung

Java erlaubt nicht, dass der Name der Enumeration mit einer Zahl wie 100A, 25K beginnt. In diesem Fall können wir den Code mit _ (Unterstrich) oder einem beliebigen zulässigen Muster anhängen und prüfen.

Examples

Aufzählung mit Namen am Anfang

```
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);
    }
}
```

Aufzählung beginnend mit der Nummer online lesen:
https://riptutorial.com/de/java/topic/10719/aufzahlung-beginnend-mit-der-nummer

Kapitel 13: Aufzählungen

Einführung

Java-Enums (deklariert mit dem Schlüsselwort `enum`) sind eine Abkürzungssyntax für beträchtliche Mengen von Konstanten einer einzelnen Klasse.

Syntax

- [public / protected / private] enum Enum_name {// Neues Enum deklarieren.
- ENUM_CONSTANT_1 [, ENUM_CONSTANT_2 ...]; // Deklariere die Enumenkonstanten. Dies muss die erste Zeile innerhalb der Enumeration sein und sollte durch Kommas getrennt werden, am Ende ein Semikolon.
- ENUM_CONSTANT_1 (param) [, ENUM_CONSTANT_2 (param) ...]; // Deklarieren Sie Enumerationskonstanten mit Parametern. Die Parametertypen müssen zum Konstruktor passen.
- ENUM_CONSTANT_1 {...} [, ENUM_CONSTANT_2 {...} ...]; // Deklarieren Sie Enumenkonstanten mit überschriebenen Methoden. Dies muss geschehen, wenn die Aufzählung abstrakte Methoden enthält. Alle diese Methoden müssen implementiert werden.
- ENUM_CONSTANT.name () // Gibt einen String mit dem Namen der Enumenkonstante zurück.
- ENUM_CONSTANT.ordinal () // Gibt die Ordnungszahl dieser Enumerationskonstante zurück, ihre Position in der Enumendeklaration, wobei der Anfangskonstante eine Ordnungszahl von Null zugewiesen wird.
- Enum_name.values () // Gibt ein neues Array (vom Typ Enum_name []) zurück, das bei jedem Aufruf jede Konstante dieser Enumeration enthält.
- Enum_name.valueOf ("ENUM_CONSTANT") // Die Umkehrung von ENUM_CONSTANT.name () - gibt die Enumenkonstante mit dem angegebenen Namen zurück.
- Enum.valueOf (Enum_name.class, "ENUM_CONSTANT") // Ein Synonym des vorherigen: Die Umkehrung von ENUM_CONSTANT.name () - gibt die Enumenkonstante mit dem angegebenen Namen zurück.

Bemerkungen

Beschränkungen

Enumerationen erweitern immer `java.lang.Enum` , daher ist es für eine Enumeration nicht möglich, eine Klasse zu erweitern. Sie können jedoch viele Schnittstellen implementieren.

Tipps

Aufgrund ihrer speziellen Darstellung gibt es effizientere Karten und Sets , die mit Enumerationen

als Schlüssel verwendet werden können. Diese laufen oft schneller als ihre nicht spezialisierten Kollegen.

Examples

Deklarieren und Verwenden einer grundlegenden Aufzählung

Enum kann als Syntaxzucker für eine versiegelte Klasse betrachtet werden, die nur zum Zeitpunkt der Kompilierung bekannt ist, um eine Menge von Konstanten zu definieren.

Eine einfache Aufzählung der verschiedenen Jahreszeiten würde wie folgt erklärt:

```
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

Während die Enumerationskonstanten nicht unbedingt in Großbuchstaben angegeben sein müssen, ist es die Java-Konvention, dass Konstantennamen vollständig aus Großbuchstaben bestehen und die Wörter durch Unterstriche getrennt sind.

Sie können ein Enum in einer eigenen Datei deklarieren:

```
/**
 * This enum is declared in the Season.java file.
*/
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    FALL
}
```

Sie können es aber auch in einer anderen Klasse deklarieren:

```
 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
    }

}
```

Schließlich können Sie kein Enum innerhalb eines Methodentexts oder Konstruktors deklarieren:

```
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;
    }

}
```

Doppelte Enumerationskonstanten sind nicht zulässig:

```
public enum Season {
    WINTER,
    WINTER, //Compile Time Error : Duplicate Constants
    SPRING,
    SUMMER,
    FALL
}
```

Jede Konstante von enum ist standardmäßig **`public`**, **`static`** und **`final`**. Da jede Konstante `static`, kann auf sie direkt über den Aufzählungsnamen zugegriffen werden.

Enumenkonstanten können als Methodenparameter übergeben werden:

```
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"
```

Sie können ein Array der Enumenkonstanten mit der Methode `values()` abrufen. Die Werte befinden sich garantiert in der Reihenfolge der Deklaration im zurückgegebenen Array:

```
Season[] seasons = Season.values();
```

Hinweis: Diese Methode ordnet bei jedem Aufruf ein neues Array von Werten zu.

Um die Enumenkonstanten zu durchlaufen:

```
public static void enumIterate() {
    for (Season s : Season.values()) {
        System.out.println(s.name());
    }
}
```

Sie können enums in einer `switch` Anweisung verwenden:

```
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;
    }
}
```

Sie können Enumerationskonstanten auch mit `==` :

```
Season.FALL == Season.WINTER    // false
Season.SPRING == Season.SPRING  // true
```

Eine andere Möglichkeit, die Enumenkonstanten zu vergleichen, ist die Verwendung von `equals()` wie unten, was als schlechte Praxis betrachtet wird, da Sie wie folgt leicht in Fallstricke fallen können:

```
Season.FALL.equals(Season.FALL); // true
Season.FALL.equals(Season.WINTER); // false
Season.FALL.equals("FALL"); // false and no compiler error
```

Obwohl die Menge der Instanzen in der `enum` nicht zur Laufzeit geändert werden kann, sind die Instanzen selbst nicht inhärent unveränderlich, da eine `enum` wie jede andere Klasse veränderliche Felder enthalten kann, wie unten gezeigt wird.

```
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
```

Es `enum` sich jedoch, `enum` Instanzen unveränderlich zu machen, dh, wenn sie entweder keine zusätzlichen Felder haben oder alle diese Felder als `final` und selbst unveränderlich sind. Dadurch wird sichergestellt, dass ein `enum` über die gesamte Lebensdauer der Anwendung keinen Speicher `enum` und dass die Instanzen sicher für alle Threads verwendet werden können.

Aufzählungen implementieren implizit `Serializable` und `Comparable` , weil die `Enum` - Klasse tut:

```
public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable
```

Enums mit Konstruktoren

Eine `enum` kann keinen öffentlichen Konstruktor haben. Private Konstruktoren sind jedoch zulässig (Konstruktoren für Enumerationen sind standardmäßig private Pakete):

```
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
```

Es wird empfohlen, alle Felder privat zu halten und Getter-Methoden bereitzustellen, da es eine begrenzte Anzahl von Instanzen für eine Aufzählung gibt.

Wenn Sie stattdessen ein `Enum` als `class` implementieren würden, würde es so aussehen:

```
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
```

Enumerationskonstanten sind technisch veränderbar, daher könnte ein Setter hinzugefügt werden, um die interne Struktur einer Enumerationskonstante zu ändern. Dies wird jedoch als sehr schlechte Praxis angesehen und sollte vermieden werden.

Es empfiehlt sich, Enum-Felder unveränderlich zu machen, mit `final` :

```
public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value){
        this.value = value;
    }

    ...

}
```

Sie können mehrere Konstruktoren in derselben Enumeration definieren. Wenn Sie dies tun, entscheiden die Argumente, die Sie in Ihrer Enumendeklaration übergeben, welcher Konstruktor

aufgerufen wird:

```
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;
    }

    ...

}
```

Hinweis: Alle nicht primitiven Enum-Felder sollten `Serializable` implementieren, da die `Enum` Klasse dies tut.

Verwendung von Methoden und statischen Blöcken

Eine Enumeration kann wie jede Klasse eine Methode enthalten. Um zu sehen, wie das funktioniert, erklären wir eine Enumeration wie folgt:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;
}
```

Wir haben eine Methode, die die Enumeration in die entgegengesetzte Richtung zurückgibt:

```
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;
        }
    }
}
```

Dies kann durch die Verwendung von Feldern und statischen Initialisierungsblöcken weiter

verbessert werden:

```
public enum Direction {
    NORTH, SOUTH, EAST, WEST;

    private Direction opposite;

    public Direction getOpposite(){
        return opposite;
    }

    static {
        NORTH.opposite = SOUTH;
        SOUTH.opposite = NORTH;
        WEST.opposite = EAST;
        EAST.opposite = WEST;
    }
}
```

In diesem Beispiel wird die entgegengesetzte Richtung in einem privaten Feld gespeichert Instanz `opposite` , der statisch das erste Mal , wenn eine Initialisierung `Direction` verwendet wird. In diesem speziellen Fall (weil `NORTH` auf `SOUTH` verweist und umgekehrt), können wir hier keine Enums mit Konstruktoren verwenden (Konstruktoren `NORTH(SOUTH), SOUTH(NORTH), EAST(WEST), WEST(EAST)` wären eleganter und würden das `opposite` zulassen deklariert wird `final` , würde aber selbstbezogen sein und daher nicht erlaubt).

Implementiert die Schnittstelle

Dies ist eine `enum` , die auch eine aufrufbare Funktion ist, die `String` Eingaben anhand vorkompilierter regulärer Ausdrucksmuster testet.

```
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"));
    }
}
```

Jedes Mitglied der Aufzählung kann auch die Methode implementieren:

```
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

Wenn ein Verfahren benötigen einen „dehnbar“ Satz zu akzeptieren `enum` - Werte kann der Programmierer Polymorphismus wie auf einer normalen Anwendung `class` durch eine Schnittstelle zu schaffen , die anywere verwendet werden , wo die `enum` s verwendet werden soll:

```
public interface ExtensibleEnum {
    String name();
}
```

Auf diese Weise kann jedes mit der Schnittstelle versehene `enum` als Parameter verwendet werden, wodurch der Programmierer eine variable Menge von `enum` erstellen kann, die von der Methode akzeptiert werden. Dies kann beispielsweise in APIs nützlich sein, in denen es eine standardmäßige (nicht veränderbare) `enum` und der Benutzer dieser APIs die `enum` mit weiteren Werten "erweitern" möchte.

Ein Satz von Standardaufzählungswerten kann wie folgt definiert werden:

```
public enum DefaultValues implements ExtensibleEnum {
    VALUE_ONE, VALUE_TWO;
}
```

Zusätzliche Werte können dann wie folgt definiert werden:

```
public enum ExtendedValues implements ExtensibleEnum {
    VALUE_THREE, VALUE_FOUR;
```

```
}
```

Ein Beispiel, das zeigt, wie die Aufzählungen verwendet werden - beachten Sie, wie `printEnum()` Werte von beiden `enum` akzeptiert:

```
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
```

Hinweis: Dieses Muster hindert Sie nicht daran, Aufzählungswerte, die bereits in einer Aufzählung definiert sind, in einer anderen Aufzählung neu zu definieren. Diese Aufzählungswerte wären dann unterschiedliche Instanzen. Es ist auch nicht möglich, die Aufschalt-Aufzählung zu verwenden, da wir nur die Schnittstelle haben, nicht die eigentliche `enum` .

Aufzählungen mit abstrakten Methoden

Aufzählungen können abstrakte Methoden definieren, die jedes `enum` Mitglied zu implementieren ist nicht erforderlich.

```
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);
}
```

Dies ermöglicht jedem Enumerationsmitglied, sein eigenes Verhalten für eine bestimmte Operation zu definieren, ohne Typen in einer Methode in der Top-Level-Definition aktivieren zu müssen.

Beachten Sie, dass dieses Muster eine Kurzform dessen ist, was normalerweise durch Polymorphie und / oder Implementierung von Schnittstellen erreicht wird.

Dokumentation von Aufzählungen

Nicht immer ist der Name der `enum` klar genug, um verstanden zu werden. Um eine `enum` zu dokumentieren, verwenden Sie Standard-Javadoc:

```
/**
 * 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;
    }
}
```

Die Werte einer Aufzählung erhalten

Jede Aufzählungsklasse enthält eine implizite statische Methode mit dem Namen `values()` . Diese Methode gibt ein Array zurück, das alle Werte dieser Enumeration enthält. Mit dieser Methode können Sie die Werte durchlaufen. Beachten Sie jedoch, dass diese Methode bei jedem Aufruf ein **neues** Array zurückgibt.

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

Wenn Sie ein `Set` benötigen, können Sie auch `EnumSet.allOf(Day.class)` .

Aufzählung als beschränkter Typparameter

Beim Schreiben einer Klasse mit Generics in Java kann sichergestellt werden, dass der Typparameter eine Enumeration ist. Da alle Enumerationen die `Enum` Klasse erweitern, kann die folgende Syntax verwendet werden.

```
public class Holder<T extends Enum<T>> {
    public final T value;

    public Holder(T init) {
        this.value = init;
    }
}
```

In diesem Beispiel ist der Typ `T` *muss* eine Enumeration sein.

Erhalte die Konstante nach Namen

Sagen wir, wir haben eine Aufzählung `DayOfWeek` :

```
enum DayOfWeek {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}
```

Eine Enumeration wird mit einer integrierten statischen `valueOf()` Methode erstellt, mit der eine Konstante anhand ihres Namens `valueOf()` werden kann:

```
String dayName = DayOfWeek.SUNDAY.name();
assert dayName.equals("SUNDAY");

DayOfWeek day = DayOfWeek.valueOf(dayName);
assert day == DayOfWeek.SUNDAY;
```

Dies ist auch mit einem dynamischen Aufzählungstyp möglich:

```
Class<DayOfWeek> enumType = DayOfWeek.class;
DayOfWeek day = Enum.valueOf(enumType, "SUNDAY");
assert day == DayOfWeek.SUNDAY;
```

Beide `valueOf()` -Methoden `IllegalArgumentException` eine `IllegalArgumentException` wenn das angegebene Enum keine Konstante mit einem übereinstimmenden Namen hat.

Die Guava-Bibliothek stellt eine `Enums.getIfPresent()` , die ein Guava- `Optional` zurückgibt, um die explizite Ausnahmebehandlung zu beseitigen:

```
DayOfWeek defaultDay = DayOfWeek.SUNDAY;
DayOfWeek day = Enums.valueOf(DayOfWeek.class, "INVALID").or(defaultDay);
assert day == DayOfWeek.SUNDAY;
```

Implementieren Sie das Singleton-Muster mit einer Einzelelement-Enumeration

Aufzählungskonstanten werden instanziiert, wenn zum ersten Mal auf eine Aufzählung verwiesen wird. Daher ist es möglich, das Singleton- Softwaredesignmuster mit einer Einzelelement-Enumeration zu implementieren.

```
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
    }
}
```

Laut "Effective Java" von Joshua Bloch ist eine Einzelelement-Enumeration der beste Weg, um ein Singleton zu implementieren. Dieser Ansatz hat folgende Vorteile:

- Fadensicherheit
- Garantie für einzelne Instanziierung
- Serienmäßige Serialisierung

Und wie in dem Abschnitt implements interface gezeigt , kann dieses Singleton auch eine oder mehrere Schnittstellen implementieren.

Aufzählung mit Eigenschaften (Feldern)

Für den Fall, dass wir `enum` mit mehr Informationen und nicht nur als konstanten Werten verwenden möchten, und wir möchten zwei enums vergleichen können.

Betrachten Sie das folgende Beispiel:

```
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;
    }

}
```

Hier haben wir ein `Enum` namens `Coin` das seinen Wert darstellt. Mit dem Verfahren `isGreaterThan` können wir zwei vergleichen `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
```

Konvertieren Sie Enum in String

Manchmal möchten Sie Ihr Enum in einen String konvertieren. Dafür gibt es zwei Möglichkeiten.

Angenommen, wir haben:

```
public enum Fruit {
    APPLE, ORANGE, STRAWBERRY, BANANA, LEMON, GRAPE_FRUIT;
}
```

Wie konvertieren wir so etwas wie `Fruit.APPLE` in `"APPLE"` ?

Konvertieren mit `name()`

`name()` ist eine interne Methode in `enum` , die die `String` Darstellung der Enumeration zurückgibt. Die zurückgegebene `String` repräsentiert ***genau*** wie der enum-Wert definiert wurde.

Zum Beispiel:

```
System.out.println(Fruit.BANANA.name());      // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.name()); // "GRAPE_FRUIT"
```

Konvertieren mit `toString()`

`toString()` wird *standardmäßig* überschrieben, um dasselbe Verhalten wie `name()`

`toString()` wird jedoch wahrscheinlich von *Entwicklern* überschrieben, um einen benutzerfreundlicheren `String` drucken

> Verwenden Sie `toString()` wenn Sie Ihren Code einchecken möchten. `name()` ist dafür wesentlich stabiler. Verwenden Sie `toString()` wenn Sie den Wert in Logs oder stdout oder etwas ausgeben möchten

Standardmäßig:

```
System.out.println(Fruit.BANANA.toString());      // "BANANA"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "GRAPE_FRUIT"
```

Beispiel dafür, überschrieben zu werden

```
System.out.println(Fruit.BANANA.toString());      // "Banana"
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "Grape Fruit"
```

Bestimmen Sie den spezifischen Körper

In einem `enum` ist es möglich , ein bestimmtes Verhalten für eine bestimmte Konstante des definieren `enum` , die das Standardverhalten des überschreibt `enum` wird diese Technik , bekannt als *konstanter spezifischen Körper.*

Angenommen, drei Klavierschüler - John, Ben und Luke - werden in einem `enum` namens `PianoClass` wie folgt definiert:

```
    enum PianoClass {
    JOHN, BEN, LUKE;
        public String getSex() {
            return "Male";
        }
        public String getLevel() {
            return "Beginner";
        }
    }
```

Und eines Tages kommen zwei andere Schüler - Rita und Tom - mit einem Geschlecht (weiblich) und einem Niveau (Mittelstufe) an, die nicht mit den vorherigen übereinstimmen:

```
    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
        }
    }
```

Daher ist es nicht korrekt, die neuen Schüler wie folgt der ständigen Deklaration hinzuzufügen:

```
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
```

Für jede der Konstanten Rita und Tom kann ein bestimmtes Verhalten definiert werden, das das Standardverhalten von `PianoClass2` wie folgt überschreibt:

```
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";
    }
}
```

und jetzt sind Toms Niveau und Rita Geschlecht so, wie sie sein sollten:

```
PianoClass3 tom = PianoClass3.TOM;
PianoClass3 rita = PianoClass3.RITA;
System.out.println(tom.getLevel()); // prints Intermediate
System.out.println(rita.getSex()); // prints Female
```

Eine andere Möglichkeit, einen inhaltsspezifischen Körper zu definieren, ist beispielsweise der Konstruktor:

```
enum Friend {
    MAT("Male"),
    JOHN("Male"),
    JANE("Female");

    private String gender;

    Friend(String gender) {
```

```
        this.gender = gender;
    }

    public String getGender() {
        return this.gender;
    }
}
```

und Verwendung:

```
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
```

Null-Instanz-Aufzählung

```
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...
}
```

Genauso wie `enum` für Singletons (1 Instanzklassen) verwendet werden kann, kann es für Dienstprogrammklassen (0 Instanzklassen) verwendet werden. Stellen Sie sicher, dass Sie die (leere) Liste der Enumenkonstanten mit einem `;` beenden `;` .

Siehe die Frage Zero-Instanz-Aufzählung vs. private Konstruktoren, um die Instantiierung für eine Diskussion über Pro und Con gegenüber privaten Konstruktoren zu verhindern.

Aufzählungen mit statischen Feldern

Wenn für Ihre Enum-Klasse statische Felder erforderlich sind, denken Sie daran, dass sie **nach** den Enum-Werten selbst erstellt werden. Das heißt, der folgende Code führt zu einer `NullPointerException` :

```
enum Example {
    ONE(1), TWO(2);

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

    private Example(int value) {
        integers.put(this.name(), value);
    }
}
```

Ein möglicher Weg, dies zu beheben:

```
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);
    }
}
```

Initialisieren Sie das statische Feld nicht:

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

Initialisierung:

- Erstellen Sie die Aufzählungswerte
    - Als Nebeneffekt wird putValue () aufgerufen, der Ganzzahlen initialisiert
- Die statischen Werte werden eingestellt
    - ganze Zahlen = null; // wird nach der Aufzählung ausgeführt, damit der Inhalt von Ganzzahlen verloren geht

Vergleichen und Enthält für Aufzählungswerte

Enums enthält nur Konstanten und kann direkt mit `==` verglichen werden. Es ist also nur eine Referenzprüfung erforderlich, keine `.equals` Methode zu verwenden. Darüber hinaus kann es bei `.equals` Verwendung von `.equals` zu einer `NullPointerException` während dies bei `==` check nicht der Fall ist.

```
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!");
        }

    }
}
```

Um die `EnumSet` zu gruppieren, zu ergänzen, reichen wir die `EnumSet` Klasse, die verschiedene Methoden enthält.

- `EnumSet#range` : Um eine Untermenge der Aufzählung nach einem durch zwei Endpunkte definierten `EnumSet#range` zu erhalten
- `EnumSet#of` : Eine Reihe bestimmter Enumerationen ohne Bereich. Mehrere überlastet `of` Methoden gibt es.
- `EnumSet#complementOf` : Eine Aufzählung, die die im Methodenparameter angegebenen Aufzählungswerte ergänzt

```
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");
      }
  }
}
```

Aufzählungen online lesen: https://riptutorial.com/de/java/topic/155/aufzahlungen

Kapitel 14: Ausdrücke

Einführung

Ausdrücke in Java sind das wichtigste Konstrukt für Berechnungen.

Bemerkungen

Eine Referenz zu den Operatoren, die in Ausdrücken verwendet werden können, finden Sie unter Operatoren .

Examples

Vorrang des Bedieners

Wenn ein Ausdruck mehrere Operatoren enthält, kann er auf verschiedene Arten gelesen werden. Zum Beispiel kann der mathematische Ausdruck `1 + 2 x 3` auf zwei Arten gelesen werden:

1. Addiere `1` und `2` und multipliziere das Ergebnis mit `3` . Dies gibt die Antwort `9` . Wenn wir Klammern hinzufügen, würde dies wie `( 1 + 2 ) x 3` aussehen.
2. Addiere `1` zum Ergebnis der Multiplikation von `2` und `3` . Dies gibt die Antwort `7` . Wenn wir Klammern hinzufügen, würde dies wie `1 + ( 2 x 3 )` aussehen.

In der Mathematik lautet die Konvention, den Ausdruck auf die zweite Art zu lesen. Die allgemeine Regel lautet, dass Multiplikation und Division vor Addition und Subtraktion erfolgen. Wenn eine fortgeschrittenere mathematische Notation verwendet wird, ist entweder die Bedeutung entweder "selbstverständlich" (für einen ausgebildeten Mathematiker!) Oder es werden Klammern hinzugefügt, um die Begriffsbestimmung zu erleichtern. In beiden Fällen hängt die Wirksamkeit der Notation zur Vermittlung von Bedeutung von der Intelligenz und dem gemeinsamen Wissen der Mathematiker ab.

Java hat die gleichen klaren Regeln zum Lesen eines Ausdrucks, basierend auf der *Priorität* der verwendeten Operatoren.

Im Allgemeinen wird jedem Operator ein *Vorrangwert zugeordnet* . siehe nachstehende Tabelle.

Zum Beispiel:

```
1 + 2 * 3
```

Der Vorrang von `+` ist niedriger als der Vorrang von `*` . Das Ergebnis des Ausdrucks ist also 7 und nicht 9.

Beschreibung	Operatoren / Konstrukte (primär)	Vorrang	Assoziativität
Qualifikation Klammern Instanzerstellung Feldzugang Array-Zugriff Methodenaufruf Methodenreferenz	Name . Name (Ausdruck) new primär . Name primär [ausdrücken] primär (ausdrücken, ...) primärer :: name	fünfzehn	Links nach rechts
Post-Inkrement	Ausdruck ++ , Ausdruck --	14	-
Pre-Inkrement Unary Besetzung [1]	++ Ausdruck, -- Ausdruck, + Ausdr, - ausdr, ~ ausdr, ! ausdrücken, (Typ) Ausdruck	13	- Rechts nach links Rechts nach links
Multiplikativ	* /%	12	Links nach rechts
Zusatzstoff	+ -	11	Links nach rechts
Verschiebung	<< >> >>>	10	Links nach rechts
Relational	<> <=> = instanceof	9	Links nach rechts
Gleichberechtigung	==! =	8	Links nach rechts
Bitweises AND	&	7	Links nach rechts
Bitweises exklusives ODER	^	6	Links nach rechts
Bitweises ODER	|	5	Links nach rechts
Logisches UND	&&	4	Links nach rechts
Logisches ODER	||	3	Links nach rechts
Bedingung [1]	? :	2	Rechts nach links

Beschreibung	Operatoren / Konstrukte (primär)	Vorrang	Assoziativität
Zuordnung Lambda [1]	= * = / =% = + = - = << = >> = >>> = & = ^ = | = ->	1	Rechts nach links

[1] Die Priorität von Lambda-Ausdrücken ist komplex, da sie auch nach einem Cast oder als dritter Teil des bedingten ternären Operators auftreten kann.

Konstante Ausdrücke

Ein konstanter Ausdruck ist ein Ausdruck, der einen primitiven Typ oder einen String ergibt und dessen Wert zur Kompilierzeit in ein Literal ausgewertet werden kann. Der Ausdruck muss auswerten, ohne eine Ausnahme auszulösen, und er muss nur aus folgenden Elementen bestehen:

- Primitive und String Literale.
- Typumwandlungen in primitive Typen oder `String` .
- Die folgenden unären Operatoren: `+` , `-` , `~` und `!` .
- Die folgenden binären Operatoren: `*` , `/` , `%` , `+` , `-` , `<<` , `>>` , `>>>` , `<` , `<=` , `>` , `>=` , `== !=` , `&` , `^` , `|` , `&&` und `||` .
- Der ternäre bedingte Operator `? :` .
- Konstante Ausdrücke.
- Einfache Namen, die sich auf konstante Variablen beziehen. (Eine konstante Variable ist eine als `final` deklarierte Variable, bei der der Initialisierungsausdruck selbst ein konstanter Ausdruck ist.)
- Qualifizierte Namen der Form `<TypeName> . <Identifier>` , die sich auf konstante Variablen beziehen.

Beachten Sie, dass die oben aufgeführte Liste *umfasst nicht* `++` und `--` die Zuweisungsoperatoren, `class` und `instanceof` , Methodenaufrufe und Verweise auf allgemeine Variablen oder Felder.

Konstante Ausdrücke vom Typ `String` führen zu einer "internierten" `String` , und Fließkommaoperationen in konstanten Ausdrücken werden mit FP-strikter Semantik ausgewertet.

Verwendet für konstante Ausdrücke

Konstante Ausdrücke können (fast) überall dort verwendet werden, wo ein normaler Ausdruck verwendet werden kann. Sie haben jedoch in den folgenden Zusammenhängen eine besondere Bedeutung.

Konstante Ausdrücke sind für die Bezeichnung von Beschriftungen in switch-Anweisungen

erforderlich. Zum Beispiel:

```
switch (someValue) {
case 1 + 1:            // OK
case Math.min(2, 3):   // Error - not a constant expression
    doSomething();
}
```

Wenn der Ausdruck auf der rechten Seite einer Zuweisung ein konstanter Ausdruck ist, kann die Zuweisung eine primitive Verengungsumwandlung durchführen. Dies ist zulässig, vorausgesetzt, der Wert des konstanten Ausdrucks liegt innerhalb des Bereichs des Typs auf der linken Seite. (Siehe JLS 5.1.3 und 5.2) Zum Beispiel:

```
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
```

Wenn ein konstanter Ausdruck als Bedingung in einer verwendet wird , `do` , `while` oder `for` , dann betrifft es die Lesbarkeit Analyse. Zum Beispiel:

```
while (false) {
    doSomething();           // Error - statenent not reachable
}
boolean flag = false;
while (flag) {
    doSomething();           // OK
}
```

(Beachten Sie, dass dies nicht gilt, `if` Anweisungen verwendet werden. Der Java-Compiler lässt zu, dass der `then` oder `else` Block einer `if` -Anweisung nicht erreichbar ist. Dies ist das Java-Analog der bedingten Kompilierung in C und C ++.)

Schließlich werden `static final` Felder in einer Klasse oder Schnittstelle mit konstanten Ausdrucksinitialisierern eifrig initialisiert. Somit ist garantiert, dass diese Konstanten im initialisierten Zustand beobachtet werden, selbst wenn ein Zyklus im Abhängigkeitsgraphen der Klasseninitialisierung vorliegt.

Weitere Informationen finden Sie in JLS 15.28. Konstante Ausdrücke .

Reihenfolge der Ausdrucksauswertung

Java-Ausdrücke werden nach folgenden Regeln ausgewertet:

- Operanden werden von links nach rechts ausgewertet.
- Die Operanden eines Operators werden vor dem Operator ausgewertet.
- Operatoren werden nach der Priorität des Operators bewertet
- Argumentlisten werden von links nach rechts ausgewertet.

Einfaches Beispiel

Im folgenden Beispiel:

```
int i = method1() + method2();
```

Die Reihenfolge der Bewertung lautet:

1. Der linke Operand von `=` Operator wird auf die Adresse von `i` ausgewertet.
2. Der linke Operand des `+` -Operators (`method1()`) wird ausgewertet.
3. Der rechte Operand des `+` -Operators (`method2()`) wird ausgewertet.
4. Die `+` -Operation wird ausgewertet.
5. Die `=` -Operation wird ausgewertet, wobei das Ergebnis der Addition `i` .

Wenn die Auswirkungen der Aufrufe beobachtbar sind, können Sie feststellen, dass der Aufruf von `method1` vor dem Aufruf von `method2` .

Beispiel mit einem Operator, der eine Nebenwirkung hat

Im folgenden Beispiel:

```
int i = 1;
intArray[i] = ++i + 1;
```

Die Reihenfolge der Bewertung lautet:

1. Der linke Operand des Operators `=` wird ausgewertet. Dies gibt die Adresse von `intArray[1]` .
2. Das Vorinkrement wird ausgewertet. Dies addiert `1` zu `i` und wird zu `2` ausgewertet.
3. Der rechte Operand des `+` wird ausgewertet.
4. Die `+` -Operation wird ausgewertet als: `2 + 1` -> `3` .
5. Die `=` Operation wird ausgewertet, wobei `intArray[1] 3 intArray[1]` .

Beachten Sie, dass der linke Operand von `=` zuerst ausgewertet wird und nicht durch die Nebenwirkung des `++i` Unterausdrucks beeinflusst wird.

Referenz:

- JLS 15.7 - Auswertungsreihenfolge

Grundlagen des Ausdrucks

Ausdrücke in Java sind das wichtigste Konstrukt für Berechnungen. Hier sind einige Beispiele:

```
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
```

Ein Ausdruck besteht im Allgemeinen aus den folgenden Formen:

- Ausdrucksnamen, bestehend aus:
  - Einfache Bezeichner; zB `someIdentifier`
  - Qualifizierte Identifikatoren; zB `MyClass.someField`
- Vorwahlen bestehend aus:
  - Literale; zB `1` , `1.0` , `'X'` , `"hello"` , `false` und `null`
  - Wörtliche Ausdrücke der Klasse; zB `MyClass.class`
  - `this` und `<TypeName> . this`
  - Eingeklammerte Ausdrücke; zB `( a + b )`
  - Ausdrücke zur Erstellung von Klasseninstanzen; zB `new MyClass(1, 2, 3)`
  - Ausdrücke zur Erstellung von Array-Instanzen; zB `new int[3]`
  - Feldzugriffsausdrücke; zB `obj.someField` oder `this.someField`
  - Array-Zugriffsausdrücke; zB `vector[21]`
  - Methodenaufrufe; zB `obj.doIt(1, 2, 3)`
  - Methodenreferenzen (Java 8 und höher); zB `MyClass::doIt`
- Unäre Operatorausdrücke; zB `!a` oder `i++`
- Binäre Operatorausdrücke; zB `a + b` oder `obj == null`
- Ternäre Operatorausdrücke; zB `(obj == null) ? 1 : obj.getCount()`
- Lambda-Ausdrücke (Java 8 und höher); zB `obj -> obj.getCount()`

Die Details zu den verschiedenen Ausdrucksformen finden Sie in anderen Themen.

- Das Thema Operatoren behandelt unäre, binäre und ternäre Operatorausdrücke.
- Das Thema Lambda-Ausdrücke deckt Lambda-Ausdrücke und Methodenreferenzausdrücke ab.
- Das Thema Klassen und Objekte behandelt Ausdrücke zur Erstellung von Klasseninstanzen.
- Das Thema Arrays behandelt Array-Zugriffsausdrücke und Array-Instanzerstellungsausdrücke.
- Das Thema Literale behandelt die verschiedenen Arten von literalen Ausdrücken.

Der Typ eines Ausdrucks

In den meisten Fällen hat ein Ausdruck einen statischen Typ, der zur Kompilierzeit durch Untersuchen und dessen Unterausdrücke bestimmt werden kann. Diese werden als *eigenständige* Ausdrücke bezeichnet.

Die folgenden Arten von Ausdrücken können jedoch (in Java 8 und höher) *Polyausdrücke sein* :

- Eingeklammerte Ausdrücke
- Ausdrücke für die Erstellung von Klasseninstanzen
- Methodenaufrufausdrücke
- Methodenreferenzausdrücke
- Bedingte Ausdrücke
- Lambda-Ausdrücke

Wenn ein Ausdruck ein Poly Ausdruck ist, kann seine Aktivität durch den *Zieltyp* des Ausdrucks

beeinflußt werden; dh wofür es verwendet wird.

Der Wert eines Ausdrucks

Der Wert eines Ausdrucks ist mit seinem Typ kompatibel. Die Ausnahme ist, wenn eine *Haufenverschmutzung* aufgetreten ist; z. B. weil "unsichere Konvertierungswarnungen" (unangemessen) unterdrückt oder ignoriert wurden.

Ausdrucksanweisungen

Im Gegensatz zu vielen anderen Sprachen erlaubt Java im Allgemeinen nicht, Ausdrücke als Anweisungen zu verwenden. Zum Beispiel:

```
public void compute(int i, int j) {
    i + j;   // ERROR
}
```

Da das Ergebnis der Auswertung eines Ausdrucks like nicht verwendet werden kann und die Ausführung des Programms auf keine andere Weise beeinflusst werden kann, haben die Java-Designer die Auffassung vertreten, dass eine solche Verwendung entweder ein Fehler ist oder falsch ist.

Dies gilt jedoch nicht für alle Ausdrücke. Eine Teilmenge von Ausdrücken ist als Aussagen rechtlich zulässig. Das Set umfasst:

- Zuweisungsausdruck, einschließlich Zuweisungen von *Vorgängen* .
- Inkrement- und Dekrement-Ausdrücke vor und nach dem Schreiben.
- Methodenaufrufe (`void` oder `void`).
- Ausdrücke für die Erstellung von Klasseninstanzen

Ausdrücke online lesen: https://riptutorial.com/de/java/topic/8167/ausdrucke

Kapitel 15: Ausnahmen und Ausnahmebehandlung

Einführung

Objekte vom Typ `Throwable` und deren Subtypen können mit dem Schlüsselwort `throw` auf den Stack geschickt und mit `try…catch` Anweisungen abgefangen werden.

Syntax

- void someMethod () löst SomeException {} // Methodendeklaration aus und zwingt den Aufruf von Methodenaufrufen, wenn SomeException ein geprüfter Ausnahmetyp ist
- Versuchen {

```
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
```

  }
- endlich {

```
 //code that will always run, whether try block finishes or not
```

  }

Examples

Eine Ausnahme mit Try-Catch abfangen

Eine Ausnahme kann mit der `try...catch` Anweisung abgefangen und behandelt werden. (In der Tat nehmen `try` Anweisungen andere Formen an, wie in anderen Beispielen über `try...catch...finally` und `try-with-resources` .)

Try-catch mit einem catch-Block

Die einfachste Form sieht so aus:

```
try {
    doSomething();
} catch (SomeException e) {
    handle(e);
}
// next statement
```

Das Verhalten eines einfachen `try...catch` ist wie folgt:

- Die Anweisungen im `try` Block werden ausgeführt.
- Wenn von den Anweisungen im `try` Block keine Ausnahme ausgelöst wird, wird die Steuerung nach dem `try...catch` an die nächste Anweisung übergeben.
- Wenn innerhalb des `try` Blocks eine Ausnahme ausgelöst wird.
    - Das Ausnahmeobjekt wird getestet, um zu sehen, ob es sich um eine Instanz von `SomeException` oder um einen Untertyp handelt.
    - Wenn ja, dann wird der `catch` wird Block die Ausnahme *abfangen:*
        - Die Variable `e` ist an das Ausnahmeobjekt gebunden.
        - Der Code innerhalb des `catch` Blocks wird ausgeführt.
        - Wenn dieser Code eine Ausnahme auslöst, wird die neu geworfene Ausnahme anstelle der ursprünglichen Ausnahme propagiert.
        - Ansonsten geht die Kontrolle nach dem `try...catch` zur nächsten Anweisung über.
    - Ist dies nicht der Fall, setzt sich die ursprüngliche Ausnahme fort.

Try-Catch mit mehreren Catches

Ein `try...catch` kann auch mehrere `catch` Blöcke haben. Zum Beispiel:

```
try {
    doSomething();
} catch (SomeException e) {
    handleOneWay(e)
} catch (SomeOtherException e) {
    handleAnotherWay(e);
}
// next statement
```

Wenn es mehrere `catch` Blöcke gibt, werden sie nacheinander beginnend mit dem ersten versucht, bis eine Übereinstimmung für die Ausnahme gefunden wird. Der entsprechende Handler wird ausgeführt (wie oben), und die Steuerung wird nach der `try...catch` Anweisung an die nächste Anweisung übergeben. Die `catch` Blöcke nach dem übereinstimmenden werden immer übersprungen, *auch wenn der Handlercode eine Ausnahme auslöst* .

Die Abgleichstrategie "von oben nach unten" hat Konsequenzen für Fälle, in denen die Ausnahmen in den `catch` nicht unzusammenhängend sind. Zum Beispiel:

```
try {
    throw new RuntimeException("test");
} catch (Exception e) {
    System.out.println("Exception");
```

```
} catch (RuntimeException e) {
    System.out.println("RuntimeException");
}
```

Dieses Code-Snippet gibt "Exception" statt "RuntimeException" aus. Da es sich bei `RuntimeException` um einen Untertyp von `Exception` , wird der erste (allgemeinere) `catch` abgeglichen. Der zweite (spezifischere) `catch` wird niemals ausgeführt.

Daraus können wir lernen, dass die spezifischsten `catch` (in Bezug auf die Ausnahmetypen) zuerst und die allgemeinsten als letzte angezeigt werden sollten. (Einige Java-Compiler werden Sie warnen, wenn ein `catch` niemals ausgeführt werden kann. Dies ist jedoch kein Kompilierungsfehler.)

Fangblöcke für mehrere Ausnahmen

Java SE 7

Beginnend mit Java SE 7 kann ein einzelner `catch` Block eine Liste nicht zusammenhängender Ausnahmen verarbeiten. Die Ausnahmetypen werden mit einem vertikalen Strich (`|`) angezeigt. Zum Beispiel:

```
try {
    doSomething();
} catch (SomeException | SomeOtherException e) {
    handleSomeException(e);
}
```

Das Verhalten eines Ausnahmefalls mit mehreren Ausnahmen ist eine einfache Erweiterung für den Fall mit einer Ausnahme. Der `catch` stimmt überein, wenn die geworfene Ausnahme (mindestens) mit einer der aufgeführten Ausnahmen übereinstimmt.

Die Spezifikation enthält einige zusätzliche Feinheiten. Der Typ von `e` ist eine synthetische *Vereinigung* der Ausnahmetypen in der Liste. Wenn der Wert von `e` verwendet wird, ist sein statischer Typ der am wenigsten verbreitete Supertyp der Typvereinigung. Wenn jedoch `e` innerhalb des `catch` Blocks erneut ausgelöst wird, handelt es sich bei den ausgelösten Ausnahmetypen um die Typen in der Union. Zum Beispiel:

```
public void method() throws IOException, SQLException
    try {
        doSomething();
    } catch (IOException | SQLException e) {
        report(e);
        throw e;
    }
```

In der obigen `SQLException` werden `IOException` und `SQLException` auf Ausnahmen geprüft, deren am wenigsten üblicher Supertyp `Exception` . Dies bedeutet, dass die `report` mit dem `report(Exception)` übereinstimmen muss. Der Compiler weiß jedoch, dass der Wurf nur eine `IOException` oder eine `SQLException` `throw` kann. Daher kann die `method` als `throws IOException, SQLException` und nicht als `throws Exception` deklariert werden. (Was eine gute Sache ist: siehe Pitfall - Throwable Throwable,

Exception, Error oder RuntimeException .)

Eine Ausnahme auslösen

Das folgende Beispiel zeigt die Grundlagen zum Auslösen einer Ausnahme:

```
public void checkNumber(int number) throws IllegalArgumentException {
    if (number < 0) {
        throw new IllegalArgumentException("Number must be positive: " + number);
    }
}
```

Die Ausnahme wird in der 3. Zeile geworfen. Diese Aussage kann in zwei Teile unterteilt werden:

- `new IllegalArgumentException(...)` erstellt eine Instanz der `IllegalArgumentException` Klasse mit einer Nachricht, die den von der Ausnahme gemeldeten Fehler beschreibt.
- `throw ...` wirft dann das Exception-Objekt.

Wenn die Ausnahme ausgelöst wird, werden die einschließenden Anweisungen *abnormal beendet,* bis die Ausnahme *behandelt wird* . Dies wird in anderen Beispielen beschrieben.

Es empfiehlt sich, das Ausnahmeobjekt in einer einzelnen Anweisung zu erstellen und zu werfen, wie oben gezeigt. Es ist außerdem empfehlenswert, eine aussagekräftige Fehlermeldung in die Ausnahme aufzunehmen, um dem Programmierer zu helfen, die Ursache des Problems zu verstehen. Dies ist jedoch nicht notwendigerweise die Nachricht, die Sie dem Endbenutzer zeigen sollten. (Java bietet zunächst keine direkte Unterstützung für die Internationalisierung von Ausnahmemeldungen.)

Es gibt noch ein paar weitere Punkte zu machen:

- Wir haben die `checkNumber` als `throws IllegalArgumentException` . Dies war nicht unbedingt erforderlich, da `IllegalArgumentException` eine geprüfte Ausnahme ist. Siehe Die Java-Ausnahmehierarchie - Ungeprüfte und geprüfte Ausnahmen . Es ist jedoch empfehlenswert, dies zu tun und auch die Ausnahmen zu berücksichtigen, die die Javadoc-Kommentare einer Methode auslösen.
- Code unmittelbar nach einer `throw` Anweisung ist *nicht erreichbar* . Also, wenn wir das geschrieben haben:

  ```
  throw new IllegalArgumentException("it is bad");
  return;
  ```

  Der Compiler meldet einen Kompilierungsfehler für die `return` .

Ausnahme-Verkettung

Viele Standardausnahmen haben einen Konstruktor mit einem zweiten `cause` zusätzlich zum herkömmlichen `message` . Die `cause` ermöglicht es Ihnen, Ausnahmen zu verketten. Hier ist ein

Beispiel.

Zuerst definieren wir eine ungeprüfte Ausnahme, die von unserer Anwendung ausgelöst wird, wenn ein nicht behebbarer Fehler auftritt. Beachten Sie, dass wir einen Konstruktor eingefügt haben, der ein `cause` akzeptiert.

```
public class AppErrorException extends RuntimeException {
    public AppErrorException() {
        super();
    }

    public AppErrorException(String message) {
        super(message);
    }

    public AppErrorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Nachfolgend finden Sie einen Code, der die Verkettung von Ausnahmen veranschaulicht.

```
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);
    }
}
```

Der `throw` innerhalb des `try` Blocks erkennt ein Problem und meldet es über eine Ausnahme mit einer einfachen Nachricht. Im Gegensatz dazu behandelt der `throw` innerhalb des `catch` Blocks die `IOException` indem sie eine neue (geprüfte) Ausnahme `IOException` . Die ursprüngliche Ausnahme wird jedoch nicht weggeworfen. Wenn Sie die `IOException` als `cause` , zeichnen wir sie auf, damit sie im Stacktrace gedruckt werden kann, wie unter Erstellen und Lesen von Stacktraces erläutert.

Benutzerdefinierte Ausnahmen

In den meisten Fällen ist es aus Sicht des Code-Designs einfacher, vorhandene generische `Exception` Klassen zu verwenden, wenn Ausnahmen ausgelöst werden. Dies gilt insbesondere, wenn Sie nur die Ausnahme benötigen, um eine einfache Fehlermeldung zu erhalten. In diesem Fall wird normalerweise RuntimeException bevorzugt, da es sich nicht um eine geprüfte Ausnahme handelt. Es gibt andere Ausnahmeklassen für allgemeine Fehlerklassen:

- UnsupportedOperationException - eine bestimmte Operation wird nicht unterstützt
- IllegalArgumentException - Ein ungültiger Parameterwert wurde an eine Methode übergeben
- IllegalStateException - Ihre API hat intern eine Bedingung erreicht, die niemals auftreten sollte oder die Folge einer ungültigen Verwendung Ihrer API ist

Fälle, in denen Sie eine benutzerdefinierte Ausnahmeklasse **verwenden** möchten, umfassen Folgendes:

- Sie schreiben eine API oder Bibliothek, die von anderen verwendet werden soll, und Sie möchten, dass Benutzer Ihrer API Ausnahmen spezifisch abfangen und behandeln *können und diese Ausnahmen von anderen, allgemeineren Ausnahmen unterscheiden können* .
- Sie werfen Ausnahmen für eine **bestimmte Art von Fehler** in einem Teil Ihres Programms aus, die Sie in einem anderen Teil Ihres Programms abfangen und behandeln möchten, und Sie möchten diese Fehler von anderen, allgemeineren Fehlern unterscheiden können.

Sie können Ihre eigenen benutzerdefinierten Ausnahmen erstellen, indem Sie `RuntimeException` für eine ungeprüfte Ausnahme erweitern oder die Exception prüfen, indem Sie eine `Exception` *die nicht auch eine Unterklasse von RuntimeException* ist.

> Unterklassen von Exception, die nicht ebenfalls Unterklassen von RuntimeException sind, sind geprüfte Ausnahmen

```
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;
    }
}
```

Diese können als vordefinierte Ausnahmen verwendet werden:

```
void validateString(String value){
    if (value.length() > 30){
        throw new StringTooLongException(value, 30);
    }
}
```

Die Felder können verwendet werden, wenn die Ausnahme abgefangen und behandelt wird:

```
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 );
    }
}
```

Beachten Sie, dass gemäß der Java-Dokumentation von Oracle :

> [...] Wenn von einem Client vernünftigerweise erwartet wird, dass er sich von einer

Ausnahme erholt, machen Sie ihn zu einer geprüften Ausnahme. Wenn ein Client nichts aus der Ausnahme wiederherstellen kann, machen Sie ihn zu einer ungeprüften Ausnahme.

Mehr:

- Warum erfordert RuntimeException keine explizite Ausnahmebehandlung?

Die try-with-resources-Anweisung

Java SE 7

Wie das Beispiel der try-catch-final-Anweisung veranschaulicht, ist für die Bereinigung von Ressourcen mit einer `finally` Klausel eine erhebliche Menge an "Boiler-Plate" -Code erforderlich, um die Randfälle korrekt zu implementieren. Java 7 bietet eine wesentlich einfachere Möglichkeit, dieses Problem in Form der *try-with-resources*- Anweisung zu lösen.

Was ist eine Ressource?

In Java 7 wurde die Schnittstelle `java.lang.AutoCloseable` eingeführt, mit der Klassen mit der *try-with-resources*- Anweisung verwaltet werden können. Instanzen von Klassen, die `AutoCloseable` implementieren, werden als *Ressourcen bezeichnet* . Diese müssen in der Regel rechtzeitig entsorgt werden, anstatt sich auf den Müllsammler zu verlassen.

Die `AutoCloseable` Schnittstelle definiert eine einzelne Methode:

```
public void close() throws Exception
```

Eine `close()` -Methode sollte die Ressource in geeigneter Weise entsorgen. Die Spezifikation besagt, dass es sicher sein sollte, die Methode für eine Ressource aufzurufen, die bereits entsorgt wurde. Darüber hinaus Klassen , die Umsetzung `Autocloseable` *dringend empfohlen* , den erklären *ermutigt* `close()` Methode zu werfen eine spezifischere Ausnahme als `Exception` oder keine Ausnahme überhaupt.

Eine Vielzahl von Java-Standardklassen und -Schnittstellen implementiert `AutoCloseable` . Diese schließen ein:

- `InputStream` , `OutputStream` und ihre Unterklassen
- `Reader` , `Writer` und ihre Unterklassen
- `Socket` und `ServerSocket` und ihre Unterklassen
- `Channel` und seine Unterklassen und
- die JDBC-Schnittstellen `Connection` , `Statement` und `ResultSet` und ihre Unterklassen.

Anwendungen und Kurse von Drittanbietern können dies ebenfalls tun.

Die grundlegende try-with-resource-Anweisung

Die Syntax einer *Try-With-Ressource* basiert auf den klassischen *Try-Catch-* , *Try-Final-* und *Try-Catch-Final-* Formularen. Hier ist ein Beispiel für eine "Grundform"; dh die Form ohne `catch` oder `finally` .

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}
```

Die zu verwaltenden Ressourcen werden im Abschnitt `(...)` nach der `try` Klausel als Variablen deklariert. In dem obigen Beispiel erklären wir eine Ressource variablen `stream` und initialisieren es zu einem neu erstellten `PrintStream` .

Nachdem die Ressourcenvariablen initialisiert wurden, wird der `try` Block ausgeführt. Nach Abschluss dieses `stream.close()` wird `stream.close()` automatisch aufgerufen, um sicherzustellen, dass die Ressource nicht ausläuft. Beachten Sie, dass der `close()` Aufruf unabhängig vom Abschluss des Blocks geschieht.

Die erweiterten try-with-resource-Anweisungen

Die *try-with-resources-* Anweisung kann wie bei der *Try-catch-finally-* Syntax vor Java 7 mit `catch` und `finally` Blöcken erweitert werden. Der folgende Codeausschnitt fügt unserem vorherigen einen `catch` Block hinzu, um die `FileNotFoundException` zu behandeln, die der `PrintStream` Konstruktor `PrintStream` kann:

```
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");
}
```

Wenn entweder die Ressourceninitialisierung oder der try-Block die Ausnahme `catch` , wird der `catch` Block ausgeführt. Der `finally` Block wird immer wie bei einer herkömmlichen *try-catch-finally-* Anweisung ausgeführt.

Es gibt jedoch ein paar Dinge zu beachten:

- Die Ressourcenvariable befindet sich *außerhalb des Bereichs* in `catch` und blockiert `finally` .
- Die Ressourcenbereinigung findet statt, bevor die Anweisung versucht, den `catch` Block abzugleichen.
- Wenn die automatische Ressourcenbereinigung eine Ausnahme auslöst, *könnte* dies in einem der `catch` festgehalten werden.

Verwalten mehrerer Ressourcen

Die obigen Codeausschnitte zeigen eine einzige verwaltete Ressource. *Try-with-resources* kann sogar mehrere Ressourcen in einer Anweisung verwalten. Zum Beispiel:

```
try (InputStream is = new FileInputStream(file1);
     OutputStream os = new FileOutputStream(file2)) {
    // Copy 'is' to 'os'
}
```

Das verhält sich wie erwartet. Beide, `is` und `os` werden am Ende des `try` Blocks automatisch geschlossen. Es gibt einige Punkte zu beachten:

- Die Initialisierungen erfolgen in der Codereihenfolge, und spätere Ressourcenvariablen-Initialisierer können die Werte der früheren verwenden.
- Alle Ressourcenvariablen, die erfolgreich initialisiert wurden, werden bereinigt.
- Ressourcenvariablen werden in *umgekehrter Reihenfolge* ihrer Deklarationen bereinigt.

In obigem Beispiel `is` also vor `os` initialisiert und danach bereinigt, und `is` wird bereinigt, wenn während der Initialisierung von `os` eine Ausnahme vorliegt.

Gleichwertigkeit von try-with-resource und klassischem try-catch-finally

Die Java-Sprachspezifikation gibt das Verhalten von *Try-with-Resource*- Formularen im Hinblick auf die klassische *try-catch-finally*- Anweisung an. (Weitere Informationen finden Sie in der JLS.)

Zum Beispiel diese grundlegende *Try-with-Ressource* :

```
try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}
```

ist als gleichwertig mit diesem *try-catch-finally definiert* :

```
// 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);
        }
    }
}
```

(Das JLS gibt an, dass die tatsächlichen Variablen `t` und `primaryException` für normalen Java-Code nicht sichtbar sind.)

Die erweiterte Form der *Try-With-Ressourcen* wird als Äquivalenz mit der Basisform angegeben. Zum Beispiel:

```
try (PrintStream stream = new PrintStream(fileName)) {
    stream.println("Hello world!");
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}
```

ist äquivalent zu:

```
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");
}
```

Stacktraces erstellen und lesen

Wenn ein Ausnahmeobjekt erstellt wird (dh, wenn Sie es `new Throwable`), erfasst der `Throwable` Konstruktor Informationen zu dem Kontext, in dem die Ausnahme erstellt wurde. Diese Informationen können später in Form eines Stacktraces ausgegeben werden, mit dessen Hilfe das Problem diagnostiziert werden kann, das die Ausnahme in erster Linie verursacht hat.

Stacktrace drucken

Beim `printStackTrace()` eines Stacktraces muss lediglich die `printStackTrace()` -Methode `printStackTrace()` werden. Zum Beispiel:

```
try {
    int a = 0;
    int b = 0;
    int c = a / b;
```

```
} catch (ArithmeticException ex) {
    // This prints the stacktrace to standard output
    ex.printStackTrace();
}
```

Die `printStackTrace()` -Methode ohne Argumente wird in der Standardausgabe der Anwendung gedruckt. dh das aktuelle `System.out` . Es gibt auch `printStackTrace(PrintStream)` und `printStackTrace(PrintWriter)` Überladungen, die in einem angegebenen `Stream` oder `Writer` drucken.

Anmerkungen:

1. Der Stacktrace enthält nicht die Details der Ausnahme selbst. Sie können die `toString()` -Methode verwenden, um diese Details abzurufen. z.B

   ```
   // Print exception and stacktrace
   System.out.println(ex);
   ex.printStackTrace();
   ```

2. Stacktrace-Druck sollte sparsam verwendet werden; siehe Pitfall - Übermäßige oder unangemessene Stacktraces . Es ist häufig besser, ein Protokollierungsframework zu verwenden und das zu protokollierende Ausnahmeobjekt zu übergeben.

Stapelverfolgung verstehen

Betrachten Sie das folgende einfache Programm, das aus zwei Klassen in zwei Dateien besteht. (Wir haben die Dateinamen angezeigt und Zeilennummern zur Veranschaulichung hinzugefügt.)

```
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      }
```

Wenn diese Dateien kompiliert und ausgeführt werden, erhalten Sie die folgende Ausgabe.

```
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)
```

Lassen Sie uns diese Zeile für Zeile lesen, um herauszufinden, was es uns sagt.

Zeile 1 sagt uns, dass der Thread "main" aufgrund einer nicht erfassten Ausnahme beendet wurde. Der vollständige Name der Ausnahme lautet `java.lang.ArithmeticException` , und die Ausnahmemeldung lautet "/ by zero".

Wenn wir die Javadocs für diese Ausnahme suchen, heißt es:

> Wird ausgelöst, wenn eine außergewöhnliche arithmetische Bedingung vorliegt. Beispielsweise löst eine Ganzzahl "durch Null teilen" eine Instanz dieser Klasse aus.

Tatsächlich ist die Nachricht "/ by zero" ein deutlicher Hinweis darauf, dass die Ursache der Ausnahme darin liegt, dass ein Code versucht hat, etwas durch null zu teilen. Aber was?

Die restlichen 3 Zeilen sind die Stapelverfolgung. Jede Zeile stellt einen Methodenaufruf (oder Konstruktoraufruf) auf der Aufrufliste dar. In jeder Zeile werden drei Dinge angegeben:

- Name der Klasse und Methode, die ausgeführt wurde,
- der Quellcode-Dateiname,
- Die Quellcode-Zeilennummer der Anweisung, die gerade ausgeführt wird

Diese Zeilen eines Stacktraces werden oben mit dem Frame für den aktuellen Anruf aufgelistet. Der obere Frame in unserem Beispiel oben befindet sich in der `Test.bar` Methode und in Zeile 9 der Test.java-Datei. Das ist die folgende Zeile:

```
    return a / b;
```

Wenn wir uns ein paar Zeilen früher in der Datei ansehen, wo `b` initialisiert wird, ist offensichtlich, dass `b` den Wert Null hat. Wir können ohne Zweifel sagen, dass dies die Ursache der Ausnahme ist.

Wenn wir weiter gehen mussten, können wir aus dem stacktrace ersehen, dass `bar()` von Zeile 3 von Test.java aus `foo()` aufgerufen wurde und dass `foo()` wiederum von `Main.main()` aufgerufen wurde.

Anmerkung: Die Klassen- und Methodennamen in den Stack-Frames sind die internen Namen für die Klassen und Methoden. Sie müssen die folgenden ungewöhnlichen Fälle erkennen:

- Eine verschachtelte oder innere Klasse sieht wie "OuterClass $ InnerClass" aus.
- Eine anonyme innere Klasse sieht wie "OuterClass $ 1", "OuterClass $ 2" usw. aus.
- Wenn Code in einem Konstruktor, Instanzfeldinitialisierer oder einem Instanzinitialisiererblock ausgeführt wird, lautet der Methodenname "".
- Wenn Code in einem statischen Feldinitialisierungs- oder statischen Initialisierungsblock ausgeführt wird, lautet der Methodenname "".

(In einigen Java-Versionen erkennt und entfernt der Stacktrace-Formatierungscode wiederholte Stackframe-Sequenzen, die auftreten können, wenn eine Anwendung aufgrund einer

übermäßigen Rekursion ausfällt.)

Ausnahme-Verkettung und verschachtelte Stacktraces

Java SE 1.4

Die Verkettung von Ausnahmebedingungen geschieht, wenn ein Codeabschnitt eine Ausnahme abfängt und dann eine neue erstellt und auslöst, wobei die erste Ausnahme als Ursache übergeben wird. Hier ist ein Beispiel:

```
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  }
```

Wenn die obige Klasse kompiliert und ausgeführt wird, erhalten wir die folgende Stapelverfolgung:

```
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
```

Der stacktrace beginnt mit dem Klassennamen, der Methode und dem Aufrufstack für die Ausnahme, die (in diesem Fall) zum Absturz der Anwendung geführt hat. Daran schließt sich eine Zeile "Caused by:" an, die die `cause` meldet. Der Klassenname und die Nachricht werden gemeldet, gefolgt von den Stack-Frames der Ursachenausnahme. Die Ablaufverfolgung endet mit einem "... N more", der angibt, dass die letzten N Frames mit denen der vorherigen Ausnahme identisch sind.

"Verursacht durch:" ist nur in der Ausgabe enthalten, wenn die `cause` der primären Ausnahme nicht `null` . Ausnahmen können unbegrenzt verkettet werden. In diesem Fall kann der Stacktrace mehrere "Verursacht durch:" - Spuren aufweisen.

Hinweis: Der `cause` wurde nur in der `Throwable` API in Java 1.4.0 verfügbar gemacht. Vorher musste die Verkettung von Ausnahmen von der Anwendung mithilfe eines benutzerdefinierten

Ausnahmefelds zur Darstellung der Ursache und einer benutzerdefinierten `printStackTrace` Methode `printStackTrace` werden.

Erfassen eines Stacktraces als String

Manchmal muss eine Anwendung in der Lage sein, einen Stacktrace als Java- `String` zu erfassen, damit er für andere Zwecke verwendet werden kann. Im Allgemeinen wird dazu ein temporärer `OutputStream` oder `Writer` , der in einen In-Memory-Puffer schreibt und diesen an `printStackTrace(...)` .

Die Bibliotheken von Apache Commons und Guava bieten verschiedene Methoden zum Erfassen eines Stacktraces als String:

```
org.apache.commons.lang.exception.ExceptionUtils.getStackTrace(Throwable)

com.google.common.base.Throwables.getStackTraceAsString(Throwable)
```

Wenn Sie in Ihrer Codebasis keine Bibliotheken von Drittanbietern verwenden können, führen Sie die folgende Methode mit der Aufgabe aus:

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

Wenn Sie den Stacktrace analysieren möchten, sollten Sie `getStackTrace()` und `getCause()` verwenden, `getStackTrace()` zu versuchen, einen Stacktrace zu analysieren.

Behandlung von InterruptedException

`InterruptedException` ist ein verwirrendes Tier - es erscheint in scheinbar harmlosen Methoden wie `Thread.sleep()` , aber die falsche Handhabung führt zu schwer zu verwaltendem Code, der sich in gleichzeitigen Umgebungen schlecht verhält.

Grundsätzlich bedeutet das, wenn eine `InterruptedException` aufgefangen wird, irgendjemand namens `Thread.interrupt()` in dem Thread, in dem Ihr Code gerade ausgeführt wird. Vielleicht `Thread.interrupt()` Sie sagen: "Es ist mein Code! Ich werde ihn niemals unterbrechen!" " und deshalb etwas so machen:

```
// Bad. Don't do this.
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
```

```
  // disregard
}
```

Dies ist jedoch genau der falsche Weg, um mit dem Auftreten eines "unmöglichen" Ereignisses umzugehen. Wenn Sie wissen, dass Ihre Anwendung niemals auf eine `InterruptedException` stößt, sollten Sie ein solches Ereignis als schwerwiegenden Verstoß gegen die Annahmen Ihres Programms behandeln und so schnell wie möglich beenden.

Die richtige Art, mit einem "unmöglichen" Interrupt umzugehen, ist wie folgt:

```
// When nothing will interrupt your code
try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  Thread.currentThread().interrupt();
  throw new AssertionError(e);
}
```

Das macht zwei Dinge; Zunächst wird der Interrupt-Status des Threads wiederhergestellt (als ob die `InterruptedException` nicht an erster Stelle ausgelöst worden wäre). Anschließend wird ein `AssertionError` ausgelöst, der angibt, dass die grundlegenden Invarianten Ihrer Anwendung verletzt wurden. Wenn Sie sicher sind, dass Sie den Thread niemals unterbrechen, ist dieser Code sicher, da der `catch` Block niemals erreicht werden sollte.

Guava der Verwendung von `Uninterruptibles` Klasse hilft , dieses Muster zu vereinfachen; Das Aufrufen von `Uninterruptibles.sleepUninterruptibly()` ignoriert den unterbrochenen Status eines Threads, bis die Schlafdauer abgelaufen ist (an diesem Punkt wird er für spätere Aufrufe wiederhergestellt, um die eigene `InterruptedException`). Wenn Sie wissen, dass Sie einen solchen Code niemals unterbrechen, kann auf diese Weise vermieden werden, dass Sie Ihre Schlafanrufe in einen Try-Catch-Block packen müssen.

Häufig können Sie jedoch nicht garantieren, dass Ihr Thread niemals unterbrochen wird. Insbesondere wenn Sie Code schreiben, der von einem `Executor` oder einem anderen Thread-Management ausgeführt wird, ist es wichtig, dass Ihr Code umgehend auf Interrupts reagiert. Andernfalls wird die Anwendung blockieren oder sogar blockieren.

In solchen Fällen empfiehlt es sich in der Regel, der `InterruptedException` die Weitergabe des Aufrufstapels zu erlauben, indem eine `throws InterruptedException` für jede Methode der Reihe nach hinzugefügt wird. Dies mag kludgy erscheinen, ist aber eigentlich eine wünschenswerte Eigenschaft - die Signaturen Ihrer Methode zeigen den Anrufern jetzt an, dass sie auf Interrupts umgehend antworten wird.

```
// Let the caller determine how to handle the interrupt if you're unsure
public void myLongRunningMethod() throws InterruptedException {
  ...
}
```

In einigen wenigen Fällen (zB beim Überschreiben einer Methode, die nicht `throw` alle Ausnahmen geprüft) können Sie den unterbrochenen Status zurücksetzen , ohne Auslösen einer Ausnahme, in der Erwartung , was Code wird als nächstes ausgeführt , um die Unterbrechung zu behandeln.

Dies verzögert die Behandlung der Unterbrechung, unterdrückt sie jedoch nicht vollständig.

```
// 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.
}
```

Die Java-Ausnahmehierarchie - Ungeprüfte und geprüfte Ausnahmen

Alle Java-Ausnahmen sind Instanzen von Klassen in der Exception-Klassenhierarchie. Dies kann wie folgt dargestellt werden:

- `java.lang.Throwable` - Dies ist die Basisklasse für alle Ausnahmeklassen. Seine Methoden und Konstruktoren implementieren eine Reihe von Funktionen, die allen Ausnahmen gemeinsam sind.
    - `java.lang.Exception` - Dies ist die Oberklasse aller normalen Ausnahmen.
        - verschiedene Standard- und benutzerdefinierte Ausnahmeklassen.
        - `java.lang.RuntimeException` - Dies ist die Superklasse aller normalen Ausnahmen, bei denen es sich um nicht *geprüfte Ausnahmen handelt* .
            - verschiedene Standard- und benutzerdefinierte Laufzeitausnahmeklassen.
    - `java.lang.Error` - Dies ist die Oberklasse aller "schwerwiegenden Fehlerausnahmen".

Anmerkungen:

1. Die Unterscheidung zwischen *geprüften* und *ungeprüften* Ausnahmen wird im Folgenden beschrieben.
2. Die `Throwable` , `Exception` und `RuntimeException` Klasse sollten als `abstract` behandelt werden. siehe Pitfall - Throwable, Exception, Error oder RuntimeException werfen .
3. Die `Error` werden von der JVM in Situationen ausgelöst, in denen es für eine Anwendung unsicher oder unklug wäre, eine Wiederherstellung durchzuführen.
4. Es wäre unklug, benutzerdefinierte Untertypen von `Throwable` zu deklarieren. Java-Tools und -Bibliotheken setzen möglicherweise voraus, dass `Error` und `Exception` die einzigen direkten Subtypen von `Throwable` , und `Throwable` falsch, wenn diese Annahme falsch ist.

Geprüfte versus nicht geprüfte Ausnahmen

In einigen Programmiersprachen wird kritisiert, dass Ausnahmen unterstützt werden. Es ist schwer zu wissen, welche Ausnahmen eine bestimmte Methode oder Prozedur auslösen kann. Da eine nicht behandelte Ausnahme zum Absturz eines Programms führen kann, können Ausnahmen zu einer Quelle der Fragilität werden.

Die Java-Sprache spricht dieses Problem mit dem geprüften Ausnahmemechanismus an. Erstens klassifiziert Java Ausnahmen in zwei Kategorien:

- Geprüfte Ausnahmen stellen normalerweise erwartete Ereignisse dar, mit denen eine

Anwendung umgehen kann. Beispielsweise repräsentieren `IOException` und ihre Subtypen Fehlerbedingungen, die bei E / A-Vorgängen auftreten können. Beispiele sind das Öffnen von Dateien, weil eine Datei oder ein Verzeichnis nicht vorhanden ist, das Lesen und Schreiben im Netzwerk schlägt fehl, weil eine Netzwerkverbindung unterbrochen wurde und so weiter.

- Ungeprüfte Ausnahmen stellen normalerweise unerwartete Ereignisse dar, mit denen eine Anwendung nicht umgehen kann. Dies ist normalerweise das Ergebnis eines Fehlers in der Anwendung.

("Ausgelöst" bezieht sich im Folgenden auf jede Ausnahme, die explizit (durch eine `throw` Anweisung) oder implizit (bei einer fehlgeschlagenen Dereferenzierung, Typumwandlung usw.) ausgelöst wird. In ähnlicher Weise bezieht sich "propagiert" auf eine Ausnahme, die in einem verschachtelter Anruf und nicht innerhalb des Anrufs erfasst. Der folgende Beispielcode veranschaulicht dies.)

Der zweite Teil des geprüften Ausnahmemechanismus besteht darin, dass es Einschränkungen für Methoden gibt, bei denen eine geprüfte Ausnahme auftreten kann:

- Wenn eine geprüfte Ausnahme in einer Methode ausgelöst oder propagiert wird, *muss sie* entweder von der Methode abgefangen oder in der `throws` Klausel der Methode aufgeführt werden. (Die Bedeutung der `throws` Klausel wird in diesem Beispiel beschrieben .)
- Wenn eine geprüfte Ausnahme in einem Initialisierungsblock ausgelöst oder weitergegeben wird, muss der Block abgefangen werden.
- Eine geprüfte Ausnahme kann nicht von einem Methodenaufruf in einem Feldinitialisierungsausdruck propagiert werden. (Eine solche Ausnahme kann nicht erkannt werden.)

Kurz gesagt, eine geprüfte Ausnahme muss entweder behandelt oder deklariert werden.

Diese Einschränkungen gelten nicht für ungeprüfte Ausnahmen. Dies gilt für alle Fälle, in denen eine Ausnahme implizit ausgelöst wird, da alle diese Fälle ungeprüfte Ausnahmen auslösen.

Überprüfte Ausnahmebeispiele

Diese Codeausschnitte sollen die geprüften Ausnahmebeschränkungen veranschaulichen. In jedem Fall zeigen wir eine Version des Codes mit einem Kompilierungsfehler und eine zweite Version mit dem korrigierten Fehler.

```
// 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.
}
```

Das erste Beispiel zeigt, wie explizit geworfene geprüfte Ausnahmen als "geworfen" deklariert werden können, wenn sie nicht in der Methode behandelt werden sollen.

```
// 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
    }
}
```

Das zweite Beispiel zeigt, wie mit einer propagierten geprüften Ausnahme umgegangen werden kann.

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

Das letzte Beispiel zeigt, wie mit einer geprüften Ausnahmebedingung in einem statischen Feldinitialisierer umgegangen wird.

```
// 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;
    }
}
```

Beachten Sie, dass in diesem letzten Fall müssen wir uns auch mit den Problemen, die `is` nicht zugewiesen wurde, kann als einmal mehr sein, und doch hat auch zugewiesen werden, auch im Falle einer Ausnahme.

Einführung

Ausnahmen sind Fehler, die bei der Ausführung eines Programms auftreten. Betrachten Sie das Java-Programm, unter dem zwei Ganzzahlen geteilt werden.

```
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);
    }
}
```

Jetzt kompilieren und führen wir den obigen Code aus und sehen die Ausgabe für eine versuchte Division durch Null:

```
Input two integers
7 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Division.main(Disivion.java:14)
```

Division durch Null ist eine ungültige Operation, die einen Wert erzeugen würde, der nicht als Ganzzahl dargestellt werden kann. Java behandelt dies, indem es *eine **Ausnahme auslöst*** . In diesem Fall handelt es sich bei der Ausnahme um eine Instanz der *ArithmeticException*- Klasse.

Hinweis: Das Beispiel zum Erstellen und Lesen von Stack-Traces erläutert, was die Ausgabe hinter den beiden Zahlen bedeutet.

Der Nutzen einer *Ausnahme* ist die Flusssteuerung, die sie erlaubt. Ohne Ausnahmen kann eine

typische Lösung für dieses Problem sein, zuerst zu prüfen, ob `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);
    }
}
```

Dies gibt die Nachricht aus, die `You cannot divide by zero.` auf die Konsole und beendet das Programm auf elegante Weise, wenn der Benutzer versucht, durch Null zu teilen. Eine äquivalente Möglichkeit, dieses Problem durch *Ausnahmebehandlung zu* lösen, besteht darin, die `if` Flusssteuerung durch einen `try-catch` Block zu ersetzen:

```
...

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;
}

...
```

Ein try-catch-Block wird wie folgt ausgeführt:

1. Beginnen Sie mit der Ausführung des Codes im `try` Block.
2. Wenn im try-Block eine *Ausnahme* auftritt, brechen Sie sofort ab und prüfen Sie, ob diese Ausnahme vom `catch` Block *abgefangen* wird (in diesem Fall, wenn die Ausnahme eine Instanz von `ArithmeticException`).
3. Wenn die Ausnahme *abgefangen wird* , wird sie der Variablen `e` zugewiesen und der `catch` Block wird ausgeführt.
4. Wenn entweder der `try` oder `catch` Block abgeschlossen ist (dh während der Codeausführung keine nicht erfassten Ausnahmen auftreten), fahren Sie mit der Ausführung

des Codes unterhalb des `try-catch` Blocks fort.

Im Allgemeinen wird empfohlen, die *Ausnahmebehandlung* als Teil der normalen Ablaufsteuerung einer Anwendung zu verwenden, bei der das Verhalten andernfalls undefiniert oder unerwartet wäre. Zum Beispiel ist es in der Regel besser *, eine* `null` wenn eine Methode fehlschlägt, *eine Ausnahme* auszulösen, sodass die Anwendung, die die Methode verwendet, ihre eigene Flusssteuerung für die Situation über die *Ausnahmebehandlung* der oben dargestellten Art definieren kann. In gewissem Sinne wird dadurch das Problem umgangen, dass ein bestimmter *Typ zurückgegeben werden muss* , da eine von mehreren Arten von *Ausnahmen ausgelöst werden* kann, um auf das bestimmte aufgetretene Problem hinzuweisen.

Weitere Hinweise dazu, wie und wie Exceptions nicht verwendet werden, finden Sie unter Java-Fallstricke - Exceptions

Anweisungen in try catch block zurückgeben

Obwohl dies eine schlechte Praxis ist, können in einem Ausnahmebehandlungsblock mehrere return-Anweisungen hinzugefügt werden:

```
 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;
    }
}
```

Diese Methode gibt immer 7 zurück, da der finally-Block, der dem try / catch-Block zugeordnet ist, ausgeführt wird, bevor etwas zurückgegeben wird. Nun, da hat endlich `return 7;` Dieser Wert ersetzt die Try / Catch-Rückgabewerte.

Wenn der catch-Block einen primitiven Wert zurückgibt und dieser primitive Wert anschließend im finally-Block geändert wird, wird der im catch-Block zurückgegebene Wert zurückgegeben und die Änderungen aus dem finally-Block werden ignoriert.

Im folgenden Beispiel wird "0" und nicht "1" gedruckt.

```
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;
        }
    }
}
```

Erweiterte Funktionen von Exceptions

In diesem Beispiel werden einige erweiterte Funktionen und Anwendungsfälle für Ausnahmen behandelt.

Den Callstack programmgesteuert untersuchen

Java SE 1.4

Mit Ausnahme von Stacktraces werden in erster Linie Informationen zu einem Anwendungsfehler und dessen Kontext bereitgestellt, sodass der Programmierer das Problem diagnostizieren und beheben kann. Manchmal kann es für andere Dinge verwendet werden. Beispielsweise muss eine `SecurityManager` Klasse möglicherweise den Aufrufstapel untersuchen, um zu entscheiden, ob der Code, der einen Aufruf ausführt, vertrauenswürdig ist.

Sie können Ausnahmen verwenden, um den Aufrufstapel programmatisch wie folgt zu untersuchen:

```
    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());
```

Hier gibt es einige wichtige Einschränkungen:

1. Die in einem `StackTraceElement` verfügbaren Informationen sind begrenzt. Es sind keine weiteren Informationen verfügbar, als von `printStackTrace` angezeigt werden. (Die Werte der lokalen Variablen im Frame sind nicht verfügbar.)

2. Die Javadocs für `getStackTrace()` geben an, dass eine JVM Frames auslassen `getStackTrace()` :

   > Einige virtuelle Maschinen können unter Umständen einen oder mehrere Stack-Frames aus der Stack-Ablaufverfolgung auslassen. Im Extremfall darf eine virtuelle Maschine, die über keine Stack-Trace-Informationen zu diesem auslösbaren Objekt verfügt, ein Array mit der Länge Null aus dieser Methode

zurückgeben.

Ausnahme-Konstruktion optimieren

Wie bereits an anderer Stelle erwähnt, ist das Erstellen einer Ausnahme ziemlich teuer, da dazu Informationen über alle Stack-Frames im aktuellen Thread erfasst und aufgezeichnet werden müssen. Manchmal wissen wir, dass diese Informationen niemals für eine bestimmte Ausnahme verwendet werden. Beispielsweise wird der Stacktrace niemals gedruckt. In diesem Fall gibt es einen Implementierungstrick, den wir in einer benutzerdefinierten Ausnahme verwenden können, um zu bewirken, dass die Informationen nicht erfasst werden.

Die für Stacktraces erforderlichen Stack-Frame-Informationen werden erfasst, wenn die `Throwable` Konstruktoren die `Throwable.fillInStackTrace()` -Methode `Throwable.fillInStackTrace()` . Diese Methode ist `public` , was bedeutet, dass eine Unterklasse sie überschreiben kann. Der Trick besteht darin, die von `Throwable` vererbte `Throwable` mit einer Methode zu überschreiben, die nichts tut. z.B

```
public class MyException extends Exception {
    // constructors

    @Override
    public void fillInStackTrace() {
        // do nothing
    }
}
```

Das Problem bei diesem Ansatz ist, dass eine Ausnahme, die `fillInStackTrace()` überschreibt, niemals den Stacktrace erfassen kann und in Szenarien, in denen Sie eine benötigen, nutzlos ist.

Löschen oder Ersetzen des Stacktraces

Java SE 1.4

In einigen Situationen enthält das Stacktrace für eine auf normale Weise erstellte Ausnahme entweder falsche Informationen oder Informationen, die der Entwickler dem Benutzer nicht preisgeben möchte. In diesen Szenarien kann `Throwable.setStackTrace` verwendet werden, um das Array von `StackTraceElement` Objekten zu ersetzen, das die Informationen enthält.

Zum Beispiel kann Folgendes verwendet werden, um die Stack-Informationen einer Ausnahme zu verwerfen:

```
exception.setStackTrace(new StackTraceElement[0]);
```

Unterdrückte Ausnahmen

Java SE 7

Java 7 führte das *try-with-resources*- Konstrukt und das damit verbundene Konzept der

Ausnahmeunterdrückung ein. Betrachten Sie den folgenden Ausschnitt:

```
try (Writer w = new BufferedWriter(new FileWriter(someFilename))) {
    // do stuff
    int temp = 0 / 0;    // throws an ArithmeticException
}
```

Wenn die Ausnahme ausgelöst wird, ruft der `try` `close()` auf dem `w` wodurch alle gepufferten Ausgaben `FileWriter` und der `FileWriter` . Aber was passiert, wenn eine `IOException` ausgelöst wird, während die Ausgabe `IOException` wird?

Was passiert, ist, dass jede Ausnahme, die beim Bereinigen einer Ressource ausgelöst wird, *unterdrückt wird* . Die Ausnahme wird abgefangen und der Liste der unterdrückten Ausnahmen der primären Ausnahme hinzugefügt. Als Nächstes werden die *Try-with-Ressourcen* mit der Bereinigung der anderen Ressourcen fortgesetzt. Schließlich wird die primäre Ausnahme erneut ausgegeben.

Ein ähnliches Muster tritt auf, wenn eine Ausnahme während der Ressourceninitialisierung ausgelöst wurde oder der `try` Block normal abgeschlossen wurde. Die erste geworfene Ausnahme wird zur primären Ausnahme und nachfolgende, die aus der Bereinigung resultieren, werden unterdrückt.

Die unterdrückten Ausnahmen können vom aufrufenden `getSuppressedExceptions` aus dem primären Ausnahmenobjekt `getSuppressedExceptions` .

Die Anweisungen try-finally und try-catch-finally

Die Anweisung `try...catch...finally` kombiniert die Ausnahmebehandlung mit Bereinigungscode. Der `finally` Block enthält Code, der unter allen Umständen ausgeführt wird. Dadurch sind sie für das Ressourcenmanagement und andere Arten der Bereinigung geeignet.

Versuchen Sie es endlich

Hier ist ein Beispiel für die einfachere (`try...finally`) Form:

```
try {
    doSomething();
} finally {
    cleanUp();
}
```

Das Verhalten des `try...finally` wie folgt:

- Der Code im `try` Block wird ausgeführt.
- Wenn im `try` Block keine Ausnahme ausgelöst wurde:
    - Der Code im `finally` Block wird ausgeführt.
    - Wenn der `finally` Block eine Ausnahme auslöst, wird diese Ausnahme weitergegeben.
    - Ansonsten geht die Kontrolle nach dem `try...finally` Befehl auf die nächste Anweisung `try...finally` .

- Wenn im try-Block eine Ausnahme ausgelöst wurde:
    - Der Code im `finally` Block wird ausgeführt.
    - Wenn der `finally` Block eine Ausnahme auslöst, wird diese Ausnahme weitergegeben.
    - Andernfalls setzt sich die ursprüngliche Ausnahme fort.

Der Code innerhalb `finally` Blocks wird immer ausgeführt. (Die einzigen Ausnahmen sind, wenn `System.exit(int)` aufgerufen wird, oder wenn die JVM in Panik gerät.) Daher ist ein `finally` Block der richtige `System.exit(int)` , der immer ausgeführt werden muss. zB das Schließen von Dateien und anderen Ressourcen oder das Aufheben von Sperren.

Try-Catch-Endlich

Unser zweites Beispiel zeigt, wie `catch` und `finally` verwendet werden können. Es zeigt auch, dass das Aufräumen von Ressourcen nicht einfach ist.

```
// 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
        }
    }
}
```

Die vollständigen (hypothetischen) Verhaltensweisen von `try...catch...finally` in diesem Beispiel sind zu kompliziert, um sie hier zu beschreiben. Die einfache Version ist, dass der Code im `finally` Block immer ausgeführt wird.

Betrachten wir dies aus der Perspektive des Ressourcenmanagements:

- Wir deklarieren die "Ressource" (dh `reader`) vor dem `try` Block, so dass sie für den `finally` Block gültig ist.
- Mit dem `new FileReader(...)` kann der `catch` alle `IOError` Ausnahmen behandeln, die beim Öffnen der Datei ausgelöst werden.
- Wir benötigen ein `reader.close()` im `finally` Block, da es einige Ausnahmepfade gibt, die wir weder im `try` Block noch im `catch` Block `catch` .
- Da jedoch vor der Initialisierung des `reader` *möglicherweise* eine Ausnahme ausgelöst wurde, benötigen wir auch einen expliziten `null` .
- Schließlich kann der Aufruf von `reader.close()` (hypothetisch) eine Ausnahme `reader.close()` . Das interessiert uns nicht, aber wenn wir die Ausnahme nicht an der Quelle einfangen, müssen wir uns weiter oben im Aufrufstapel befassen.

Java SE 7

Java 7 und höher bieten eine alternative Try-with-Resources-Syntax, die die Ressourcenbereinigung erheblich vereinfacht.

Die 'throws'-Klausel in einer Methodendeklaration

Java *geprüfte Ausnahme* Mechanismus erfordert die Programmierer , dass bestimmte Methoden zu erklären specifed geprüfte Ausnahmen werfen *konnte.* Dies geschieht mit der `throws` Klausel. Zum Beispiel:

```
public class OddNumberException extends Exception { // a checked exception
}

public void checkEven(int number) throws OddNumberException {
    if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

Das `throws OddNumberException` erklärt , dass ein Aufruf `checkEven` *könnte* eine Ausnahme auslösen , die vom Typ ist `OddNumberException` .

Eine `throws` Klausel kann eine Liste von Typen deklarieren und ungeprüfte Ausnahmen sowie geprüfte Ausnahmen enthalten.

```
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();
    }
}
```

Was ist der Sinn, ungeprüfte Ausnahmen als geworfen zu erklären?

Die `throws` Klausel in einer Methodendeklaration dient zwei Zwecken:

1. Dem Compiler wird mitgeteilt, welche Ausnahmen ausgelöst werden, damit der Compiler nicht erfasste (geprüfte) Ausnahmen als Fehler melden kann.
2. Sie teilt einem Programmierer mit, der Code schreibt, der die Methode aufruft, welche Ausnahmen zu erwarten sind. Zu diesem Zweck ist es oft sinnvoll, ungeprüfte Ausnahmen in eine `throws` .

Hinweis: Die `throws` wird auch vom Javadoc-Tool beim Generieren der API-Dokumentation und von den typischen "Hover-Text" -Methode-Tipps der IDE verwendet.

Würfe und Überschreiben der Methode

Die `throws` Klausel bildet einen Teil der Signatur einer Methode zum Zwecke des Überschreibens von Methoden. Eine Überschreibungsmethode kann mit derselben Gruppe von aktivierten Ausnahmen wie mit der überschriebenen Methode oder mit einer Teilmenge deklariert werden. Die Überschreibungsmethode kann jedoch keine zusätzlichen geprüften Ausnahmen hinzufügen. Zum Beispiel:

```
@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
```

Der Grund für diese Regel besteht darin, dass die Überschreibungsmethode die Typenersetzbarkeit beeinträchtigen könnte, wenn eine überschriebene Methode eine geprüfte Ausnahme auslösen kann, die die überschriebene Methode nicht auslösen konnte.

Ausnahmen und Ausnahmebehandlung online lesen:
https://riptutorial.com/de/java/topic/89/ausnahmen-und-ausnahmebehandlung

Kapitel 16: Autoboxing

Einführung

Autoboxing ist die automatische Konvertierung, die der Java-Compiler zwischen primitiven Typen und den entsprechenden Objekt-Wrapper-Klassen vornimmt. Beispiel: Konvertierung von int -> Integer, double -> Double ... Wenn die Konvertierung anders verläuft, wird dies als Unboxing bezeichnet. In der Regel wird dies in Sammlungen verwendet, die nur Objekte enthalten können, wobei primitive Boxtypen erforderlich sind, bevor sie in der Sammlung festgelegt werden.

Bemerkungen

Autoboxing kann bei häufiger Verwendung in Ihrem Code zu Leistungsproblemen führen.

- http://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html
- Integer-Auto-Unboxing und Auto-Boxing führen zu Leistungsproblemen?

Examples

Int und Integer austauschbar verwenden

Wenn Sie generische Typen mit Dienstprogrammklassen verwenden, stellen Sie oft fest, dass Nummerntypen nicht sehr hilfreich sind, wenn sie als Objekttypen angegeben werden, da sie nicht ihren primitiven Gegenstücken entsprechen.

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

Java SE 7

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

Glücklicherweise können Ausdrücke, die als `int` ausgewertet werden, anstelle einer `Integer` wenn sie benötigt wird.

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

Die `ints.add(i);` Aussage ist äquivalent zu:

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

Und behält Eigenschaften von `Integer#valueOf` z. B. dass die gleichen `Integer` Objekte von der JVM zwischengespeichert werden, wenn sie sich innerhalb des Zahlen-Caching-Bereichs befinden.

Das gilt auch für:

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

In mehrdeutigen Situationen ist jedoch Vorsicht geboten. Betrachten Sie den folgenden Code:

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

Die `java.util.List` Schnittstelle enthält sowohl ein `remove(int index)` (`List` Interface - Methode) und `remove(Object o)` (Methode von vererbten `java.util.Collection`). In diesem Fall findet kein Boxing statt und `remove(int index)` wird aufgerufen.

Ein weiteres Beispiel für ein merkwürdiges Verhalten von Java-Code, das durch die automatische Integierung von Autoboxen mit Werten im Bereich von `-128` bis `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
```

Dies geschieht, weil der Operator `>=` implizit `intValue()` das `int` zurückgibt, während `==` die **Referenzen** und nicht die `int` Werte vergleicht.

Standardmäßig speichert Java Werte im Bereich `[-128, 127]` , sodass der Operator `==` funktioniert, da die `Integers` Werte in diesem Bereich auf dieselben Objekte verweisen, wenn ihre Werte gleich sind. Der maximale Wert des zwischengespeicherten Bereichs kann mit der JVM-Option `-XX:AutoBoxCacheMax` definiert werden. Wenn Sie das Programm mit `-XX:AutoBoxCacheMax=1000` , wird der folgende Code also `true` `-XX:AutoBoxCacheMax=1000` :

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

Boolesche Anweisung in if verwenden

Aufgrund des automatischen Unboxing kann ein `Boolean` in einer `if` Anweisung verwendet werden:

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

Das funktioniert auch für `while` , `do while` und die Bedingung in den `for` Anweisungen.

Wenn der `Boolean` `NullPointerException` `null` , wird eine `NullPointerException` in die Konvertierung geworfen.

Auto-Unboxing kann zu NullPointerException führen

Dieser Code kompiliert:

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

Es stürzt jedoch zur Laufzeit mit einer `java.lang.NullPointerException` in der zweiten Zeile ab.

Das Problem ist, dass ein primitives `int` keinen `null` kann.

Dies ist ein minimalistisches Beispiel, aber in der Praxis manifestiert es sich oft in komplexeren Formen. Die `NullPointerException` ist nicht sehr intuitiv und hilft oft bei der Suche nach solchen Fehlern.

Stellen Sie sicher, dass Autoboxing und Auto-Unboxing mit Vorsicht ausgeführt werden, und stellen Sie sicher, dass nicht gepackte Werte zur Laufzeit keine `null` haben.

Speicher- und Rechenaufwand für Autoboxing

Autoboxing kann einen erheblichen Speicheraufwand verursachen. Zum Beispiel:

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

verbraucht normalerweise eine beträchtliche Menge an Speicher (etwa 60 KB für 6 KB der tatsächlichen Daten).

Darüber hinaus erfordern Boxed Integer normalerweise zusätzliche Roundtrips im Speicher, wodurch die CPU-Caches weniger effektiv werden. In obigem Beispiel ist der Speicher, auf den zugegriffen wird, auf fünf verschiedene Speicherorte verteilt, die sich in ganz unterschiedlichen Speicherbereichen befinden können: 1. das `HashMap` Objekt, 2. das Map-Objekt `Entry[] table` , 3. das `Entry` Objekt, 4. das entrys `key` (Box der primitiven Schlüssel), 5. das entrys `value` Objekt (Box den Grundwertes).

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

```
}
```

Das Lesen von `boxed` erfordert zwei Speicherzugriffe, wobei nur einer auf das `primitive` zugreifen kann.

Beim Abrufen von Daten aus dieser Karte der scheinbar unschuldige Code

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

ist äquivalent zu:

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

Normalerweise verursacht der obige Code die *Erstellung und Garbage Collection* eines `Integer` Objekts für jede `Map#get(Integer)` -Operation. (Weitere Informationen finden Sie im Hinweis unten.)

Um diesen Aufwand zu reduzieren, bieten mehrere Bibliotheken optimierte Sammlungen für primitive Typen, für die *kein* Boxen erforderlich ist. Zusätzlich zum Vermeiden des Box-Overheads erfordert diese Sammlung etwa viermal weniger Speicher pro Eintrag. Java Hotspot *kann* zwar das Autoboxing durch Arbeiten mit Objekten auf dem Stapel statt mit dem Heap optimieren, es ist jedoch nicht möglich, den Speicheraufwand und die daraus resultierende Speicherumleitung zu optimieren.

Java 8-Streams verfügen außerdem über optimierte Schnittstellen für primitive Datentypen, z. B. `IntStream` , für die kein Boxen erforderlich ist.

Hinweis: Bei einer typischen Java-Laufzeitumgebung wird ein einfacher Cache aus `Integer` und anderen primitiven Wrapper-Objekten verwaltet, der von den Factory-Methoden von `valueOf` und von Autoboxing verwendet wird. Für `Integer` liegt der Standardbereich dieses Caches zwischen -128 und +127. Einige JVMs bieten eine JVM-Befehlszeilenoption zum Ändern der Cachegröße / des Cache-Bereichs.

Verschiedene Fälle, wenn Integer und Int austauschbar verwendet werden können

Fall 1: Bei Verwendung anstelle von Methodenargumenten.

Wenn für eine Methode ein Objekt der Wrapper-Klasse als Argument erforderlich ist, kann dem Argument eine Variable des jeweiligen primitiven Typs übergeben werden und umgekehrt.

Beispiel:

```
int i;
```

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

Fall 2: Beim Übergeben von Rückgabewerten:

Wenn eine Methode eine primitive Typvariable zurückgibt, kann ein Objekt der entsprechenden Wrapper-Klasse austauschbar als Rückgabewert und umgekehrt übergeben werden.

Beispiel:

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

Fall 3: Während der Durchführung von Operationen.

Bei jeder Ausführung von Operationen mit Zahlen können die primitive Typvariable und das Objekt der jeweiligen Wrapper-Klasse austauschbar verwendet werden.

```
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
```

Fallstricke : Denken Sie daran, ein Objekt der Wrapper-Klasse zu initialisieren oder einem Wert zuzuweisen.

Wenn Sie das Wrapper-Klassenobjekt und die primitive Variable austauschbar verwenden, vergessen Sie niemals, das Wrapper-Klassenobjekt zu initialisieren oder ihm einen Wert zuzuweisen. Andernfalls kann es zur Laufzeit zu einer Nullzeiger-Ausnahme kommen.

Beispiel:

```
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
    }
```

Im obigen Beispiel ist der Wert des Objekts nicht zugewiesen und nicht initialisiert, und daher wird

das Programm zur Laufzeit in einer Nullzeiger-Ausnahme ausgeführt. Wie aus dem obigen Beispiel hervorgeht, sollte der Wert des Objekts niemals uninitialisiert und nicht zugewiesen bleiben.

Autoboxing online lesen: https://riptutorial.com/de/java/topic/138/autoboxing

Kapitel 17: Befehlszeile Argumentverarbeitung

Syntax

- public static void main (String [] args)

Parameter

Parameter	Einzelheiten
args	Die Befehlszeilenargumente. Unter der Annahme, dass die `main` vom Java-Launcher aufgerufen wird, ist `args` nicht null und enthält keine `null` .

Bemerkungen

Wenn eine reguläre Java-Anwendung mit dem `java` Befehl (oder einem gleichwertigen Befehl) gestartet wird, wird eine `main` aufgerufen, die die Argumente von der Befehlszeile im Array `args` übergibt.

Leider bieten die Java SE-Klassenbibliotheken keine direkte Unterstützung für die Verarbeitung von Befehlsargumenten. Damit bleiben Ihnen zwei Alternativen:

- Implementieren Sie die Argumentverarbeitung von Hand in Java.
- Nutzen Sie eine Drittanbieter-Bibliothek.

Dieses Thema wird versuchen, einige der beliebtesten Drittanbieter-Bibliotheken zu behandeln. Eine ausführliche Liste der Alternativen finden Sie in dieser Antwort auf die StackOverflow-Frage "Wie werden Befehlszeilenargumente in Java analysiert?" .

Examples

Argumentverarbeitung mit GWT ToolBase

Wenn Sie komplexere Befehlszeilenargumente analysieren möchten, z. B. mit optionalen Parametern, sollten Sie den GWT-Ansatz von Google am besten verwenden. Alle Kurse sind öffentlich verfügbar unter:

https://gwt.googlesource.com/gwt/+/2.8.0-beta1/dev/core/src/com/google/gwt/util/tools/ToolBase.java

Ein Beispiel für die Handhabung des Befehlszeile- `myprogram -dir "~/Documents" -port 8888` lautet:

```
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` hat auch eine Methode `isRequired()` die überschrieben werden kann, um zu sagen, dass das Befehlszeilenargument erforderlich ist (Standardrückgabe ist `false` sodass das Argument optional ist.

Argumente von Hand bearbeiten

Wenn die Befehlszeilensyntax für eine Anwendung einfach ist, ist es sinnvoll, die Befehlsargumentverarbeitung vollständig in benutzerdefiniertem Code auszuführen.

In diesem Beispiel stellen wir eine Reihe einfacher Fallstudien vor. In jedem Fall gibt der Code Fehlermeldungen aus, wenn die Argumente nicht akzeptabel sind, und ruft dann `System.exit(1)` auf, um der Shell mitzuteilen, dass der Befehl fehlgeschlagen ist. (Wir gehen in jedem Fall davon aus, dass der Java-Code mit einem Wrapper mit dem Namen "myapp" aufgerufen wird.)

Ein Befehl ohne Argumente

In dieser Fallstudie erfordert der Befehl keine Argumente. Der Code veranschaulicht, dass wir mit `args.length` die Anzahl der Befehlszeilenargumente erhalten.

```
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");
    }
}
```

Ein Befehl mit zwei Argumenten

In dieser Fallstudie erfordert der Befehl genau zwei Argumente.

```
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]);
    }
}
```

Beachten Sie, dass der Befehl abstürzen würde, wenn der Benutzer `args.length` überprüft `args.length` , wenn er mit zu wenigen Befehlszeilenargumenten ausgeführt wurde.

Ein Befehl mit "Flag" -Optionen und mindestens einem Argument

In dieser Fallstudie hat der Befehl mehrere (optionale) Flag-Optionen und erfordert nach den Optionen mindestens ein Argument.

```
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);
    }
}
```

Wie Sie sehen, wird die Verarbeitung der Argumente und Optionen ziemlich umständlich, wenn die Befehlssyntax kompliziert ist. Es ist ratsam, eine "Kommandozeilenanalyse" -Bibliothek zu verwenden. siehe die anderen Beispiele.

Befehlszeile Argumentverarbeitung online lesen:
https://riptutorial.com/de/java/topic/4775/befehlszeile-argumentverarbeitung

Kapitel 18: Benchmarks

Einführung

Das Schreiben von Leistungsbenchmarks in Java ist nicht so einfach wie das `System.currentTimeMillis()` am Anfang und am Ende und Berechnen der Differenz. Um gültige Leistungsbenchmarks zu schreiben, sollte man geeignete Werkzeuge verwenden.

Examples

Einfaches JMH-Beispiel

Eines der Tools zum Schreiben richtiger Benchmark-Tests ist JMH . `HashSet` , wir möchten die Leistung eines `HashSet` in `HashSet` und `TreeSet` .

Der einfachste Weg, um JHM in Ihr Projekt zu integrieren - besteht darin, das Plug-in für Maven und Schatten zu verwenden . Sie können auch `pom.xml` aus JHM-Beispielen sehen .

```
<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>
```

Danach müssen Sie die Benchmark-Klasse selbst schreiben:

```
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));
    }
}
```

Bitte beachte dieses `blackhole.consume()` , wir werden später darauf zurückkommen. Wir brauchen auch die Hauptklasse für das Laufen von 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();
    }
}
```

Und wir sind fertig. Wir müssen nur das `mvn package` ausführen (es erstellt `benchmarks.jar` in Ihrem `/target` Ordner) und führen unseren Benchmark-Test aus:

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

Und nach einigen Aufwärm- und Berechnungs-Iterationen haben wir unsere Ergebnisse:

```
# 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
```

Über dieses `blackhole.consume()` . Wenn Ihre Berechnungen den Status Ihrer Anwendung nicht ändern, wird Java ihn höchstwahrscheinlich einfach ignorieren. Um dies zu vermeiden, können Sie Ihre Benchmark-Methoden entweder dazu bringen, einen bestimmten Wert zurückzugeben, oder das Objekt mit dem `Blackhole` Objekt verbrauchen.

Weitere Informationen zum Schreiben der richtigen Benchmarks finden Sie im Blog von Aleksey Shipilëv, im Blog von Jacob Jenkov und im Java-Performance-Blog: 1 , 2 .

Benchmarks online lesen: https://riptutorial.com/de/java/topic/9514/benchmarks

Kapitel 19: BigDecimal

Einführung

Die BigDecimal- Klasse bietet Operationen für Arithmetik (Addieren, Subtrahieren, Multiplizieren, Dividieren), Skalierungsmanipulation, Rundung, Vergleich, Hashing und Formatkonvertierung. Das BigDecimal steht für unveränderliche, beliebig genaue, vorzeichenbehaftete Dezimalzahlen. Diese Klasse wird für die Notwendigkeit einer hochgenauen Berechnung verwendet.

Examples

BigDecimal-Objekte sind unveränderlich

Wenn Sie mit BigDecimal berechnen möchten, müssen Sie den zurückgegebenen Wert verwenden, da BigDecimal-Objekte unveränderlich sind:

```
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
```

Vergleich von BigDecimals

Die Methode `compareTo` sollte zum Vergleichen von `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
```

Normalerweise sollten Sie die `equals` Methode **nicht** verwenden, da zwei `BigDecimals` nur dann als gleich betrachtet werden, wenn sie gleichwertig und ebenfalls **skaliert sind** :

```
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
```

Mathematische Operationen mit BigDecimal

Dieses Beispiel zeigt, wie grundlegende mathematische Operationen mit BigDecimals ausgeführt werden.

1.Zusatz

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

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

Ergebnis: 12

2. Subtraktion

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

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

Ergebnis: -2

3. Multiplikation

Beim Multiplizieren von zwei `BigDecimal` erhält das Ergebnis eine Skalierung, die der Summe der Skalen der Operanden entspricht.

```
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);
```

Ergebnis: 36.89931

Um den Maßstab des Ergebnisses zu ändern, verwenden Sie die überladene Multiplikationsmethode, mit der `MathContext` - ein Objekt, das die Regeln für Operatoren beschreibt, insbesondere die Genauigkeit und den Rundungsmodus des Ergebnisses. Weitere Informationen zu den verfügbaren Rundungsmodi finden Sie in der Oracle-Dokumentation.

```
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);
```

Ergebnis: 36,90

4.Division

Die Division ist etwas komplizierter als die anderen Rechenoperationen. Betrachten Sie beispielsweise das folgende Beispiel:

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

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

Wir würden erwarten, dass dies etwas ähnelt: 0.7142857142857143, aber wir würden bekommen:

> Ergebnis: **java.lang.ArithmeticException: Dezimale Erweiterung ohne Endung; kein genau darstellbares Dezimalergebnis.**

Dies würde perfekt funktionieren, wenn das Ergebnis eine abschließende Dezimalzahl wäre, wenn ich 5 durch 2 teilen wollte, aber für jene Zahlen, die nach der Division ein nicht abschließendes Ergebnis ergeben würden, würden wir eine `ArithmeticException` . Im realen Szenario kann man die Werte, die während der Division auftreten würden, nicht vorhersagen. Daher müssen die **Skalierung** und der **Rundungsmodus** für die Division BigDecimal angegeben werden. Weitere Informationen zum Skalierungs- und Rundungsmodus finden Sie in der Oracle-Dokumentation .

Zum Beispiel könnte ich Folgendes tun:

```
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);
```

> Ergebnis: 0.7142857143

5.Reminder oder Modul

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

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

> Ergebnis: 5

6.Power

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

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

Ergebnis: 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);
```

Ergebnis: 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);
```

Ergebnis: 5

9.Ziehpunkt nach links

```
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);
```

Ergebnis: 52.3449843776

10.Ziehpunkt nach rechts

```
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);
```

Ergebnis: 5234498.43776

Es gibt viele weitere Optionen und Kombinationen von Parametern für die oben genannten Beispiele (z. B. gibt es 6 Variationen der Divide-Methode). Dieser Satz ist nicht erschöpfend und enthält einige grundlegende Beispiele.

Verwenden Sie BigDecimal anstelle von Float

Aufgrund der Darstellung des Float-Typs im Computerspeicher können die Ergebnisse von Operationen, die diesen Typ verwenden, ungenau sein - einige Werte werden als Näherungen gespeichert. Gute Beispiele dafür sind monetäre Berechnungen. Wenn eine hohe Präzision erforderlich ist, sollten andere Typen verwendet werden. zB bietet Java 7 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));
}
```

Ausgabe dieses Programms ist:

```
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
```

Für ein Startguthaben von 10000,00 nach 1000 Operationen für 1,99 erwarten wir, dass das Guthaben 8010,00 beträgt. Die Verwendung des Float-Typs gibt uns eine Antwort um 8009,77, was bei monetären Berechnungen unannehmbar ungenau ist. Die Verwendung von BigDecimal liefert das richtige Ergebnis.

BigDecimal.valueOf ()

Die BigDecimal-Klasse enthält einen internen Cache mit häufig verwendeten Nummern, z. B. 0 bis 10. Die BigDecimal.valueOf () -Methoden werden Konstruktoren mit ähnlichen Typparametern vorgezogen, dh im folgenden Beispiel wird a bevorzugt 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
```

Initialisierung von BigDecimals mit dem Wert Null, Eins oder Zehn

BigDecimal bietet statische Eigenschaften für die Zahlen Null, Eins und Zehn. Es ist empfehlenswert, diese anstelle der tatsächlichen Zahlen zu verwenden:

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

Durch die Verwendung der statischen Eigenschaften vermeiden Sie eine unnötige Instantiierung. Außerdem enthält Ihr Code ein Literal anstelle einer 'magischen Zahl'.

```
//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;
```

BigDecimal online lesen: https://riptutorial.com/de/java/topic/1667/bigdecimal

Kapitel 20: BigInteger

Einführung

Die `BigInteger` Klasse wird für mathematische Operationen verwendet, bei denen große Ganzzahlen mit zu großen Beträgen für primitive Datentypen verwendet werden. Zum Beispiel sind 100-Fakultäten 158-stellig - viel größer als ein `long` kann. `BigInteger` bietet Analoga für alle primitiven Integer-Operatoren von Java und alle relevanten Methoden von `java.lang.Math` sowie einige andere Operationen.

Syntax

- BigInteger Variablenname = neuer BigInteger ("12345678901234567890"); // eine dezimale Ganzzahl als Zeichenfolge
- BigInteger Variablenname = new BigInteger ("10101011010101001010100110001100111010110001111100001010101010010"), 2) // eine binäre Ganzzahl als Zeichenfolge
- BigInteger Variablenname = new BigInteger ("ab54a98ceb1f0800", 16) // eine hexadezimale Ganzzahl als Zeichenfolge
- BigInteger Variablenname = new BigInteger (64, new Random ()); // ein Pseudozufallszahlengenerator, der 64 Bits liefert, um eine Ganzzahl zu konstruieren
- BigInteger Variablenname = new BigInteger (neues Byte [] {0, -85, 84, -87, -116, -21, 31, 10, -46}); // signierte Zweierkomplementdarstellung einer Ganzzahl (Big Endian)
- BigInteger Variablenname = new BigInteger (1, neues Byte [] {- 85, 84, -87, -116, -21, 31, 10, -46}); // vorzeichenlose Zweierkomplementdarstellung einer positiven Ganzzahl (Big Endian)

Bemerkungen

`BigInteger` ist unveränderlich. Daher kann man seinen Zustand nicht ändern. Das Folgende funktioniert beispielsweise nicht, da die `sum` aufgrund von Unveränderlichkeit nicht aktualisiert wird.

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

Weisen Sie das Ergebnis der `sum` zu, damit es funktioniert.

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

Java SE 8

In der offiziellen Dokumentation zu `BigInteger` heißt es, dass `BigInteger` Implementierungen alle

Ganzzahlen zwischen -2 ^{2147483647} und 2 ^{2147483647} (exklusiv) unterstützen sollen. Dies bedeutet, dass `BigInteger` mehr als 2 *Milliarden* Bits haben kann!

Examples

Initialisierung

Die Klasse `java.math.BigInteger` stellt Operationsanaloga für alle primitiven Ganzzahloperatoren von Java und für alle relevanten Methoden aus `java.lang.Math` . Da das Paket `java.math` nicht automatisch verfügbar ist, müssen Sie möglicherweise `java.math.BigInteger` importieren, bevor Sie den einfachen Klassennamen verwenden können.

Um `long` oder `int` Werte in `BigInteger` zu konvertieren, verwenden Sie:

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

oder für ganze Zahlen:

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

der das *verbreitern* `intValue` Ganzzahl zu langen, Vorzeichenbit - Erweiterung für negative Werte verwendet, so dass negative Werte negativ bleiben.

So konvertieren Sie einen numerischen `String` in `BigInteger` :

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

Der folgende Konstruktor wird verwendet, um die String-Darstellung eines `BigInteger` in der angegebenen Basis in einen `BigInteger` .

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

Java unterstützt auch die direkte Konvertierung von Bytes in eine Instanz von `BigInteger` . Derzeit können nur signierte und nicht signierte Big Endian-Codierungen verwendet werden:

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

Dadurch wird eine `BigInteger` Instanz mit dem Wert -128 generiert, da das erste Bit als Vorzeichenbit interpretiert wird.

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

Dadurch wird eine `BigInteger` Instanz mit dem Wert 128 generiert, da die Bytes als vorzeichenlose Zahl interpretiert werden und das Vorzeichen explizit auf 1 (positive Zahl) gesetzt wird.

Es gibt vordefinierte Konstanten für allgemeine Werte:

- `BigInteger.ZERO` - Wert von "0".
- `BigInteger.ONE` - Wert von "1".
- `BigInteger.TEN` - Wert von "10".

Es gibt auch `BigInteger.TWO` (Wert von "2"), aber Sie können es nicht in Ihrem Code verwenden, da es `private` .

BigIntegers vergleichen

Sie können `BigIntegers` genauso vergleichen, wie Sie `String` oder andere Objekte in Java vergleichen.

Zum Beispiel:

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

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

Ausgabe:

```
Not Equal
```

Hinweis:

Verwenden Sie im Allgemeinen **nicht** den Operator `==` , um BigIntegers zu vergleichen

- `==` Operator: vergleicht Referenzen; ob sich zwei Werte auf dasselbe Objekt beziehen
- `equals()` -Methode: vergleicht den Inhalt von zwei BigIntegers.

Beispielsweise sollten BigIntegers **nicht** auf folgende Weise verglichen werden:

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

Dies kann zu einem unerwarteten Verhalten führen, da der Operator `==` nur die Referenzgleichheit überprüft. Wenn beide BigIntegers denselben Inhalt enthalten, sich aber nicht auf dasselbe Objekt beziehen, schlägt **dies fehl.** Vergleichen Sie stattdessen BigIntegers mithilfe der Methoden `equals` (siehe oben).

Sie können Ihren `BigInteger` mit konstanten Werten wie 0,1,10 vergleichen.

zum Beispiel:

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

Sie können auch zwei BigIntegers mit der Methode `compareTo()` wie folgt vergleichen: `compareTo()` liefert 3 Werte.

- **0:** Wenn beide **gleich sind** .
- **1:** Wenn der erste Wert **größer als der** zweite ist (der in Klammern).
- **-1:** Wenn erster als **weniger ist** .

```
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.
}
```

BigInteger-Beispiele für mathematische Operationen

BigInteger befindet sich in einem unveränderlichen Objekt. Sie müssen also die Ergebnisse aller mathematischen Operationen einer neuen BigInteger-Instanz zuordnen.

Zugabe: *10 + 10 = 20*

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

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

Ausgabe: 20

Substraktion: *10 - 9 = 1*

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

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

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

Ausgabe: 1

Division: *10/5 = 2*

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

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

Ausgabe: 2

Division: *17/4 = 4*

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

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

Ausgabe: 4

Multiplikation: *10 * 5 = 50*

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

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

Ausgabe: 50

Leistung: *10 ^ 3 = 1000*

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

Ausgabe: 1000

Rest: *10% 6 = 4*

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

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

Ausgabe: 4

GCD: Der größte gemeinsame Teiler (GCD) für `12` und `18` ist `6` .

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

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

Ausgabe: 6

Maximal zwei BigIntegers:

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

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

Ausgabe: 11

Minimum von zwei BigIntegers:

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

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

Ausgabe: 10

Binäre Logikoperationen auf BigInteger

BigInteger unterstützt die binären Verknüpfungen, die auch für `Number` Typen verfügbar sind. Wie bei allen Operationen werden sie durch Aufrufen einer Methode implementiert.

Binär oder:

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

val1.or(val2);
```

Ausgabe: 11 (entspricht `10 | 9`)

Binär und:

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

val1.and(val2);
```

Ausgabe: 8 (entspricht `10 & 9`)

Binäres Xor:

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

val1.xor(val2);
```

Ausgabe: 3 (entspricht `10 ^ 9`)

Rechte Shifttaste:

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

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

Ausgabe: 5 (entspricht `10 >> 1`)

Linksverschiebung:

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

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

Ausgabe: 20 (entspricht `10 << 1`)

Binäre Inversion (nicht):

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

val1.not();
```

Ausgabe: 5

*NAND (And-Not): **

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

val1.andNot(val2);
```

Ausgabe: 7

Zufällige BigIntegers generieren

Die `BigInteger` Klasse verfügt über einen Konstruktor, der zum Generieren von zufälligen `BigIntegers` , wenn eine Instanz von `java.util.Random` und ein `int` angegeben wird, die angibt, wie viele Bits `BigInteger` soll. Die Verwendung ist recht einfach - wenn Sie den Konstruktor `BigInteger(int, Random)` wie folgt aufrufen:

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

Dann erhalten Sie einen `BigInteger` dessen Wert zwischen 0 (einschließlich) und 2 `bitCount` (exklusiv) liegt.

Dies bedeutet auch, dass der `new BigInteger(2147483647, sourceOfRandomness)` alle positiven `BigInteger` `new BigInteger(2147483647, sourceOfRandomness)` kann, wenn er genügend Zeit hat.

Was wird der `sourceOfRandomness` sein, liegt bei Ihnen. Zum Beispiel ist ein `new Random()` in den meisten Fällen gut genug:

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

Wenn Sie bereit sind, Geschwindigkeit für Zufallszahlen höherer Qualität aufzugeben, können Sie stattdessen ein `new SecureRandom ()` verwenden:

```
import java.security.SecureRandom;

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

Sie können sogar einen Algorithmus mit einer anonymen Klasse direkt implementieren! Beachten Sie, dass Sie **Ihren eigenen RNG** - **Algorithmus Ausrollen *werden* Sie mit geringer Qualität Zufälligkeit am Ende,** also immer sicher sein , einen Algorithmus zu verwenden , die anständig erwiesen ist , wenn Sie die resultierende wollen `BigInteger` (n) berechenbar sein.

```
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;
    }
});
```

BigInteger online lesen: https://riptutorial.com/de/java/topic/1514/biginteger

Kapitel 21: Bilder programmgesteuert erstellen

Bemerkungen

`BufferedImage.getGraphics()` immer `Graphics2D` .

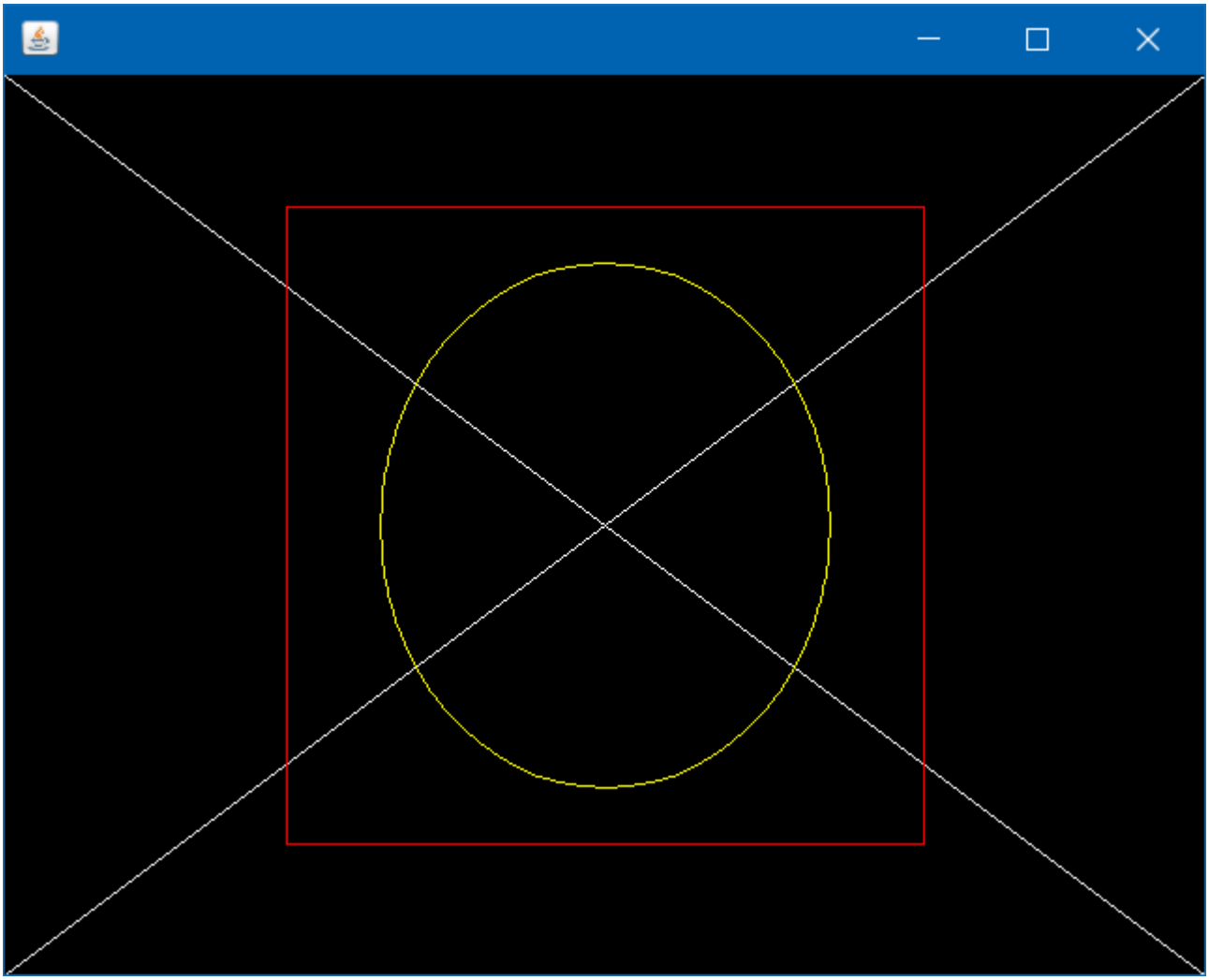
Die Verwendung von `VolatileImage` kann die Geschwindigkeit von Zeichenvorgängen erheblich verbessern, hat jedoch auch Nachteile: Der Inhalt kann jederzeit verloren gehen und muss möglicherweise von Grund auf neu erstellt werden.

Examples

Ein einfaches Bild programmgesteuert erstellen und anzeigen

```
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);
    }
}
```



Speichern Sie ein Image auf der Festplatte

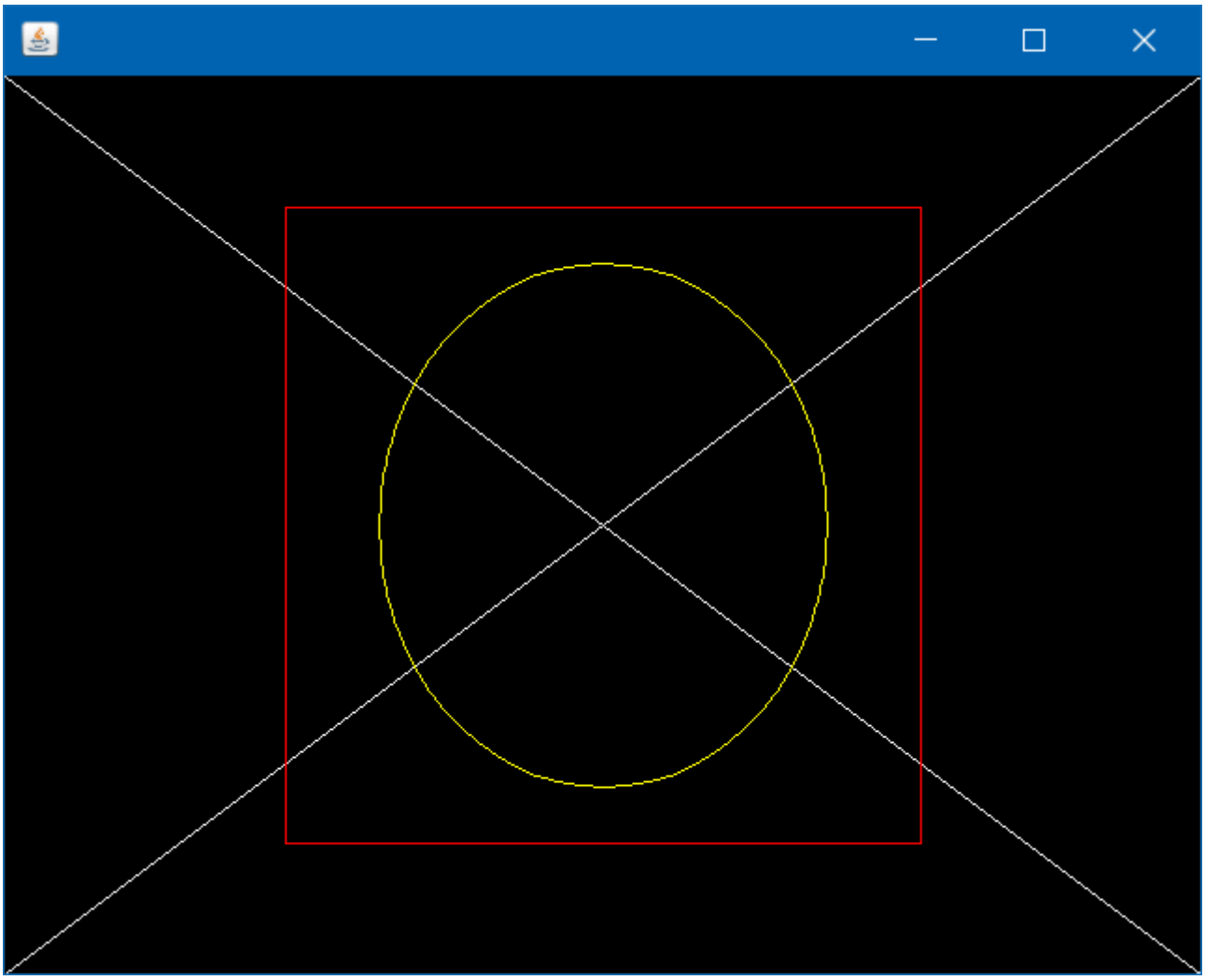
```
public static void saveImage(String destination) throws IOException {
    // method implemented in "Creating a simple image Programmatically and displaying it"
example
    BufferedImage img = createSampleImage();

    // ImageIO provides several write methods with different outputs
    ImageIO.write(img, "png", new File(destination));
}
```

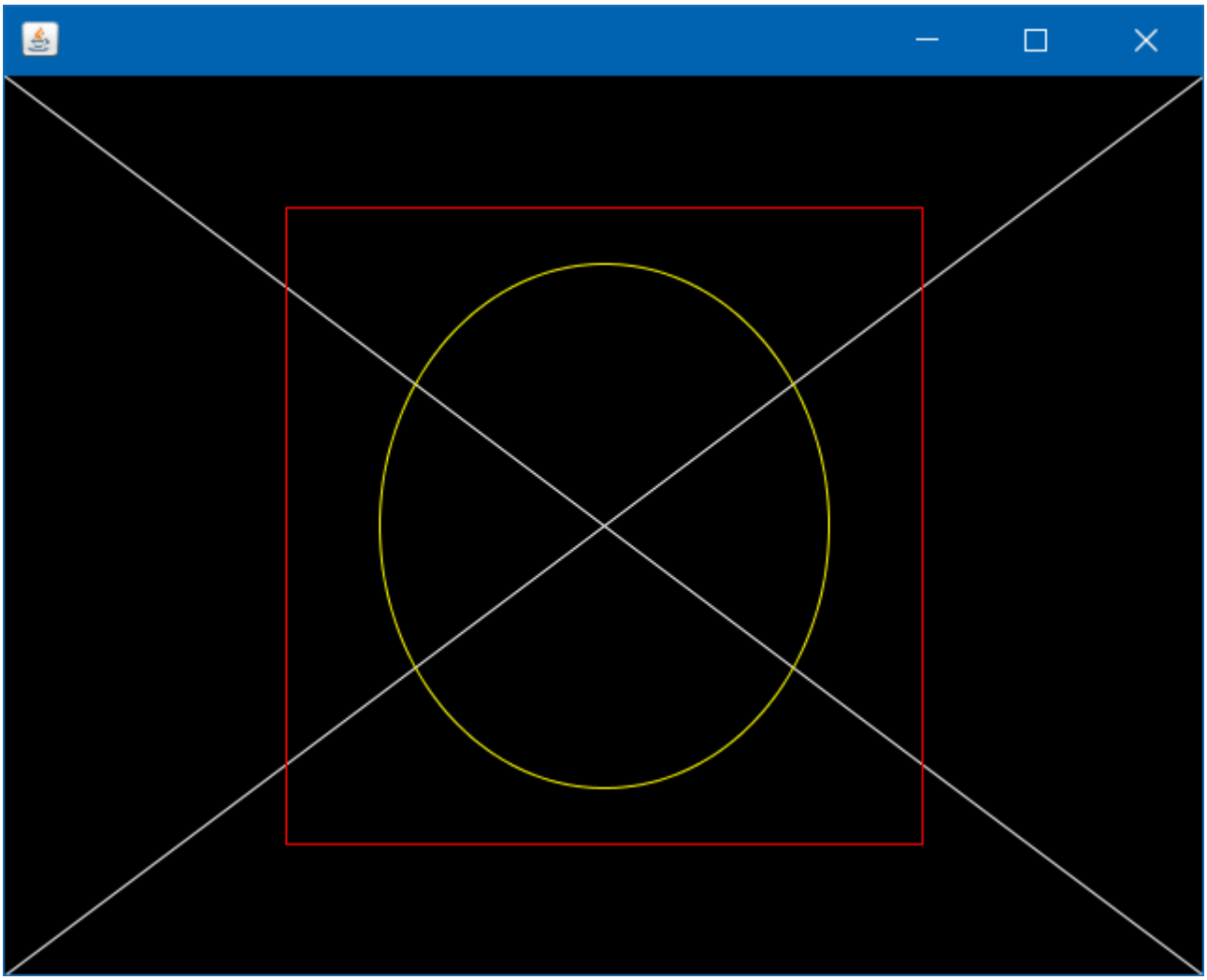
Festlegen der Bildwiedergabequalität

```
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);
}
```

Ein Vergleich von QUALITY und SPEED Rendering des Beispielbildes:





Erstellen eines Bildes mit der BufferedImage-Klasse

```
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"));
```

Ausgabe:

Bild mit BufferedImage bearbeiten und erneut verwenden

```
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"));
```

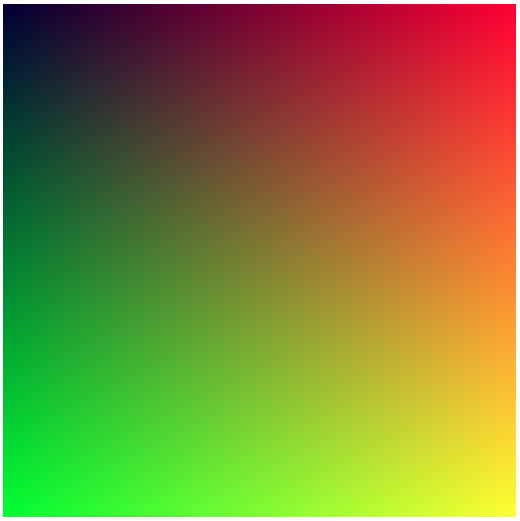
Ursprüngliche Katzendatei:

Produzierte Datei:

Festlegen der Farbe einzelner Pixel in BufferedImage

```
BufferedImage image = new BufferedImage(256, 256, BufferedImage.TYPE_INT_ARGB);

//you don't have to use the Graphics object, you can read and set pixel color individually
for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 256; j++) {
        int alpha = 255; //don't forget this, or use BufferedImage.TYPE_INT_RGB instead
        int red = i; //or any formula you like
        int green = j; //or any formula you like
        int blue = 50; //or any formula you like
        int color = (alpha << 24) | (red << 16) | (green << 8) | blue;
        image.setRGB(i, j, color);
    }
}

ImageIO.write(image, "png", new File("computed.png"));
```

Ausgabe:



So skalieren Sie ein BufferedImage

```
/**
* Resizes an image using a Graphics2D object backed by a BufferedImage.
```

```
 * @param srcImg - source image to scale
 * @param w - desired width
 * @param h - desired height
 * @return - the new resized image
 */
private BufferedImage getScaledImage(Image srcImg, int w, int h){

    //Create a new image with good size that contains or might contain arbitrary alpha values
between and including 0.0 and 1.0.
    BufferedImage resizedImg = new BufferedImage(w, h, BufferedImage.TRANSLUCENT);

    //Create a device-independant object to draw the resized image
    Graphics2D g2 = resizedImg.createGraphics();

    //This could be changed, Cf. http://stackoverflow.com/documentation/java/5482/creating-
images-programmatically/19498/specifying-image-rendering-quality
    g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
RenderingHints.VALUE_INTERPOLATION_BILINEAR);

    //Finally draw the source image in the Graphics2D with the desired size.
    g2.drawImage(srcImg, 0, 0, w, h, null);

    //Disposes of this graphics context and releases any system resources that it is using
    g2.dispose();

    //Return the image used to create the Graphics2D
    return resizedImg;
}
```

Bilder programmgesteuert erstellen online lesen: https://riptutorial.com/de/java/topic/5482/bilder-programmgesteuert-erstellen

Kapitel 22: Bit-Manipulation

Bemerkungen

- Im Gegensatz zu C / C ++ ist Java in Bezug auf die zugrunde liegende Maschinenhardware vollständig endianeutral. Standardmäßig erhalten Sie kein großes oder kleines Endian-Verhalten. Sie müssen explizit angeben, welches Verhalten Sie möchten.
- Der `byte` Typ ist mit einem Bereich von -128 bis +127 vorzeichenbehaftet. Um einen Byte-Wert in sein vorzeichenloses Äquivalent zu konvertieren, maskieren Sie ihn mit 0xFF wie folgt: `(b & 0xFF)` .

Examples

Werte als Bitfragmente packen / entpacken

Es ist üblich, dass die Speicherleistung mehrere Werte in einen einzigen Grundwert komprimiert. Dies kann nützlich sein, um verschiedene Informationen in eine einzige Variable zu übergeben.

Zum Beispiel kann man 3 Bytes - z. B. Farbcode in RGB - in ein einzelnes int packen.

Werte packen

```
// 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;
```

Werte auspacken

```
// 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)
};
```

Einzelne Bits prüfen, einstellen, löschen und umschalten. Verwenden Sie als Bitmaske

Angenommen, wir möchten das Bit `n` eines Ganzzahlprimitivs ändern (`i` (Byte, Kurzzeichen, Zeichen, Ganzzahl oder Langzeichen)

```
(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'
```

Long / int / short / byte als Bitmaske verwenden:

```
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));
    }
}
```

Drucke

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

welche der Maske entspricht, die wir als `checkBitMask` Parameter übergeben haben: `FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55` .

Die Kraft von 2 ausdrücken

Um die Potenz von 2 (2 ^ n) von ganzen Zahlen auszudrücken, kann eine Bitverschiebungsoperation verwendet werden, die es erlaubt, das `n` explizit anzugeben.

Die Syntax lautet im Wesentlichen:

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

Beispiele:

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

Dies ist besonders nützlich, wenn Sie konstante Werte definieren, aus denen hervorgeht, dass anstelle von hexadezimalen oder dezimalen Werten eine Potenz von 2 verwendet wird.

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

Eine einfache Methode zur Berechnung der int Potenz von 2 wäre

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

Überprüfen, ob eine Zahl eine Potenz von 2 ist

Wenn eine ganze Zahl `x` eine Potenz von 2 ist, wird nur ein Bit gesetzt, während bei `x-1` alle Bits danach gesetzt sind. Zum Beispiel: `4` ist `100` und `3` ist `011` als Binärzahl, was die oben genannte Bedingung erfüllt. Null ist keine Potenz von 2 und muss explizit geprüft werden.

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

Verwendung für Left und Right Shift

Nehmen wir an, wir haben drei Arten von Berechtigungen: **READ** , **WRITE** und **EXECUTE** . Jede Berechtigung kann zwischen 0 und 7 liegen. (Angenommen, ein 4-Bit-Zahlensystem)

> RESOURCE = READ WRITE EXECUTE (12-Bit-Nummer)
>
> RESOURCE = 0100 0110 0101 = 4 6 5 (12-Bit-Nummer)

Wie können wir die (12-Bit-Nummer) Berechtigungen erhalten, die oben gesetzt sind (12-Bit-

Nummer)?

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

Auf diese Weise können wir die **EXECUTE-** Berechtigungen der **Ressource erhalten** . Nun, was ist, wenn wir **READ-** Berechtigungen der **Ressource erhalten** möchten?

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

Recht? Vermutlich nehmen Sie das an? Berechtigungen ergeben sich jedoch in 1024. Wir möchten nur READ-Berechtigungen für die Ressource erhalten. Keine Sorge, deshalb hatten wir Schichtbetreiber. Wenn wir sehen, liegen die READ-Berechtigungen 8 Bit hinter dem tatsächlichen Ergebnis. Wenn Sie also einen Schichtoperator anwenden, der READ-Berechtigungen ganz rechts neben dem Ergebnis bringt? Was ist, wenn wir tun:

> 0100 0000 0000 >> 8 => 0000 0000 0100 (Da es sich um eine positive Zahl handelt, die durch 0 ersetzt wird, verwenden Sie einfach den vorzeichenlosen Rechtsverschiebungsoperator.)

Wir haben jetzt tatsächlich die **Leseberechtigung** , die 4 ist.

Nun erhalten wir beispielsweise die **Berechtigungen READ** , **WRITE** , **EXECUTE** für eine **Ressource** . Was können wir tun, um Berechtigungen für diese **Ressource zu erstellen** ?

Nehmen wir zunächst das Beispiel für binäre Berechtigungen. (Immer noch ein 4-Bit-Zahlensystem vorausgesetzt)

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

Wenn Sie denken, dass wir einfach tun werden:

`READ | WRITE | EXECUTE` , Sie haben etwas recht, aber nicht genau. Was passiert, wenn wir READ | durchführen SCHREIBEN | AUSFÜHREN

0001 | 0100 | 0110 => 0111

Berechtigungen werden jedoch (in unserem Beispiel) tatsächlich als 0001 0100 0110 dargestellt

Um dies zu tun, wissen wir, dass **READ** 8 Bits hinter, **WRITE** 4 Bits hinter und **PERMISSIONS** an

letzter Stelle platziert werden. Das Zahlensystem, das für **RESOURCE-** Berechtigungen verwendet wird, ist tatsächlich 12 Bit (in unserem Beispiel). Sie kann in verschiedenen Systemen unterschiedlich sein (werden).

(LESEN SIE << 8) | (SCHREIBEN SIE << 4) | (AUSFÜHREN)

0000 0000 0001 << 8 (READ)

0001 0000 0000 (Linksverschiebung um 8 Bit)

0000 0000 0100 << 4 (SCHREIBEN)

0000 0100 0000 (Linksverschiebung um 4 Bit)

0000 0000 0001 (AUSFÜHRUNG)

Wenn wir nun die Ergebnisse der obigen Verschiebung hinzufügen, wird es ungefähr so sein;

0001 0000 0000 (READ)

0000 0100 0000 (SCHREIBEN)

0000 0000 0001 (AUSFÜHRUNG)

0001 0100 0001 (Berechtigungen)

java.util.BitSet-Klasse

Seit 1.7 gibt es eine java.util.BitSet- Klasse, die eine einfache und benutzerfreundliche Bit-Speicher- und Bearbeitungsschnittstelle bietet:

```
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` implementiert `Clonable` und `Serializable` . Unter der Haube werden alle `Clonable` im Feld `long[] words` gespeichert, das automatisch erweitert wird.

Es unterstützt auch ganze Satz logische Operationen `and` , `or` , `xor` , `andNot` :

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

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

Signiert gegen unsignierte Schicht

In Java sind alle Zahlenprimitive signiert. Zum Beispiel kann ein int immer Werte aus [-2 ^ 31 - 1, 2 ^ 31] darstellen, wobei das erste Bit den Wert -1 für den negativen Wert und 0 für den positiven Wert signiert.

Grundlegende Schichtoperatoren `>>` und `<<` sind signierte Operatoren. Sie werden das Vorzeichen des Wertes erhalten.

Es ist jedoch üblich, dass Programmierer Zahlen verwenden, um *vorzeichenlose Werte* zu speichern. Für ein int bedeutet dies, den Bereich auf [0, 2 ^ 32 - 1] zu verschieben, um einen doppelt so großen Wert wie mit einem vorzeichenbehafteten int zu haben.

Für diese Power-User bedeutet das Bit für Vorzeichen keine Bedeutung. Aus diesem Grund fügte Java `>>>` hinzu, einen Operator für die Linksverschiebung, der dieses Vorzeichen-Bit ignorierte.

```
                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)
```

Warum gibt es kein `<<<` ?

Dies ergibt sich aus der beabsichtigten Definition der Rechtsverschiebung. Da es die leeren Stellen auf der linken Seite ausfüllt, gibt es keine Entscheidung, was das Zeichenzeichen angeht. Infolgedessen sind keine zwei verschiedenen Bediener erforderlich.

Siehe diese Frage für eine detailliertere Antwort.

Bit-Manipulation online lesen: https://riptutorial.com/de/java/topic/1177/bit-manipulation

Kapitel 23: BufferedWriter

Syntax

- neuer BufferedWriter (Writer); // Der Standardkonstruktor
- BufferedWriter.write (int c); // Schreibt ein einzelnes Zeichen
- BufferedWriter.write (String str); // Schreibt einen String
- BufferedWriter.newLine (); // Schreibt ein Zeilentrennzeichen
- BufferedWriter.close (); // Schließt den BufferedWriter

Bemerkungen

- Wenn Sie versuchen, aus einem `BufferedWriter` (mit `BufferedWriter.write()`) nach dem Schließen des `BufferedWriter` (mit `BufferedWriter.close()`) zu schreiben, wird eine `IOException` .
- Der `BufferedWriter(Writer)` `IOException` löst KEINE `IOException` . Der `FileWriter(File)` `FileNotFoundException` `IOException` eine `FileNotFoundException` , die die `IOException` . `IOException` `FileNotFoundException` `IOException` wird auch `FileNotFoundException` Eine zweite Catch-Anweisung ist nur dann `FileNotFoundException` , wenn Sie mit `FileNotFoundException` etwas anderes `FileNotFoundException` .

Examples

Schreiben Sie eine Textzeile in Datei

Dieser Code schreibt die Zeichenfolge in eine Datei. Es ist wichtig, den Schreiber zu schließen, so dass dies in einem `finally` Block erfolgt.

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

Beachten Sie auch, dass `write(String s)` nach dem `write(String s)` keine Zeilenvorschubzeichen setzt. Verwenden Sie dazu die `newLine()` -Methode.

Java SE 7

Java 7 fügt das Paket `java.nio.file` und try-with-resources hinzu :

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

BufferedWriter online lesen: https://riptutorial.com/de/java/topic/3063/bufferedwriter

Kapitel 24: ByteBuffer

Einführung

Die `ByteBuffer` Klasse wurde in Java 1.4 eingeführt, um das Arbeiten mit binären Daten zu erleichtern. Es ist besonders für die Verwendung mit primitiven Daten geeignet. Es erlaubt die Erzeugung, aber auch die nachfolgende Manipulation eines `byte[]` auf einer höheren Abstraktionsebene

Syntax

- Byte [] Arr = neues Byte [1000];
- ByteBuffer buffer = ByteBuffer.wrap (arr);
- ByteBuffer Puffer = ByteBuffer.allocate (1024);
- ByteBuffer buffer = ByteBuffer.allocateDirect (1024);
- Byte b = Puffer.get ();
- Byte b = Puffer.get (10);
- kurz s = buffer.getShort (10);
- Puffer-Eingang ((Byte) 120);
- buffer.putChar ('a');

Examples

Grundlegende Verwendung - Erstellen eines ByteBuffers

Es gibt zwei Möglichkeiten, einen `ByteBuffer` zu erstellen, bei dem einer erneut unterteilt werden kann.

Wenn Sie bereits ein `byte[]` , können Sie es in einen `ByteBuffer` *"einhüllen"* , um die Verarbeitung zu vereinfachen:

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

Dies wäre eine Möglichkeit für Code, der Netzwerkinteraktionen auf niedriger Ebene abwickelt

---

Wenn Sie noch kein `byte[]` , können Sie einen `ByteBuffer` über einem Array erstellen, das speziell für den Puffer wie `ByteBuffer` reserviert ist:

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

Wenn der Code-Pfad extrem leistungskritisch ist und Sie **direkten Systemspeicherzugriff** benötigen, kann der `ByteBuffer` sogar *direkte* Puffer mit `#allocateDirect()`

Grundlegende Verwendung - Schreiben Sie Daten in den Puffer

Wenn eine `ByteBuffer` Instanz gegeben ist, kann man Primitive-Typ-Daten mit *relativen* und *absoluten* `put ByteBuffer` sie schreiben. Der markante Unterschied besteht darin , dass die Daten unter Verwendung der *relativen* Methode verfolgt die Spuren des Index setzen die Daten an für Sie eingefügt wird, während die absolute Methode immer einen Index erfordert geben zu `put` , die Daten an.

Beide Methoden erlauben das *"Verketten" von* Aufrufen. Bei einem ausreichend großen Puffer kann man dementsprechend Folgendes tun:

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

was äquivalent ist zu:

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

Beachten Sie, dass die Methode, die für `byte` wird, nicht speziell benannt wird. Außerdem beachten Sie, dass es auch gültig ist sowohl eine passieren `ByteBuffer` und ein `byte[]` zu `put` . Ansonsten haben alle primitiven Typen spezielle `put` Methoden.

Ein zusätzlicher Hinweis: Der bei Verwendung des absoluten `put*` angegebene Index wird immer in `byte` s gezählt.

Grundlegende Verwendung - Verwendung von DirectByteBuffer

`DirectByteBuffer` ist eine spezielle Implementierung von `ByteBuffer` , unter der kein `byte[]` .

Wir können solche ByteBuffer durch Aufruf von:

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

Diese Operation belegt 16 Byte Speicher. Der Inhalt der direkten Puffer *kann* sich außerhalb des normalen Abfallsammelhaufens befinden.

Wir können überprüfen, ob ByteBuffer direkt ist, indem Sie Folgendes aufrufen:

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

Die Hauptmerkmale von `DirectByteBuffer` sind, dass JVM versucht, den zugewiesenen Speicher ohne zusätzliche Pufferung nativ zu bearbeiten, sodass die darauf durchgeführten Vorgänge möglicherweise schneller sind als die für ByteBuffers mit darunter liegenden Arrays.

Es wird empfohlen, `DirectByteBuffer` mit `DirectByteBuffer` Operationen zu verwenden, die von der Ausführungsgeschwindigkeit abhängig sind, wie z. B. Echtzeitkommunikation.

Wir müssen uns bewusst sein, dass wir, wenn wir die `array()` -Methode versuchen, `UnsupportedOperationException` . Daher ist es eine gute Praxis, zu prüfen, ob unser ByteBuffer (Byte-Array) vorhanden ist, bevor wir versuchen, darauf zuzugreifen:

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

Eine andere Verwendung des direkten Bytepuffers ist das Interop über die JNI. Da ein direkter Byte-Puffer kein `byte[]` , sondern einen tatsächlichen Speicherblock verwendet, ist es möglich, über einen Zeiger im nativen Code direkt auf diesen Speicher zuzugreifen. Dies kann ein wenig Ärger und Mehraufwand beim Marshalling zwischen der Java- und der nativen Darstellung von Daten ersparen.

Die JNI-Schnittstelle definiert mehrere Funktionen, um direkte Bytepuffer zu behandeln: NIO-Unterstützung .

ByteBuffer online lesen: https://riptutorial.com/de/java/topic/702/bytebuffer

Kapitel 25: Bytecode-Änderung

Examples

Was ist Bytecode?

Bytecode ist der Satz von Anweisungen, der von der JVM verwendet wird. Um dies zu veranschaulichen, nehmen wir dieses Hello World-Programm.

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

Das ist, woraus es wird, wenn es in Bytecode kompiliert wird.

```
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
```

Was ist die Logik dahinter?

getstatic - Erhält den Wert eines statischen Felds einer Klasse. In diesem Fall ist der *PrintStream "Out"* des *Systems* .

ldc - **Schiebe** eine Konstante auf den Stapel. In diesem Fall ist der String "Hello World"

invokevirtual - Ruft eine Methode für eine geladene Referenz im Stapel auf und legt das Ergebnis auf dem Stapel ab. Die Parameter der Methode werden ebenfalls vom Stack übernommen.

Nun, da muss mehr sein?

Es gibt 255 Opcodes, aber noch nicht alle sind implementiert. Eine Tabelle mit allen aktuellen Opcodes finden Sie hier: Java-Bytecode-Befehlslisten .

Wie kann ich Bytecode schreiben / bearbeiten?

Es gibt mehrere Möglichkeiten, Bytecode zu schreiben und zu bearbeiten. Sie können einen Compiler, eine Bibliothek oder ein Programm verwenden.

Zum Schreiben:

- Jasmin
- Krakatau

Für die Bearbeitung:

- Bibliotheken
  - ASM
  - Javassist
  - BCEL - *Unterstützt kein Java 8+*
- Werkzeuge
  - Bytecode-Viewer
  - JBytedit
  - reJ - *Unterstützt nicht Java 8+*
  - JBE - *Java 8+ wird nicht unterstützt*

Ich möchte mehr über Bytecode erfahren!

Es gibt wahrscheinlich eine spezifische Dokumentationsseite speziell für Bytecode. Diese Seite konzentriert sich auf die Änderung von Bytecode mit verschiedenen Bibliotheken und Werkzeugen.

So bearbeiten Sie JAR-Dateien mit ASM

Zunächst müssen die Klassen aus dem Glas geladen werden. Wir werden drei Methoden für diesen Prozess verwenden:

- loadClasses (Datei)
- 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;
}
```

Mit diesen Methoden wird das Laden und Ändern einer JAR-Datei zum einfachen Ändern von ClassNodes in einer Map. In diesem Beispiel werden wir alle Strings in der Dose mit Hilfe der Tree-API durch großgeschriebene ersetzen.

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

Nachdem nun alle Zeichenfolgen des ClassNode geändert wurden, müssen die Änderungen gespeichert werden. Um die Änderungen zu speichern und einen Arbeitsausgang zu haben, müssen einige Dinge getan werden:

- Exportieren Sie ClassNodes in Bytes
- Laden Sie Nicht-Klassen-JAR-Einträge *(z.* B. *Manifest.mf / andere binäre Ressourcen in Jar)*

als Bytes
- Speichern Sie alle Bytes in einem neuen Glas

Im letzten Abschnitt werden wir drei Methoden erstellen.

- processNodes (Map <String, ClassNode> Knoten)
- loadNonClasses (Datei jarFile)
- saveAsJar (Map <String, Byte []> outBytes, String Dateiname)

Verwendungszweck:

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

Die verwendeten Methoden:

```
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();
    }
}
```

Das ist es. Alle Änderungen werden in "sample-edit.jar" gespeichert.

So laden Sie einen ClassNode als Klasse

```
/**
 * 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;
    }
}
```

So benennen Sie Klassen in einer JAR-Datei um

```
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 ist eine vorhandene Klasse in der ASM-Bibliothek. Es können jedoch nur Klassennamen geändert werden. Wenn Sie Felder und Methoden umbenennen möchten, sollten Sie Ihre eigene Implementierung der Remapper-Klasse erstellen.

Javassist Basic

Javassist ist eine Bytecode-Instrumentierungsbibliothek, mit der Sie den Bytecode ändern können, der Java-Code einfügt, der von Javassist in Bytecode konvertiert und zur Laufzeit der instrumentierten Klasse / Methode hinzugefügt wird.

Schreiben wir den ersten Transformer, der eine hypothetische Klasse "com.my.to.be.instrument.MyClass" enthält, und fügt den Anweisungen jeder Methode einen Protokollaufruf hinzu.

```
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;
    }
}
```

Um diesen Transformator zu verwenden (damit unsere JVM die Methodentransformation zum Zeitpunkt des Ladens in jeder Klasse aufruft), müssen Sie dieses Instrument hinzufügen oder dies mit einem Agenten:

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

public class EasyAgent {

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

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

Der letzte Schritt, um unser erstes Instrumentor-Experiment zu starten, ist das Registrieren dieser Agentenklasse bei der JVM-Maschinenausführung. Der einfachste Weg, dies zu tun, ist die Registrierung mit einer Option im Shell-Befehl:

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

Wie wir sehen, wird das Agent / Transformer-Projekt als Jar zur Ausführung einer Anwendung mit dem Namen MyJavaApplication hinzugefügt, die eine Klasse mit dem Namen "com.my.to.be.instrumented.MyClass" enthalten soll, um den eingefügten Code tatsächlich auszuführen.

Bytecode-Änderung online lesen: https://riptutorial.com/de/java/topic/3747/bytecode-anderung

Kapitel 26: C ++ - Vergleich

Einführung

Java und C ++ sind ähnliche Sprachen. Dieses Thema dient als Kurzanleitung für Java- und C ++ - Engineer.

Bemerkungen

In anderen Konstrukten definierte Klassen

Innerhalb einer anderen Klasse definiert

C ++

Geschachtelte Klasse [ref] (benötigt eine Referenz zum Einschließen der Klasse)

```
class Outer {
   class Inner {
      public:
         Inner(Outer* o) :outer(o) {}

      private:
         Outer*  outer;
   };
};
```

Java

[nicht statisch] Geschachtelte Klasse (auch bekannt als Inner Class oder Member Class)

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

Statisch innerhalb einer anderen Klasse definiert

C ++

Statische verschachtelte Klasse

```
class Outer {
   class Inner {
      ...
   };
};
```

Java

Statische verschachtelte Klasse (auch bekannt als statische Member-Klasse) [ref]

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

Innerhalb einer Methode definiert

(zB Eventhandling)

C ++

Lokale Klasse [ref]

```
void fun() {
   class Test {
      /* members of Test class */
   };
}
```

Siehe auch Lambda-Ausdrücke

Java

Lokale Klasse [ref]

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
            }
        }).start();
```

```
    }
}
```

Überschreiben vs Überladen

Die folgenden Overriding- und Overloading-Punkte gelten sowohl für C ++ als auch für Java:

- Eine überschriebene Methode hat denselben Namen und dieselben Argumente wie ihre Basismethode.
- Eine überladene Methode hat denselben Namen, jedoch unterschiedliche Argumente und ist nicht auf die Vererbung angewiesen.
- Zwei Methoden mit demselben Namen und Argumenten, aber unterschiedlichen Rückgabetypen sind ungültig. Siehe verwandte Stackoverflow-Fragen zum Thema "Überladen mit anderem Rückgabetyp in Java" - Frage 1 ; Frage 2

Polymorphismus

Polymorphismus ist die Fähigkeit von Objekten verschiedener Klassen, die durch Vererbung miteinander in Beziehung stehen, auf denselben Methodenaufruf unterschiedlich zu reagieren. Hier ist ein Beispiel:

- Basisklasse Form mit Fläche als abstrakte Methode
- Zwei abgeleitete Klassen, Square und Circle, implementieren Flächenmethoden
- Formbezugspunkte auf Quadrat und Fläche werden aufgerufen

In C ++ wird Polymorphismus durch virtuelle Methoden aktiviert. In Java sind die Methoden standardmäßig virtuell.

Reihenfolge der Konstruktion / Zerstörung

Objektbereinigung

In C ++ empfiehlt es sich, einen Destruktor als virtuell zu deklarieren, um sicherzustellen, dass der Destruktor der Unterklasse aufgerufen wird, wenn der Basisklassenzeiger gelöscht wird.

In Java ist eine Finalize-Methode ein Destruktor in C ++. Finalizer sind jedoch unvorhersehbar (sie sind auf GC angewiesen). Bewährte Methode: Verwenden Sie eine "close" -Methode, um explizit zu bereinigen.

```
protected void close() {
    try {
       // do subclass cleanup
    }
    finally {
```

```
        isClosed = true;
        super.close();
    }
}

protected void finalize() {
    try {
        if(!isClosed) close();
    }
    finally {
        super.finalize();
    }
}
```

Abstrakte Methoden und Klassen

Konzept	C ++	Java
Abstrakte Methode ohne Implementierung deklariert	reine virtuelle Methode `virtual void eat(void) = 0;`	abstrakte Methode `abstract void draw();`
Abstrakte Klasse kann nicht instanziiert werden	kann nicht instanziiert werden; hat mindestens 1 reine virtuelle Methode `class AB {public: virtual void f() = 0;};`	kann nicht instanziiert werden; kann nicht abstrakte Methoden haben `abstract class GraphicObject {}`
Schnittstelle keine Instanzfelder	Kein "interface" - Schlüsselwort, kann jedoch eine Java-Schnittstelle mit Einrichtungen einer abstrakten Klasse nachahmen	sehr ähnlich der abstrakten Klasse, aber 1) unterstützt Mehrfachvererbung; 2) keine Instanzfelder `interface TestInterface {}`

Eingabehilfen-Modifikatoren

Modifikator	C ++	Java
Öffentlich - für alle zugänglich	*keine besonderen Hinweise*	*keine besonderen Hinweise*
Geschützt - durch Unterklassen zugänglich	auch für Freunde zugänglich	auch innerhalb desselben Pakets zugänglich
Privat - für Mitglieder	auch für Freunde zugänglich	*keine besonderen*

Modifikator	C ++	Java
zugänglich		*Hinweise*
Standard	Klassenstandard ist privat; struct default ist public	zugänglich für alle Klassen innerhalb desselben Pakets
andere	Freund - eine Möglichkeit, privaten und geschützten Mitgliedern Zugriff ohne Erbschaft zu gewähren (siehe unten)	

C ++ Freund Beispiel

```
class Node {
  private:
    int key;  Node *next;
    // LinkedList::search() can access "key" & "next"
    friend int LinkedList::search();
};
```

Das gefürchtete Diamantproblem

> Das Raute-Problem ist eine Mehrdeutigkeit, die entsteht, wenn zwei Klassen B und C von A erben und Klasse D sowohl von B als auch C erbt. Wenn es eine Methode in A gibt, die B und C überschrieben haben, überschreibt D sie nicht Welche Version der Methode erbt D: die von B oder die von C? (aus Wikipedia)

Während C ++ für Diamantenprobleme immer anfällig war, war Java bis Java 8 anfällig. Ursprünglich unterstützte Java keine Mehrfachvererbung, aber mit dem Aufkommen von Standardschnittstellenmethoden können Java-Klassen die "Implementierung" nicht von mehreren Klassen erben .

java.lang.Object-Klasse

In Java erben alle Klassen implizit oder explizit von der Object-Klasse. Jede Java-Referenz kann in den Objekttyp umgewandelt werden.

C ++ hat keine vergleichbare "Object" -Klasse.

Java-Sammlungen und C ++ - Container

Java Collections sind mit C ++ Containern identisch.

Flussdiagramm für Java-Sammlungen

C ++ Container-Flussdiagramm

Integer-Typen

Bits	Mindest	Max	C ++ - Typ (auf LLP64 oder LP64)	Java-Typ
8	-2 (8-1) = -128	2 (8-1) -1 = 127	verkohlen	Byte
8	0	2 (8) -1 = 255	unsigniertes Zeichen	-
16	-2 (16-1) = -32,768	2 (16-1) -1 = 32.767	kurz	kurz
16	0 (\ u0000)	2 (16) -1 = 65.535 (\ uFFFF)	unsigniert kurz	char (unsigniert)
32	-2 (32-1) = -2,147 Milliarden	2 (32-1) -1 = 2,147 Milliarden	int	int
32	0	2 (32) -1 = 4,295 Milliarden	unsigned int	-
64	-2 (64-1)	2 (16-1) -1	lange*	lang Lang
64	0	2 (16) -1	unsigniert lang unsigniert lange	-

* Win64-API ist nur 32 Bit

Viele weitere C ++ - Typen

Examples

Statische Klassenmitglieder

Statische Member haben Klassenbereich im Gegensatz zum Objektbereich

C ++ - Beispiel

```
// define in header
```

```
class Singleton {
   public:
      static Singleton *getInstance();

   private:
      Singleton() {}
      static Singleton *instance;
};

// initialize in .cpp
Singleton* Singleton::instance = 0;
```

Java- Beispiel

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

In anderen Konstrukten definierte Klassen

Innerhalb einer anderen Klasse definiert

C ++

Geschachtelte Klasse [ref] (benötigt eine Referenz zum Einschließen der Klasse)

```
class Outer {
   class Inner {
      public:
         Inner(Outer* o) :outer(o) {}

      private:
         Outer*  outer;
   };
};
```

Java

[nicht statisch] Geschachtelte Klasse (auch bekannt als Inner Class oder Member Class)

```
class OuterClass {
```

```
    ...
    class InnerClass {
        ...
    }
}
```

Statisch innerhalb einer anderen Klasse definiert

C ++

Statische verschachtelte Klasse

```
class Outer {
   class Inner {
      ...
   };
};
```

Java

Statische verschachtelte Klasse (auch bekannt als statische Member-Klasse) [ref]

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

Innerhalb einer Methode definiert

(zB Eventhandling)

C ++

Lokale Klasse [ref]

```
void fun() {
   class Test {
      /* members of Test class */
   };
}
```

Java

Lokale Klasse [ref]

```
class Test {
    void f() {
        new Thread(new Runnable() {
            public void run() {
                doSomethingBackgroundish();
            }
        }).start();
    }
}
```

Pass-by-Value & Pass-by-Referenz

Viele behaupten, dass Java NUR ein Wert für Vorübergehen ist, aber es ist nuancierter als das. Vergleichen Sie die folgenden C ++ - und Java-Beispiele, um die vielen Varianten von Pass-by-Value (aka copy) und Pass-by-Reference (Alias) zu sehen.

C ++ - Beispiel (vollständiger Code)

```
  // 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;
  }
```

Java-Beispiel (vollständiger Code)

```
   // 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);
   }
```

Vererbung vs. Zusammensetzung

C ++ und Java sind beide objektorientierte Sprachen, daher gilt das folgende Diagramm für beide.

Ausgestoßener Downcasting

Vorsicht bei der Verwendung von "Downcasting" - Downcasting führt die Vererbungshierarchie von einer Basisklasse in eine Unterklasse (dh entgegengesetzt zum Polymorphismus). Verwenden Sie im Allgemeinen Polymorphie und Überschreiben anstelle von Instanzierung und Downcasting.

C ++ - Beispiel

```
// explicit type case required
Child *pChild =  (Child *) &parent;
```

Java-Beispiel

```
if(mySubClass instanceof SubClass) {
   SubClass mySubClass = (SubClass)someBaseClass;
   mySubClass.nonInheritedMethod();
}
```

Abstrakte Methoden und Klassen

Abstrakte Methode

ohne Implementierung deklariert

C ++

reine virtuelle Methode

```
virtual void eat(void) = 0;
```

Java

abstrakte Methode

```
abstract void draw();
```

Abstrakte Klasse

kann nicht instanziiert werden

C ++

kann nicht instanziiert werden; hat mindestens 1 reine virtuelle Methode

```
class AB {public: virtual void f() = 0;};
```

Java

kann nicht instanziiert werden; kann nicht abstrakte Methoden haben

```
abstract class GraphicObject {}
```

Schnittstelle

keine Instanzfelder

C ++

nichts vergleichbar mit Java

Java

sehr ähnlich der abstrakten Klasse, aber 1) unterstützt Mehrfachvererbung; 2) keine Instanzfelder

```
interface TestInterface {}
```

C ++ - Vergleich online lesen: https://riptutorial.com/de/java/topic/10849/c-plusplus---vergleich

Kapitel 27: CompletableFuture

Einführung

CompletableFuture ist eine in Java SE 8 hinzugefügte Klasse, die die Future-Schnittstelle von Java SE 5 implementiert. Neben der Unterstützung der Future-Schnittstelle werden viele Methoden hinzugefügt, die einen asynchronen Rückruf ermöglichen, wenn die Zukunft abgeschlossen ist.

Examples

Konvertieren Sie die Blockierungsmethode in asynchron

Die folgende Methode dauert je nach Verbindung eine oder zwei Sekunden, um eine Webseite abzurufen und die Textlänge zu zählen. Was auch immer der Thread aufruft, wird für diesen Zeitraum blockiert. Außerdem wird eine Ausnahme zurückgegeben, die später nützlich ist.

```
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);
    }
}
```

Dadurch wird es in eine Methode konvertiert, die sofort zurückgegeben wird, indem der Aufruf der blockierenden Methode in einen anderen Thread verschoben wird. Standardmäßig wird die supplyAsync-Methode den Lieferanten im allgemeinen Pool ausführen. Für eine Blockierungsmethode ist dies wahrscheinlich keine gute Wahl, da die Threads in diesem Pool möglicherweise erschöpft sind, weshalb ich den optionalen Parameter service hinzugefügt habe.

```
static private ExecutorService service = Executors.newCachedThreadPool();

static public CompletableFuture<Long> asyncGetWebPageLength(String url) {
    return CompletableFuture.supplyAsync(() -> blockingGetWebPageLength(url), service);
}
```

Um die Funktion asynchron zu verwenden, sollte eine der Methoden verwendet werden, die das Aufrufen eines Lamdas mit dem Ergebnis des Lieferanten zulässt, wenn es abgeschlossen ist, beispielsweise thenAccept. Es ist auch wichtig, ausnahmsweise oder eine Methode zum Protokollieren der möglicherweise aufgetretenen Ausnahmen zu verwenden.

```
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;
            });

}
```

Einfaches Beispiel für CompletableFuture

In dem folgenden Beispiel `calculateShippingPrice` Methode 'dispatchShippingPrice' die Versandkosten, was einige Verarbeitungszeit in Anspruch nimmt. In einem realen Beispiel würde dies zum Beispiel die Kontaktaufnahme mit einem anderen Server sein, der den Preis basierend auf dem Gewicht des Produkts und der Versandart zurückgibt.

Indem Sie dies über `CompletableFuture` asynchrone Weise modellieren, können wir verschiedene Arbeiten in der Methode fortsetzen (z. B. Berechnung der Verpackungskosten).

```
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!
    });
}
```

CompletableFuture online lesen: https://riptutorial.com/de/java/topic/10935/completablefuture

Kapitel 28: Datei I / O

Einführung

Java I / O (Eingabe und Ausgabe) wird verwendet, um die Eingabe zu verarbeiten und die Ausgabe zu erzeugen. Java verwendet das Stream-Konzept, um den E / A-Vorgang schnell zu machen. Das Paket java.io enthält alle für Eingabe- und Ausgabeoperationen erforderlichen Klassen. Die Handhabung von Dateien erfolgt auch in Java über die Java I / O-API.

Examples

Lesen aller Bytes in ein Byte []

Java 7 führte die sehr nützliche Klasse Files ein

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();
}
```

Ein Bild aus einer Datei lesen

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

...

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

Ein Byte [] in eine Datei schreiben

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) {}
    }
}
```

Die meisten java.io-Datei-APIs akzeptieren sowohl `String` als auch `File` als Argumente. Daher können Sie dies auch tun

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

Stream vs Writer / Reader API

Streams bieten den direktesten Zugriff auf den binären Inhalt. `OutputStream` alle `InputStream` / `OutputStream` Implementierungen immer auf `int` s und `byte` `OutputStream` .

```
// 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);
}
```

Es gibt einige Ausnahmen, vor allem der `PrintStream` , der "die Möglichkeit `PrintStream` , Darstellungen verschiedener Datenwerte bequem zu drucken". Dies ermöglicht die Verwendung von `System.out` sowohl als binärer `InputStream` als auch als `System.out.println()` mit Methoden wie `System.out.println()` .

Einige Stream-Implementierungen funktionieren auch als Schnittstelle zu übergeordneten Inhalten wie Java-Objekten (siehe Serialisierung) oder `DataOutputStream` Typen, z. B. `DataOutputStream` / `DataInputStream` .

Mit den Klassen `Writer` und `Reader` bietet Java auch eine API für explizite Zeichenströme. Obwohl die meisten Anwendungen diese Implementierungen auf Streams stützen, stellt die Zeichenstrom-API keine Methoden für binären Inhalt bereit.

```
// 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();
```

Wenn Zeichen in binäre Daten codiert werden müssen (z. B. bei Verwendung der `InputStreamWriter` / `OutputStreamWriter` Klassen), sollten Sie einen Zeichensatz angeben, wenn Sie nicht vom Standardzeichensatz der Plattform abhängig sein möchten. Verwenden Sie im Zweifelsfall eine Unicode-kompatible Codierung, z. B. UTF-8, die auf allen Java-Plattformen unterstützt wird. Daher sollten Sie sich wahrscheinlich nicht an Klassen wie `FileWriter` und `FileReader` da diese immer den Standardplattformzeichensatz verwenden. Eine bessere Möglichkeit, mit Zeichenströmen auf Dateien zuzugreifen, ist folgende:

```
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();
```

Einer der am häufigsten verwendeten `Reader` ist `BufferedReader` der eine Methode zum Lesen ganzer Textzeilen von einem anderen Reader bietet und vermutlich der einfachste Weg ist, einen Zeichenstrom Zeile für Zeile zu lesen:

```
// 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);
}
```

Eine ganze Datei auf einmal lesen

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

\ Z ist das EOF-Symbol (Dateiende). Als Trennzeichen wird der Scanner die Füllung lesen, bis das EOF-Flag erreicht ist.

Eine Datei mit einem Scanner lesen

Eine Datei Zeile für Zeile lesen

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

Wort für Wort

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

Sie können das Delimeter auch mithilfe der scanner.useDelimeter () -Methode ändern

Durchlaufen eines Verzeichnisses und Filtern nach Dateierweiterung

```
    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 von java.io.File zu Java 7 NIO (java.nio.file.Path)

In diesen Beispielen wird davon ausgegangen, dass Sie bereits wissen, was NIO von Java 7 generell ist, und Sie sind daran `java.io.File` mit `java.io.File` Code zu `java.io.File` . Anhand dieser Beispiele können Sie schnell weitere NIO-basierte Dokumentation für die Migration finden.

Das NIO von Java 7 bietet noch viel mehr, z. B. speicherzugeordnete Dateien oder das Öffnen einer ZIP- oder JAR-Datei mit FileSystem . Diese Beispiele behandeln nur eine begrenzte Anzahl grundlegender Anwendungsfälle.

Wenn Sie `java.io.File` eine Dateisystem-Lese- / Schreiboperation mit einer `java.io.File` Instanzmethode `java.io.File` , finden Sie sie als statische Methode in `java.nio.file.Files` .

Zeigen Sie auf einen Pfad

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

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

Pfade relativ zu einem anderen Pfad

```
// 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");
```

Konvertieren der Datei von / in den Pfad zur Verwendung mit Bibliotheken

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

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

Überprüfen Sie, ob die Datei vorhanden ist, und löschen Sie sie gegebenenfalls

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

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

Schreiben Sie über einen OutputStream in eine Datei

Es gibt mehrere Möglichkeiten, aus einer Datei mit NIO zu schreiben und zu lesen, um unterschiedliche Leistungs- und Speicherbeschränkungen, Lesbarkeit und Anwendungsfälle zu `FileChannel` , wie z. B. `FileChannel` , `Files.write(Path path, byte\[\] bytes, OpenOption... options)` . In diesem Beispiel wird nur `OutputStream` behandelt. Es wird jedoch dringend `OutputStream` , sich über speicherzugeordnete Dateien und die verschiedenen statischen Methoden zu `java.nio.file.Files` in `java.nio.file.Files` verfügbar `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));
    }
}
```

Iteration für jede Datei innerhalb eines Ordners

```
// -> 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;
    }
});
```

Rekursive Ordner-Iteration

```
// -> 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);
        }
    }
}
```

Datei lesen / schreiben mit FileInputStream / FileOutputStream

Schreiben Sie in eine Datei 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();
}
```

Lesen aus Datei 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();
}
```

Beachten Sie, dass seit Java 1.7 die try-with-resources- Anweisung eingeführt wurde, was die Implementierung der Lese- / Schreiboperation wesentlich vereinfacht:

Schreiben Sie in eine Datei 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();
}
```

Lesen aus Datei 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();
}
```

Lesen aus einer Binärdatei

Sie können eine Binärdatei mit diesem Code in allen aktuellen Java-Versionen lesen:

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();
```

Wenn Sie Java 7 oder höher verwenden, gibt es eine einfachere Möglichkeit, die `nio API` :

Java SE 7

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

Sperren

Eine Datei kann mit dem gesperrten `FileChannel` - API , die von Input Output erworben werden kann `streams` und `readers`

Beispiel mit `streams`

// Öffnet einen Dateistream FileInputStream ios = new FileInputStream (Dateiname);

```
    // 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
```

Eine Datei mit InputStream und OutputStream kopieren

Wir können Daten mithilfe einer Schleife direkt von einer Quelle in eine Datensenke kopieren. In diesem Beispiel lesen wir Daten von einem InputStream und schreiben gleichzeitig in einen OutputStream. Sobald wir mit dem Lesen und Schreiben fertig sind, müssen wir die Ressource schließen.

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

Eine Datei mit Channel und Buffer lesen

`Channel` verwendet einen `Buffer` zum Lesen / Schreiben von Daten. Ein Puffer ist ein Container mit

fester Größe, in den wir einen Datenblock gleichzeitig schreiben können. `Channel` ist ziemlich schnell als Stream-basierte E / A.

Um Daten aus einer Datei mit `Channel` zu lesen, müssen Sie die folgenden Schritte ausführen:

1. Wir benötigen eine Instanz von `FileInputStream` . `FileInputStream` hat eine Methode namens `getChannel()` die einen Channel zurückgibt.
2. Rufen Sie die `getChannel()` Methode von FileInputStream auf und rufen Sie den Channel ab.
3. Erstellen Sie einen ByteBuffer. ByteBuffer ist ein Container mit fester Größe.
4. Channel hat eine Lesemethode und wir müssen ein ByteBuffer als Argument für diese Lesemethode angeben. ByteBuffer verfügt über zwei Modi: Nur-Lese-Stimmung und Nur-Lese-Stimmung. Wir können den Modus mit dem Aufruf der Methode `flip()` ändern. Puffer hat eine Position, ein Limit und eine Kapazität. Sobald ein Puffer mit fester Größe erstellt wurde, stimmen sein Limit und seine Kapazität mit der Größe überein und die Position beginnt bei Null. Während ein Puffer mit Daten geschrieben wird, nimmt seine Position allmählich zu. Ändern des Modus bedeutet, die Position zu ändern. Um Daten vom Anfang eines Puffers aus lesen zu können, müssen wir die Position auf Null setzen. Die Flip () - Methode ändert die Position
5. Wenn wir die Lesemethode des `Channel` aufrufen, wird der Puffer mit Daten gefüllt.
6. Wenn wir die Daten aus dem `ByteBuffer` lesen `ByteBuffer` , müssen wir den Puffer umdrehen, um seinen Modus in den Nur-Lese-Modus zu ändern, und nur dann den schreibgeschützten Modus verwenden.
7. Wenn keine Daten mehr zu lesen sind, gibt die `read()` Methode des Kanals 0 oder -1 zurück.

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

Eine Datei mit Channel kopieren

Wir können `Channel` dazu verwenden, Dateiinhalte schneller zu kopieren. Dazu können wir die `transferTo()` Methode von `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();
        }
    }
}
```

Eine Datei mit BufferedInputStream lesen

Das Lesen einer Datei mit einem `BufferedInputStream` Allgemeinen schneller als mit `FileInputStream` da ein interner Puffer zum Speichern der aus dem darunterliegenden Eingabestrom gelesenen Bytes verwaltet wird.

```
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();
        }

    }
}
```

Eine Datei mit Channel und Buffer schreiben

Um Daten mit `Channel` in eine Datei zu schreiben, müssen wir die folgenden Schritte ausführen:

1. Zuerst müssen wir ein Objekt von `FileOutputStream`
2. `FileChannel` , um die `getChannel()` Methode vom `FileOutputStream`
3. Erstellen Sie einen `ByteBuffer` und füllen Sie ihn mit Daten
4. Dann müssen wir die `flip()` -Methode des `ByteBuffer` und als Argument der `write()` -Methode des `FileChannel`
5. Sobald wir mit dem Schreiben fertig sind, müssen wir die Ressource schließen

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

Eine Datei mit PrintStream schreiben

Wir können die `PrintStream` Klasse verwenden, um eine Datei zu schreiben. Es gibt mehrere Methoden, mit denen Sie beliebige Datentypwerte drucken können. `println()` -Methode fügt eine neue Zeile an. Sobald wir mit dem Drucken fertig sind, müssen wir den `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();
        }

    }
}
```

Durchlaufen Sie ein Verzeichnis, in dem Unterverzeichnisse zum Drucken gedruckt werden

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

Verzeichnisse hinzufügen

Um ein neues Verzeichnis aus einer `File` Instanz zu `mkdirs()` müssen Sie eine der beiden Methoden verwenden: `mkdirs()` oder `mkdir()` .

- `mkdir()` - Erstellt das durch diesen abstrakten Pfadnamen benannte Verzeichnis. (Quelle)
- `mkdirs()` - Erstellt das durch diesen abstrakten Pfadnamen benannte Verzeichnis, einschließlich aller erforderlichen, jedoch nicht vorhandenen übergeordneten Verzeichnisse. Wenn dieser Vorgang fehlschlägt, ist es möglicherweise gelungen, einige der erforderlichen übergeordneten Verzeichnisse zu erstellen. (Quelle)

Hinweis: `createNewFile()` erstellt kein neues Verzeichnis, sondern nur eine Datei.

```
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
```

Standardausgabe / Fehler blockieren oder umleiten

Manchmal schreibt eine schlecht entworfene Bibliothek eines Drittanbieters unerwünschte Diagnosen in die Streams von `System.out` oder `System.err` . Die empfohlenen Lösungen für dieses Problem sind, entweder eine bessere Bibliothek zu finden oder (im Fall von Open Source) das Problem zu beheben und den Entwicklern einen Patch zur Verfügung zu stellen.

Wenn die oben genannten Lösungen nicht durchführbar sind, sollten Sie die Streams umleiten.

Umleitung in der Befehlszeile

Unter UNIX kann ein Linux- oder MacOSX-System von der Shell aus mit `>` Umleitung ausgeführt werden. Zum Beispiel:

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

Der erste leitet die Standardausgabe und den Standardfehler nach "/ dev / null" um, wodurch alles, was in diese Streams geschrieben wird, weggeworfen wird. Die zweite Version leitet die Standardausgabe in "out.log" und den Standardfehler in "error.log" um.

(Weitere Informationen zur Umleitung finden Sie in der Dokumentation der von Ihnen verwendeten Befehlsshell. Ähnliches gilt für Windows.)

Alternativ können Sie die Umleitung in einem Wrapper-Skript oder einer Batch-Datei implementieren, die die Java-Anwendung startet.

Umleitung innerhalb einer Java-Anwendung

Es ist auch möglich, die Streams *innerhalb* einer Java-Anwendung mithilfe von `System.setOut()` und `System.setErr()` . Das folgende Snippet leitet beispielsweise die Standardausgabe und den Standardfehler in zwei Protokolldateien um:

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

Wenn Sie die Ausgabe vollständig wegwerfen möchten, können Sie einen Ausgabestrom erstellen, der in einen ungültigen Dateideskriptor "schreibt". Dies entspricht funktional dem Schreiben in "/ dev / null" unter UNIX.

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

Achtung: `setOut` Sie vorsichtig, wie Sie `setOut` und `setErr` :

1. Die Umleitung wirkt sich auf die gesamte JVM aus.
2. Dadurch nehmen Sie dem Benutzer die Möglichkeit, die Streams von der Befehlszeile umzuleiten.

Auf den Inhalt einer ZIP-Datei zugreifen

Die FileSystem-API von Java 7 ermöglicht das Lesen und Hinzufügen von Einträgen aus oder zu einer Zip-Datei mithilfe der Java-NIO-Datei-API auf dieselbe Weise wie bei einem anderen Dateisystem.

Das Dateisystem ist eine Ressource, die nach der Verwendung ordnungsgemäß geschlossen werden sollte. Daher sollte der Try-with-Resources-Block verwendet werden.

Lesen aus einer vorhandenen Datei

```
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();
}
```

Neue Datei erstellen

```
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();
}
```

Datei I / O online lesen: https://riptutorial.com/de/java/topic/93/datei-i---o

Kapitel 29: Datum und Uhrzeit (java.time. *)

Examples

Einfache Datumsmanipulationen

Holen Sie sich das aktuelle Datum.

```
LocalDate.now()
```

Holen Sie sich das Datum von gestern.

```
LocalDate y = LocalDate.now().minusDays(1);
```

Holen Sie sich das Datum von morgen

```
LocalDate t = LocalDate.now().plusDays(1);
```

Holen Sie sich ein bestimmtes Datum.

```
LocalDate t = LocalDate.of(1974, 6, 2, 8, 30, 0, 0);
```

Neben den `plus` und `minus` Methoden gibt es eine Reihe von "with" -Methoden, mit denen ein bestimmtes Feld in einer `LocalDate` Instanz festgelegt werden kann.

```
LocalDate.now().withMonth(6);
```

Das obige Beispiel gibt eine neue Instanz mit dem Monat Juni zurück (dies unterscheidet sich von `java.util.Date` wo `setMonth` eine 0 wurde, die den 5. Juni bildet).

Da LocalDate-Manipulationen unveränderliche LocalDate-Instanzen zurückgeben, können diese Methoden auch miteinander verkettet werden.

```
LocalDate ld = LocalDate.now().plusDays(1).plusYears(1);
```

Dies würde uns das Datum von morgen in einem Jahr geben.

Datum und Uhrzeit

Datum und Uhrzeit ohne Zeitzonenangabe

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 27, 8, 0);
LocalDateTime now = LocalDateTime.now();
LocalDateTime parsed = LocalDateTime.parse("2016-07-27T07:00:00");
```

Datum und Uhrzeit mit Zeitzoneninformationen

```
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]");
```

Datum und Uhrzeit mit Offset-Informationen (dh es werden keine Änderungen der Sommerzeit berücksichtigt)

```
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");
```

Vorgänge nach Datum und Uhrzeit

```
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]"))
```

Sofortig

Stellt einen Zeitpunkt dar. Kann als Wrapper für einen Unix-Zeitstempel betrachtet werden.

```
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
```

Verwendung verschiedener Klassen von Date Time API

Das folgende Beispiel enthält auch Erläuterungen, die zum Verständnis des Beispiels erforderlich sind.

```
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);

}
}
```

Date Time Formatierung

Vor Java 8 gab es im `DateFormat` `SimpleDateFormat` Klassen `DateFormat` und `SimpleDateFormat` Dieser ältere Code wird für `java.text` weiter verwendet.

Java 8 bietet jedoch einen modernen Ansatz für die Verarbeitung von Formatierung und Analyse.

Beim Formatieren und Analysieren übergeben Sie zuerst ein `String` Objekt an `DateTimeFormatter` und verwenden es wiederum zum Formatieren oder Analysieren.

```
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));
    }
}
```

Ein wichtiger Hinweis, anstatt benutzerdefinierte Muster zu verwenden, ist es ratsam, vordefinierte Formatierer zu verwenden. Ihr Code sieht klarer aus und die Verwendung von ISO8061 wird Ihnen definitiv auf lange Sicht helfen.

Differenz zwischen 2 LocalDates berechnen

Verwenden Sie `LocalDate` und `ChronoUnit` :

```
LocalDate d1 = LocalDate.of(2017, 5, 1);
LocalDate d2 = LocalDate.of(2017, 5, 18);
```

Da die Methode `between` dem `ChronoUnit` Enumerator 2 `Temporal` Parameter als Parameter `LocalDate` können Sie die `LocalDate` Instanzen problemlos `LocalDate`

```
long days = ChronoUnit.DAYS.between(d1, d2);
System.out.println( days );
```

Datum und Uhrzeit (java.time. *) online lesen: https://riptutorial.com/de/java/topic/4813/datum-und-uhrzeit--java-time----

Kapitel 30: Datumsklasse

Syntax

- `Date object = new Date();`
- `Date object = new Date(long date);`

Parameter

Parameter	Erläuterung
Kein Parameter	Erzeugt ein neues Date-Objekt mit der Zuordnungszeit (auf die nächste Millisekunde)
langes Datum	Erzeugt ein neues Date-Objekt mit einer auf "Millisekunden" seit "der Epoche" (1. Januar 1970, 00:00:00 GMT) eingestellten Zeit.

Bemerkungen

Darstellung

Intern wird ein Java Date-Objekt als Long dargestellt. es ist die Anzahl der Millisekunden seit einer bestimmten Zeit (als *Epoche bezeichnet*). Die ursprüngliche Java Date-Klasse verfügte über Methoden zum Umgang mit Zeitzonen usw., die jedoch zugunsten der damals neuen Calendar-Klasse verworfen wurden.

Wenn Sie also nur eine bestimmte Zeit in Ihrem Code verwenden möchten, können Sie eine Date-Klasse erstellen und speichern usw. Wenn Sie eine vom Menschen lesbare Version dieses Datums ausdrucken möchten, erstellen Sie jedoch eine Calendar-Klasse Verwenden Sie die Formatierung, um Stunden, Minuten, Sekunden, Tage, Zeitzonen usw. zu erstellen. Beachten Sie, dass eine bestimmte Millisekunde als unterschiedliche Stunden in verschiedenen Zeitzonen angezeigt wird. Normalerweise möchten Sie eine in der "lokalen" Zeitzone anzeigen, aber die Formatierungsmethoden müssen berücksichtigen, dass Sie sie möglicherweise für eine andere Zeitzone anzeigen möchten.

Beachten Sie außerdem, dass die von JVMs verwendeten Uhren normalerweise keine Millisekundengenauigkeit haben. Die Uhr kann nur alle 10 Millisekunden "ticken". Wenn Sie also die Zeit steuern, können Sie sich nicht darauf verlassen, die Dinge auf diesem Pegel genau zu messen.

Anweisung importieren

```
import java.util.Date;
```

Die `Date` Klasse kann aus dem `java.util` Paket importiert werden.

Vorsicht

`Date` sind veränderlich, sodass die Verwendung dieser Codes das Schreiben von threadsicherem Code erschweren oder versehentlich Schreibzugriff auf den internen Status ermöglichen kann. In der untenstehenden Klasse ermöglicht die Methode `getDate()` beispielsweise dem Aufrufer, das Transaktionsdatum zu ändern:

```
public final class Transaction {
  private final Date date;

  public Date getTransactionDate() {
    return date;
  }
}
```

Die Lösung ist, entweder eine Kopie des `date` oder die in Java 8 eingeführten neuen APIs in `java.time` .

Die meisten Konstruktormethoden in der `Date` Klasse sind veraltet und sollten nicht verwendet werden. In fast allen Fällen ist es ratsam, die `Calendar` Klasse für Datumsvorgänge zu verwenden.

Java 8

Java 8 führt eine neue Zeit- und Datums-API in das Paket `java.time` , einschließlich LocalDate und LocalTime . Die Klassen im Paket `java.time` bieten eine überarbeitete API, die einfacher zu verwenden ist. Wenn Sie in Java 8 schreiben, wird dringend empfohlen, dass Sie diese neue API verwenden. Siehe Datum und Uhrzeit (java.time. *) .

Examples

Date-Objekte erstellen

```
Date date = new Date();
System.out.println(date); // Thu Feb 25 05:03:59 IST 2016
```

Dieses `Date` Objekt enthält hier das aktuelle Datum und die Uhrzeit, zu der dieses Objekt erstellt wurde.

```
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` werden am besten über eine `Calendar` da die Verwendung der Datenkonstruktoren veraltet und nicht empfohlen wird. Dazu benötigen wir eine Instanz der `Calendar` Klasse von der Factory-Methode. Dann können Sie Jahr, Monat und Tag des Monats mithilfe von Zahlen oder im Fall von Monaten festgelegter Konstanten in der Klasse Calendar festlegen, um die Lesbarkeit zu verbessern und Fehler zu reduzieren.

```
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
```

Neben dem Datum können wir auch die Zeit in der Reihenfolge Stunde, Minuten und Sekunden übergeben.

Date-Objekte vergleichen

Kalender, Datum und LocalDate

Java SE 8

vor, nach, compareTo und gleich Methoden

```
//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 und equals Methoden

```
//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)));
```

Datumsvergleich vor Java 8

Vor Java 8 konnten Datumsangaben mit den Klassen java.util.Calendar und java.util.Date verglichen werden. Datumsklasse bietet 4 Methoden zum Vergleich von Datumsangaben:

- nach (Datum wann)
- vor (Datum wann)
- compareTo (Date anotherDate)
- gleich (Objekt obj)

`after` , `before` , `compareTo` und `equals` Methoden , um die Werte zurückgegeben durch Vergleichen getTime () Methode für jedes Datum.

`compareTo` Methode gibt eine positive ganze Zahl zurück.

- Wert größer als 0: Wenn das Datum nach dem Datumsargument liegt
- Wert größer als 0: Wenn das Datum vor dem Datumsargument liegt
- Wert gleich 0: wenn Date gleich dem Date-Argument ist

`equals` Ergebnisse können, wie im Beispiel gezeigt, überraschend sein, da Werte wie Millisekunden nicht mit dem gleichen Wert initialisiert werden, wenn dies nicht ausdrücklich angegeben ist.

Seit Java 8

Mit Java 8 ist ein neues Objekt für die Arbeit mit Date java.time.LocalDate verfügbar. `LocalDate` implementiert ChronoLocalDate , die abstrakte Darstellung eines Datums, an dem das Chronologie- oder Kalendersystem steckbar ist.

Um die Datumszeitgenauigkeit zu erhalten, muss das Objekt java.time.LocalDateTime verwendet werden. `LocalDate` und `LocalDateTime` verwenden zum Vergleich den gleichen Methodennamen.

Der Vergleich von Datumsangaben mit einem `LocalDate` unterscheidet sich von der Verwendung von `ChronoLocalDate` da die Chronologie oder das Kalendersystem nicht als erstes berücksichtigt werden.

Da die meisten Anwendungen `LocalDate` verwenden `LocalDate` , ist `ChronoLocalDate` nicht in den Beispielen enthalten. Lesen Sie hier weiter .

> Die meisten Anwendungen sollten Methodensignaturen, Felder und Variablen als LocalDate deklarieren, nicht diese Schnittstelle [ChronoLocalDate].

`LocalDate` bietet 5 Methoden zum Vergleich von Daten:

- isAfter (ChronoLocalDate andere)
- isBefore (ChronoLocalDate andere)
- isEqual (ChronoLocalDate andere)
- compareTo (ChronoLocalDate andere)
- gleich (Objekt obj)

Im Falle des `LocalDate` Parameters ist `isAfter` , `isBefore` , `isEqual` , `equals` und `compareTo` nun diese Methode:

```
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` Methode prüft, ob die Parameterreferenz dem Datum zuerst entspricht, während `isEqual` direkt `compareTo0`

Bei einer anderen Klasseninstanz von `ChronoLocalDate` die Datumsangaben anhand des `Epoch Day` verglichen. Die Epochen-Tageszählung ist eine einfache Erhöhung der Anzahl von Tagen, an denen Tag 0 1970-01-01 (ISO) ist.

Datum in ein bestimmtes String-Format konvertieren

`format()` aus der `SimpleDateFormat` Klasse hilft beim Konvertieren eines `Date` Objekts in ein bestimmtes Format- `String` Objekt mithilfe der angegebenen *Musterzeichenfolge* .

```
Date today = new Date();

SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MMM-yy"); //pattern is specified here
System.out.println(dateFormat.format(today)); //25-Feb-16
```

Muster können mit `applyPattern()` erneut angewendet werden

```
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
```

Hinweis: Hier steht `mm` (kleiner Buchstabe m) für Minuten und `MM` (Großbuchstabe M) für Monat.

Achten Sie besonders bei der Formatierung Jahre: Kapital „Y“ (`Y`) gibt die „Woche im Jahr“ , während Klein „y“ (`y`) das Jahr angibt.

String in Datum konvertieren

`SimpleDateFormat` `parse()` aus der `SimpleDateFormat` Klasse hilft beim Konvertieren eines `String` Musters in ein `Date` Objekt.

```
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
```

Es gibt 4 verschiedene Formatvorlagen für das Textformat: `SHORT` , `MEDIUM` (Standardeinstellung), `LONG` und `FULL` , die jeweils vom Gebietsschema abhängen. Wenn kein Gebietsschema angegeben ist, wird das Systemgebietsschema verwendet.

Stil	Locale.US	Locale.France
KURZ	30.6.09	30/06/09
MITTEL	30. Juni 2009	30. Juni 2009
LANGE	30. Juni 2009	30. Juni 2009
VOLL	Dienstag, 30. Juni 2009	30. Juni 2009

Eine grundlegende Datumsausgabe

Wenn Sie den folgenden Code mit dem Formatstring `yyyy/MM/dd hh:mm.ss` , erhalten Sie die folgende Ausgabe

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()));
```

Konvertieren Sie die formatierte Zeichenfolgendarstellung des Datums in das Date-Objekt

Mit dieser Methode kann eine formatierte Zeichenfolgendarstellung eines Datums in ein `Date` Objekt konvertiert werden.

```
    /**
     * 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;
    }
```

Ein bestimmtes Datum erstellen

Während die Java Date-Klasse mehrere Konstruktoren hat, werden Sie feststellen, dass die meisten veraltet sind. Die einzige akzeptable Methode zum direkten Erstellen einer Date-Instanz ist entweder die Verwendung des leeren Konstruktors oder die Übergabe eines langen Zeitraums (Anzahl Millisekunden seit Standardzeit). Beides ist nicht praktisch, es sei denn, Sie suchen nach dem aktuellen Datum oder haben bereits eine andere Datumsinstanz in der Hand.

Um ein neues Datum zu erstellen, benötigen Sie eine Kalenderinstanz. Von dort aus können Sie die Kalenderinstanz auf das gewünschte Datum einstellen.

```
Calendar c = Calendar.getInstance();
```

Dadurch wird eine neue Kalenderinstanz auf die aktuelle Uhrzeit zurückgegeben. Kalender verfügt über viele Methoden, um Datum und Uhrzeit zu ändern oder ganz einfach einzustellen. In diesem Fall setzen wir es auf ein bestimmtes Datum.

```
c.set(1974, 6, 2, 8, 0, 0);
Date d = c.getTime();
```

Die `getTime` Methode gibt die Date-Instanz zurück, die wir benötigen. Beachten Sie, dass mit den Kalendersatzmethoden nur ein oder mehrere Felder festgelegt werden, sie jedoch nicht alle. Das heißt, wenn Sie das Jahr einstellen, bleiben die anderen Felder unverändert.

FALLE

In vielen Fällen erfüllt dieses Code-Snippet seinen Zweck. Beachten Sie jedoch, dass zwei wichtige Teile des Datums / der Uhrzeit nicht definiert sind.

- Die Parameter `(1974, 6, 2, 8, 0, 0)` werden innerhalb der an anderer Stelle definierten Standardzeitzone interpretiert.
- Die Millisekunden werden nicht auf Null gesetzt, sondern zum Zeitpunkt der Erstellung der Kalenderinstanz von der Systemuhr aufgefüllt.

Java 8 LocalDate- und LocalDateTime-Objekte

Date- und LocalDate-Objekte **können nicht** *genau miteinander* konvertiert werden, da ein Date-Objekt einen bestimmten Tag und eine bestimmte Uhrzeit darstellt, während ein LocalDate-Objekt keine Zeit- oder Zeitzoneninformationen enthält. Es kann jedoch nützlich sein, zwischen den beiden zu konvertieren, wenn Sie nur die tatsächlichen Datumsinformationen und nicht die Zeitangaben berücksichtigen.

Erzeugt ein 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());
```

Erzeugt eine LocalDateTime

```
// Create a default date time
LocalDateTime lDateTime = LocalDateTime.now();

// Creates a date time from values
lDateTime = LocalDateTime.of(2017, 12, 15, 11, 30);

// create a date time from string
lDateTime = LocalDateTime.parse("2017-12-05T11:30:30");

// create a date time from zone
LocalDateTime.now(ZoneId.systemDefault());
```

LocalDate to Date und umgekehrt

```
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 und umgekehrt

```
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());
```

Zeitzonen und java.util.Date

Ein `java.util.Date` Objekt *verfügt nicht* über ein Konzept der Zeitzone.

- Es gibt keine Möglichkeit , eine Zeitzone für ein Datum **festlegen**
- Es ist nicht möglich, die Zeitzone eines Date-Objekts zu **ändern**
- Ein Date-Objekt, das mit dem `new Date()` Standardkonstruktor `new Date()` wird mit der aktuellen Uhrzeit in der Standardzeitzone des Systems initialisiert

Es ist jedoch möglich, das durch den Date-Objekt beschriebene Datum in einer anderen Zeitzone `java.text.SimpleDateFormat` , z. B. mit `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));
```

Ausgabe:

```
Central European Time
2016-07-21 22:50:56
2016-07-21 21:50:56
```

Konvertieren Sie java.util.Date in java.sql.Date

`java.sql.Date` Konvertierung von `java.util.Date` in `java.sql.Date` ist normalerweise erforderlich, wenn ein Date-Objekt in eine Datenbank geschrieben werden muss.

`java.sql.Date` ist ein Wrapper um den Millisekundenwert und wird von `JDBC` zur Identifizierung eines `SQL DATE` Typs verwendet

Im folgenden Beispiel verwenden wir den Konstruktor `java.util.Date()` , der ein Date-Objekt erstellt und es initialisiert, um die Zeit auf die nächste Millisekunde darzustellen. Dieses Datum wird in der `convert(java.util.Date utilDate)` -Methode verwendet, um ein `java.sql.Date` Objekt zurückzugeben

Beispiel

```
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;
    }

}
```

Ausgabe

```
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` enthält sowohl Datums- als auch Zeitinformationen, wohingegen `java.sql.Date` nur `java.sql.Date` enthält

Ortszeit

Um nur den Zeitteil eines Datums zu verwenden, verwenden Sie LocalTime. Sie können ein LocalTime-Objekt auf mehrere Arten instanziieren

1. `LocalTime time = LocalTime.now();`
2. `time = LocalTime.MIDNIGHT;`
3. `time = LocalTime.NOON;`
4. `time = LocalTime.of(12, 12, 45);`

`LocalTime` verfügt außerdem über eine integrierte toString-Methode, die das Format sehr gut anzeigt.

```
System.out.println(time);
```

Sie können Stunden, Minuten, Sekunden und Nanosekunden auch vom LocalTime-Objekt abrufen, addieren und subtrahieren

```
time.plusMinutes(1);
time.getMinutes();
time.minusMinutes(1);
```

Sie können es mit dem folgenden Code in ein Date-Objekt umwandeln:

```
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);
```

Diese Klasse funktioniert sehr gut innerhalb einer Timer-Klasse, um einen Wecker zu simulieren.

Datumsklasse online lesen: https://riptutorial.com/de/java/topic/164/datumsklasse

Kapitel 31: Demontieren und Dekompilieren

Syntax

- javap [Optionen] <Klassen>

Parameter

Name	Beschreibung
`<classes>`	Liste der zu zerlegenden Klassen. Kann entweder `package1.package2.Classname` Format `package1.package2.Classname` *oder* im Format `package1/package2/Classname` vorliegen. `.class` Sie **nicht** die Erweiterung `.class` .
`-help` , `--help` , `-?`	Diese Nutzungsnachricht ausdrucken
`-version`	Versionsinformation
`-v` , `-verbose`	Zusätzliche Informationen drucken
`-l`	Drucken Sie Zeilennummern und lokale Variablentabellen
`-public`	Nur öffentliche Klassen und Mitglieder anzeigen
`-protected`	Geschützte / öffentliche Klassen und Mitglieder anzeigen
`-package`	Paket / protected / public-Klassen und -Mitglieder anzeigen (Standard)
`-p` , `-private`	Zeige alle Klassen und Mitglieder
`-c`	Zerlegen Sie den Code
`-s`	Interne Typunterschriften drucken
`-sysinfo`	Zeigt Systeminformationen (Pfad, Größe, Datum, MD5-Hash) der verarbeiteten Klasse an
`-constants`	Endkonstanten anzeigen
`-classpath <path>`	Geben Sie an, wo Benutzerklassendateien gesucht werden sollen
`-cp <path>`	Geben Sie an, wo Benutzerklassendateien gesucht werden sollen
`-bootclasspath <path>`	Überschreibt den Speicherort der Bootstrap-Klassendateien

Examples

Anzeige des Bytecodes mit Javap

Wenn Sie den generierten Bytecode für ein Java-Programm `javap` möchten, können Sie ihn mit dem bereitgestellten Befehl `javap` anzeigen.

Angenommen, wir haben die folgende Java-Quelldatei:

```
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");
    }
}
```

Nach dem Kompilieren der Quelldatei ist die einfachste Verwendung:

```
cd <directory containing classes> (e.g. target/classes)
javap com/stackoverflow/documentation/SpringExample
```

Was erzeugt die Ausgabe

```
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();
}
```

Dies listet alle nicht privaten Methoden in der Klasse auf. Dies ist jedoch für die meisten Zwecke nicht besonders nützlich. Der folgende Befehl ist viel nützlicher:

```
javap -p -c -s -constants -l -v com/stackoverflow/documentation/HelloWorldService
```

Was erzeugt die Ausgabe:

```
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()
```

Demontieren und Dekompilieren online lesen:
https://riptutorial.com/de/java/topic/2318/demontieren-und-dekompilieren

Kapitel 32: Dequeue-Schnittstelle

Einführung

Ein Deque ist eine lineare Sammlung, die das Einfügen und Entfernen von Elementen an beiden Enden unterstützt.

Der Name deque ist eine Abkürzung für "double ended queue" und wird normalerweise als "deck" ausgesprochen.

Bei den meisten Deque-Implementierungen sind der Anzahl der Elemente, die sie enthalten können, keine festen Grenzen gesetzt, aber diese Schnittstelle unterstützt kapazitätsbeschränkte Deques sowie solche, die keine feste Größenbegrenzung aufweisen.

Die Deque-Schnittstelle ist ein umfangreicherer abstrakter Datentyp als sowohl Stack als auch Queue, da sie gleichzeitig Stacks und Warteschlangen implementiert

Bemerkungen

Generics können mit Deque verwendet werden.

```
Deque<Object> deque = new LinkedList<Object>();
```

Wenn ein Deque als Warteschlange verwendet wird, führt dies zu einem FIFO-Verhalten (First In In First Out).

Deques können auch als LIFO-Stapel (Last-In-First-Out) verwendet werden.

Weitere Informationen zu Methoden finden Sie in dieser Dokumentation.

Examples

Elemente zum Deque hinzufügen

```
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");
```

Elemente aus Deque entfernen

```
//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();
```

Element abrufen ohne zu entfernen

```
//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();
```

Iteration durch 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;
}
```

Dequeue-Schnittstelle online lesen: https://riptutorial.com/de/java/topic/10156/dequeue-schnittstelle

Kapitel 33: Der Java-Befehl - 'Java' und 'Javaw'

Syntax

- `java [ <opt> ... ] <class-name> [ <argument> ... ]`
- `java [ <opt> ... ] -jar <jar-file-pathname> [ <argument> ... ]`

Bemerkungen

Der `java` Befehl wird zum Ausführen einer Java-Anwendung von der Befehlszeile aus verwendet. Es ist als Bestandteil von Java SE JRE oder JDK verfügbar.

Auf Windows-Systemen gibt es zwei Varianten des `java` Befehls:

- Die `java` Variante startet die Anwendung in einem neuen Konsolenfenster.
- Die `javaw` Variante startet die Anwendung, ohne ein neues Konsolenfenster zu erstellen.

Auf anderen Systemen (z. B. Linux, Mac OSX, UNIX) wird nur der `java` Befehl bereitgestellt, und es wird kein neues Konsolenfenster geöffnet.

Das `<opt>` -Symbol in der Syntax kennzeichnet eine Option in der `java` Befehlszeile. Die Themen "Java-Optionen" und "Heap- und Stack-Sizing-Optionen" behandeln die am häufigsten verwendeten Optionen. Weitere werden im Thema JVM-Flags behandelt .

Examples

Ausführen einer ausführbaren JAR-Datei

Ausführbare JAR-Dateien sind der einfachste Weg, Java-Code in einer einzigen Datei zusammenzustellen, die ausgeführt werden kann. * (Anmerkung der Redaktion: Die Erstellung von JAR-Dateien sollte in einem eigenen Thema behandelt werden.) *

Vorausgesetzt, Sie haben eine ausführbare JAR-Datei mit dem Pfadnamen `<jar-path>` , sollten Sie sie wie folgt ausführen können:

```
java -jar <jar-path>
```

Wenn für den Befehl Befehlszeilenargumente erforderlich sind, fügen Sie sie nach dem `<jar-path>` . Zum Beispiel:

```
java -jar <jar-path> arg1 arg2 arg3
```

Wenn Sie zusätzliche JVM-Optionen in der `java` Befehlszeile `-jar` müssen, müssen diese *vor* der

Option `-jar` . Beachten Sie, dass eine Option `-cp` / `-classpath` ignoriert wird, wenn Sie `-jar` . Der Klassenpfad der Anwendung wird durch das Manifest der JAR-Datei bestimmt.

Ausführen von Java-Anwendungen über eine "main" -Klasse

Wenn eine Anwendung nicht als ausführbare JAR-Datei gepackt wurde, müssen Sie den Namen einer Einstiegspunktklasse in der `java` Befehlszeile angeben.

Die HelloWorld-Klasse ausführen

Das Beispiel "HelloWorld" wird unter Erstellen eines neuen Java-Programms beschrieben . Sie besteht aus einer einzigen Klasse namens `HelloWorld` die die Anforderungen für einen Einstiegspunkt erfüllt.

Unter der Annahme, dass sich die (kompilierte) Datei "HelloWorld.class" im aktuellen Verzeichnis befindet, kann sie wie folgt gestartet werden:

```
java HelloWorld
```

Einige wichtige Dinge zu beachten sind:

- Wir müssen den Namen der Klasse angeben, nicht den Pfadnamen für die ".class" -Datei oder die ".java" -Datei.
- Wenn die Klasse in einem Paket deklariert ist (wie bei den meisten Java-Klassen), muss der Klassenname, den wir dem Befehl `java` angeben, der vollständige Klassenname sein. Wenn `SomeClass` beispielsweise im Paket `SomeClass` deklariert ist, `com.example` der vollständige Klassenname `com.example.SomeClass` .

Klassenpfad angeben

Wenn wir nicht in der `java -jar` Befehlssyntax verwenden, sucht der `java` Befehl nach der zu ladenden Klasse, indem er den Klassenpfad durchsucht. siehe The Classpath . Der obige Befehl setzt voraus, dass der Standardklassenpfad das aktuelle Verzeichnis ist (oder das aktuelle Verzeichnis enthält). Wir können dies expliziter machen, indem wir den Klassenpfad `-cp` , der mit der Option `-cp` verwendet werden `-cp` .

```
java -cp . HelloWorld
```

Dies bedeutet, dass das aktuelle Verzeichnis (worauf sich "." Bezieht) der einzige Eintrag im Klassenpfad ist.

Die Option `-cp` ist eine Option, die vom Befehl `java` verarbeitet wird. Alle Optionen, die für den `java` Befehl vorgesehen sind, sollten vor dem Klassennamen stehen. Alles nach der Klasse wird als Befehlszeilenargument für die Java-Anwendung behandelt und an die Anwendung in der `String[]` , die an die `main` wird.

(Wenn keine Option `-cp` ist, verwendet `java` den Klassenpfad, der von der Umgebungsvariablen

`CLASSPATH` wird. Wenn diese Variable nicht festgelegt oder leer ist, verwendet `java` "." Als Standardklassenpfad.)

Einstiegspunktklassen

Eine Java - Entry-Point - Klasse hat ein `main` mit folgenden Signatur und Modifikatoren:

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

> Randbemerkung: aufgrund der Funktionsweise von Arrays kann es auch sein `(String args[])`

Wenn der `java` Befehl die virtuelle Maschine startet, lädt sie die angegebenen Einstiegsklassen und versucht, nach `main` zu suchen. Bei Erfolg werden die Argumente von der Befehlszeile in Java- `String` Objekte konvertiert und zu einem Array zusammengefügt. Wenn `main` wie diese aufgerufen wird, wird das Array *nicht* `null` und wird keine enthalten `null` Einträge.

Eine gültige Einstiegspunkt-Klassenmethode muss Folgendes tun:

- `main` (Groß- und Kleinschreibung beachten)
- Seien Sie `public` und `static`
- Einen `void` Rückgabetyp haben
- Habe ein einzelnes Argument mit einem Array `String[]` . Das Argument muss vorhanden sein und nicht mehr als ein Argument ist zulässig.
- Seien Sie generisch: Typparameter sind nicht zulässig.
- Verfügt über eine nicht generische Klasse der obersten Ebene (nicht verschachtelt oder inner)

Es ist üblich, die Klasse als `public` zu deklarieren, dies ist jedoch nicht unbedingt erforderlich. Ab Java 5 kann der Argumenttyp der `main` eine `String` Varargs anstelle eines String-Arrays sein. `main` kann optional Ausnahmen auslösen, und der Parameter kann beliebig benannt werden, üblicherweise sind dies jedoch `args` .

JavaFX-Einstiegspunkte

Ab Java 8 kann der `java` Befehl auch eine JavaFX-Anwendung direkt starten. JavaFX ist im JavaFX- Tag dokumentiert, ein JavaFX-Einstiegspunkt muss jedoch Folgendes ausführen:

- Erweitern Sie `javafx.application.Application`
- Seien Sie `public` und nicht `abstract`
- Nicht generisch oder verschachtelt sein
- Einen expliziten oder impliziten `public` No-Args-Konstruktor haben

Fehlerbehebung beim Befehl 'java'

In diesem Beispiel werden häufig auftretende Fehler bei der Verwendung des Befehls 'java' behandelt.

"Befehl nicht gefunden"

Wenn Sie eine Fehlermeldung erhalten, wie:

```
java: command not found
```

Wenn Sie versuchen, den `java` Befehl auszuführen, bedeutet dies, dass sich im Befehlssuchpfad der Shell kein `java` Befehl befindet. Die Ursache könnte sein:

- Sie haben überhaupt keine Java JRE oder JDK installiert,
- Sie haben die Umgebungsvariable `PATH` in Ihrer Shell-Initialisierungsdatei nicht (richtig) aktualisiert, oder
- Sie haben die entsprechende Initialisierungsdatei in der aktuellen Shell nicht "beschafft".

Informationen zu den erforderlichen Schritten finden Sie unter "Java installieren" .

"Hauptklasse konnte nicht gefunden oder geladen werden"

Diese Fehlermeldung wird vom Befehl `java` ausgegeben, wenn die angegebene Einstiegspunktklasse nicht gefunden / geladen werden konnte. Im Allgemeinen gibt es drei Gründe, warum dies passieren kann:

- Sie haben eine Einstiegspunktklasse angegeben, die nicht vorhanden ist.
- Die Klasse ist vorhanden, aber Sie haben sie falsch angegeben.
- Die Klasse ist vorhanden und Sie haben sie richtig angegeben. Java kann sie jedoch nicht finden, da der Klassenpfad falsch ist.

Hier ist ein Verfahren zur Diagnose und Lösung des Problems:

1. Finden Sie den vollständigen Namen der Einstiegspunktklasse heraus.

    - Wenn Sie über Quellcode für eine Klasse verfügen, besteht der vollständige Name aus dem Paketnamen und dem einfachen Klassennamen. Die Instanz der "Main" -Klasse wird im Paket "com.example.myapp" deklariert. Der vollständige Name lautet "com.example.myapp.Main".
    - Wenn Sie eine kompilierte Klassendatei haben, können Sie den Klassennamen finden, indem Sie `javap` darauf `javap` .
    - Wenn sich die Klassendatei in einem Verzeichnis befindet, können Sie den vollständigen Klassennamen aus den Verzeichnisnamen ableiten.
    - Wenn sich die Klassendatei in einer JAR- oder ZIP-Datei befindet, können Sie den vollständigen Klassennamen aus dem Dateipfad in der JAR- oder ZIP-Datei ableiten.

2. Sehen Sie sich die Fehlermeldung des `java` Befehls an. Die Nachricht sollte mit dem vollständigen Klassennamen enden, den `java` verwendet.

    - Stellen Sie sicher, dass er genau mit dem vollständigen Klassennamen für die Einstiegsklasse übereinstimmt.

- Es sollte nicht mit ".java" oder ".class" enden.
- Es sollte keine Schrägstriche oder andere Zeichen enthalten, die in einem Java-Bezeichner nicht zulässig sind [1] .
- Das Gehäuse des Namens sollte genau mit dem vollständigen Klassennamen übereinstimmen.

3. Wenn Sie den richtigen Klassennamen verwenden, stellen Sie sicher, dass sich die Klasse tatsächlich im Klassenpfad befindet:

    - Ermitteln Sie den Pfadnamen, dem der Klassenname zugeordnet ist. Siehe Klassennamen zu Pfadnamen zuordnen
    - Ermitteln Sie, was der Klassenpfad ist. Sehen Sie sich dieses Beispiel an: Verschiedene Möglichkeiten, den Klassenpfad anzugeben
    - Sehen Sie sich die JAR- und ZIP-Dateien im Klassenpfad an, um zu sehen, ob sie eine Klasse mit dem erforderlichen Pfadnamen enthalten.
    - Sehen Sie sich jedes Verzeichnis an, um zu sehen, ob der Pfadname in eine Datei innerhalb des Verzeichnisses aufgelöst wird.

Wenn der Klassenpfad von Hand nicht gefunden wurde, konnten Sie die Optionen `-Xdiag` und `-XshowSettings` hinzufügen. Ersteres listet alle geladenen Klassen auf, und letzteres gibt Einstellungen aus, die den effektiven Klassenpfad für die JVM enthalten.

Schließlich gibt es einige *dunkle* Ursachen für dieses Problem:

- Eine ausführbare JAR-Datei mit einem Hauptklassenattribut, das eine nicht vorhandene `Main-Class` angibt.
- Eine ausführbare JAR-Datei mit einem falschen `Class-Path` Attribut.
- Wenn Sie vermasseln [2] die Optionen vor dem Klassennamen, die `java` versucht Befehl kann eine von ihnen als Klassennamen zu interpretieren.
- Wenn jemand Java-Stilregeln ignoriert und Paket- oder Klassenbezeichner verwendet hat, die sich nur in Groß- und Kleinschreibung unterscheiden, und Sie auf einer Plattform ausgeführt werden, die Groß- / Kleinschreibung in Dateinamen als nicht signifikant behandelt.
- Probleme mit Homoglyphen in Klassennamen im Code oder in der Befehlszeile.

"Hauptmethode in Klasse <Name> nicht gefunden"

Dieses Problem tritt auf, wenn der `java` Befehl die von Ihnen benannte Klasse finden und laden kann, aber keine Einstiegspunktmethode finden kann.

Es gibt drei mögliche Erklärungen:

- Wenn Sie versuchen, eine ausführbare JAR-Datei auszuführen, hat das JAR-Manifest ein falsches "Main-Class" -Attribut, das eine Klasse angibt, die keine gültige Einstiegspunktklasse ist.
- Sie haben dem `java` Befehl eine Klasse mitgeteilt, die keine Einstiegspunktklasse ist.
- Die Einstiegspunktklasse ist falsch. Weitere Informationen finden Sie unter Einstiegspunktklassen .

Andere Ressourcen

- Was bedeutet "Konnte Hauptklasse nicht finden oder laden"?
- http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html

---

1 - Ab Java 8 ordnet der `java` Befehl hilfreich ein Dateinamen-Trennzeichen ("/" oder "") einem Punkt (".") Zu. Dieses Verhalten ist jedoch nicht in den Handbuchseiten dokumentiert.

2 - Ein wirklich verdeckter Fall liegt vor, wenn Sie einen Befehl aus einem formatierten Dokument kopieren und einfügen, wenn der Texteditor einen "langen Bindestrich" anstelle eines normalen Bindestrichs verwendet hat.

Ausführen einer Java-Anwendung mit Bibliotheksabhängigkeiten

Dies ist eine Fortsetzung der Beispiele "Hauptklasse" und "ausführbare JAR" .

Typische Java-Anwendungen bestehen aus einem anwendungsspezifischen Code und verschiedenen wiederverwendbaren Bibliothekscodes, die Sie implementiert haben oder von Dritten implementiert wurden. Letztere werden im Allgemeinen als Bibliotheksabhängigkeiten bezeichnet und werden normalerweise als JAR-Dateien verpackt.

Java ist eine dynamisch gebundene Sprache. Wenn Sie eine Java-Anwendung mit Bibliotheksabhängigkeiten ausführen, muss die JVM wissen, wo sich die Abhängigkeiten befinden, damit Klassen nach Bedarf geladen werden können. Im Großen und Ganzen gibt es zwei Möglichkeiten, damit umzugehen:

- Die Anwendung und ihre Abhängigkeiten können in eine einzige JAR-Datei gepackt werden, die alle erforderlichen Klassen und Ressourcen enthält.
- Der JVM kann über den Laufzeitklassenpfad mitgeteilt werden, wo die abhängigen JAR-Dateien zu finden sind.

Für eine ausführbare JAR-Datei wird der Laufzeitklassenpfad durch das Manifestattribut "Class-Path" angegeben. *(Anmerkung der Redaktion: Dies sollte in einem eigenen Thema im `jar` Befehl beschrieben werden.)* Andernfalls muss der Klassenpfad der Laufzeit mithilfe der Option `-cp` oder der Umgebungsvariable `CLASSPATH` werden.

Angenommen, wir haben eine Java-Anwendung in der Datei "myApp.jar", deren Einstiegspunktklasse `com.example.MyApp` . Angenommen, die Anwendung hängt von den Bibliotheks-JAR-Dateien "lib / library1.jar" und "lib / library2.jar" ab. Wir könnten die Anwendung mit dem `java` Befehl wie folgt in einer Befehlszeile starten:

```
$ # 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
```

(Unter Windows würden Sie ; anstelle von : als Klassenpfad-Trennzeichen verwenden und die

(lokale) `CLASSPATH` Variable mit `set` statt mit `export set` .)

Ein Java-Entwickler wäre damit zwar vertraut, aber nicht "benutzerfreundlich". Daher ist es üblich, ein einfaches Shellskript (oder eine Windows-Batchdatei) zu schreiben, um die Details auszublenden, über die der Benutzer nicht informiert sein muss. Wenn Sie beispielsweise das folgende Shell-Skript in eine Datei namens "myApp" ablegen, diese ausführbar machen und in einem Verzeichnis im Befehlssuchpfad ablegen:

```
#!/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
```

dann könntest du es wie folgt ausführen:

```
$ myApp arg1 arg2 ...
```

Alle Argumente in der Befehlszeile werden über die Erweiterung `"$@"` an die Java-Anwendung übergeben. (Mit einer Windows-Batchdatei können Sie etwas ähnliches machen, die Syntax ist jedoch unterschiedlich.)

Leerzeichen und andere Sonderzeichen in Argumenten

Zunächst ist das Problem der Behandlung von Leerzeichen in Argumenten NICHT ein Java-Problem. Es ist eher ein Problem, das von der Befehlsshell behandelt werden muss, die Sie beim Ausführen eines Java-Programms verwenden.

Nehmen wir als Beispiel an, wir haben das folgende einfache Programm, das die Größe einer Datei ausgibt:

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

Angenommen, wir möchten die Größe einer Datei drucken, deren Pfadname Leerzeichen enthält. zB `/home/steve/Test File.txt` . Wenn wir den Befehl so ausführen:

```
$ java PrintFileSizes /home/steve/Test File.txt
```

Die Shell weiß nicht, dass `/home/steve/Test File.txt` tatsächlich ein Pfadname ist. Stattdessen werden zwei verschiedene Argumente an die Java-Anwendung übergeben, die versucht, ihre

jeweilige Dateigröße zu ermitteln, und schlagen fehl, da Dateien mit diesen Pfaden (wahrscheinlich) nicht vorhanden sind.

Lösungen mit einer POSIX-Shell

POSIX-Shells umfassen `sh` sowie Derivate wie `bash` und `ksh` . Wenn Sie eine dieser Shells verwenden, können Sie das Problem lösen, *indem Sie* das Argument *zitieren* .

```
$ java PrintFileSizes "/home/steve/Test File.txt"
```

Die Anführungszeichen um den Pfadnamen geben der Shell an, dass sie als einzelnes Argument übergeben werden sollte. Die Anführungszeichen werden in diesem Fall entfernt. Es gibt mehrere Möglichkeiten, dies zu tun:

```
$ java PrintFileSizes '/home/steve/Test File.txt'
```

Einfache (gerade) Anführungszeichen werden wie Anführungszeichen behandelt, mit der Ausnahme, dass sie auch verschiedene Erweiterungen innerhalb des Arguments unterdrücken.

```
$ java PrintFileSizes /home/steve/Test\ File.txt
```

Ein Backslash entkommt dem folgenden Leerzeichen und bewirkt, dass es nicht als Argumenttrennzeichen interpretiert wird.

Eine umfassendere Dokumentation, einschließlich Beschreibungen zum Umgang mit anderen Sonderzeichen in Argumenten, finden Sie im Zitierthema der Bash- Dokumentation.

Lösung für Windows

Das grundlegende Problem für Windows ist, dass auf Betriebssystemebene die Argumente als einzelne Zeichenfolge (Quelle) an einen untergeordneten Prozess übergeben werden. Dies bedeutet, dass die endgültige Verantwortung für das Parsen (oder das erneute Parsen) der Befehlszeile entweder dem Programm oder dessen Laufzeitbibliotheken obliegt. Es gibt viele Unstimmigkeiten.

Im Java-Fall, um es kurz zu machen:

- Sie können ein Argument in einem `java` Befehl in doppelte Anführungszeichen setzen, um Argumente mit Leerzeichen zu übergeben.
- Anscheinend `java` der `java` Befehl selbst die Befehlszeichenfolge, und er erhält mehr oder weniger Recht
- Wenn Sie jedoch versuchen, dies mit `SET` und der Variablensubstitution in einer Batchdatei zu kombinieren, wird es sehr kompliziert, ob doppelte Anführungszeichen entfernt werden.
- Die Shell `cmd.exe` verfügt offenbar über andere Escape-Mechanismen. zB Verdoppelung

doppelte Anführungszeichen, und mit `^` entkommt.

Weitere Informationen finden Sie in der Batch-File- Dokumentation.

Java-Optionen

Der `java` Befehl unterstützt eine Vielzahl von Optionen:

- Alle Optionen beginnen mit einem einzelnen Bindestrich oder Minuszeichen (`-`): Die GNU / Linux-Konvention der Verwendung von `--` for "long" -Optionen wird nicht unterstützt.
- Optionen müssen vor dem Argument `<classname>` oder `-jar <jarfile>` , damit sie erkannt werden. Alle nachfolgenden Argumente werden als Argumente behandelt, die an die ausgeführte Java-App übergeben werden.
- Optionen, die nicht mit `-X` oder `-XX` sind Standardoptionen. Sie können sich auf alle Java-Implementierungen [1] verlassen , um jede Standardoption zu unterstützen.
- Optionen, die mit `-X` sind keine Standardoptionen und können von einer Java-Version zur nächsten zurückgezogen werden.
- Optionen, die mit `-XX` sind erweiterte Optionen und können auch zurückgezogen werden.

Systemeigenschaften mit `-D`

Die Option `-D<property>=<value>` wird verwendet, um eine Eigenschaft im Eigenschaftenobjekt des Systems `Properties` . Dieser Parameter kann wiederholt werden, um andere Eigenschaften festzulegen.

Optionen für Speicher, Stack und Garbage Collector

Die wichtigsten Optionen zum Steuern der Heap- und Stackgrößen sind in Einstellen der Heap-, PermGen- und Stackgrößen dokumentiert. *(Anmerkung der Redaktion: Garbage Collector-Optionen sollten in demselben Thema beschrieben werden.)*

Assertions aktivieren und deaktivieren

Die `-ea` und `-da` Optionen bzw. aktivieren und Java deaktivieren `assert` Überprüfung:

- Alle Assertionsprüfungen sind standardmäßig deaktiviert.
- Die Option `-ea` ermöglicht die Überprüfung aller Zusicherungen
- Die `-ea:<packagename>...` ermöglicht die Überprüfung von Zusicherungen in einem Paket *und allen Unterpaketen* .
- Die `-ea:<classname>...` ermöglicht die Überprüfung von Assertions in einer Klasse.
- Die Option `-da` deaktiviert die Prüfung aller Zusicherungen
- Die `-da:<packagename>...` deaktiviert die Prüfung von Zusicherungen in einem Paket *und allen untergeordneten Paketen* .

- Die `-da:<classname>...` deaktiviert die Prüfung von Assertions in einer Klasse.
- Die Option `-esa` ermöglicht die Überprüfung aller `-esa` .
- Die Option `-dsa` deaktiviert die Prüfung für alle `-dsa` .

Die Optionen können kombiniert werden. Zum Beispiel.

```
$ # 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
```

Beachten Sie, dass die Aktivierung der Assertionsprüfung das Verhalten einer Java-Programmierung ändern kann.

- Es ist wahrscheinlich, dass die Anwendung im Allgemeinen langsamer wird.
- Dies kann dazu führen, dass bestimmte Methoden länger dauern, was das Timing von Threads in einer Multithread-Anwendung ändern kann.
- Es kann *zufällige* Ereignisse *vor den* Beziehungen einleiten, die dazu führen können, dass Anomalien des Gedächtnisses verschwinden.
- Eine falsch implementierte `assert` Anweisung kann unerwünschte Nebenwirkungen haben.

Auswahl des VM-Typs

Mit den Optionen `-client` und `-server` können Sie zwischen zwei verschiedenen Formen der HotSpot-VM wählen:

- Das "Client" -Formular ist auf Benutzeranwendungen abgestimmt und ermöglicht einen schnelleren Start.
- Das "Server" -Formular ist auf lang laufende Anwendungen abgestimmt. Während des Aufwärmens der JVM dauert die Erfassungsstatistik länger, wodurch der JIT-Compiler die Optimierung des nativen Codes verbessern kann.

Standardmäßig wird die JVM, sofern möglich, je nach den Fähigkeiten der Plattform im 64-Bit-Modus ausgeführt. Mit den Optionen `-d32` und `-d64` können Sie den Modus explizit auswählen.

---

1 - Überprüfen Sie das offizielle Handbuch für den `java` Befehl. Manchmal wird eine *Standardoption* als "Änderungen vorbehalten" bezeichnet.

Der Java-Befehl - 'Java' und 'Javaw' online lesen: https://riptutorial.com/de/java/topic/5791/der-java-befehl----java--und--javaw-

Kapitel 34: Der Klassenpfad

Einführung

Der Klassenpfad listet die Bereiche auf, an denen die Java-Laufzeitumgebung nach Klassen und Ressourcen suchen soll. Der Klassenpfad wird auch vom Java-Compiler verwendet, um zuvor kompilierte und externe Abhängigkeiten zu finden.

Bemerkungen

Java-Klasse wird geladen

Die JVM (Java Virtual Machine) lädt Klassen, sobald die Klassen erforderlich sind (dies wird als Lazy-Loading bezeichnet). Die Standorte der zu verwendenden Klassen sind an drei Stellen angegeben: -

1. Die von der Java-Plattform benötigten werden zuerst geladen, z. B. diejenigen in der Java-Klassenbibliothek und ihre Abhängigkeiten.
2. Als nächstes werden Erweiterungsklassen geladen (dh die in `jre/lib/ext/`)
3. Benutzerdefinierte Klassen über den Klassenpfad werden dann geladen

Klassen werden mit Klassen geladen, die Untertypen von `java.lang.ClassLoader` . Dies wurde in diesem Thema ausführlicher beschrieben: Classloader .

Klassenpfad

Der Klassenpfad ist ein Parameter, der von der JVM oder dem Compiler verwendet wird und die Speicherorte benutzerdefinierter Klassen und Pakete angibt. Dies kann wie in den meisten dieser Beispiele in der Befehlszeile oder über eine Umgebungsvariable (`CLASSPATH`) festgelegt werden.

Examples

Es gibt verschiedene Möglichkeiten, den Klassenpfad anzugeben

Es gibt drei Möglichkeiten, den Klassenpfad festzulegen.

1. Es kann mit der Umgebungsvariable `CLASSPATH` werden:

```
set CLASSPATH=...          # Windows and csh
export CLASSPATH=...       # Unix ksh/bash
```

2. Sie kann in der Befehlszeile wie folgt festgelegt werden

```
java -classpath ...
javac -classpath ...
```

Beachten Sie, dass die Option `-classpath` (oder `-cp`) Vorrang vor der Umgebungsvariable `CLASSPATH` .

3. Der Klassenpfad für eine ausführbare JAR-Datei wird mit dem `Class-Path` Element in `MANIFEST.MF` :

   ```
   Class-Path: jar1-name jar2-name directory-name/jar3-name
   ```

   Beachten Sie, dass dies nur gilt, wenn die JAR-Datei folgendermaßen ausgeführt wird:

   ```
   java -jar some.jar ...
   ```

   In diesem Ausführungsmodus werden die Option `-classpath` und die Umgebungsvariable CLASSPATH ignoriert, auch wenn die JAR-Datei kein `Class-Path` Element enthält.

Wenn kein Klassenpfad angegeben ist, ist der Standardklassenpfad bei Verwendung von `java -jar` die ausgewählte JAR-Datei oder ansonsten das aktuelle Verzeichnis.

Verbunden:

- https://docs.oracle.com/javase/tutorial/deployment/jar/downman.html
- http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html

Hinzufügen aller JARs in einem Verzeichnis zum Klassenpfad

Wenn Sie alle JARs im Verzeichnis zum Klassenpfad hinzufügen möchten, können Sie dies mit der Klassenpfad-Platzhaltersyntax auf einfache Weise tun. zum Beispiel:

```
someFolder/*
```

Dadurch wird die JVM `someFolder` , alle JAR- und ZIP-Dateien im Verzeichnis `someFolder` zum Klassenpfad hinzuzufügen. Diese Syntax kann in verwendet wird `-cp` Argument, ein `CLASSPATH` - Umgebungsvariable oder ein `Class-Path` - Attribut in einer ausführbaren JAR - Datei Manifest file.See *Class Path Wild Cards: Einstellen des Klassenpfades* für Beispiele und Einsprüche.

Anmerkungen:

1. Klassenpfad-Platzhalter wurden erstmals in Java 6 eingeführt. Frühere Versionen von Java behandeln "*" nicht als Platzhalter.
2. Sie können keine anderen Zeichen vor oder nach dem " *" setzen; zB "someFolder* / .jar" ist kein Platzhalter.
3. Ein Platzhalter entspricht nur Dateien mit dem Suffix ".jar" oder ".JAR". ZIP-Dateien werden ignoriert, ebenso wie JAR-Dateien mit anderen Suffixen.
4. Ein Platzhalter entspricht nur JAR-Dateien im Verzeichnis selbst, nicht in seinen Unterverzeichnissen.
5. Wenn eine Gruppe von JAR-Dateien mit einem Platzhaltereintrag übereinstimmt, wird ihre relative Reihenfolge im Klassenpfad nicht angegeben.

Klassenpfadpfad-Syntax

Der Klassenpfad ist eine Folge von Einträgen, die Verzeichnispfadnamen, JAR- oder ZIP-Dateipfadnamen oder JAR / ZIP-Platzhalterspezifikationen sind.

- Für einen in der Befehlszeile angegebenen Klassenpfad (z. B. `-classpath`) oder als Umgebungsvariable müssen die Einträge mit `;` getrennt werden `;` (Semikolon) Zeichen unter Windows oder `:` (Doppelpunkt) Zeichen auf anderen Plattformen (Linux, UNIX, MacOSX usw.).

- Verwenden Sie für das `Class-Path` Element in `MANIFEST.MF` einer JAR-Datei ein einzelnes Leerzeichen, um die Einträge zu trennen.

Manchmal ist es notwendig, ein Leerzeichen in einen Klassenpfadeintrag einzubetten

- Wenn der Klassenpfad in der Befehlszeile angegeben wird, müssen Sie lediglich die entsprechenden Shell-Anführungszeichen verwenden. Zum Beispiel:

  ```
  export CLASSPATH="/home/user/My JAR Files/foo.jar:second.jar"
  ```

  (Die Details können von der von Ihnen verwendeten Befehlsshell abhängen.)

- Wenn der Klassenpfad in einer "MANIFEST.MF" -Datei einer JAR-Datei angegeben ist, muss die URL-Codierung verwendet werden.

  ```
  Class-Path: /home/user/My%20JAR%20Files/foo.jar second.jar
  ```

Dynamischer Klassenpfad

Manchmal reicht es nicht aus, nur alle JARs aus einem Ordner hinzuzufügen, beispielsweise wenn Sie über systemeigenen Code verfügen und eine Untermenge von JARs auswählen müssen. In diesem Fall benötigen Sie zwei `main()` -Methoden. Die erste erstellt einen Classloader und verwendet dann diesen Classloader, um das zweite `main()` aufzurufen.

Hier ein Beispiel, in dem Sie die richtige SWT-native JAR für Ihre Plattform auswählen, alle JARs Ihrer Anwendung hinzufügen und dann die echte `main()` Methode aufrufen: Erstellen Sie plattformübergreifende Java-SWT-Anwendung

Laden Sie eine Ressource aus dem Klassenpfad

Es kann nützlich sein, eine Ressource (Bild, Textdatei, Eigenschaften, KeyStore, ...) zu laden, die sich in einem JAR befindet. Zu diesem Zweck können wir die `Class` und `ClassLoader` .

Angenommen, wir haben die folgende Projektstruktur:

```
program.jar
|
\-com
```

```
  \-project
    |
    |-file.txt
    \-Test.class
```

Und wir möchten über die Klasse `Test` auf den Inhalt von `file.txt` zugreifen. Wir können dies tun, indem wir den Klassenlader fragen:

```
InputStream is = Test.class.getClassLoader().getResourceAsStream("com/project/file.txt");
```

Durch die Verwendung des Klassenladers müssen wir den vollständig qualifizierten Pfad unserer Ressource (jedes Paket) angeben.

Alternativ können wir das Klassenobjekt Test direkt abfragen

```
InputStream is = Test.class.getResourceAsStream("file.txt");
```

Bei Verwendung des Klassenobjekts ist der Pfad relativ zur Klasse selbst. `Test.class` unsere `Test.class` sich im Paket `com.project` und mit `file.txt` identisch ist, müssen wir überhaupt keinen Pfad angeben.

Wir können jedoch absolute Pfade aus dem Klassenobjekt verwenden:

```
 is = Test.class.getResourceAsStream("/com/project/file.txt");
```

Zuordnung von Klassennamen zu Pfadnamen

Die Standard-Java-Toolchain (und von Drittanbietern für die Interaktion mit ihnen entwickelte Tools) enthält spezielle Regeln für die Zuordnung der Namen von Klassen zu den Pfadnamen von Dateien und anderen Ressourcen, die diese repräsentieren.

Die Zuordnungen sind wie folgt

- Für Klassen im Standardpaket sind die Pfadnamen einfache Dateinamen.
- Bei Klassen in einem benannten Paket werden die Komponenten der Paketnamen den Verzeichnissen zugeordnet.
- Bei benannten verschachtelten und inneren Klassen wird die Dateinamenkomponente gebildet, indem die Klassennamen mit einem `$` -Zeichen verbunden werden.
- Bei anonymen inneren Klassen werden anstelle von Namen Zahlen verwendet.

Dies ist in der folgenden Tabelle dargestellt:

Klassenname	Quellpfadname	Pfadname der Klassendatei
`SomeClass`	`SomeClass.java`	`SomeClass.class`
`com.example.SomeClass`	`com/example/SomeClass.java`	`com/example/SomeClass.class`
`SomeClass.Inner`	(in `SomeClass.java`)	`SomeClass$Inner.class`

Klassenname	Quellpfadname	Pfadname der Klassendatei
`SomeClass` innere Klassen	(in `SomeClass.java`)	`SomeClass$1.class` , `SomeClass$2.class` usw

Was der Klassenpfad bedeutet: Wie Suchvorgänge funktionieren

Der Klassenpfad dient dazu, einer JVM mitzuteilen, wo Klassen und andere Ressourcen zu finden sind. Die Bedeutung des Klassenpfads und des Suchprozesses sind miteinander verknüpft.

Der Klassenpfad ist eine Form eines Suchpfads, der eine Abfolge von *Positionen* angibt, in denen nach Ressourcen gesucht werden soll. In einem Standardklassenpfad sind diese Bereiche entweder ein Verzeichnis im Host-Dateisystem, eine JAR-Datei oder eine ZIP-Datei. In jedem Fall ist der Speicherort der Stamm eines *Namespaces* , der durchsucht wird.

Das Standardverfahren für die Suche nach einer Klasse im Klassenpfad lautet wie folgt:

1. Ordnen Sie den Klassennamen einem relativen Klassendateipfad `RP` . Die Zuordnung von Klassennamen zu Klassendateinamen wird an anderer Stelle beschrieben.

2. Für jeden Eintrag `E` im Klassenpfad:

    - Wenn der Eintrag ein Dateisystemverzeichnis ist:
        - Löse `RP` relativ zu `E` , um einen absoluten Pfadnamen `AP` .
        - Testen Sie, ob `AP` ein Pfad für eine vorhandene Datei ist.
        - Wenn ja, laden Sie die Klasse aus dieser Datei
    - Wenn der Eintrag eine JAR- oder ZIP-Datei ist:
        - Nachschlagen von `RP` im JAR / ZIP-Dateiindex.
        - Wenn der entsprechende JAR / ZIP-Dateieintrag vorhanden ist, laden Sie die Klasse aus diesem Eintrag.

Das Verfahren zum Suchen einer Ressource im Klassenpfad hängt davon ab, ob der Ressourcenpfad absolut oder relativ ist. Für einen absoluten Ressourcenpfad ist das Verfahren wie oben beschrieben. Bei einem relativen Ressourcenpfad, der mit `Class.getResource` oder `Class.getResourceAsStream` , wird der Pfad für das Klassenpaket vor der Suche vorangestellt.

(Beachten Sie, dass dies die Prozeduren sind, die von den standardmäßigen Java-Klassenladern implementiert werden. Ein benutzerdefinierter Klassenlader führt die Suche möglicherweise anders aus.)

Der Bootstrap-Klassenpfad

Die normalen Java-Klassenladeprogramme suchen zuerst im Bootstrap-Klassenpfad nach Klassen, bevor sie nach Erweiterungen und dem Klassenpfad der Anwendung suchen. Der Bootstrap-Klassenpfad besteht standardmäßig aus der Datei "rt.jar" und einigen anderen wichtigen JAR-Dateien, die von der JRE-Installation bereitgestellt werden. Diese enthalten alle Klassen in der Standard-Java SE-Klassenbibliothek sowie verschiedene "interne" Implementierungsklassen.

Unter normalen Umständen müssen Sie sich nicht darum kümmern. Standardmäßig verwenden Befehle wie `java` , `javac` usw. die entsprechenden Versionen der Laufzeitbibliotheken.

In bestimmten Fällen ist es erforderlich, das normale Verhalten der Java-Laufzeitumgebung zu überschreiben, indem eine alternative Version einer Klasse in den Standardbibliotheken verwendet wird. Beispielsweise kann es in den Laufzeitbibliotheken zu einem "show stopper" - Fehler kommen, den Sie mit normalen Mitteln nicht umgehen können. In einer solchen Situation ist es möglich, eine JAR-Datei zu erstellen, die die geänderte Klasse enthält, und sie dem Bootstrap-Klassenpfad hinzuzufügen, der die JVM startet.

Der Befehl `java` bietet die folgenden `-X` Optionen zum Ändern des Bootstrap-Klassenpfads:

- `-Xbootclasspath:<path>` ersetzt den aktuellen Startklassenpfad durch den angegebenen Pfad.
- `-Xbootclasspath/a:<path>` hängt den angegebenen Pfad an den aktuellen Startklassenpfad an.
- `-Xbootclasspath/p:<path>` fügt den angegebenen Pfad dem aktuellen Startklassenpfad hinzu.

Beachten Sie, dass Sie bei Verwendung der Bootclasspath-Optionen zum Ersetzen oder Überschreiben einer Java-Klasse (usw.) Java technisch modifizieren. Wenn Sie Ihren Code dann verteilen, *kann dies* Auswirkungen auf *die* Lizenzierung haben. (Beachten Sie die Bedingungen der Java Binary License ... und wenden Sie sich an einen Anwalt.)

Der Klassenpfad online lesen: https://riptutorial.com/de/java/topic/3720/der-klassenpfad

Kapitel 35: Die java.util.Objects-Klasse

Examples

Grundlegende Verwendung für die Objektnullprüfung

Für die Nulleincheckmethode

```
Object nullableObject = methodReturnObject();
if (Objects.isNull(nullableObject)) {
    return;
}
```

Für nicht null Check-in-Methode

```
Object nullableObject = methodReturnObject();
if (Objects.nonNull(nullableObject)) {
    return;
}
```

Verwendung der Objects.nonNull () -Methode in Stream-API

Auf die alte Art und Weise für die Überprüfung der Sammlung

```
List<Object> someObjects = methodGetList();
for (Object obj : someObjects) {
    if (obj == null) {
        continue;
    }
    doSomething(obj);
}
```

Mit der `Objects.nonNull` Methode und der Java8 Stream-API können Sie dies auf folgende Weise tun:

```
List<Object> someObjects = methodGetList();
someObjects.stream()
           .filter(Objects::nonNull)
           .forEach(this::doSomething);
```

Die java.util.Objects-Klasse online lesen: https://riptutorial.com/de/java/topic/5768/die-java-util-objects-klasse

Kapitel 36: Durchsetzung

Syntax

- assert *expression1* ;
- assert *expression1* : *expression2* ;

Parameter

Parameter	Einzelheiten
expression1	Die Assertionsanweisung löst einen `AssertionError` wenn dieser Ausdruck als `false` ausgewertet wird.
expression2	Wahlweise. Bei der Verwendung von `AssertionError` s, die von der assert-Anweisung ausgelöst werden, wird diese Nachricht angezeigt.

Bemerkungen

Standardmäßig sind Assertions zur Laufzeit deaktiviert.

Um Assertionen zu aktivieren, müssen Sie Java mit der `-ea` .

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

Assertionen sind Anweisungen, die einen Fehler auslösen, wenn ihr Ausdruck als `false` ausgewertet wird. Zusicherungen sollten nur zum *Testen von* Code verwendet werden. Sie sollten niemals in der Produktion verwendet werden.

Examples

Prüfung der Arithmetik mit 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;
```

Durchsetzung online lesen: https://riptutorial.com/de/java/topic/407/durchsetzung

Kapitel 37: Dynamischer Methodenversand

Einführung

Was ist dynamischer Methodenversand?

Dynamic Method Dispatch ist ein Prozess, bei dem der Aufruf einer überschriebenen Methode zur Laufzeit und nicht zur Kompilierungszeit aufgelöst wird. Wenn eine überschriebene Methode von einer Referenz aufgerufen wird, bestimmt Java anhand des Objekttyps, auf die sie verweist, welche Version dieser Methode ausgeführt werden soll. Dies ist auch als Laufzeitpolymorphismus bekannt.

Wir werden das an einem Beispiel sehen.

Bemerkungen

- Dynamische Bindung = Späte Bindung
- Abstrakte Klassen können nicht instanziiert werden, sie können jedoch Unterklassen zugeordnet werden (Basis für eine untergeordnete Klasse).
- Eine abstrakte Methode ist eine Methode, die ohne Implementierung deklariert wird
- Abstrakte Klassen können eine Mischung von Methoden enthalten, die mit oder ohne Implementierung deklariert wurden
- Wenn eine abstrakte Klasse eine Unterklasse ist, stellt die Unterklasse normalerweise Implementierungen für alle abstrakten Methoden ihrer übergeordneten Klasse bereit. Wenn dies nicht der Fall ist, muss die Unterklasse auch als abstrakt deklariert werden
- Dynamic Method Dispatch ist ein Mechanismus, durch den ein Aufruf einer überschriebenen Methode zur Laufzeit aufgelöst wird. So implementiert Java Laufzeitpolymorphismus.
- Upcasting: Umwandeln eines Untertyps in einen Supertyp, aufwärts zum Vererbungsbaum.
- Laufzeitpolymorphismus = dynamischer Polymorphismus

Examples

Dynamic Method Dispatch - Beispielcode

Abstrakte Klasse :

```
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");
    }
}
```

Überschreiben der abstrakten Methode in der Kindklasse:

```
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
```

Dynamische / späte Bindung führt zum dynamischen Methodenversand:

```
//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();

    }
}
```

Ergebnis:

```
Scan for Internet Security
Detect Virus & Malware
```

Upcasting:

```
objVS = new ClsFreeVersion();
objVS = new ClsPaidVersion();
objVS = new ClsTrialVersion()
```

Dynamischer Methodenversand online lesen:
https://riptutorial.com/de/java/topic/9204/dynamischer-methodenversand

Kapitel 38: Eigenschaften Klasse

Einführung

Das Eigenschaftenobjekt enthält ein Schlüssel- und ein Wertepaar als Zeichenfolge. Die Klasse java.util.Properties ist die Unterklasse von Hashtable.

Es kann verwendet werden, um den Eigenschaftswert basierend auf dem Eigenschaftsschlüssel abzurufen. Die Properties-Klasse stellt Methoden zum Abrufen von Daten aus der Eigenschaftendatei und zum Speichern von Daten in der Eigenschaftendatei bereit. Darüber hinaus kann es verwendet werden, um Eigenschaften des Systems zu erhalten.

Vorteil der Eigenschaftendatei

Eine Neukompilierung ist nicht erforderlich, wenn Informationen aus der Eigenschaftendatei geändert werden: Wenn Informationen geändert werden

Syntax

- In einer Eigenschaftsdatei:
- Schlüssel = Wert
- #Kommentar

Bemerkungen

Ein Eigenschaftenobjekt ist eine Map, deren Schlüssel und Werte nach Konvention Strings sind. Obwohl die Methoden von Map für den Zugriff auf die Daten verwendet werden können, werden in der Regel stattdessen die typsicheren Methoden getProperty , setProperty und stringPropertyNames verwendet.

Eigenschaften werden häufig in Java-Eigenschaftendateien gespeichert, bei denen es sich um einfache Textdateien handelt. Ihr Format ist in der Properties.load-Methode ausführlich dokumentiert. In Summe:

- Jeder Schlüssel / Wert - Paar eine Textzeile mit Leerzeichen ist, gleich (`=`) oder Doppelpunkt (`:`) zwischen dem Schlüssel und dem Wert. Das Gleiche oder der Doppelpunkt kann vor und nach dem Whitespace beliebig sein, was ignoriert wird.
- Führende Leerzeichen werden immer ignoriert, abschließende Leerzeichen sind immer enthalten.
- Ein Backslash kann verwendet werden, um beliebige Zeichen zu umgehen (außer Kleinbuchstaben `u`).
- Ein umgekehrter Schrägstrich am Ende der Zeile zeigt an, dass die nächste Zeile eine Fortsetzung der aktuellen Zeile ist. Wie bei allen Zeilen werden jedoch führende Leerzeichen in der Fortsetzungszeile ignoriert.
- Genau wie im Java-Quellcode steht `\u` gefolgt von vier hexadezimalen Ziffern, für ein UTF-

16-Zeichen.

Die meisten Frameworks, einschließlich der Java SE-eigenen Einrichtungen wie java.util.ResourceBundle, laden Eigenschaftsdateien als InputStreams. Beim Laden einer Eigenschaftsdatei aus einem InputStream darf diese Datei nur ISO 8859-1-Zeichen enthalten (d. H. Zeichen im Bereich 0–255). Alle anderen Zeichen müssen als `\u` escape dargestellt werden. Sie können jedoch eine Textdatei in beliebiger Kodierung schreiben und das native2ascii- Tool (das mit jedem JDK mitgeliefert wird) verwenden, um dieses Escape für Sie auszuführen .

Wenn Sie eine Eigenschaftendatei mit Ihrem eigenen Code laden, kann diese in beliebiger Kodierung vorliegen, sofern Sie einen Reader (z. B. einen InputStreamReader) erstellen, der auf dem entsprechenden Zeichensatz basiert. Sie können die Datei dann mit load (Reader) anstelle der alten load (InputStream) -Methode laden.

Sie können Eigenschaften auch in einer einfachen XML-Datei speichern, sodass die Datei selbst die Kodierung definieren kann. Eine solche Datei kann mit der loadFromXML- Methode geladen werden. Die DTD, die die Struktur solcher XML-Dateien beschreibt, befindet sich unter http://java.sun.com/dtd/properties.dtd .

Examples

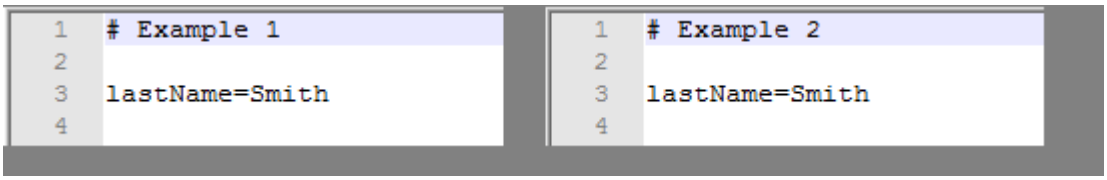
Eigenschaften laden

So laden Sie eine mit Ihrer Anwendung gebündelte Eigenschaftendatei:

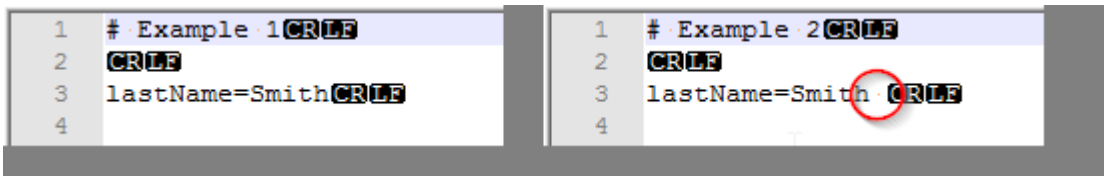
```
public class Defaults {

    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);
        }
    }

}
```

Eigenschaftsvorbehalt: Nachlaufende Leerzeichen

Sehen Sie sich diese beiden Property-Dateien an, die scheinbar völlig identisch sind:



außer sie sind wirklich nicht identisch:



(Screenshots stammen aus Notepad ++)

Da nachfolgenden Leerzeichen, um den Wert des erhaltenen `lastName` wäre `"Smith"` im ersten Fall und `"Smith "` im zweiten Fall.

Sehr selten erwarten Benutzer dies und können nur spekulieren, warum dies das Standardverhalten der `Properties` Klasse ist. Es ist jedoch einfach, eine erweiterte Version von `Properties` zu erstellen, die dieses Problem behebt. Die folgende Klasse, **TrimmedProperties** , macht genau das. Es ist ein Drop-In-Ersatz für die Standardeigenschaftenklasse.

```
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;
    }
}
```

Eigenschaften als XML speichern

Eigenschaften in einer XML-Datei speichern

Die Art und Weise, in der Sie Eigenschaftsdateien als XML-Dateien speichern, ist der Art und Weise, wie Sie sie als `.properties` Dateien speichern würden, sehr ähnlich. Anstelle von `store()` Sie `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");
}
```

Wenn Sie die Datei öffnen, sieht es so aus.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
3  <properties>
4  <comment>testing properties with xml</comment>
5  <entry key="age">23</entry>
6  <entry key="color">green</entry>
7  <entry key="name">Steve</entry>
8  </properties>
9
```

Eigenschaften aus einer XML-Datei laden

`loadFromXML()` diese Datei als `properties` zu laden, müssen Sie `loadFromXML()` anstelle von `load()` `.propeties` , die Sie mit regulären `.propeties` Dateien verwenden würden.

```
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);
    }
}
```

Wenn Sie diesen Code ausführen, erhalten Sie Folgendes in der Konsole:

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

Eigenschaften Klasse online lesen: https://riptutorial.com/de/java/topic/576/eigenschaften-klasse

Kapitel 39: Einstellungen

Examples

Ereignis-Listener hinzufügen

Es gibt zwei Arten von Ereignissen, die von einem `Preferences` Objekt `NodeChangeEvent` : `PreferenceChangeEvent` und `NodeChangeEvent` .

PreferenceChangeEvent

Ein `PreferenceChangeEvent` wird bei jeder Änderung eines Schlüssel-Wert-Paars des Knotens von einem `Properties` Objekt ausgegeben. `PreferenceChangeEvent` s kann mit einem `PreferenceChangeListener` abgehört werden:

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();
    }
});
```

Dieser Listener hört nicht auf geänderte Schlüsselwertpaare untergeordneter Knoten.

NodeChangeEvent

Dieses Ereignis wird ausgelöst, wenn ein untergeordneter Knoten eines `Properties` hinzugefügt oder entfernt wird.

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

Unterknoten der Voreinstellungen abrufen

`Preferences` repräsentieren immer einen bestimmten Knoten in einem gesamten `Preferences` , wie folgt:

```
/userRoot
├── com
│   └── mycompany
│       └── myapp
│           ├── darkApplicationMode=true
│           ├── showExitConfirmation=false
│           └── windowMaximized=true
└── org
    └── myorganization
        └── anotherapp
            ├── defaultFont=Helvetica
            ├── defaultSavePath=/home/matt/Documents
            └── exporting
                ├── defaultFormat=pdf
                └── openInBrowserAfterExport=false
```

So wählen Sie den Knoten /com/mycompany/myapp :

1. Konvention basierend auf dem Paket einer Klasse:

   ```
   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. Durch relativen Pfad:

   ```
   Preferences myApp = Preferences.userRoot().node("com/mycompany/myapp");
   ```

   Wenn Sie einen relativen Pfad verwenden (ein Pfad, der nicht mit einem / beginnt), wird der Pfad relativ zum übergeordneten Knoten aufgelöst, auf dem er aufgelöst wird. Das folgende Beispiel gibt beispielsweise den Knoten des Pfads /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. Über den absoluten Pfad:

   ```
   Preferences myApp = Preferences.userRoot().node("/com/mycompany/myapp");
   ```

   Die Verwendung eines absoluten Pfads auf dem Wurzelknoten unterscheidet sich nicht von der Verwendung eines relativen Pfads. Der Unterschied besteht darin, dass der Pfad relativ zu

dem Wurzelknoten aufgelöst wird, wenn er auf einem Unterknoten aufgerufen wird.

```
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
```

Koordinieren Sie den Zugriff auf Einstellungen auf mehrere Anwendungsinstanzen

Alle Instanzen von Preferences sind immer Thread-sicher in den Threads einer einzelnen Java Virtual Machine (JVM). Da Preferences mehreren JVMs gemeinsam genutzt werden können, gibt es spezielle Methoden, um Änderungen zwischen virtuellen Maschinen zu synchronisieren.

Wenn Sie eine Anwendung haben, die nur in einer **Instanz ausgeführt werden soll** , ist **keine externe Synchronisierung** erforderlich.

Wenn Sie eine Anwendung, die auf einem einzigen System in **mehreren Instanzen** ausgeführt wird und daher Preferences zugreifen muss zwischen den JVMs auf dem System koordiniert werden, dann ist die **sync() Verfahren** nach Preferences können Knoten verwendet werden , um Änderungen an den , um sicherzustellen , Preferences Knoten sind sichtbar für andere JVMs im System:

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

Einstellungen exportieren

Preferences können in ein XML-Dokument exportiert werden, das diesen Knoten darstellt. Die resultierende XML-Struktur kann erneut importiert werden. Das resultierende XML – Dokument wird sich erinnern , ob es von den Benutzer oder System exportiert wurde Preferences .

So exportieren Sie einen einzelnen Knoten, jedoch **nicht seine untergeordneten Knoten** :

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) {}
    }
}
```

So exportieren Sie einen einzelnen Knoten **mit seinen untergeordneten Knoten** :

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) {}
    }
}
```

Präferenzen importieren

Preferences können aus einem XML-Dokument importiert werden. Importieren soll in Verbindung mit der Exportfunktionalität von Preferences , da die richtigen XML-Dokumente erstellt werden.

Die XML - Dokumente werden sich erinnern , ob sie von den Benutzer oder System exportiert wurden Preferences . Daher können sie erneut in ihre jeweiligen Preferences importiert werden, ohne dass Sie herausfinden müssen, woher sie stammen. Die statische Funktion wird automatisch feststellen, ob das XML - Dokument aus den Benutzer oder System exportiert wurde Preferences und wird sie automatisch importieren in den Baum , den sie aus exportiert wurden.

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) {}
    }
}
```

Ereignis-Listener entfernen

Ereignis-Listener können wieder von jedem Properties werden, aber die Instanz des Listeners muss dafür in der Nähe bleiben.

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);
```

Gleiches gilt für NodeChangeListener .

Präferenzwerte abrufen

Ein Wert eines Preferences Knotens kann vom Typ String , boolean , byte[] , double , float , int oder long . Alle Beschwörungen müssen einen Standardwert zur Verfügung stellen, falls der angegebene Wert nicht in dem ist Preferences Knoten.

```
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);
```

Voreinstellungen festlegen

Um einen Wert im Knoten " Preferences zu speichern, wird eine der putXXX() Methoden verwendet. Ein Wert eines Preferences Knotens kann vom Typ String , boolean , byte[] , double , float , int oder 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);
```

Präferenzen verwenden

Preferences können verwendet werden, um Benutzereinstellungen zu speichern, die die persönlichen Anwendungseinstellungen eines Benutzers widerspiegeln, z. B. die Schriftart des Editors, ob die Anwendung lieber im Vollbildmodus gestartet werden soll, ob sie das Kontrollkästchen "Nicht mehr anzeigen" und andere Dinge aktiviert haben so wie das.

```
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);
        }
    }
}
```

Einstellungen online lesen: https://riptutorial.com/de/java/topic/582/einstellungen

Kapitel 40: Enum Map

Einführung

Die Java EnumMap-Klasse ist die spezialisierte Map-Implementierung für Enumerationsschlüssel. Es erbt die Klassen Enum und AbstractMap.

Die Parameter für die Klasse java.util.EnumMap.

K: Dies ist der Typ der von dieser Karte verwalteten Schlüssel. V: Dies ist der Typ der zugeordneten Werte.

Examples

Enum Map Book Beispiel

```
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);

        }
}
}
```

Enum Map online lesen: https://riptutorial.com/de/java/topic/10158/enum-map

Kapitel 41: EnumSet-Klasse

Einführung

Die Java EnumSet-Klasse ist die spezialisierte Set-Implementierung zur Verwendung mit Aufzählungstypen. Sie erbt die AbstractSet-Klasse und implementiert die Set-Schnittstelle.

Examples

Aufzählungsbeispiel

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

EnumSet-Klasse online lesen: https://riptutorial.com/de/java/topic/10159/enumset-klasse

Kapitel 42: Erbe

Einführung

Vererbung ist ein grundlegendes objektorientiertes Feature, bei dem eine Klasse die Eigenschaften einer anderen Klasse mit dem Schlüsselwort extends erfasst und extends . Interfaces und die Schlüsselwörter implements finden Sie unter Interfaces .

Syntax

- Klasse ClassB erweitert ClassA {...}
- Klasse ClassB implementiert InterfaceA {...}
- interface InterfaceB erweitert InterfaceA {...}
- class ClassB erweitert ClassA implementiert InterfaceC, InterfaceD {...}
- abstrakte Klasse AbstractClassB erweitert ClassA {...}
- abstrakte Klasse AbstractClassB erweitert AbstractClassA {...}
- abstrakte Klasse AbstractClassB erweitert ClassA implementiert InterfaceC, InterfaceD {...}

Bemerkungen

Vererbung wird häufig mit Generika kombiniert, sodass die Basisklasse einen oder mehrere Typparameter hat. Siehe Generische Klasse erstellen .

Examples

Abstrakte Klassen

Eine abstrakte Klasse ist eine Klasse, die mit dem abstract Schlüsselwort markiert ist. Im Gegensatz zu nicht abstrakten Klassen kann es abstrakte - implementierungslose - Methoden enthalten. Es ist jedoch zulässig, eine abstrakte Klasse ohne abstrakte Methoden zu erstellen.

Eine abstrakte Klasse kann nicht instanziiert werden. Sie kann unterklassiert (erweitert) werden, solange die Unterklasse entweder abstrakt ist oder alle von Superklassen als abstrakt markierten Methoden implementiert.

Ein Beispiel für eine abstrakte Klasse:

```
public abstract class Component {
    private int x, y;

    public setPosition(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public abstract void render();
}
```

Die Klasse muss als abstrakt markiert sein, wenn sie mindestens eine abstrakte Methode hat. Eine abstrakte Methode ist eine Methode, die keine Implementierung hat. Andere Methoden können innerhalb einer abstrakten Klasse deklariert werden, die implementiert ist, um allgemeinen Code für alle Unterklassen bereitzustellen.

Beim Versuch, diese Klasse zu instanziieren, wird ein Kompilierungsfehler ausgegeben:

```
//error: Component is abstract; cannot be instantiated
Component myComponent = new Component();
```

Eine Klasse, die Component und eine Implementierung für alle ihre abstrakten Methoden bereitstellt und instanziiert werden kann.

```
public class Button extends Component {

    @Override
    public void render() {
        //render a button
    }
}

public class TextBox extends Component {

    @Override
    public void render() {
        //render a textbox
    }
}
```

Instanzen, die Klassen erben, können auch als übergeordnete Klasse umgewandelt werden (normale Vererbung) und bieten einen polymorphen Effekt, wenn die abstrakte Methode aufgerufen wird.

```
Component myButton = new Button();
Component myTextBox = new TextBox();

myButton.render(); //renders a button
myTextBox.render(); //renders a text box
```

Abstrakte Klassen gegen Schnittstellen

Abstrakte Klassen und Schnittstellen bieten eine Möglichkeit, Methodensignaturen zu definieren, während die Erweiterungs- / Implementierungsklasse für die Implementierung erforderlich ist.

Es gibt zwei Hauptunterschiede zwischen abstrakten Klassen und Schnittstellen:

- Eine Klasse kann nur eine einzelne Klasse erweitern, aber viele Schnittstellen implementieren.
- Eine abstrakte Klasse kann Instanzfelder (nicht static) enthalten, Schnittstellen jedoch nur static Felder.

Java SE 8

In Interfaces deklarierte Methoden konnten keine Implementierungen enthalten. Daher wurden abstrakte Klassen verwendet, wenn es sinnvoll war, zusätzliche Methoden bereitzustellen, die als abstrakte Methoden bezeichnet wurden.

Java SE 8

In Java 8 können Schnittstellen Standardmethoden enthalten, die normalerweise mit den anderen Methoden der Schnittstelle implementiert werden , wodurch Schnittstellen und abstrakte Klassen in dieser Hinsicht gleichermaßen mächtig werden.

Anonyme Unterklassen von abstrakten Klassen

Aus praktischen Gründen ermöglicht Java die Instantiierung anonymer Instanzen von Unterklassen abstrakter Klassen, die Implementierungen für die abstrakten Methoden beim Erstellen des neuen Objekts bereitstellen. Das obige Beispiel könnte so aussehen:

```
Component myAnonymousComponent = new Component() {
    @Override
    public void render() {
        // render a quick 1-time use component
    }
```

```
}
```

Statische Vererbung

Statische Methoden können ähnlich wie normale Methoden vererbt werden. Im Gegensatz zu normalen Methoden ist es jedoch nicht möglich, " abstrakte " Methoden zu erstellen, um das Überschreiben statischer Methoden zu erzwingen. Das Schreiben einer Methode mit derselben Signatur wie eine statische Methode in eine Superklasse scheint eine Form des Überschreibens zu sein, aber in Wirklichkeit wird dadurch einfach eine neue Funktion erstellt, die die andere versteckt.

```
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");
    }

}
```

Das Ausführen einer dieser Klassen erzeugt die Ausgabe:

```
Hello
BaseClass's num: 5
Hello
Hey
```

```
Static says Hi
StaticOverride's num: test
```

Beachten Sie, dass im Gegensatz zur normalen Vererbung die Methoden der statischen Vererbung nicht ausgeblendet werden. Sie können die sayHello BaseClass.sayHello() immer mit BaseClass.sayHello() . Klassen erben jedoch statische Methoden, *wenn* in der Unterklasse keine Methoden mit derselben Signatur gefunden werden. Wenn zwei Signaturen der Methode variieren, können beide Methoden von der Unterklasse ausgeführt werden, auch wenn der Name gleich ist.

Statische Felder verbergen sich auf ähnliche Weise.

Verwenden Sie 'final', um die Vererbung und das Überschreiben zu beschränken

Abschlussunterricht

Bei Verwendung in einer class verhindert der final Modifizierer, dass andere Klassen deklariert werden, extend die Klasse erweitern. Eine final Klasse ist eine "Blatt" -Klasse in der Vererbungsklassenhierarchie.

```
// This declares a final class
final class MyFinalClass {
    /* some code */
}

// Compilation error: cannot inherit from final MyFinalClass
class MySubClass extends MyFinalClass {
    /* more code */
}
```

Anwendungsfälle für Abschlussklassen

Abschließende Klassen können mit einem private Konstruktor kombiniert werden, um die Instantiierung einer Klasse zu steuern oder zu verhindern. Dies kann verwendet werden, um eine sogenannte "Utility-Klasse" zu erstellen, die nur statische Member definiert. dh Konstanten und statische Methoden.

```
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;
    }

}
```

Unveränderliche Klassen sollten auch als final deklariert werden. (Eine unveränderliche Klasse ist eine, deren Instanzen nicht mehr geändert werden können, nachdem sie erstellt wurden; siehe Thema Informierbare Objekte .) Dadurch wird das Erstellen einer veränderlichen Unterklasse einer unveränderlichen Klasse unmöglich. Dies würde gegen das Liskov-Substitutionsprinzip verstoßen, das verlangt, dass ein Subtyp dem "Verhaltensvertrag" seiner Supertypen gehorcht.

Aus praktischer Sicht ist es einfacher, eine unveränderliche Klasse als final deklarieren, um das Programmverhalten zu begründen. Außerdem werden Sicherheitsbedenken in dem Szenario angesprochen, in dem nicht vertrauenswürdiger Code in einer Sicherheits-Sandbox ausgeführt wird. (Da String zum Beispiel als final deklariert ist, muss eine vertrauenswürdige Klasse nicht befürchten, dass sie dazu verleitet wird, veränderliche Unterklassen zu akzeptieren, die der nicht vertrauenswürdige Aufrufer dann unbemerkt ändern könnte.)

Ein Nachteil der final ist, dass sie nicht mit einigen spöttischen Frameworks wie Mockito funktionieren. Update: Mockito Version 2 unterstützt jetzt das Verspotten der Abschlussklassen.

Endgültige Methoden

Der final Modifikator kann auch auf Methoden angewendet werden, um zu verhindern, dass sie in Unterklassen überschrieben werden:

```
public class MyClassWithFinalMethod {

    public final void someMethod() {
    }
}

public class MySubClass extends MyClassWithFinalMethod {

    @Override
    public void someMethod() { // Compiler error (overridden method is final)
    }
}
```

Endgültige Methoden werden normalerweise verwendet, wenn Sie einschränken möchten, was eine Unterklasse in einer Klasse ändern kann, ohne Unterklassen vollständig zu verbieten.

Der final Modifikator kann auch auf Variablen angewendet werden, die Bedeutung von final für Variablen steht jedoch nicht in Zusammenhang mit der Vererbung.

Das Liskov-Substitutionsprinzip

Ersetzbarkeit ist ein Prinzip in der objektorientierten Programmierung von Barbara Liskov in einem 1987 Konferenz Keynote eingeführt besagt , dass, wenn die Klasse B eine Unterklasse der Klasse ist A , dann überall dort , wo A erwartet wird, B kann stattdessen verwendet werden:

```
class A {...}
class B extends A {...}

public void method(A obj) {...}

A a = new B(); // Assignment OK
method(new B()); // Passing as parameter OK
```

Dies gilt auch, wenn es sich bei dem Typ um eine Schnittstelle handelt, bei der keine hierarchische Beziehung zwischen den Objekten erforderlich ist:

```
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
```

Die Liste enthält jetzt Objekte, die nicht zu derselben Klassenhierarchie gehören.

Erbe

Mit der Verwendung der extends Schlüsselwort unter Klassen, alle Eigenschaften der Oberklasse sind in der Unterklasse vorhanden ist (auch als *übergeordnete Klasse* oder *Basisklasse* genannt) (auch als *Child - Klasse* oder *Abgeleitete Klasse* bekannt)

```
public class BaseClass {

    public void baseMethod(){
        System.out.println("Doing base class stuff");
    }
}

public class SubClass extends BaseClass {

}
```

Instanzen von SubClass haben die Methode baseMethod() *geerbt* :

```
SubClass s = new SubClass();
s.baseMethod();  //Valid, prints "Doing base class stuff"
```

Zusätzlicher Inhalt kann zu einer Unterklasse hinzugefügt werden. Dies ermöglicht zusätzliche Funktionen in der Unterklasse, ohne die Basisklasse oder andere Unterklassen derselben Basisklasse zu ändern:

```
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"
```

Felder werden auch vererbt:

```
public class BaseClassWithField {

    public int x;

}

public class SubClassWithField extends BaseClassWithField {

    public SubClassWithField(int x) {
        this.x = x; //Can access fields
    }
}
```

private Felder und Methoden sind in der Unterklasse noch vorhanden, aber nicht zugänglich:

```
public class BaseClassWithPrivateField {

    private int x = 5;
```

```
    public int getX() {
        return x;
    }
}

public class SubClassInheritsPrivateField extends BaseClassWithPrivateField {

    public void printX() {
        System.out.println(x); //Illegal, can't access private field x
        System.out.println(getX()); //Legal, prints 5
    }
}

SubClassInheritsPrivateField s = new SubClassInheritsPrivateField();
int x = s.getX(); //x will have a value of 5.
```

In Java kann jede Klasse höchstens eine andere Klasse erweitern.

```
public class A{}
public class B{}
public class ExtendsTwoClasses extends A, B {} //Illegal
```

Dies wird als Mehrfachvererbung bezeichnet. In einigen Sprachen ist dies zwar zulässig, in Java ist dies jedoch nicht für Klassen zulässig.

Infolgedessen verfügt jede Klasse über eine unverzweigte vorgelagerte Klassenkette, die zu `Object` , von der alle Klassen abstammen.

Vererbung und statische Methoden

In Java können übergeordnete und untergeordnete Klassen statische Methoden mit demselben Namen verwenden. Aber in solchen Fällen der Umsetzung von statischer Methode in Kind versteckt Elternklasse Implementierung, es ist kein Verfahren überwiegende. Zum Beispiel:

```
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");
    }
  }
}
```

Statische Methoden sind an eine Klasse gebunden, nicht an eine Instanz, und diese Methodenbindung erfolgt zur Kompilierzeit. Da in dem ersten Aufruf von `staticMethod()` , Elternklasse Referenz `p` wurde verwendet, Parent ,s Version von `staticMethod()` aufgerufen wird.

Im zweiten Fall haben wir p in die Child Klasse umgewandelt, die staticMethod() Child ausgeführt hat.

Variable Abschattung

Variablen sind SHADOWED und Methoden sind OVERRIDDEN. Welche Variable verwendet wird, hängt von der Klasse ab, für die die Variable deklariert ist. Welche Methode verwendet wird, hängt von der tatsächlichen Klasse des Objekts ab, auf die die Variable verweist.

```
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
    }
}
```

Verengen und Erweitern von Objektreferenzen

Umwandlung einer Instanz einer Basisklasse in eine Unterklasse wie in: b = (B) a; wird als *Verengung bezeichnet* (wenn Sie versuchen, das Basisklassenobjekt auf ein spezifischeres Klassenobjekt zu beschränken) und erfordert eine explizite Typumwandlung.

Umwandlung einer Instanz einer Unterklasse in eine Basisklasse wie in: A a = b; wird als *Verbreiterung bezeichnet* und benötigt keinen Typcast.

Zur Veranschaulichung betrachten Sie die folgenden Klassendeklarationen und den Testcode:

```
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
    }
}
```

Die Aussage Vehicle vehicle = new Car(); ist eine gültige Java-Anweisung. Jede Instanz Car ist auch ein Vehicle . Daher ist die Zuweisung legal, ohne dass eine explizite Typisierung erforderlich ist.

Andererseits ist Car c = vehicle; ist ungültig. Der statische Typ der vehicle ist Vehicle Dies bedeutet, dass sie sich auf eine Instanz von Car , Truck , MotorCycle , or any other current or future subclass of Vehicle beziehen kann . (Or indeed, an instance of Vehicle itself, since we did not declare it as an abstrakte class.) The assignment cannot be allowed, since that might lead to itself, since we did not declare it as an class.) The assignment cannot be allowed, since that might lead to car referring to a Truck-Instanz verweist.

Um dies zu vermeiden, müssen wir eine explizite Typumwandlung hinzufügen:

```
Car c = (Car) vehicle;
```

Der Typ-Cast teilt dem Compiler mit, dass wir davon ausgehen, *dass* der Wert eines vehicle ein Car oder eine Unterklasse des Car . Bei Bedarf fügt der Compiler Code ein, um eine Laufzeitprüfung durchzuführen. Wenn die Prüfung fehlschlägt, wird eine ClassCastException ausgelöst, wenn der Code ausgeführt wird.

Beachten Sie, dass nicht alle Typumwandlungen gültig sind. Zum Beispiel:

```
String s = (String) vehicle;  // not valid
```

Der Java-Compiler weiß, dass eine Instanz, die mit Vehicle kompatibel *ist, nicht immer* mit String kompatibel sein kann. Die Typumwandlung konnte niemals erfolgreich sein, und die JLS schreibt vor, dass dies zu einem Kompilierungsfehler führt.

Programmierung an einer Schnittstelle

Die Idee hinter der Programmierung einer Schnittstelle besteht darin, den Code hauptsächlich auf Schnittstellen aufzubauen und nur zum Zeitpunkt der Instantiierung konkrete Klassen zu verwenden. In diesem Zusammenhang sieht guter Code, der sich beispielsweise mit Java-Sammlungen befasst, ungefähr so aus (nicht, dass die Methode selbst von Nutzen ist, sondern nur zur Veranschaulichung):

```
public <T> Set<T> toSet(Collection<T> collection) {
  return Sets.newHashSet(collection);
}
```

während schlechter Code so aussehen könnte:

```
public <T> HashSet<T> toSet(ArrayList<T> collection) {
  return Sets.newHashSet(collection);
}
```

Nicht nur der erstere kann auf eine breitere Auswahl von Argumenten angewendet werden, sondern seine Ergebnisse werden mit dem von anderen Entwicklern bereitgestellten Code, der im Allgemeinen dem Konzept der Programmierung einer Schnittstelle entspricht, besser kompatibel. Die wichtigsten Gründe für die Verwendung des ersteren sind jedoch:

- Meistens benötigt und sollte der Kontext, in dem das Ergebnis verwendet wird, nicht so viele Details wie die konkrete Implementierung vorsieht.
- Das Befolgen einer Schnittstelle erzwingt einen saubereren Code, und weniger Hacks wie beispielsweise eine weitere öffentliche Methode werden einer Klasse hinzugefügt, die ein bestimmtes Szenario abdeckt.
- Der Code ist besser zu testen, da Schnittstellen leicht zu spotteln sind.
- Schließlich hilft das Konzept auch, wenn nur eine Implementierung erwartet wird (zumindest für die Testbarkeit).

Wie kann man also das Konzept der Programmierung auf eine Schnittstelle anwenden, wenn man neuen Code schreibt und dabei eine bestimmte Implementierung vor Augen hat? Eine Option, die wir häufig verwenden, ist eine Kombination der folgenden Muster:

- Programmierung an einer Schnittstelle
- Fabrik
- Baumeister

Das folgende, auf diesen Prinzipien basierende Beispiel ist eine vereinfachte und verkürzte Version einer RPC-Implementierung, die für verschiedene Protokolle geschrieben wurde:

```
public interface RemoteInvoker {
  <RQ, RS> CompletableFuture<RS> invoke(RQ request, Class<RS> responseClass);
}
```

Die obige Schnittstelle soll nicht direkt über eine Factory instanziiert werden. Stattdessen leiten wir weitere konkretere Schnittstellen ab, eine für den HTTP-Aufruf und eine für AMQP. Jede dieser Instanzen verfügt über eine Factory und einen Builder, um Instanzen zu erstellen, die wiederum Instanzen sind die obige Schnittstelle:

```
public interface AmqpInvoker extends RemoteInvoker {
  static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
    return new AmqpInvokerBuilder(instanceId, factory);
  }
}
```

Instanzen von RemoteInvoker für die Verwendung mit AMQP können jetzt so einfach erstellt werden (oder je nach Builder mehr involviert sein):

```
RemoteInvoker invoker = AmqpInvoker.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

Und ein Aufruf einer Anfrage ist so einfach wie:

```
Response res = invoker.invoke(new Request(data), Response.class).get();
```

Da Java 8 die AmqpInvoker.with() bietet, statische Methoden direkt in Schnittstellen AmqpInvoker.with() , wurde die Zwischenfactory im obigen Code implizit durch AmqpInvoker.with() . In Java vor Version 8 kann derselbe Effekt mit einer inneren Factory Klasse erzielt werden:

```
public interface AmqpInvoker extends RemoteInvoker {
  class Factory {
    public static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {
      return new AmqpInvokerBuilder(instanceId, factory);
    }
  }
```

```
}
```

Die entsprechende Instanziierung würde sich dann in

```
RemoteInvoker invoker = AmqpInvoker.Factory.with(instanceId, factory)
  .requestRouter(router)
  .build();
```

Der oben verwendete Builder könnte so aussehen (obwohl dies eine Vereinfachung darstellt, da die tatsächliche Definition von bis zu 15 Parametern erlaubt, die von den Standardwerten abweichen). Beachten Sie, dass das Konstrukt nicht öffentlich ist und daher nur von der obigen AmqpInvoker Schnittstelle aus verwendet werden kann:

```
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);
  }
}
```

Im Allgemeinen kann ein Builder auch mit einem Tool wie FreeBuilder erstellt werden.

Die standardmäßige (und die einzig erwartete) Implementierung dieser Schnittstelle wird schließlich als paketlokale Klasse definiert, um die Verwendung der Schnittstelle, der Factory und des Builders zu erzwingen:

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

In der Zwischenzeit erwies sich dieses Muster als sehr effizient bei der Entwicklung unseres gesamten neuen Codes, unabhängig davon, wie einfach oder komplex die Funktionalität ist.

Abstrakte Klassen- und Interface-Nutzung: "Is-a" Relation vs. "Has-a" -Funktion

Wann sollten abstrakte Klassen verwendet werden: Um dasselbe oder unterschiedliches Verhalten bei mehreren verwandten Objekten zu implementieren

Wann sind Schnittstellen zu verwenden: um einen Vertrag durch mehrere nicht zusammenhängende Objekte zu implementieren

Abstrakte Klassen erstellen "ist eine" Beziehung, während Schnittstellen "eine Funktion" haben.

Dies ist im folgenden Code zu sehen:

```
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;
    }
}
```

Ausgabe:

```
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
```

Wichtige Hinweise:

1. Animal ist eine abstrakte Klasse mit gemeinsamen Attributen: name und lifeExpectancy und abstrakte Methoden: remember() und protectOwner() . Dog und Cat sind Animals , die die Methoden remember() und protectOwner() implementiert haben.

2. Cat kann climb() , Dog jedoch nicht. Dog kann think() , Cat aber nicht. Diese spezifischen Funktionen werden Cat und Dog bei der Implementierung hinzugefügt.

3. Man ist kein Animal aber er kann Think , Learn , sich Apply und Climb .

4. Cat ist kein Man aber sie kann Climb .

5. Dog ist kein Man aber er kann Learn

6. Man ist weder eine Cat noch ein Dog , kann jedoch einige der Fähigkeiten der beiden letztgenannten besitzen, ohne Animal , Cat oder Dog . Dies geschieht mit Interfaces.

7. Obwohl Animal eine abstrakte Klasse ist, hat sie im Gegensatz zu einer Schnittstelle einen Konstruktor.

TL; DR:

Nicht verbundene Klassen können über Schnittstellen über Funktionen verfügen, verwandte Klassen ändern jedoch das Verhalten durch Erweiterung der Basisklassen.

Auf der Java-Dokumentation finden Sie Informationen darüber , welche in einem bestimmten Anwendungsfall verwendet werden soll.

Erwägen Sie die Verwendung abstrakter Klassen, wenn ...

1. Sie möchten Code unter mehreren eng miteinander verwandten Klassen teilen.
2. Sie erwarten, dass Klassen, die Ihre abstrakte Klasse erweitern, viele allgemeine Methoden oder Felder aufweisen oder andere Zugriffsmodifizierer als public (wie protected und private) erfordern.
3. Sie möchten nicht statische oder nicht abschließende Felder deklarieren.

Erwägen Sie die Verwendung von Schnittstellen, wenn ...

1. Sie erwarten, dass nicht verknüpfte Klassen Ihre Schnittstelle implementieren. Beispielsweise können viele nicht zusammenhängende Objekte die Serializable Schnittstelle implementieren.
2. Sie möchten das Verhalten eines bestimmten Datentyps angeben, machen sich jedoch keine Gedanken darüber, wer sein Verhalten implementiert.
3. Sie möchten die Mehrfachvererbung des Typs nutzen.

Überschreibung bei Vererbung

Das Überschreiben in Vererbung wird verwendet, wenn Sie eine bereits definierte Methode aus einer Superklasse in einer Unterklasse verwenden, jedoch auf eine andere Weise als bei der ursprünglichen Erstellung der Methode in der Superklasse. Durch das Überschreiben kann der Benutzer den Code wiederverwenden, indem er vorhandenes Material verwendet und es an die Bedürfnisse des Benutzers anpasst.

Das folgende Beispiel ClassA , wie ClassB die Funktionalität von ClassA indem ClassA wird, was durch die Druckmethode gesendet wird:

Beispiel:

```
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");
    }
}
```

Ausgabe:

```
EIN

B
```

Erbe online lesen: https://riptutorial.com/de/java/topic/87/erbe

Kapitel 43: Executor-, ExecutorService- und Thread-Pools

Einführung

Die Executor- Schnittstelle in Java bietet die Möglichkeit, die Aufgabe von Aufgaben von den Mechanismen zu koppeln, wie die einzelnen Aufgaben ausgeführt werden, einschließlich Details zur Verwendung des Threads, zur Planung usw. Ein Executor wird normalerweise verwendet, anstatt Threads explizit zu erstellen. Mit Executors müssen Entwickler ihren Code nicht wesentlich umschreiben, um die Task-Ausführungsrichtlinie ihres Programms einfach anpassen zu können.

Bemerkungen

Fallstricke

- Wenn Sie eine Task für die wiederholte Ausführung planen, wird Ihre Task je nach verwendetem ScheduledExecutorService möglicherweise vor jeder weiteren Ausführung angehalten, wenn die Ausführung Ihrer Task eine Ausnahme verursacht, die nicht behandelt wird. Siehe Mutter F ** k beim ScheduledExecutorService!

Examples

Feuer und Vergessen - ausführbare Aufgaben

Executoren akzeptieren eine java.lang.Runnable die (möglicherweise rechnerisch oder anderweitig andauernd oder stark) Code enthält, der in einem anderen Thread ausgeführt werden soll.

Verwendung wäre:

```
Executor exec = anExecutor;
exec.execute(new Runnable() {
    @Override public void run() {
        //offloaded work, no need to get result back
    }
});
```

Beachten Sie, dass Sie mit diesem Executor keine Möglichkeit haben, einen berechneten Wert zurückzubekommen.
Mit Java 8 kann man Lambdas verwenden, um das Codebeispiel zu verkürzen.

Java SE 8

```
Executor exec = anExecutor;
exec.execute(() -> {
    //offloaded work, no need to get result back
});
```

ThreadPoolExecutor

Ein häufig verwendeter Executor ist der ThreadPoolExecutor , der sich um die Thread-Behandlung kümmert. Sie können die minimale Anzahl an Threads konfigurieren, die der Executor immer beibehalten muss, wenn nicht viel zu tun ist (Core-Größe genannt) und eine maximale Thread-Größe, auf die der Pool wachsen kann, wenn mehr Arbeit zu erledigen ist. Wenn die Arbeitslast nachlässt, verringert der Pool die Thread-Anzahl langsam wieder, bis die Mindestgröße erreicht ist.

```
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
    }
});
```

Hinweis Wenn Sie den ThreadPoolExecutor mit einer *unbegrenzten* Warteschlange konfigurieren, überschreitet die Thread-Anzahl nicht corePoolSize da neue Threads nur erstellt werden, wenn die Warteschlange voll ist:

ThreadPoolExecutor mit allen Parametern:

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

von JavaDoc

> Wenn mehr als corePoolSize, aber weniger als maximumPoolSize-Threads ausgeführt werden, wird ein neuer Thread nur erstellt, wenn die Warteschlange voll ist.

Vorteile:

1. Die Größe der BlockingQueue kann gesteuert werden, und Szenarien mit zu wenig Arbeitsspeicher können vermieden werden. Die Anwendungsleistung wird durch die begrenzte beschränkte Warteschlangengröße nicht beeinträchtigt.
2. Sie können vorhandene Richtlinien verwenden oder neue Richtlinien für Ablehnungshandler erstellen.
    1. In der Standardeinstellung ThreadPoolExecutor.AbortPolicy löst der Handler bei Ablehnung eine Laufzeit-RejectedExecutionException aus.
    2. In ThreadPoolExecutor.CallerRunsPolicy führt der Thread, der die Ausführung ausführt, die Aufgabe aus. Dies stellt einen einfachen Mechanismus zur Regelung der Rückkopplung bereit, der die Geschwindigkeit der Übergabe neuer Aufgaben verlangsamt.
    3. In ThreadPoolExecutor.DiscardPolicy wird eine Aufgabe, die nicht ausgeführt werden kann, einfach gelöscht.
    4. Wenn der Executor in ThreadPoolExecutor.DiscardOldestPolicy nicht heruntergefahren ist, wird die Task am Anfang der Arbeitswarteschlange gelöscht und die Ausführung wird erneut versucht (dies kann erneut fehlschlagen und führt dazu, dass dies wiederholt wird.)
3. Benutzerdefinierte ThreadFactory kann konfiguriert werden, was nützlich ist:
    1. So legen Sie einen aussagekräftigeren Threadnamen fest
    2. So legen Sie den Thread-Daemon-Status fest
    3. Thread-Priorität einstellen

Hier ist ein Beispiel für die Verwendung von ThreadPoolExecutor

Wert aus der Berechnung abrufen – Aufrufbar

Wenn Ihre Berechnung einen Rückgabewert liefert, der später erforderlich ist, reicht eine einfache ausführbare Aufgabe nicht aus. In solchen Fällen können Sie ExecutorService.submit(Callable <T>) das nach Abschluss der Ausführung einen Wert zurückgibt.

Der Service gibt eine Future der Sie das Ergebnis der Taskausführung abrufen können.

```
// 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
```

Wenn Sie das Ergebnis der Zukunft benötigen, rufen Sie future.get()

- Warten Sie unbegrenzt auf die Zukunft, um mit einem Ergebnis fertig zu werden.

  ```
  try {
      // Blocks current thread until future is completed
      Integer result = future.get();
  catch (InterruptedException || ExecutionException e) {
      // handle appropriately
  }
  ```

- Warten Sie, bis die Zukunft abgeschlossen ist, jedoch nicht länger als die angegebene Zeit.

  ```
  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
  }
  ```

Wenn das Ergebnis einer geplanten oder laufenden Aufgabe nicht mehr erforderlich ist, können Sie Future.cancel(boolean) aufrufen, um den Future.cancel(boolean) abzubrechen.

- Durch Aufrufen von cancel(false) wird die Aufgabe nur aus der Warteschlange der auszuführenden Aufgaben entfernt.
- Wenn Sie cancel(true) aufrufen cancel(true) wird die Task *auch* unterbrochen, wenn sie gerade ausgeführt wird.

Planen von Aufgaben zur Ausführung zu einer festgelegten Zeit, nach einer Verzögerung oder wiederholt

Die ScheduledExecutorService Klasse stellt Methoden zum Planen einzelner oder wiederholter Aufgaben auf verschiedene Arten bereit. Im folgenden Codebeispiel wird davon ausgegangen, dass der pool wie folgt deklariert und initialisiert wurde:

```
ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);
```

Zusätzlich zu den normalen ExecutorService Methoden fügt die ScheduledExecutorService API vier Methoden hinzu, die Aufgaben planen und ScheduledFuture Objekte zurückgeben. Letzteres kann zum Abrufen von Ergebnissen (in einigen Fällen) und zum Abbrechen von Aufgaben verwendet werden.

Start einer Aufgabe nach einer festen Verzögerung

Im folgenden Beispiel wird der Start einer Aufgabe nach zehn Minuten geplant.

```
ScheduledFuture<Integer> future = pool.schedule(new Callable<>() {
        @Override public Integer call() {
            // do something
            return 42;
        }
    },
    10, TimeUnit.MINUTES);
```

Aufgaben zu einem festen Preis beginnen

Im folgenden Beispiel wird eine Task so geplant, dass sie nach zehn Minuten startet und dann alle 1 Minute wiederholt wird.

```
ScheduledFuture<?> future = pool.scheduleAtFixedRate(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

Die Task-Ausführung wird gemäß dem Zeitplan fortgesetzt, bis der pool heruntergefahren wird, die future abgebrochen wird oder eine der Aufgaben auf eine Ausnahme stößt.

Es ist garantiert, dass sich die von einem bestimmten scheduledAtFixedRate Aufruf scheduledAtFixedRate Tasks nicht zeitlich überlappen. Wenn eine Task länger als die vorgeschriebene Zeit dauert, können die nächste und die nachfolgenden Taskausführungen zu spät beginnen.

Aufgaben werden mit einer festen Verzögerung gestartet

Im folgenden Beispiel wird eine Task so geplant, dass sie nach zehn Minuten startet und dann wiederholt mit einer Verzögerung von einer Minute zwischen einer Taskende und der nächsten Task startet.

```
ScheduledFuture<?> future = pool.scheduleWithFixedDelay(new Runnable() {
        @Override public void run() {
            // do something
        }
    },
    10, 1, TimeUnit.MINUTES);
```

Die Task-Ausführung wird gemäß dem Zeitplan fortgesetzt, bis der pool heruntergefahren wird, die future abgebrochen wird oder eine der Aufgaben auf eine Ausnahme stößt.

Abgelehnte Ausführung von Handle

Ob

1. Sie versuchen, Aufgaben an einen heruntergefahrenen Executor zu übergeben oder
2. Die Warteschlange ist gesättigt (nur bei beschränkten möglich) und die maximale Anzahl Threads wurde erreicht.

RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor) wird aufgerufen.

Das Standardverhalten ist, dass beim Aufrufer eine RejectedExecutionException ausgelöst wird. Es sind jedoch weitere vordefinierte Verhaltensweisen verfügbar:

- **ThreadPoolExecutor.AbortPolicy** (standardmäßig REE werfen)
- **ThreadPoolExecutor.CallerRunsPolicy** (führt eine Task im Thread des Aufrufers aus - *blockiert ihn*)

- **ThreadPoolExecutor.DiscardPolicy** (Task im **Hintergrund** verwerfen)
- **ThreadPoolExecutor.DiscardOldestPolicy** (**älteste** Aufgabe in der Warteschlange **automatisch** verwerfen und Ausführung der neuen Aufgabe erneut **versuchen**)

Sie können sie mit einem der ThreadPool- Konstruktoren festlegen:

```
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) // <--
```

Sie können auch Ihr eigenes Verhalten implementieren, indem Sie die RejectedExecutionHandler-Schnittstelle erweitern:

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
```

Unterschiede in der Übergabe von Ausnahmen () im Vergleich zur Ausführung ()

Der Befehl execute () wird im Allgemeinen für Feuer- und Vergissaufrufe verwendet (ohne dass das Ergebnis analysiert werden muss), und der Befehl submit () wird zum Analysieren des Ergebnisses des Future-Objekts verwendet.

Wir sollten uns der Hauptunterschiede der Ausnahmebehandlungsmechanismen zwischen diesen beiden Befehlen bewusst sein.

Ausnahmen von submit () werden vom Framework verschluckt, wenn Sie sie nicht abgefangen haben.

Codebeispiel, um den Unterschied zu verstehen:

Fall 1: Übergeben Sie den Befehl Runnable with execute (), der die Ausnahme meldet.

```
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);
   }
 }
```

Ausgabe:

```
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)
```

Fall 2: Ersetzen von service.submit(new Runnable(){ **():** service.submit(new Runnable(){ In diesem Fall werden Ausnahmen vom Framework verschluckt, da die run () - Methode sie nicht explizit service.submit(new Runnable(){ .

Ausgabe:

```
creating service
a and b=4:0
a and b=4:0
```

Fall 3: Ändern Sie den neuenFixedThreadPool in ExtendedExecutor

```
//ExecutorService service = Executors.newFixedThreadPool(2);
 ExtendedExecutor service = new ExtendedExecutor();
```

Ausgabe:

```
creating service
a and b=4:0
java.lang.ArithmeticException: / by zero
a and b=4:0
java.lang.ArithmeticException: / by zero
```

Ich habe dieses Beispiel für zwei Themen gezeigt: Verwenden Sie Ihren benutzerdefinierten ThreadPoolExecutor und behandeln Sie Exectpion mit einem benutzerdefinierten ThreadPoolExecutor.

Eine andere einfache Lösung für das obige Problem: Wenn Sie den normalen ExecutorService- & Submit-Befehl verwenden, rufen Sie das Future-Objekt über den submit () - Befehlsaufruf get () für Future ab. Erfassen Sie die drei Ausnahmen, die in der Implementierung der afterExecute-Methode angegeben wurden. Vorteil des benutzerdefinierten ThreadPoolExecutor gegenüber diesem Ansatz: Sie müssen den Ausnahmebehandlungsmechanismus nur an einer Stelle behandeln - Custom ThreadPoolExecutor.

Anwendungsfälle für verschiedene Arten von Parallelitätskonstrukten

1. ExecutorService

   ExecutorService executor = Executors.newFixedThreadPool(50);

   Es ist einfach und leicht zu bedienen. Es verdeckt ThreadPoolExecutor Details von ThreadPoolExecutor .

   Ich ziehe dieses vor, wenn die Anzahl der Callable/Runnable Aufgaben gering ist und die Anzahl der Aufgaben in der unbegrenzten Warteschlange den Speicher nicht erhöht und die Leistung des Systems nicht beeinträchtigt. Wenn Sie über CPU/Memory verfügen, ziehe ich es vor, ThreadPoolExecutor mit Kapazitätsbeschränkungen und RejectedExecutionHandler zu verwenden, um die Ablehnung von Aufgaben auszuführen.

2. CountDownLatch

   CountDownLatch wird mit einer bestimmten Anzahl initialisiert. Diese Anzahl wird durch Aufrufe der countDown() -Methode countDown() . Threads, die darauf warten, dass diese Anzahl Null erreicht, können eine der await() Methoden aufrufen. Der Aufruf von await() blockiert den Thread, bis der Zähler null erreicht. *Diese Klasse ermöglicht es einem Java-Thread, zu warten, bis eine andere Gruppe von Threads ihre Aufgaben abgeschlossen hat.*

   Anwendungsfälle:

   1. Maximale Parallelität erreichen: Manchmal möchten wir mehrere Threads gleichzeitig starten, um maximale Parallelität zu erreichen
   2. Warten Sie, bis N-Threads abgeschlossen sind, bevor Sie die Ausführung starten
   3. Deadlock-Erkennung

3. ThreadPoolExecutor : Es bietet mehr Kontrolle. Wenn die Anwendung durch die Anzahl der ausstehenden ausgeführten / aufrufbaren Aufgaben eingeschränkt wird, können Sie die beschränkte Warteschlange verwenden, indem Sie die maximale Kapazität festlegen. Sobald die Warteschlange die maximale Kapazität erreicht hat, können Sie RejectionHandler definieren. Java bietet vier Arten von RejectedExecutionHandler Richtlinien .

   1. ThreadPoolExecutor.AbortPolicy , der Handler ThreadPoolExecutor.AbortPolicy bei Ablehnung eine Laufzeit-RejectedExecutionException aus.

2. ThreadPoolExecutor.CallerRunsPolicy`, der Thread, der die Ausführung ausführt, führt die Aufgabe aus. Dies stellt einen einfachen Mechanismus zur Regelung der Rückkopplung bereit, der die Geschwindigkeit der Übergabe neuer Aufgaben verlangsamt.

3. In ThreadPoolExecutor.DiscardPolicy wird eine Aufgabe, die nicht ausgeführt werden kann, einfach gelöscht.

4. ThreadPoolExecutor.DiscardOldestPolicy : Wenn der Executor nicht heruntergefahren wird, wird die Task am Anfang der Arbeitswarteschlange gelöscht und die Ausführung wird erneut versucht (dies kann erneut fehlschlagen und führt dazu, dass dies wiederholt wird.)

Wenn Sie das CountDownLatch Verhalten simulieren möchten, können Sie die invokeAll() Methode verwenden.

4. Ein weiterer Mechanismus, den Sie nicht zitiert haben, ist ForkJoinPool

   Der ForkJoinPool wurde Java in Java 7 hinzugefügt. Der ForkJoinPool ähnelt dem Java ExecutorService jedoch mit einem Unterschied. Mit dem ForkJoinPool können Aufgaben einfach in kleinere Aufgaben aufgeteilt werden, die dann ebenfalls an den ForkJoinPool werden. Das Stehlen von Aufgaben geschieht in ForkJoinPool wenn freie Worker-Threads Aufgaben aus der beschäftigten Worker-Warteschlange stehlen.

   Java 8 hat in ExecutorService eine weitere API zum Erstellen von Workstealing-Pools eingeführt. Sie müssen RecursiveTask und RecursiveAction nicht erstellen, können jedoch weiterhin ForkJoinPool .

   ```
   public static ExecutorService newWorkStealingPool()
   ```

   Erstellt einen Worksteal-Thread-Pool, der alle verfügbaren Prozessoren als Ziel-Parallelitätsebene verwendet.

   Standardmäßig wird die Anzahl der CPU-Kerne als Parameter verwendet.

Alle diese vier Mechanismen ergänzen einander. Abhängig von der Granularität, die Sie steuern möchten, müssen Sie die richtige auswählen.

Warten Sie auf den Abschluss aller Aufgaben in ExecutorService

Schauen wir uns die verschiedenen Optionen an, um zu warten, bis die an Executor gesendeten Aufgaben abgeschlossen sind

1. ExecutorService invokeAll()

   Führt die angegebenen Aufgaben aus und gibt eine Liste der Futures zurück, die ihren Status und die Ergebnisse enthalten, wenn alles abgeschlossen ist.

Beispiel:

```
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

   Eine Synchronisationshilfe, mit der ein oder mehrere Threads warten können, bis ein Satz von Vorgängen in anderen Threads abgeschlossen ist.

   Ein **CountDownLatch** wird mit einer bestimmten Anzahl initialisiert. Die wait-Methoden werden blockiert, bis der aktuelle Zählerstand null ist, weil countDown() der countDown() -Methode countDown() , wonach alle wartenden Threads freigegeben werden und alle nachfolgenden Aufrufe von await sofort zurückgegeben werden. Dies ist ein einmaliges Phänomen - die Zählung kann nicht zurückgesetzt werden. Wenn Sie eine Version benötigen, die die Zählung zurücksetzt, sollten Sie einen **CyclicBarrier verwenden** .

3. ForkJoinPool oder newWorkStealingPool() in Executors

4. Durchlaufen Sie alle Future Objekte, die nach dem Senden an ExecutorService

5. Empfohlene Methode zum Herunterfahren von der Oracle-Dokumentationsseite von 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(): Startet ein ordnungsgemäßes Herunterfahren, bei dem zuvor gesendete Aufgaben ausgeführt werden, jedoch keine neuen Aufgaben angenommen werden.

shutdownNow(): Versucht, alle aktiv ausgeführten Aufgaben zu stoppen, hält die Verarbeitung wartender Aufgaben an und gibt eine Liste der Aufgaben zurück, die auf ihre Ausführung warteten.

Wenn im obigen Beispiel Ihre Aufgaben mehr Zeit in Anspruch nehmen, können Sie den if-Zustand in den while-Zustand ändern

Ersetzen

```
if (!pool.awaitTermination(60, TimeUnit.SECONDS))
```

mit

```
while(!pool.awaitTermination(60, TimeUnit.SECONDS)) {
  Thread.sleep(60000);
```

}

Anwendungsfälle für verschiedene Typen von ExecutorService

Executors gibt verschiedene Arten von ThreadPools zurück, die auf die jeweiligen Bedürfnisse zugeschnitten sind.

1. public static ExecutorService newSingleThreadExecutor()

   Erstellt einen Executor, der einen einzelnen Worker-Thread verwendet, der in einer unbegrenzten Warteschlange ausgeführt wird

   Es gibt einen Unterschied zwischen newFixedThreadPool(1) und newSingleThreadExecutor() wie das Java-Dokument für Letzteres sagt:

   Im Gegensatz zum ansonsten äquivalenten newFixedThreadPool (1) ist der zurückgegebene Executor garantiert nicht für die Verwendung zusätzlicher Threads rekonfigurierbar.

   newFixedThreadPool bedeutet, dass ein newFixedThreadPool später im Programm neu newFixedThreadPool kann durch: ((ThreadPoolExecutor) fixedThreadPool).setMaximumPoolSize(10) Dies ist für newSingleThreadExecutor nicht möglich

   Anwendungsfälle:

   1. Sie möchten die übergebenen Aufgaben in einer Reihenfolge ausführen.
   2. Sie benötigen nur einen Thread, um alle Ihre Anfragen zu bearbeiten

   Nachteile:

   1. Ungebundene Warteschlangen sind schädlich

2. public static ExecutorService newFixedThreadPool(int nThreads)

   Erstellt einen Thread-Pool, der eine feste Anzahl von Threads wiederverwendet, die in einer gemeinsam genutzten, nicht gebundenen Warteschlange ausgeführt werden. Zu jedem Zeitpunkt sind höchstens nThreads-Threads aktive Verarbeitungsaufgaben. Wenn zusätzliche Tasks übergeben werden, während alle Threads aktiv sind, warten sie in der Warteschlange, bis ein Thread verfügbar ist

   Anwendungsfälle:

   1. Effektive Nutzung der verfügbaren Kerne. Konfigurieren Sie nThreads als Runtime.getRuntime().availableProcessors()

2. Wenn Sie entscheiden, dass die Anzahl der Threads eine Anzahl im Thread-Pool nicht überschreiten soll

Nachteile:

1. Ungebundene Warteschlangen sind schädlich.

3. `public static ExecutorService newCachedThreadPool()`

   Erstellt einen Thread-Pool, der nach Bedarf neue Threads erstellt, zuvor erstellte Threads jedoch wieder verwendet, wenn sie verfügbar sind

Anwendungsfälle:

1. Für kurzlebige asynchrone Aufgaben

Nachteile:

1. Ungebundene Warteschlangen sind schädlich.
2. Jede neue Aufgabe erstellt einen neuen Thread, wenn alle vorhandenen Threads belegt sind. Wenn die Task längere Zeit in Anspruch nimmt, werden mehr Threads erstellt, was die Leistung des Systems beeinträchtigt. Alternative in diesem Fall: newFixedThreadPool

4. `public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`

   Erstellt einen Thread-Pool, der Befehle planen kann, die nach einer bestimmten Verzögerung ausgeführt werden oder regelmäßig ausgeführt werden.

Anwendungsfälle:

1. Behandlung wiederkehrender Ereignisse mit Verzögerungen, die in bestimmten Zeitabständen in der Zukunft auftreten werden

Nachteile:

1. Ungebundene Warteschlangen sind schädlich.

5. `public static ExecutorService newWorkStealingPool()`

   Erstellt einen Worksteal-Thread-Pool, der alle verfügbaren Prozessoren als Ziel-Parallelitätsebene verwendet

Anwendungsfälle:

1. Für das Teilen und Erobern von Problemen.
2. Effektive Verwendung von im Leerlauf befindlichen Threads. Leere Threads stehlen Aufgaben aus ausgelasteten Threads.

Nachteile:

1. Nicht gebundene Warteschlangengröße ist schädlich.

In all diesen ExecutorService können Sie einen Nachteil feststellen: die unbegrenzte Warteschlange. Dies wird mit ThreadPoolExecutor angesprochen

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

Mit ThreadPoolExecutor können Sie

1. Kontrollieren Sie die Thread-Poolgröße dynamisch
2. Stellen Sie die Kapazität für BlockingQueue
3. Definieren Sie RejectionExecutionHander wenn die Warteschlange voll ist
4. CustomThreadFactory , um einige zusätzliche Funktionen während der Thread-Erstellung

hinzuzufügen (public Thread newThread(Runnable r)

Thread-Pools verwenden

Thread-Pools werden hauptsächlich in ExecutorService .

Die folgenden Methoden können verwendet werden, um Arbeit zur Ausführung einzureichen:

Methode	Beschreibung
submit	Führt die eingereichte Arbeit aus und gibt eine Zukunft zurück, die zum Abrufen des Ergebnisses verwendet werden kann
execute	Führen Sie die Aufgabe irgendwann in der Zukunft aus, ohne einen Rückgabewert zu erhalten
invokeAll	Führen Sie eine Liste mit Aufgaben aus und geben Sie eine Liste mit Futures zurück
invokeAny	Führt alle aus, gibt jedoch nur das Ergebnis eines erfolgreichen Ergebnisses zurück (ohne Ausnahmen)

Wenn Sie mit dem Thread-Pool fertig sind, können Sie shutdown() aufrufen, um den Thread-Pool zu beenden. Dadurch werden alle anstehenden Aufgaben ausgeführt. Um auf die Ausführung aller Tasks zu warten, können Sie awaitTermination oder isShutdown() .

Executor-, ExecutorService- und Thread-Pools online lesen:
https://riptutorial.com/de/java/topic/143/executor---executorservice--und-thread-pools

Kapitel 44: FileUpload in AWS

Einführung

Laden Sie die Datei mithilfe der Federauflagen-API in den AWS S3-Bucket hoch.

Examples

Laden Sie die Datei in den S3-Bucket hoch

Hier erstellen wir ein Rest-APi, das das Dateiobjekt als mehrteiligen Parameter vom Frontend übernimmt und mit der Java-Rest-API in den S3-Bucket lädt.

Voraussetzung : - Geheimschlüssel und Zugriffsschlüssel für den S3-Bucket, in den Sie Ihre Datei hochladen möchten.

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;

    }


}
```

Frontend-Funktion

```
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);
});
```

FileUpload in AWS online lesen: https://riptutorial.com/de/java/topic/10589/fileupload-in-aws

Kapitel 45: Fließende Schnittstelle

Bemerkungen

Tore

Das Hauptziel einer Fluent-Schnittstelle ist die Verbesserung der Lesbarkeit.

Bei der Verwendung zum Erstellen von Objekten können die für den Anrufer verfügbaren Auswahlmöglichkeiten klar festgelegt und durch Überprüfungen während der Kompilierungszeit durchgesetzt werden. Betrachten Sie zum Beispiel die folgende Optionsstruktur, die Schritte entlang des Pfads darstellt, um ein komplexes Objekt zu erstellen:

```
A -> B
  -> C -> D -> Done
       -> E -> Done
       -> F -> Done.
       -> G -> H -> I -> Done.
```

Mit einem Builder, der eine fließende Schnittstelle verwendet, kann der Anrufer leicht erkennen, welche Optionen bei jedem Schritt verfügbar sind. Zum Beispiel ist **A -> B** möglich, aber **A -> C** ist nicht und würde zu einem Kompilierungsfehler führen.

Examples

Wahrheit - Fließendes Test-Framework

Über "So verwenden Sie die Wahrheit" 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();
```

Fließender Programmierstil

Im fließenden Programmierstil geben Sie this von fließenden (Setter-) Methoden zurück, die nichts im nicht fließenden Programmierstil zurückgeben würden.

Dadurch können Sie die verschiedenen Methodenaufrufe verketten, wodurch Ihr Code für die Entwickler kürzer und einfacher zu handhaben ist.

Betrachten Sie diesen nicht fließenden Code:

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

Da die Setter-Methoden nichts zurückgeben, benötigen wir 4 Anweisungen in der main , um eine Person mit einigen Daten zu instanziieren und zu drucken. Mit einem fließenden Stil kann dieser Code folgendermaßen geändert werden:

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

Die Idee ist, immer ein Objekt zurückzugeben, um die Erstellung einer Methodenaufrufkette zu ermöglichen und Methodennamen zu verwenden, die das natürliche Sprechen widerspiegeln. Dieser flüssige Stil macht den Code lesbarer.

Fließende Schnittstelle online lesen: https://riptutorial.com/de/java/topic/5090/flie-ende-schnittstelle

Kapitel 46: FTP (File Transfer Protocol)

Syntax

- FTPClient-Verbindung (InetAddress-Host, int-Port)
- FTPClient-Login (String-Benutzername, String-Passwort)
- FTPClient disconnect ()
- FTPReply getReplyStrings ()
- boolean storeFile (String remote, InputStream local)
- OutputStream storeFileStream (Zeichenfolge remote)
- boolean setFileType (int fileType)
- boolean completePendingCommand ()

Parameter

Parameter	Einzelheiten
Wirt	Entweder der Hostname oder die IP-Adresse des FTP-Servers
Hafen	Der FTP-Server-Port
Nutzername	Der FTP-Server-Benutzername
Passwort	Das FTP-Server-Passwort

Examples

Anschließen und Anmelden an einem FTP-Server

Um mit der Verwendung von FTP mit Java zu beginnen, müssen Sie einen neuen FTPClient erstellen und sich dann mit .connect(String server, int port) und .login(String username, String password) mit dem Server verbinden.

```
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);
    }
}
```

Jetzt haben wir die Grundlagen fertig. Was ist, wenn wir einen Fehler beim Verbindungsaufbau zum Server haben? Wir möchten wissen, wann etwas schief geht, und die Fehlermeldung erhalten. Fügen Sie etwas Code hinzu, um Fehler beim Verbindungsaufbau abzufangen.

```
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 {

}
```

Lassen Sie uns das, was wir gerade gemacht haben, Schritt für Schritt zusammenbrechen.

```
showServerReply(ftp);
```

Dies bezieht sich auf eine Funktion, die wir in einem späteren Schritt ausführen werden.

```
int replyCode = ftp.getReplyCode();
```

Dadurch wird der Antwort- / Fehlercode vom Server abgerufen und als Ganzzahl gespeichert.

```
if (!FTPReply.isPositiveCompletion(replyCode)) {
    System.out.printIn("Operation failed. Server reply code: " + replyCode)
    return;
}
```

Dies überprüft den Antwortcode, um festzustellen, ob ein Fehler aufgetreten ist. Wenn ein Fehler aufgetreten ist, wird einfach "Vorgang fehlgeschlagen. Antwortcode des Servers:" gefolgt vom Fehlercode gedruckt. Wir haben auch einen Try / Catch-Block hinzugefügt, den wir im nächsten Schritt hinzufügen werden. Als Nächstes erstellen wir eine Funktion, die ftp.login() auf Fehler überprüft.

```
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");
    }
```

Lassen Sie uns auch diesen Block abbauen.

```
boolean success = ftp.login(user, pass);
```

Dadurch wird nicht nur versucht, sich beim FTP-Server anzumelden, sondern das Ergebnis wird auch als Boolean gespeichert.

```
showServerReply(ftp);
```

Dadurch wird geprüft, ob der Server uns Nachrichten gesendet hat. Im nächsten Schritt müssen Sie jedoch zuerst die Funktion erstellen.

```
if (!success) {
System.out.println("Failed to log into the server");
    return;
```

```
} else {
    System.out.println("LOGGED IN SERVER");
}
```

Diese Anweisung überprüft, ob wir uns erfolgreich angemeldet haben. Wenn ja, wird "LOGGED IN SERVER" gedruckt, andernfalls "Anmeldung beim Server fehlgeschlagen". Dies ist unser bisheriges Skript:

```
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 {

        }
    }
}
```

Nun erstellen wir als Nächstes den Catch-Block, falls im gesamten Prozess Fehler auftreten.

```
} catch (IOException ex) {
    System.out.println("Oops! Something went wrong.");
    ex.printStackTrace();
}
```

Der abgeschlossene Auffangblock druckt jetzt "Oops! Etwas ist schief gelaufen". und der stacktrace, wenn ein Fehler vorliegt. Nun ist der letzte Schritt das Erstellen des showServerReply() wir seit einiger Zeit verwenden.

```
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);
        }
```

```
    }
}
```

Diese Funktion nimmt einen FTPClient als Variable und nennt ihn "ftp". Danach speichert er alle Serverantworten vom Server in einem String-Array. Als nächstes wird geprüft, ob Nachrichten gespeichert wurden. Wenn es welche gibt, werden diese als "SERVER: [Antwort]" ausgegeben. Nun, da wir diese Funktion erledigt haben, ist dies das vollständige Skript:

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

Wir müssen zuerst einen neuen FTPClient erstellen und versuchen, eine Verbindung zum Server FTPClient und ihn unter Verwendung von .connect(String server, int port) und .login(String username, String password) . Es ist wichtig, sich über einen try / catch-Block zu verbinden und sich anzumelden, falls unser Code keine Verbindung zum Server herstellen kann. Wir müssen auch eine Funktion erstellen, die alle Nachrichten überprüft und anzeigt, die wir vom Server erhalten, wenn wir versuchen, eine Verbindung herzustellen und sich showServerReply(FTPClient ftp) . Wir nennen diese Funktion " 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();
        }
    }
}
```

Danach sollten Sie jetzt Ihren FTP-Server mit Ihrem Java-Skript verbunden haben.

FTP (File Transfer Protocol) online lesen: https://riptutorial.com/de/java/topic/5228/ftp--file-transfer-protocol-

Kapitel 47: Funktionale Schnittstellen

Einführung

In Java 8+ ist eine *funktionale Schnittstelle* eine Schnittstelle, die nur eine abstrakte Methode hat (abgesehen von den Methoden von Object). Siehe JLS §9.8. Funktionale Schnittstellen .

Examples

Liste der Standardfunktionsschnittstellen der Java Runtime Library nach Signatur

Parametertypen	Rückgabetyp	Schnittstelle
()	Leere	Lauffähig
()	T	Lieferant
()	boolean	BooleanSupplier
()	int	IntSupplier
()	lange	LongLieferant
()	doppelt	DoubleSupplier
(T)	Leere	Verbraucher <T>
(T)	T	UnaryOperator <T>
(T)	R	Funktion <T, R>
(T)	boolean	Prädikat <T>
(T)	int	ToIntFunction <T>
(T)	lange	ToLongFunction <T>
(T)	doppelt	ToDoubleFunction <T>
(T, T)	T	BinaryOperator <T>
(T, U)	Leere	BiConsumer <T, U>
(T, U)	R	BiFunktion <T, U, R>
(T, U)	boolean	BiPredicate <T, U>
(T, U)	int	ToIntBiFunction <T, U>
(T, U)	lange	ToLongBiFunction <T, U>

Parametertypen	Rückgabetyp	Schnittstelle
(T, U)	doppelt	ToDoubleBiFunction <T, U>
(T, int)	Leere	ObjIntConsumer <T>
(T, lang)	Leere	ObjLongConsumer <T>
(T, doppelt)	Leere	ObjDoubleConsumer <T>
(int)	Leere	IntConsumer
(int)	R	IntFunction <R>
(int)	boolean	IntPredicate
(int)	int	IntUnaryOperator
(int)	lange	IntToLongFunction
(int)	doppelt	IntToDoubleFunction
(int, int)	int	IntBinaryOperator
(lange)	Leere	LongConsumer
(lange)	R	LongFunction <R>
(lange)	boolean	LongPredicate
(lange)	int	LongToIntFunction
(lange)	lange	LongUnaryOperator
(lange)	doppelt	LongToDoubleFunction
(lang Lang)	lange	LongBinaryOperator
(doppelt)	Leere	DoubleConsumer
(doppelt)	R	DoubleFunction <R>
(doppelt)	boolean	DoublePredicate
(doppelt)	int	DoubleToIntFunction
(doppelt)	lange	DoubleToLongFunction
(doppelt)	doppelt	DoubleUnaryOperator
(doppelt, doppelt)	doppelt	DoubleBinaryOperator

Funktionale Schnittstellen online lesen:
https://riptutorial.com/de/java/topic/10001/funktionale-schnittstellen

Kapitel 48: Generics

Einführung

Generics sind eine Möglichkeit der generischen Programmierung, die das Java-Typsystem erweitert, um es einem Typ oder einer Methode zu ermöglichen, Objekte verschiedener Typen zu bearbeiten und gleichzeitig die Sicherheit der Kompilierzeit zu gewährleisten. Insbesondere unterstützt das Java Collections Framework Generics, um den Typ von Objekten anzugeben, die in einer Collection-Instanz gespeichert sind.

Syntax

- class ArrayList <E> {} // eine generische Klasse mit dem Typparameter E
- class HashMap <K, V> {} // eine generische Klasse mit zwei Typparametern K und V
- <E> void print (E-Element) {} // eine generische Methode mit dem Typparameter E
- ArrayList <String> Namen; // Deklaration einer generischen Klasse
- ArrayList <?> Objekte; // Deklaration einer generischen Klasse mit einem unbekannten Typparameter
- new ArrayList <String> () // Instantiierung einer generischen Klasse
- neue ArrayList <> () // Instantiierung mit Typinferenz "Raute" (Java 7 oder höher)

Bemerkungen

Generics werden in Java durch Typlöschung implementiert. Dies bedeutet, dass zur Laufzeit die in der Instanziierung einer generischen Klasse angegebenen Typinformationen nicht verfügbar sind. Zum Beispiel die Anweisung List<String> names = new ArrayList<>(); erzeugt ein Listenobjekt, aus dem der Elementtyp String zur Laufzeit nicht wiederhergestellt werden kann. Wenn jedoch die Liste in einem Feld vom Typ gespeichert ist List<String> , oder ein Verfahren / Konstruktorparameter des gleichen Typs übergeben oder von einem Verfahren dieser Rückgabetyp zurückgegeben, dann ist die vollständige Typinformationen zur Laufzeit zurückgewonnen werden über die Java Reflection-API.

Dies bedeutet auch, dass beim Casting auf einen generischen Typ (z. B. (List<String>) list) der Cast ein *ungeprüfter Cast ist* . Da der Parameter <String> gelöscht wird, kann die JVM nicht prüfen, ob eine Umwandlung von einer List<?> eine List<String> korrekt ist. Die JVM sieht zur Laufzeit nur einen Cast für List to List .

Examples

Generische Klasse erstellen

Generics ermöglichen Klassen, Schnittstellen und Methoden, andere Klassen und Schnittstellen als Typparameter zu verwenden.

In diesem Beispiel wird die generische Klasse Param , um einen einzelnen **Typparameter** T , der durch spitze Klammern (<>) getrennt ist:

```
public class Param<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

Geben Sie zum Instanziieren dieser Klasse ein **Typargument** anstelle von T . Zum Beispiel Integer :

```
Param<Integer> integerParam = new Param<Integer>();
```

Das Typargument kann ein beliebiger Referenztyp sein, einschließlich Arrays und andere generische Typen:

```
Param<String[]> stringArrayParam;
Param<int[][]> int2dArrayParam;
Param<Param<Object>> objectNestedParam;
```

In Java SE 7 und höher kann das Typargument durch einen leeren Satz von Typargumenten (<>) ersetzt werden, der als *Raute bezeichnet wird* :

Java SE 7

```
Param<Integer> integerParam = new Param<>();
```

Im Gegensatz zu anderen Bezeichnern haben Typparameter keine Namensbeschränkungen. Ihre Namen sind jedoch in der Regel der erste Buchstabe ihres Zwecks in Großbuchstaben. (Dies gilt sogar für alle offiziellen JavaDocs.)
Beispiele sind T für "Typ" , E für "Element" und K / V für "Schlüssel" / "Wert" .

Eine generische Klasse erweitern

```
public abstract class AbstractParam<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

AbstractParam ist eine abstrakte Klasse , die mit einem Typparameter von T deklariert wird. Beim Erweitern dieser Klasse kann dieser Typparameter durch ein in <> geschriebenes <> , oder der Typparameter kann unverändert bleiben. In den ersten und zweiten Beispielen ersetzen String und Integer den Parameter type. Im dritten Beispiel bleibt der Typparameter unverändert. Im vierten Beispiel werden keine Generics verwendet, es ist also ähnlich, wenn die Klasse einen Object Parameter hätte. Der Compiler warnt davor, dass AbstractParam ein Raw-Typ ist, kompiliert jedoch die ObjectParam Klasse. Das fünfte Beispiel verfügt über 2 Typparameter (siehe "Multiple Type-Parameter" weiter unten), wobei der zweite Parameter als der an die Oberklasse übergebene Typparameter ausgewählt wird.

```
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> {
    // ...
}
```

Folgendes ist die Verwendung:

```
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);
```

Beachten Sie, dass die T getValue() -Methode in der Email Klasse so T getValue() als hätte sie eine Signatur von String getValue() und die void setValue(T) -Methode so, als wäre sie für void setValue(String) deklariert worden.

Es ist auch möglich, mit anonymen inneren Klassen mit leeren geschweiften Klammern ({}) zu instanziieren:

```
AbstractParam<Double> height = new AbstractParam<Double>(){};
height.setValue(198.6);
```

Beachten Sie, dass die Verwendung des Diamanten mit anonymen inneren Klassen nicht zulässig ist.

Mehrere Typparameter

Java bietet die Möglichkeit, mehrere Typparameter in einer generischen Klasse oder Schnittstelle zu verwenden. Mehrere Typparameter können in einer Klasse oder einem Interface verwendet werden, indem Sie eine durch **Kommas getrennte Liste** von Typen in die spitzen Klammern setzen. Beispiel:

```
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;
    }
}
```

Die Verwendung kann wie folgt erfolgen:

```
MultiGenericParam<String, String> aParam = new MultiGenericParam<String, String>("value1",
"value2");
MultiGenericParam<Integer, Double> dayOfWeekDegrees = new MultiGenericParam<Integer,
Double>(1, 2.6);
```

Eine generische Methode deklarieren

Methoden können auch generische Typparameter haben.

```
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);
    }
}
```

Beachten Sie, dass wir kein tatsächliches Typargument an eine generische Methode übergeben müssen. Der Compiler leitet das Typargument für uns ein, basierend auf dem Zieltyp (z. B. der Variablen, der wir das Ergebnis zuweisen) oder den Typen der tatsächlichen Argumente. In der Regel wird auf das spezifischste Typargument geschlossen, das den Aufruftypkorrigiert.

Manchmal, wenn auch selten, kann es notwendig sein, diese Typeninferenz mit expliziten Typargumenten zu überschreiben:

```
void usage() {
    consumeObjects(this.<Object>makeList("Jeff", "Atwood").stream());
}

void consumeObjects(Stream<Object> stream) { ... }
```

Dies ist in diesem Beispiel notwendig, da der Compiler nach dem Aufruf von stream() nicht nach vorne schauen kann, um zu sehen, dass Object für T erwünscht ist, und dass andernfalls String

basierend auf den makeList Argumenten abgeleitet würde. Beachten Sie, dass die Java - Sprache nicht unterstützt , die Klasse oder das Objekt Weglassen , auf dem das Verfahren (genannt wird this in dem obigen Beispiel) , wenn Typargumente explizit vorgesehen sind.

Der Diamant

Java SE 7

Java 7 führte den *Diamond* [1] ein , um einige Boilerplatten für die generische Klasseninstanziierung zu entfernen. Mit Java 7+ können Sie schreiben:

```
List<String> list = new LinkedList<>();
```

Wo Sie in früheren Versionen schreiben mussten:

```
List<String> list = new LinkedList<String>();
```

Eine Einschränkung gilt für anonyme Klassen , bei denen Sie den type-Parameter in der Instanziierung noch angeben müssen:

```
// 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

Obwohl die Raute mit anonymen inneren Klassen in Java 7 und 8 nicht unterstützt wird, **wird sie als neue Funktion in Java 9 aufgenommen** .

Fußnote:

1 - Manche Leute nennen das <> Nutzung der „Diamant _ *Operator*". Das ist falsch. Der Diamant verhält sich nicht wie ein Operator und wird nicht als Operator in der JLS oder den (offiziellen) Java-Tutorials beschrieben. In der Tat ist <> kein eindeutiges Java-Token. Es ist eher ein < Token, gefolgt von einem > Token, und es ist legal (wenn auch schlecht), Leerzeichen oder Kommentare zwischen den beiden zu haben. Das JLS und die Tutorials beziehen sich stets auf <> als "den Diamanten", und das ist daher der richtige Begriff dafür.

Mehrere obere Grenzen erforderlich ("erweitert A & B")

Sie können einen generischen Typ benötigen, um mehrere obere Grenzen zu erweitern.

Beispiel: Wir möchten eine Liste von Zahlen sortieren, aber Number implementiert Comparable .

```
public <T extends Number & Comparable<T>> void sortNumbers( List<T> n ) {
  Collections.sort( n );
```

```
}
```

In diesem Beispiel muss T die Number *und* Comparable<T> implementieren, die in alle "normalen" integrierten Striped64 wie Integer oder BigDecimal passen sollte, nicht jedoch für exotischere wie Striped64 .

Da Mehrfachvererbung nicht zulässig ist, können Sie höchstens eine Klasse als Begrenzung verwenden. Es muss die zuerst aufgelistete Klasse sein. Beispielsweise ist <T extends Comparable<T> & Number> nicht zulässig, da Comparable eine Schnittstelle und keine Klasse ist.

Erstellen einer begrenzten generischen Klasse

Sie können die gültigen Typen einschränken, die in einer **generischen Klasse verwendet werden**, indem Sie diesen Typ in der Klassendefinition begrenzen. In Anbetracht der folgenden einfachen Typhierarchie:

```
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";
    }
}
```

Ohne **beschränkte Generika** können wir keine Containerklasse erstellen, die sowohl generisch ist als auch weiß, dass jedes Element ein Tier ist:

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

Mit generisch gebundenen Klassendefinitionen ist dies jetzt möglich.

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

Dies beschränkt auch die gültigen Instanziierungen des generischen Typs:

```
// 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>();
```

Entscheidung zwischen "T", "?" Super T` und `? verlängert T`

Die Syntax für Java-Generics beschränkt Wildcards, die den unbekannten Typ durch ? ist:

- ? extends T für einen Platzhalter mit Oberbegrenzung. Der unbekannte Typ steht für einen Typ, der ein Subtyp von T oder T sein muss.
- ? super T für einen Platzhalter mit unterer Begrenzung. Der unbekannte Typ steht für einen Typ, der ein Supertyp von T oder T sein muss.

Als Faustregel sollten Sie verwenden

- ? extends T wenn Sie nur Lesezugriff benötigen ("Eingabe")
- ? super T wenn Sie "Schreibzugriff" benötigen ("Ausgabe")
- T wenn Sie beides benötigen ("ändern")

Die Verwendung von extends oder super ist in der Regel *besser,* da der Code dadurch flexibler wird (z. B. die Verwendung von Subtypen und Supertypen), wie Sie unten sehen werden.

```
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) {}
}
```

Der Compiler kann nun eine bestimmte fehlerhafte Verwendung erkennen:

```
 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!
}
```

Wahl des richtigen T ? super T oder ? extends T ist *notwendig* , um die Verwendung mit Untertypen zu ermöglichen. Der Compiler kann dann die Typsicherheit gewährleisten; Sie sollten keine Umwandlung durchführen (was nicht typsicher ist und Programmierfehler verursachen kann), wenn Sie sie ordnungsgemäß verwenden.

Wenn es nicht leicht zu verstehen ist, denken **Sie** bitte an die **PECS-** Regel:

Der Produzent verwendet " **E** xtends" und der **C** onsumer " **S** uper".

(Der Produzent hat nur Schreibzugriff und Consumer hat nur Lesezugriff.)

Vorteile der generischen Klasse und Schnittstelle

Code, der Generics verwendet, hat viele Vorteile gegenüber nicht generischem Code. Nachfolgend sind die wichtigsten Vorteile aufgeführt

Stärkere Typprüfungen zur Kompilierzeit

Ein Java-Compiler führt eine starke Typüberprüfung für generischen Code durch und gibt Fehler aus, wenn der Code die Typsicherheit verletzt. Das Beheben von Fehlern bei der Kompilierung ist einfacher als das Beheben von Laufzeitfehlern, die schwer zu finden sind.

Eliminierung von Abgüssen

Das folgende Code-Snippet ohne Generics erfordert ein Casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Wenn der Code für die *Verwendung von Generics* umgeschrieben wurde, ist kein Casting erforderlich:

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0);   // no cast
```

Programmierern die Implementierung generischer Algorithmen ermöglichen

Durch die Verwendung von Generics können Programmierer generische Algorithmen implementieren, die mit Auflistungen verschiedener Typen arbeiten, anpassbar sind und typsicher sind und einfacher zu lesen sind.

Generischer Parameter an mehr als einen Typ binden

Generische Parameter können auch an mehrere Typen gebunden werden, indem die Syntax `T extends Type1 & Type2 & ... .`

Closeable , Sie möchten eine Klasse erstellen, deren generischer Typ sowohl Flushable als auch Closeable implementieren Closeable . Sie können schreiben

```
class ExampleClass<T extends Flushable & Closeable> {
}
```

Nun akzeptiert die ExampleClass nur generische Parameter, die sowohl Flushable **als auch** 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.
```

Die Klassenmethoden können generische Closeable entweder als Closeable oder 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.
    }
}
```

Hinweis:

Sie können den generischen Parameter nicht mit der *OR* (|) - Klausel an einen der Typen binden. Es wird nur die *AND* (&) - Klausel unterstützt. Der generische Typ kann nur eine Klasse und viele Schnittstellen erweitern. Die Klasse muss am Anfang der Liste stehen.

Instanziieren eines generischen Typs

Aufgrund der Typenlöschung funktioniert Folgendes nicht:

```
public <T> void genericMethod() {
    T t = new T(); // Can not instantiate the type T.
}
```

Der Typ T wird gelöscht. Da die JVM zur Laufzeit nicht weiß, was T ursprünglich war, weiß sie nicht, welchen Konstruktor sie aufrufen soll.

Problemumgehungen

1. Übergeben der Klasse T beim Aufruf von 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);
   ```

   Gibt Ausnahmen aus, da nicht bekannt ist, ob die übergebene Klasse einen Standardkonstruktor für den Zugriff hat.

Java SE 8

2. Übergabe einer Referenz an den Konstruktor von T :

   ```
   public <T> void genericMethod(Supplier<T> cons) {
       T t = cons.get();
   }
   ```

   ```
   genericMethod(String::new);
   ```

Verweist auf den deklarierten generischen Typ in seiner eigenen Deklaration

Wie gehen Sie vor, wenn Sie eine Instanz eines (möglicherweise weiteren) geerbten generischen

Typs innerhalb einer Methodendeklaration im generischen Typ selbst verwenden, der gerade deklariert wird? Dies ist eines der Probleme, mit denen Sie konfrontiert werden, wenn Sie ein wenig tiefer in die Generika einsteigen, aber immer noch ein häufiges Problem.

Angenommen, wir haben einen DataSeries<T> -Typ (Schnittstelle hier), der eine generische Datenreihe definiert, die Werte des Typs T . Es ist umständlich, direkt mit diesem Typ zu arbeiten, wenn viele Vorgänge mit z. B. Doppelwerten ausgeführt werden sollen. DoubleSeries extends DataSeries<Double> definieren wir DoubleSeries extends DataSeries<Double> . DataSeries<T> wir nun an, der ursprüngliche DataSeries<T> Typ DataSeries<T> hat eine Methode add(values) die eine weitere Reihe derselben Länge hinzufügt und eine neue zurückgibt. Wie können wir die Art der Durchsetzung values und die Art der Rückkehr in seine DoubleSeries anstatt DataSeries<Double> in unserer abgeleiteten Klasse?

Das Problem kann gelöst werden, indem ein generischer Typparameter hinzugefügt wird, der sich auf den deklarierten Typ bezieht (dieser wird hier auf ein Interface angewendet, gilt jedoch für Klassen):

```
public interface DataSeries<T, DS extends DataSeries<T, DS>> {
    DS add(DS values);
    List<T> data();
}
```

Hier steht T für den Datentyp der Serie, z. B. Double und DS die Serie selbst. Ein vererbte Typ (oder Typen) kann nun leicht durch Substitution des oben genannten Parameters durch eine entsprechende abgeleiteten Art implementiert werden, also eine konkrete Nachgeben Double - basierte Definition der Form:

```
public interface DoubleSeries extends DataSeries<Double, DoubleSeries> {
    static DoubleSeries instance(Collection<Double> data) {
        return new DoubleSeriesImpl(data);
    }
}
```

In diesem Moment implementiert sogar eine IDE die obige Schnittstelle mit den richtigen Typen, die nach einiger Inhaltsfüllung folgendermaßen aussehen können:

```
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);
    }
}
```

Wie Sie sehen, ist die add Methode als DoubleSeries add(DoubleSeries values) und der Compiler ist zufrieden.

Das Muster kann bei Bedarf weiter verschachtelt werden.

Verwendung von instanceof mit Generics

Generics verwenden, um den Typ in instanceof zu definieren

Betrachten Sie die folgende generische Klasse Example , die mit dem Formalparameter <T> deklariert wurde:

```
class Example<T> {
    public boolean isTypeAString(String s) {
        return s instanceof T; // Compilation error, cannot use T as class type here
    }
}
```

Dies führt immer zu einem Kompilierungsfehler, da der Compiler, sobald er die *Java-Quelle* in *Java-Bytecode* kompiliert, einen als *Typ Erasure bezeichneten* Prozess anwendet, der den generischen Code in nicht generischen Code umwandelt, so dass die T-Typen zur Laufzeit nicht unterschieden werden können. Der mit instanceof verwendete Typ muss wieder *verwendbar sein* . Das bedeutet, dass alle Informationen zum Typ zur Laufzeit verfügbar sein müssen. Dies ist bei generischen Typen normalerweise nicht der Fall.

Die folgende Klasse stellt dar, wie zwei verschiedene Klassen von Example , Example<String> und Example<Number> aussehen, nachdem Generics durch *Typlöschung entfernt wurden* :

```
class Example { // formal parameter is gone
    public boolean isTypeAString(String s) {
        return s instanceof Object; // Both <String> and <Number> are now Object
    }
}
```

Da Typen weg sind, kann die JVM nicht wissen, welcher Typ T .

Ausnahme zur vorherigen Regel

Sie können einen *unbegrenzten Platzhalter* (?) Immer verwenden, um einen Typ in der instanceof wie folgt anzugeben:

```
    public boolean isAList(Object obj) {
        return obj instanceof List<?>;
    }
```

Dies kann nützlich sein, um auszuwerten, ob eine Instanz obj eine List oder nicht:

```
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
```

Unbegrenzte Wildcards werden tatsächlich als wiederverwendbare Typen betrachtet.

Verwenden einer generischen Instanz mit instanceof

Die andere Seite der Münze ist, dass die Verwendung einer Instanz t von T mit instanceof legal ist, wie im folgenden Beispiel gezeigt:

```
class Example<T> {
```

```
    public boolean isTypeAString(T t) {
        return t instanceof String; // No compilation error this time
    }
}
```

denn nach der Typlöschung sieht die Klasse wie folgt aus:

```
class Example { // formal parameter is gone
    public boolean isTypeAString(Object t) {
        return t instanceof String; // No compilation error this time
    }
}
```

Da die JVM, selbst wenn das Löschen des Typs trotzdem auftritt, jetzt verschiedene Typen im Speicher unterscheiden kann, auch wenn sie denselben Referenztyp (Object) verwenden, wie der folgende Ausschnitt zeigt:

```
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
```

Verschiedene Möglichkeiten zur Implementierung einer generischen Schnittstelle (oder zur Erweiterung einer generischen Klasse)

Angenommen, die folgende generische Schnittstelle wurde deklariert:

```
public interface MyGenericInterface<T> {
    public void foo(T t);
}
```

Nachfolgend sind die möglichen Implementierungsmöglichkeiten aufgeführt.

Nicht-generische Klassenimplementierung mit einem bestimmten Typ

Wählen Sie einen bestimmten Typ aus, um den MyGenericClass Parameter <T> von MyGenericClass zu ersetzen, und implementieren Sie ihn wie im folgenden Beispiel:

```
public class NonGenericClass implements MyGenericInterface<String> {
    public void foo(String t) { } // type T has been replaced by String
}
```

Diese Klasse befasst sich nur mit String . Dies bedeutet, dass die Verwendung von MyGenericInterface mit anderen Parametern (z. B. Integer , Object usw.) nicht kompiliert wird, wie das folgende Snippet zeigt:

```
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
```

Generische Klassenimplementierung

Deklarieren Sie eine weitere generische Schnittstelle mit dem formalen Typparameter <T> der MyGenericInterface wie folgt implementiert:

```
public class MyGenericSubclass<T> implements MyGenericInterface<T> {
    public void foo(T t) { } // type T is still the same
    // other methods...
}
```

Beachten Sie, dass möglicherweise ein anderer Parameter für den formalen Typ verwendet wurde:

```
public class MyGenericSubclass<U> implements MyGenericInterface<U> { // equivalent to the
previous declaration
    public void foo(U t) { }
    // other methods...
}
```

Raw-Typ-Klassenimplementierung

Deklarieren Sie eine nicht-generische Klasse , die implementiert MyGenericInteface als *Ausgangstyp* (nicht bei allen mit Generika), wie folgt:

```
public class MyGenericSubclass implements MyGenericInterface {
    public void foo(Object t) { } // type T has been replaced by Object
    // other possible methods
}
```

Auf diese Weise wird **nicht** empfohlen, da es nicht 100% sicher zur Laufzeit ist , weil sie *roh Art* mischt (die Unterklasse) mit *Generika* (der Schnittstelle) und es ist auch verwirrend. Moderne Java-Compiler geben bei dieser Art der Implementierung eine Warnung aus, der Code wird jedoch aus Kompatibilitätsgründen mit älteren JVM (1.4 oder früher) kompiliert.

Alle oben aufgeführten Möglichkeiten sind auch zulässig, wenn eine generische Klasse als Supertyp anstelle einer generischen Schnittstelle verwendet wird.

Verwenden von Generics zum automatischen Casting

Mit Generics ist es möglich, zurückzugeben, was der Anrufer erwartet:

```
private Map<String, Object> data;
public <T> T get(String key) {
    return (T) data.get(key);
}
```

Die Methode wird mit einer Warnung kompiliert. Der Code ist tatsächlich sicherer als er aussieht, da die Java-Laufzeitumgebung bei der Verwendung eine Umwandlung durchführt:

```
Bar bar = foo.get("bar");
```

Es ist weniger sicher, wenn Sie generische Typen verwenden:

```
List<Bar> bars = foo.get("bars");
```

In diesem ClassCastException , wenn der zurückgegebene Typ irgendeine Art von List (dh die Rückgabe von List<String> würde keine ClassCastException auslösen; Sie würden sie schließlich erhalten, wenn Elemente aus der Liste entfernt werden).

Um dieses Problem zu umgehen, können Sie eine API erstellen, die eingegebene Schlüssel verwendet:

```
public final static Key<List<Bar>> BARS = new Key<>("BARS");
```

zusammen mit dieser put() Methode:

```
public <T> T put(Key<T> key, T value);
```

Bei diesem Ansatz können Sie nicht den falschen Typ in die Map einfügen. Das Ergebnis ist also immer korrekt (es sei denn, Sie erstellen aus Versehen zwei Schlüssel mit demselben Namen, aber unterschiedlichen Typen).

Verbunden:

- Typensichere Karte

Rufen Sie eine Klasse ab, die zur Laufzeit die generischen Parameter erfüllt

Viele ungebundene generische Parameter, wie sie in einer statischen Methode verwendet werden, können zur Laufzeit nicht wiederhergestellt werden (siehe *Andere Threads* zum *Löschen*). Es gibt jedoch eine gemeinsame Strategie, um auf den Typ zuzugreifen, der zur Laufzeit einen generischen Parameter einer Klasse erfüllt. Dies ermöglicht generischen Code, der vom Zugriff auf den Typ abhängt, *ohne* dass bei jedem Aufruf die Typinformationen eingezogen werden müssen.

Hintergrund

Die generische Parametrisierung für eine Klasse kann durch Erstellen einer anonymen inneren Klasse überprüft werden. Diese Klasse erfasst die Typinformationen. Im Allgemeinen wird dieser Mechanismus als **Supertyp-Token bezeichnet** , die in Neal Gafters Blogpost detailliert beschrieben werden.

Implementierungen

Drei gängige Implementierungen in Java sind:

- Guavas TypeToken
- Die ParameterizedTypeReference von Spring
- Jacksons TypReferenz

Verwendungsbeispiel

```
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();
    }
}
```

Generics online lesen: https://riptutorial.com/de/java/topic/92/generics

Kapitel 49: Getter und Setter

Einführung

Dieser Artikel beschreibt Getter und Setter. die Standardmethode für den Zugriff auf Daten in Java-Klassen.

Examples

Getter und Setter hinzufügen

Die Kapselung ist ein grundlegendes Konzept in OOP. Es geht darum, Daten und Code als einzelne Einheit zu verpacken. In diesem Fall empfiehlt es sich, die Variablen als private zu deklarieren und dann über Getters und Setters darauf zuzugreifen, um sie anzuzeigen und / oder zu ändern.

```
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;
  }
}
```

Auf diese privaten Variablen kann nicht direkt von außerhalb der Klasse zugegriffen werden. Sie sind somit vor unbefugtem Zugriff geschützt. Wenn Sie sie jedoch anzeigen oder ändern möchten, können Sie Getter und Setter verwenden.

getXxx() -Methode gibt den aktuellen Wert der Variablen xxx , während Sie den Wert der Variablen xxx mit setXxx() .

Die Namenskonvention der Methoden lautet (in Beispielvariable heißt variableName):

- Alle nicht boolean Variablen

```
getVariableName()   //Getter, The variable name should start with uppercase
setVariableName(..) //Setter, The variable name should start with uppercase
```

- boolean Variablen

```
 isVariableName()     //Getter, The variable name should start with uppercase
 setVariableName(...) //Setter, The variable name should start with uppercase
```

Öffentliche Getter und Setter sind Teil der Property- Definition eines Java-Beans.

Verwenden eines Setter oder Getter zum Implementieren einer Einschränkung

Setter und Getter ermöglichen, dass ein Objekt private Variablen enthält, auf die mit Einschränkungen zugegriffen und diese geändert werden können. Zum Beispiel,

```
public class Person {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        if(name!=null && name.length()>2)
           this.name = name;
    }
}
```

In dieser Person Klasse gibt es eine einzige Variable: name . Auf diese Variable kann mit der Methode getName() zugegriffen und mit der Methode setName(String) geändert werden. setName(String) jedoch einen Namen setName(String) , muss der neue Name setName(String) als 2 Zeichen sein und darf nicht null sein. Eine Setter - Methode anstatt macht die variablen name Öffentlichkeit andere ermöglicht , den Wert zu setzen name mit gewissen Einschränkungen. Dasselbe kann auf die Getter-Methode angewendet werden:

```
public String getName(){
   if(name.length()>16)
      return "Name is too large!";
   else
      return name;
}
```

In der modifizierten getName() -Methode oben wird der name nur zurückgegeben, wenn seine Länge kleiner oder gleich 16 ist. Andernfalls wird "Name is too large" zurückgegeben. Dies ermöglicht dem Programmierer das Erstellen von Variablen, die beliebig erreichbar und modifizierbar sind, und verhindert, dass Client-Klassen die Variablen ungewollt bearbeiten.

Warum verwenden Sie Getter und Setter?

Betrachten Sie eine Basisklasse, die ein Objekt mit Gettern und Setters in Java enthält:

```
public class CountHolder {
  private int count = 0;

  public int getCount() { return count; }
  public void setCount(int c) { count = c; }
}
```

Wir können nicht auf die count Variable zugreifen, weil sie privat ist. Wir können jedoch auf die getCount() und setCount(int) , da sie öffentlich sind. Für einige könnte dies die Frage aufwerfen; warum den Mittelsmann vorstellen? Warum machen Sie sie nicht einfach öffentlich?

```
public class CountHolder {
  public int count = 0;
}
```

In jeder Hinsicht sind diese beiden Funktionen in Bezug auf die Funktionalität identisch. Der Unterschied zwischen ihnen ist die Erweiterbarkeit. Überlegen Sie, was jede Klasse sagt:

- **Erstens** : "Ich habe eine Methode, die Ihnen einen int Wert gibt, und eine Methode, die diesen Wert auf einen anderen int ".

- **Zweitens** : "Ich habe ein int , das Sie einstellen und erhalten können, wie Sie möchten."

Diese klingen vielleicht ähnlich, aber die erste ist in ihrer Natur viel mehr bewacht. Es lässt Sie nur mit seiner inneren Natur interagieren, während **es** diktiert. Der Ball bleibt auf seinem Platz. Es entscheidet, wie die internen Interaktionen stattfinden. Die zweite hat seine interne Implementierung von außen ausgesetzt ist , und ist nun nicht nur anfällig für externe Benutzer, aber im Fall einer API, die **mich** zu , dass die Umsetzung aufrechterhalten (oder auf andere Weise ein nicht-rückwärtskompatible API Freigabe).

Überlegen wir, ob wir den Zugriff synchronisieren möchten, um die Zählung zu ändern und darauf zuzugreifen. Im ersten ist das einfach:

```
public class CountHolder {
  private int count = 0;

  public synchronized int getCount() { return count; }
  public synchronized void setCount(int c) { count = c; }
}
```

aber im zweiten Beispiel, das ist jetzt fast unmöglich , ohne durch zu gehen und Modifizieren jeden Ort , an dem die count referenziert wird. Schlimmer noch, wenn dies ein Element, das Sie in einer Bibliothek sind die Bereitstellung von anderen verbraucht werden, müssen Sie **nicht** auf eine Art und Weise , dass die Modifikation der Durchführung, und die harte Wahl oben erwähnt machen gezwungen.

So stellt sich die Frage; sind öffentliche Variablen immer eine gute Sache (oder zumindest nicht böse)?

Ich bin mir nicht sicher Auf der einen Seite sehen Sie Beispiele für öffentliche Variablen, die sich bewährt haben (IE: die out Variable, auf die in System.out verwiesen wird). Andererseits bietet die Bereitstellung einer öffentlichen Variablen keinen Vorteil außerhalb eines extrem geringen Overheads und einer möglichen Verringerung der Wortlautstärke. Meine Leitlinie hier wäre, wenn Sie vorhaben, eine Variable öffentlich zu machen, sollten Sie sie anhand dieser Kriterien mit **extremen** Vorurteilen beurteilen:

1. Die Variable sollte keinen Grund haben, ihre Implementierung **jemals zu** ändern. Dies ist etwas, das extrem einfach zu vermasseln ist (und selbst wenn Sie es richtig machen, können sich die Anforderungen ändern). Wenn Sie über eine öffentliche Variable verfügen, muss dies wirklich durchdacht werden, insbesondere wenn sie in einer Library / Framework / API veröffentlicht wird.
2. Die Variable muss so häufig referenziert werden, dass die minimalen Gewinne durch die Reduzierung der Ausführlichkeit dies rechtfertigen. Ich denke nicht, dass der Aufwand für die Verwendung einer Methode im Vergleich zur direkten Referenzierung hier berücksichtigt werden sollte. Es ist viel zu vernachlässigbar für das, was ich zu 99,9% der Bewerbungen schätzen würde.

Wahrscheinlich gibt es mehr, als ich mir überlegt habe. Wenn Sie jemals Zweifel haben, verwenden Sie immer Getter / Setter.

Getter und Setter online lesen: https://riptutorial.com/de/java/topic/3560/getter-und-setter

Kapitel 50: Gleichzeitige Programmierung (Threads)

Einführung

Gleichzeitiges Rechnen ist eine Form des Rechnens, bei der mehrere Berechnungen gleichzeitig statt sequentiell ausgeführt werden. Die Programmiersprache Java unterstützt die gleichzeitige Programmierung durch die Verwendung von Threads. Auf Objekte und Ressourcen kann von mehreren Threads zugegriffen werden. Jeder Thread kann potenziell auf jedes Objekt im Programm zugreifen, und der Programmierer muss sicherstellen, dass der Lese- und Schreibzugriff auf Objekte ordnungsgemäß zwischen Threads synchronisiert wird.

Bemerkungen

Verwandte Themen zu StackOverflow:

- Atomtypen
- Executor-, ExecutorService- und Thread-Pools
- Erweiterung des Thread gegenüber der Implementierung von Runnable

Examples

Grundlegendes Multithreading

Wenn Sie viele Aufgaben ausführen müssen und alle diese Aufgaben nicht vom Ergebnis der vorangegangenen Aufgaben abhängig sind, können Sie **Multithreading** für Ihren Computer verwenden, um alle diese Aufgaben gleichzeitig auszuführen und mehr Prozessoren zu verwenden, sofern Ihr Computer dies kann. Dies kann die Programmausführung **beschleunigen**, wenn Sie große unabhängige Aufgaben haben.

```
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);
        }
    }
}
```

Der Code der run-Methode der verschiedenen CountAndPrint Instanzen wird in nicht vorhersagbarer Reihenfolge ausgeführt. Ein Ausschnitt einer Beispielausführung könnte folgendermaßen aussehen:

```
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
...
```

Produzent-Verbraucher

Ein einfaches Beispiel für eine Hersteller-Verbraucher-Problemlösung. Beachten Sie, dass JDK-Klassen (AtomicBoolean und BlockingQueue) für die Synchronisierung verwendet werden, wodurch die Wahrscheinlichkeit verringert wird, dass eine ungültige Lösung erstellt wird. Fragen Sie Javadoc nach verschiedenen Arten von BlockingQueue . Die Wahl einer anderen Implementierung kann das Verhalten dieses Beispiels drastisch ändern (wie DelayQueue oder 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();
    }
}
```

ThreadLocal verwenden

Ein nützliches Werkzeug in Java Concurrency ist ThreadLocal Damit können Sie eine Variable haben, die für einen bestimmten Thread eindeutig ist. Wenn also derselbe Code in verschiedenen Threads ausgeführt wird, teilen diese Ausführungen den Wert nicht, sondern jeder Thread hat seine eigene Variable, die *lokal für den Thread ist* .

Dies wird beispielsweise häufig verwendet, um den Kontext (z. B. Berechtigungsinformationen) für die Bearbeitung einer Anforderung in einem Servlet festzulegen. Sie könnten so etwas tun:

```
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
    }
}
```

Anstatt MyUserContext an jede einzelne Methode zu übergeben, können Sie MyServlet.getContext() stattdessen dort verwenden, wo Sie es benötigen. Natürlich führt dies nun eine Variable ein, die dokumentiert werden muss, aber sie ist Thread-sicher, was viele Nachteile bei der Verwendung einer solchen Variablen mit großem Umfang beseitigt.

Der entscheidende Vorteil hierbei ist, dass jeder Thread eine eigene lokale Threadvariable in diesem contexts . Solange Sie ihn von einem definierten Einstiegspunkt aus verwenden (z. B. die Anforderung, dass jedes Servlet seinen Kontext beibehält, oder indem Sie einen Servlet-Filter hinzufügen), können Sie sich darauf verlassen, dass dieser Kontext vorhanden ist, wenn Sie ihn benötigen.

CountDownLatch

CountDownLatch

> Eine Synchronisationshilfe, mit der ein oder mehrere Threads warten können, bis ein Satz von Vorgängen in anderen Threads abgeschlossen ist.

1. Ein CountDownLatch wird mit einer bestimmten Anzahl initialisiert.
2. Die wait-Methoden werden blockiert, bis der aktuelle Zählerstand null ist, weil countDown() der countDown() -Methode countDown() , wonach alle wartenden Threads freigegeben werden und alle nachfolgenden Aufrufe von await sofort zurückgegeben werden.
3. Dies ist ein einmaliges Phänomen - die Zählung kann nicht zurückgesetzt werden. Wenn Sie eine Version benötigen, die die Zählung zurücksetzt, sollten Sie einen CyclicBarrier .

Hauptmethoden:

```
public void await() throws InterruptedException
```

> Bewirkt, dass der aktuelle Thread wartet, bis der Latch auf null heruntergezählt ist, es sei denn, der Thread ist unterbrochen.

```
public void countDown()
```

> Verringert die Anzahl der Latches und gibt alle wartenden Threads frei, wenn die Anzahl null erreicht.

Beispiel:

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

Ausgabe:

```
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
```

Erläuterung:

1. CountDownLatch wird mit einem Zähler von 5 im Hauptthread initialisiert
2. Der Hauptthread wartet mit der Methode await() .
3. Fünf Instanzen von DoSomethingInAThread wurden erstellt. Bei jeder Instanz wurde der Zähler mit der countDown() Methode dekrementiert.
4. Sobald der Zähler Null wird, wird der Hauptthread fortgesetzt

Synchronisation

In Java gibt es einen eingebauten Sperrmechanismus auf Sprachebene: Der synchronized Block, der ein beliebiges Java-Objekt als intrinsische Sperre verwenden kann (dh jedem Java-Objekt kann ein Monitor zugeordnet sein).

Intrinsische Sperren verleihen Gruppen von Aussagen Atomizität. Um zu verstehen, was das für uns bedeutet, werfen wir einen Blick auf ein Beispiel, in dem eine synchronized nützlich ist:

```
private static int t = 0;
private static Object mutex = new Object();

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

```
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread
count is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            synchronized (mutex) {
                t++;
                System.out.println(MessageFormat.format("t: {0}", t));
            }
        });
    }
    executorService.shutdown();
}
```

In diesem Fall wären ohne den synchronized Block mehrere Parallelitätsprobleme aufgetreten. Der erste wäre mit dem Post-Inkrement-Operator (er ist nicht in sich atomar) und der zweite wäre, dass wir den Wert von t beobachten würden, nachdem eine beliebige Anzahl anderer Threads die Möglichkeit hatte, ihn zu ändern. Da wir jedoch eine intrinsische Sperre erworben haben, gibt es hier keine Race-Bedingungen und die Ausgabe enthält Zahlen von 1 bis 100 in ihrer normalen Reihenfolge.

Intrinsische Sperren in Java sind *Mutexe* (dh gegenseitige Ausführungssperren). Gegenseitige Ausführung bedeutet, dass, wenn ein Thread die Sperre erworben hat, der zweite gezwungen ist, auf den ersten Thread zu warten, bevor er freigegeben wird, bevor er die Sperre für sich selbst erwerben kann. Hinweis: Eine Operation, die den Thread möglicherweise in den Wartezustand versetzt, wird als *Sperroperation bezeichnet* . Daher ist der Erwerb einer Sperre eine Sperroperation.

Intrinsische Sperren in Java sind *wiedereintrittsfähig* . Das heißt, wenn ein Thread versucht, eine Sperre zu erwerben, die er bereits besitzt, wird er nicht blockiert und er wird sie erfolgreich abrufen. Beispielsweise wird der folgende Code beim Aufruf *nicht* blockiert:

```
public void bar(){
    synchronized(this){
        ...
    }
}
public void foo(){
    synchronized(this){
        bar();
    }
}
```

Neben synchronized Blöcken gibt es auch synchronized Methoden.

Die folgenden Codeblöcke sind praktisch gleichwertig (obwohl der Bytecode anders zu sein scheint):

1. synchronized Block this :

```
public void foo() {
    synchronized(this) {
        doStuff();
    }
}
```

2. synchronized Methode:

```
public synchronized void foo() {
    doStuff();
}
```

Ebenso für static Methoden:

```
class MyClass {
    ...
    public static void bar() {
        synchronized(MyClass.class) {
            doSomeOtherStuff();
        }
    }
}
```

hat den gleichen Effekt wie dieser:

```
class MyClass {
    ...
    public static synchronized void bar() {
        doSomeOtherStuff();
    }
}
```

Atomare Operationen

Eine atomare Operation ist eine Operation, die "alle gleichzeitig" ausgeführt wird, ohne dass andere Threads während der Ausführung der atomaren Operation den Zustand beobachten oder ändern können.

Betrachten **wir** ein **schlechtes Beispiel** .

```
private static int t = 0;

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread
count is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            t++;
            System.out.println(MessageFormat.format("t: {0}", t));
        });
    }
    executorService.shutdown();
}
```

In diesem Fall gibt es zwei Probleme. Das erste Problem ist, dass der Post-Inkrement-Operator *nicht* atomar ist. Es besteht aus mehreren Operationen: Holen Sie den Wert, addieren Sie 1 zum Wert, setzen Sie den Wert. Wenn wir also das Beispiel ausführen, sehen wir wahrscheinlich nicht t: 100 in der Ausgabe - zwei Threads können gleichzeitig den Wert abrufen, erhöhen und einstellen: Nehmen wir an, der Wert von t ist 10 und zwei Threads erhöhen t. Beide Threads setzen den Wert von t auf 11, da der zweite Thread den Wert von t feststellt, bevor der erste Thread das Inkrementieren beendet hat.

Die zweite Frage ist, wie wir t beobachten. Wenn wir den Wert von t drucken, wurde der Wert nach der Inkrementierungsoperation dieses Threads möglicherweise bereits von einem anderen Thread geändert.

Um diese Probleme zu beheben, verwenden wir den java.util.concurrent.atomic.AtomicInteger , der viele atomare Operationen enthält.

```
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();
}
```

Die incrementAndGet Methode von AtomicInteger inkrementiert den neuen Wert atomar und gibt diesen zurück, wodurch die vorherige AtomicInteger beseitigt wird. Bitte beachten Sie, dass in diesem Beispiel die Zeilen immer noch nicht in Ordnung sind, da wir uns nicht bemühen, die println Aufrufe zu sequenzieren. Dies fällt nicht in den Anwendungsbereich dieses Beispiels, da dies eine Synchronisation erfordern würde AtomicInteger , um Race-Bedingungen bezüglich des Zustands zu beseitigen.

Ein grundlegendes Deadlock-System erstellen

Ein Deadlock tritt auf, wenn zwei konkurrierende Aktionen darauf warten, dass die andere beendet wird. In Java ist jedem Objekt eine Sperre zugeordnet. Um gleichzeitige Änderungen zu vermeiden, die von mehreren Threads für ein einzelnes Objekt ausgeführt werden, können wir synchronized Schlüsselwörter verwenden, aber alles kostet einen Preis. Die falsche Verwendung von synchronized Schlüsselwörtern kann dazu führen, dass Systeme stecken bleiben, die als Deadlock-System bezeichnet werden.

Angenommen, es gibt 2 Threads, die in einer Instanz arbeiten, lassen Sie uns Threads als First und Second aufrufen, und sagen wir, wir haben 2 Ressourcen R1 und R2. First erwirbt R1 und benötigt außerdem R2 für die Fertigstellung, während Second R2 beschafft und R1 für die Fertigstellung benötigt.

also zum Zeitpunkt t = 0 sagen,

First hat R1 und Second hat R2. Jetzt wartet First auf R2, während Second auf R1 wartet. Dieses Warten ist unbegrenzt und führt zu einem Deadlock.

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

Ausgabe dieses Programms:

```
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
```

Ausführung unterbrechen

Thread.sleep bewirkt, dass der aktuelle Thread die Ausführung für einen bestimmten Zeitraum Thread.sleep . Dies ist ein effizientes Mittel, um den anderen Threads einer Anwendung oder

anderen Anwendungen, die auf einem Computersystem ausgeführt werden, Prozessorzeit zur Verfügung zu stellen. Die Thread-Klasse enthält zwei überladene sleep Methoden.

Eine, die die Schlafzeit in Millisekunden angibt

```
public static void sleep(long millis) throws InterruptedException
```

Eine, die die Schlafzeit in Nanosekunden angibt

```
public static void sleep(long millis, int nanos)
```

Die Ausführung wird für 1 Sekunde angehalten

```
Thread.sleep(1000);
```

Es ist wichtig zu beachten, dass dies ein Hinweis auf den Scheduler des Kernels des Betriebssystems ist. Dies muss nicht unbedingt genau sein, und einige Implementierungen berücksichtigen nicht einmal den Nanosekundenparameter (möglicherweise wird auf die nächste Millisekunde gerundet).

Es wird empfohlen, einen Aufruf von Thread.sleep in try / catch Thread.sleep und InterruptedException Thread.sleep .

Visualisierung von Lese- / Schreibbarrieren bei gleichzeitiger Verwendung von synchronisiert / flüchtig

Wir wissen, dass wir ein synchronized Schlüsselwort verwenden sollten, um die Ausführung einer Methode oder eines Blocks exklusiv zu machen. Wenigen von uns ist jedoch ein wichtiger Aspekt der Verwendung synchronized und volatile Schlüsselwörter nicht bewusst: *Abgesehen von der Bildung einer Einheit aus Code-Einheiten bietet sie auch eine Lese- / Schreibbarriere* . Was ist diese Lese- / Schreibsperre? Lassen Sie uns dies an einem Beispiel besprechen:

```
class Counter {

  private Integer count = 10;

  public synchronized void incrementCount() {
    count++;
  }

  public Integer getCount() {
    return count;
  }
}
```

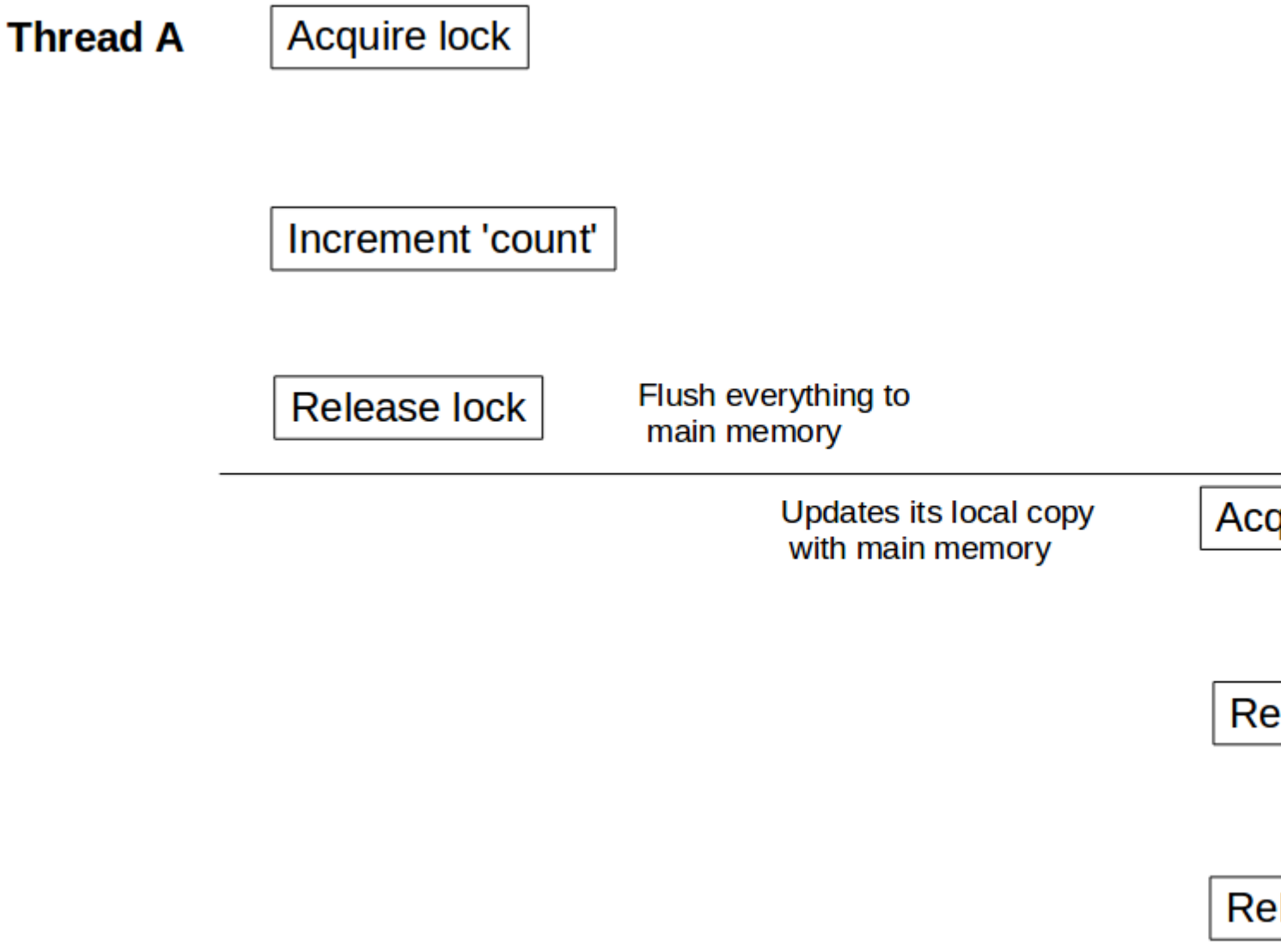
Nehmen wir an, ein Thread *A* ruft zuerst incrementCount() dann ruft ein anderer Thread *B* getCount() . In diesem Szenario gibt es keine Garantie, dass B einen aktualisierten count erhält. Es kann immer noch count als 10 , auch ist es möglich, dass nie aktualisierte Werte für die count angezeigt werden.

Um dieses Verhalten zu verstehen, müssen wir verstehen, wie das Java-Speichermodell in die Hardwarearchitektur integriert wird. In Java hat jeder Thread einen eigenen Threadstapel. Dieser Stack enthält: Method Call Stack und lokale Variable, die in diesem Thread erstellt wurden. In einem Multi-Core-System ist es durchaus möglich, dass zwei Threads gleichzeitig in separaten Cores laufen. In einem solchen Szenario ist es möglich, dass ein Teil des Stapels eines Threads im Register / Cache eines Kerns liegt. Wenn innerhalb eines Threads auf ein Objekt mit synchronized (oder volatile) Schlüsselwörtern zugegriffen wird, synchronized dieser Thread nach dem synchronized Block seine lokale Kopie dieser Variablen mit dem Hauptspeicher. Dadurch wird eine Lese- / Schreibsperre erstellt und sichergestellt, dass der Thread den neuesten Wert dieses Objekts erkennt.

Da Thread B jedoch keinen synchronisierten Zugriff für count , verweist er in diesem Fall möglicherweise auf den im Register gespeicherten Wert von count und kann niemals Aktualisierungen von Thread A getCount() synchronisiert.

```
public synchronized Integer getCount() {
  return count;
}
```

Wenn nun Faden A mit der Aktualisierung erfolgt count entriegelt es Counter Beispiel zugleich Schreibbarriere erzeugt und spült alle innerhalb dieses Blocks in den Hauptspeicher erfolgen Änderungen. Wenn Thread B eine Sperrung für dieselbe Instanz von Counter erlangt, tritt er auf ähnliche Weise in die Lesebarriere ein, liest den count aus dem Hauptspeicher und sieht alle Aktualisierungen.



Der gleiche Sichtbarkeitseffekt gilt auch für volatile Lesen / Schreiben. Alle vor dem Schreiben in volatile Variablen aktualisierten Variablen werden in den Hauptspeicher geschrieben, und alle Lesevorgänge nach dem Lesen von volatile Variablen werden aus dem Hauptspeicher stammen.

Erstellen einer java.lang.Thread-Instanz

Es gibt zwei Hauptansätze zum Erstellen eines Threads in Java. Im Grunde ist das Erstellen eines Threads so einfach wie das Schreiben des Codes, der darin ausgeführt wird. Die beiden Ansätze unterscheiden sich darin, wo Sie diesen Code definieren.

In Java wird ein Thread durch ein Objekt dargestellt - eine Instanz von java.lang.Thread oder dessen Unterklasse. Der erste Ansatz besteht darin, diese Unterklasse zu erstellen und die *run()* -Methode zu überschreiben.

Hinweis : Ich verwende *Thread* , um auf die Klasse java.lang.Thread zu verweisen, und *Thread* , um auf das logische Konzept von Threads zu verweisen.

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread running!");
        }
    }
}
```

Da wir den auszuführenden Code bereits definiert haben, kann der Thread einfach erstellt werden als:

```
MyThread t = new MyThread();
```

Die Thread- Klasse enthält auch einen Konstruktor, der eine Zeichenfolge akzeptiert, die als Name des Threads verwendet wird. Dies kann besonders hilfreich sein, wenn Sie ein Multi-Thread-Programm debuggen.

```
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");
```

Der zweite Ansatz besteht darin, den Code mit java.lang.Runnable und seiner einzigen Methode *run() zu definieren* . Die Thread- Klasse ermöglicht es Ihnen dann, diese Methode in einem separaten Thread auszuführen. Um dies zu erreichen, erstellen Sie den Thread mit einem Konstruktor, der eine Instanz der ausführbaren Schnittstelle akzeptiert.

```
Thread t = new Thread(aRunnable);
```

Dies kann sehr effektiv sein, wenn es mit Lambdas oder Methodenreferenzen kombiniert wird (nur Java 8):

```
Thread t = new Thread(operator::hardWork);
```

Sie können auch den Namen des Threads angeben.

```
Thread t = new Thread(operator::hardWork, "Pi operator");
```

Praktisch können Sie beide Ansätze ohne Sorgen verwenden. Die allgemeine Weisheit sagt jedoch, letzteres zu verwenden.

Für jeden der vier genannten Konstruktoren gibt es auch eine Alternative, die eine Instanz von java.lang.ThreadGroup als ersten Parameter akzeptiert.

```
ThreadGroup tg = new ThreadGroup("Operators");
Thread t = new Thread(tg, operator::hardWork, "PI operator");
```

Die ThreadGroup repräsentiert eine Gruppe von Threads. Sie können nur hinzufügen Thema zu einem Thread mit einem Thema ,s - Konstruktor. Die ThreadGroup kann dann verwendet werden, um alle Threads zusammen zu verwalten, und der Thread kann Informationen von seiner ThreadGroup erhalten .

Um dies zu sumarisieren, kann der Thread mit einem dieser öffentlichen Konstruktoren erstellt werden:

```
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)
```

Mit dem letzten können wir die gewünschte Stapelgröße für den neuen Thread definieren.

Häufig leidet die Lesbarkeit des Codes, wenn viele Threads mit denselben Eigenschaften oder mit demselben Muster erstellt und konfiguriert werden. Dann kann java.util.concurrent.ThreadFactory verwendet werden. Mit dieser Schnittstelle können Sie die Prozedur zum Erstellen des Threads durch das Factory-Pattern und die einzige Methode *newThread (Runnable) kapseln* .

```
class WorkerFactory implements ThreadFactory {
    private int id = 0;

    @Override
    public Thread newThread(Runnable r) {
        return new Thread(r, "Worker " + id++);
    }
}
```

Thread-Unterbrechung / Stoppen von Threads

Jeder Java-Thread hat ein Interrupt-Flag, das anfangs falsch ist. Das Unterbrechen eines Threads ist im Wesentlichen nichts anderes als das Setzen des Flags auf true. Der Code, der in diesem Thread ausgeführt wird, kann die Flagge gelegentlich überprüfen und darauf reagieren. Der Code kann ihn auch komplett ignorieren. Aber warum sollte jeder Thread eine solche Flagge haben? Eine boolesche Fahne in einem Thread zu haben, ist etwas, was wir uns selbst organisieren können, wenn wir es brauchen. Nun, es gibt Methoden, die sich auf besondere Weise verhalten, wenn der Thread, auf dem sie laufen, unterbrochen wird. Diese Methoden werden Sperrmethoden genannt. Dies sind Methoden, die den Thread in den Status WAITING oder TIMED_WAITING setzen. Wenn sich ein Thread in diesem Zustand befindet, wird bei Unterbrechung eine InterruptedException für den unterbrochenen Thread ausgelöst, anstatt dass das Interrupt-Flag auf true gesetzt ist, und der Thread wird erneut zu RUNNABLE. Code, der eine Blockierungsmethode aufruft, muss die InterruptedException behandeln, da es sich um eine geprüfte Ausnahme handelt. Ein Interrupt kann also und damit auch der Name eine WAIT unterbrechen und effektiv beenden. Beachten Sie, dass nicht alle Methoden, die irgendwie warten (z. B. das Blockieren von E / A), auf diese Weise auf eine Unterbrechung reagieren, da sie den Thread nicht in einen Wartezustand versetzen. Ein Thread, dessen Interrupt-Flag gesetzt ist und in eine Sperrmethode eintritt (dh versucht, in einen Wartezustand zu gelangen), löst sofort eine InterruptedException aus und das Interrupt-Flag wird gelöscht.

Mit Ausnahme dieser Mechanismen ordnet Java der Unterbrechung keine besondere semantische Bedeutung zu. Der Code kann einen Interrupt beliebig interpretieren. In den meisten Fällen wird eine Unterbrechung verwendet, um einem Thread zu signalisieren, dass er so schnell wie möglich nicht mehr läuft. Wie sich jedoch aus dem Vorstehenden ergibt, ist es Sache des Codes in diesem

Thread, auf diese Unterbrechung entsprechend zu reagieren, um die Ausführung zu stoppen. Das Stoppen eines Threads ist eine Kollaboration. Wenn ein Thread unterbrochen wird, kann sein laufender Code mehrere Ebenen tief im Stacktrace liegen. Die meisten Codes rufen keine Blockierungsmethode auf und werden rechtzeitig beendet, um das Stoppen des Threads nicht unnötig zu verzögern. Der Code, der hauptsächlich darauf angesprochen werden soll, auf eine Unterbrechung zu reagieren, ist Code, der sich in einer Schleife befindet, in der Aufgaben ausgeführt werden, bis keine mehr vorhanden ist oder bis ein Flag gesetzt ist, das signalisiert, dass die Schleife angehalten wird. Schleifen, die möglicherweise unendliche Aufgaben bearbeiten (dh sie laufen grundsätzlich weiter), sollten das Interrupt-Flag prüfen, um die Schleife zu verlassen. Bei endlichen Schleifen kann die Semantik vorschreiben, dass alle Aufgaben vor dem Beenden abgeschlossen werden müssen, oder es kann angebracht sein, einige Aufgaben nicht bearbeitet zu lassen. Code, der blockierende Methoden aufruft, wird gezwungen, mit der InterruptedException umzugehen. Wenn überhaupt semantisch möglich, kann die InterruptedException einfach propagiert und zum Auslösen deklariert werden. Als solches wird es selbst gegenüber seinen Anrufern zu einer Sperrmethode. Wenn es die Ausnahme nicht weitergeben kann, sollte es zumindest das unterbrochene Flag setzen, sodass Anrufer oberhalb des Stacks auch wissen, dass der Thread unterbrochen wurde. In einigen Fällen muss die Methode unabhängig von der InterruptedException weiter warten. In diesem Fall muss das Setzen des unterbrochenen Flags so lange verzögert werden, bis das Warten beendet ist. Dazu muss möglicherweise eine lokale Variable festgelegt werden, die vor dem Verlassen der Methode überprüft werden muss Dann unterbrechen Sie den Thread.

Beispiele:

Beispiel für Code, der die Bearbeitung von Aufgaben nach einer Unterbrechung stoppt

```
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
    }
}
```

Beispiel für Code, der das Setzen des Interrupt-Flags verzögert, bis der Vorgang abgeschlossen ist:

```
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
    }
}
```

Beispiel für Code, der eine feste Liste von Aufgaben enthält, bei Unterbrechung jedoch vorzeitig beendet wird

```
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
    }
}
```

Beispiel für mehrere Produzenten / Konsumenten mit gemeinsam genutzter globaler Warteschlange

Der folgende Code zeigt mehrere Producer / Consumer-Programme. Sowohl Producer- als auch Consumer-Threads verwenden dieselbe globale Warteschlange.

```
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();
            }
        }
```

```
    }
}
```

Ausgabe:

```
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
```

und so weiter ....

Erläuterung:

1. sharedQueue , ein LinkedBlockingQueue wird von allen Producer- und Consumer-Threads gemeinsam genutzt.
2. Produzententhreads erzeugen alle 200 sharedQueue fortlaufend eine ganze Zahl und sharedQueue sie an sharedQueue
3. Consumer Thread verbraucht fortlaufend Ganzzahlen aus sharedQueue .
4. Dieses Programm wird ohne explizite synchronized oder Lock Konstrukte implementiert. BlockingQueue ist der Schlüssel, um dies zu erreichen.

   BlockingQueue-Implementierungen sind hauptsächlich für die Verwendung in Warteschlangen von Produzenten und Verbrauchern vorgesehen.

   BlockingQueue-Implementierungen sind Thread-sicher. Alle Warteschlangenmethoden erzielen ihre Auswirkungen atomar durch interne Sperren oder andere Formen der Parallelitätssteuerung.

Exklusiver Schreib- / gleichzeitiger Lesezugriff

Es ist manchmal erforderlich, dass ein Prozess die gleichen "Daten" gleichzeitig schreibt und liest.

Die ReadWriteLock Schnittstelle und ihre ReentrantReadWriteLock Implementierung ermöglicht ein Zugriffsmuster, das wie folgt beschrieben werden kann:

1. Es können beliebig viele gleichzeitige Leser der Daten vorhanden sein. Wenn mindestens ein Leserzugriff gewährt wird, ist kein Schreibzugriff möglich.
2. Es kann höchstens einen einzigen Schreiber für die Daten geben. Wenn ein Schreiberzugriff gewährt wird, kann kein Leser auf die Daten zugreifen.

Eine Implementierung könnte folgendermaßen aussehen:

```
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();
    }
}

}
```

ANMERKUNG 1 : Dieser genaue Anwendungsfall hat eine sauberere Lösung, die AtomicInteger hier beschriebene Zugriffsmuster funktioniert jedoch unabhängig von der Tatsache, dass Daten hier eine Ganzzahl sind, die als atomische Variante gilt.

ANMERKUNG 2 : Die Sperre für den Leseteil ist wirklich erforderlich, obwohl sie für den Gelegenheitsleser nicht so aussieht. Wenn Sie sich nicht auf der Leserseite verriegeln, kann eine beliebige Anzahl von Dingen schief gehen, unter anderem:

1. Es ist nicht garantiert, dass die Schreibvorgänge der Grundwerte auf allen JVMs atomar sind, sodass der Leser z. B. nur 32 Bit eines 64-Bit-Schreibvorgangs sehen könnte, wenn data einen 64-Bit-langen Typ haben
2. Die Sichtbarkeit des Schreibvorgangs von einem Thread, der ihn nicht ausgeführt hat, wird von der JVM nur garantiert, wenn die *Beziehung Vor* dem Schreibvorgang und dem Lesevorgang festgelegt wird. Diese Beziehung wird hergestellt, wenn sowohl Leser als auch Schreiber ihre jeweiligen Sperren verwenden, jedoch nicht anders

Java SE 8

Falls eine höhere Leistung erforderlich ist, ist unter bestimmten Nutzungsarten ein schnellerer StampedLock verfügbar, der als StampedLock wird, der unter anderem einen optimistischen StampedLock implementiert. Diese Sperre unterscheidet sich stark von ReadWriteLock , und dieses Beispiel ist nicht transponierbar.

Lauffähiges Objekt

Die Runnable Schnittstelle definiert eine einzige Methode, run() , die den im Thread ausgeführten Code enthalten soll.

Das Runnable Objekt wird an den Thread Konstruktor übergeben. Und die start() -Methode von Thread wird aufgerufen.

Beispiel

```
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();
    }
}
```

Beispiel in Java8:

```
public static void main(String[] args) {
    Runnable r = () -> System.out.println("Hello world");
    new Thread(r).start();
}
```

Runable vs Thread-Unterklasse

Die Verwendung eines Runnable Objekts ist allgemeiner, da das Runnable Objekt eine andere Klasse als Thread Runnable kann.

Thread Unterklassen sind einfacher in einfachen Anwendungen zu verwenden, sind jedoch durch die Tatsache eingeschränkt, dass Ihre Task-Klasse von Thread .

Ein Runnable Objekt kann auf die Runnable Thread-Verwaltungs-APIs Runnable werden.

Semaphor

Ein Semaphor ist ein übergeordneter Synchronisierer, der eine Reihe von *Genehmigungen enthält* , die von Threads erworben und freigegeben werden können. Ein Semaphor kann man sich als Zähler für *Genehmigungen* vorstellen, der beim Erwerb eines Threads dekrementiert und beim Freigeben eines Threads inkrementiert wird. Wenn die Anzahl der *Genehmigungen* 0 wenn ein Thread versucht, zuzugreifen, wird der Thread blockiert, bis eine Genehmigung verfügbar gemacht wird (oder bis der Thread unterbrochen wird).

Ein Semaphor wird wie folgt initialisiert:

Semaphore semaphore = new Semaphore(1); // The int value being the number of permits

Der Semaphor-Konstruktor akzeptiert einen zusätzlichen booleschen Parameter für Fairness. Wenn false festgelegt ist, gibt diese Klasse keine Garantie für die Reihenfolge ab, in der Threads Genehmigungen erhalten. Wenn Fairness auf true gesetzt ist, garantiert das Semaphor, dass Threads, die eine der Erfassungsmethoden aufrufen, ausgewählt werden, um Genehmigungen in der Reihenfolge zu erhalten, in der ihr Aufruf dieser Methoden verarbeitet wurde. Es wird auf folgende Weise erklärt:

Semaphore semaphore = new Semaphore(1, true);

Betrachten wir nun ein Beispiel aus Javadocs, in dem Semaphore verwendet wird, um den Zugriff auf einen Pool von Elementen zu steuern. In diesem Beispiel wird ein Semaphor verwendet, um Blockierungsfunktionen bereitzustellen, um sicherzustellen, dass beim getItem() immer Elemente abgerufen werden.

```
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
    }
}
```

Fügen Sie mit einem Threadpool zwei `int`-Arrays hinzu

Ein Threadpool hat eine Warteschlange mit Tasks, von denen jede auf einem dieser Threads ausgeführt wird.

Das folgende Beispiel zeigt, wie Sie mithilfe eines Threadpools zwei int Arrays hinzufügen.

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));
```

Anmerkungen:

1. Dieses Beispiel ist rein illustrativ. In der Praxis wird es keine Beschleunigung geben, wenn Sie Threads für eine so kleine Aufgabe verwenden. Eine Verlangsamung ist wahrscheinlich, da der Aufwand für die Erstellung und Zeitplanung von Aufgaben die für die Ausführung einer Aufgabe benötigte Zeit überschreitet.

2. Wenn Sie Java 7 oder eine frühere Version verwendet haben, würden Sie anstelle von Lambdas anonyme Klassen verwenden, um die Aufgaben zu implementieren.

Status aller Threads abrufen, die von Ihrem Programm gestartet wurden, ausgenommen System-Threads

Code-Auszug:

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

Ausgabe:

```
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
```

Erläuterung:

Thread.getAllStackTraces().keySet() gibt alle Thread einschließlich Anwendungsthreads und Systemthreads zurück. Wenn Sie nur am Status der Threads interessiert sind, die von Ihrer Anwendung gestartet wurden, iterieren Sie den Thread Satz, indem Sie die Thread-Gruppe eines bestimmten Threads mit Ihrem Hauptprogramm- Thread überprüfen.

Ohne obige ThreadGroup-Bedingung gibt das Programm den Status unter System Threads zurück:

```
Reference Handler
Signal Dispatcher
Attach Listener
Finalizer
```

Callable und Future

Runnable bietet zwar die Möglichkeit, Code in einem anderen Thread auszuführen, hat jedoch die Einschränkung, dass er kein Ergebnis aus der Ausführung zurückgeben kann. Die einzige Möglichkeit, einen Rückgabewert aus der Ausführung eines Runnable besteht darin, das Ergebnis einer Variablen zuzuweisen, auf die in einem Bereich außerhalb des Runnable .

Callable wurde in Java 5 als Peer to Runnable . Callable ist im Wesentlichen der gleiche , außer es einen hat call - Methode statt run . Die call hat die zusätzliche Fähigkeit, ein Ergebnis zurückzugeben, und darf auch geprüfte Ausnahmen auslösen.

Das Ergebnis einer *Callable-Aufgabe* kann über eine Zukunft abgerufen werden

Future kann als ein Container von Sorten betrachtet werden, der das Ergebnis der Callable Berechnung enthält. Die Berechnung der aufrufbaren Daten kann in einem anderen Thread fortgesetzt werden. Jeder Versuch, das Ergebnis einer Future wird blockiert und liefert das Ergebnis nur, wenn es verfügbar ist.

Aufrufbare Schnittstelle

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Zukunft

```
interface Future<V> {
    V get();
    V get(long timeout, TimeUnit unit);
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
}
```

Callable und Future Beispiel verwenden:

```
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());
}
```

Unser Callable führt eine langwierige Berechnung aus

```
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
    }
}
```

Ausgabe

```
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
```

Andere Operationen sind für Future zulässig

Während get() die Methode ist, um das tatsächliche Ergebnis zu extrahieren, hat Future eine Provision

- get(long timeout, TimeUnit unit) definiert die maximale Zeitdauer, während der der aktuelle Thread auf ein Ergebnis wartet.
- So cancel(mayInterruptIfRunning) den Task-Aufruf ab cancel(mayInterruptIfRunning) . Das Flag mayInterrupt zeigt an, dass die Task unterbrochen werden sollte, wenn sie gerade gestartet wurde und gerade ausgeführt wird.
- Um zu überprüfen, ob die Aufgabe abgeschlossen / abgeschlossen ist, rufen Sie isDone() .
- Um zu überprüfen, ob die langwierige Aufgabe abgebrochen wurde, isCancelled() .

Sperren als Synchronisationshilfen

Vor der Einführung von Java 5 für gleichzeitige Pakete war das Threading eher untergeordnet. Bei der Einführung dieses Pakets wurden mehrere Programmierhilfen / Konstrukte auf höherer Ebene bereitgestellt.

Sperren sind Thread-Synchronisationsmechanismen, die im Wesentlichen dem gleichen Zweck wie synchronisierte Blöcke oder Schlüsselwörter dienen.

Intrinsic Locking

```
int count = 0; // shared among multiple threads

public void doSomething() {
    synchronized(this) {
        ++count; // a non-atomic operation
    }
}
```

Synchronisation mit Sperren

```
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
    }
}
```

Für Sperren stehen auch Funktionen zur Verfügung, die das intrinsische Sperren nicht bietet, z. B. Sperren, die jedoch auf Unterbrechungen ansprechen oder versuchen zu sperren.

Sperrung, reagiert auf Unterbrechung

```
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
            }
        }
    }
}
```

Machen Sie nur etwas, wenn Sie sperren können

```
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
            }
        }
    }
}
```

Es gibt mehrere Varianten von Schloss available.For mehr Details um die API - Dokumentation finden Sie hier

Gleichzeitige Programmierung (Threads) online lesen:
https://riptutorial.com/de/java/topic/121/gleichzeitige-programmierung--threads-

Kapitel 51: Gleichzeitige Sammlungen

Einführung

Eine *gleichzeitige Auflistung* ist eine [Auflistung] [1], die den gleichzeitigen Zugriff mehrerer Threads ermöglicht. Verschiedene Threads können normalerweise den Inhalt der Auflistung durchlaufen und Elemente hinzufügen oder entfernen. Die Sammlung ist dafür verantwortlich, dass die Sammlung nicht beschädigt wird. [1]:
http://stackoverflow.com/documentation/java/90/collections#t=20161222193649729848 4

Examples

Fadensichere Sammlungen

Standardmäßig sind die verschiedenen Collection-Typen nicht threadsicher.

Es ist jedoch ziemlich einfach, eine Sammlung threadsicher zu machen.

```
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>());
```

Wenn Sie eine threadsichere Sammlung erstellen, sollten Sie niemals über die ursprüngliche Sammlung, sondern nur über den fadensicheren Wrapper darauf zugreifen.

Java SE 5

Ab Java 5 gibt es in java.util.collections mehrere neue thread-sichere Sammlungen, die die verschiedenen Collections.synchronized Methoden nicht benötigen.

```
List<String> threadSafeList = new CopyOnWriteArrayList<String>();
Set<String> threadSafeSet = new ConcurrentHashSet<String>();
Map<String, String> threadSafeMap = new ConcurrentHashMap<String, String>();
```

Gleichzeitige Sammlungen

Gleichzeitige Auflistungen sind eine Verallgemeinerung von Thread-sicheren Auflistungen, die eine breitere Verwendung in einer gleichzeitigen Umgebung ermöglichen.

Während für thread-sichere Auflistungen sichere Elemente hinzugefügt oder aus mehreren Threads entfernt werden, ist für sie nicht unbedingt eine sichere Iteration im selben Kontext erforderlich (eine Collection kann möglicherweise nicht durch die Collection in einem Thread sicher durchlaufen werden, während eine andere sie durch Hinzufügen von / ändert). Elemente entfernen).

Hier werden gleichzeitige Sammlungen verwendet.

Da die Iteration häufig die addAll mehrerer Massenmethoden in Sammlungen ist, wie addAll , removeAll oder auch das Kopieren von Sammlungen (durch einen Konstruktor oder auf andere Weise), Sortieren, ... ist der Anwendungsfall für gleichzeitige Sammlungen tatsächlich ziemlich groß.

Die Java SE 5 java.util.concurrent.CopyOnWriteArrayList ist beispielsweise eine Thread-sichere und gleichzeitige Lis t-Implementierung. In ihrem Javadoc heißt es:

> Die Iterator-Methode "Snapshot" verwendet eine Referenz auf den Status des Arrays an dem Punkt, an dem der Iterator erstellt wurde. Dieses Array ändert sich während der Lebensdauer des Iterators nie. Daher sind Interferenzen nicht möglich, und es wird garantiert, dass der Iterator keine ConcurrentModificationException auslöst.

Daher ist der folgende Code sicher:

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

Eine weitere gleichzeitige Auflistung bezüglich der Iteration ist ConcurrentLinkedQueue , die besagt:

> Iteratoren sind schwach konsistent und geben Elemente zurück, die den Status der Warteschlange zu einem bestimmten Zeitpunkt bei der Erstellung des Iterators widerspiegeln. Sie lösen keine java.util.ConcurrentModificationException aus und können gleichzeitig mit anderen Vorgängen fortfahren. Elemente, die seit der Erstellung des Iterators in der Warteschlange enthalten sind, werden genau einmal zurückgegeben.

Man sollte die Javadocs überprüfen, um zu sehen, ob eine Sammlung gleichzeitig ist oder nicht. Die Attribute des Iterators, die von der iterator() -Methode zurückgegeben werden ("Fail Fast", "schwach konsistent", ...), sind das wichtigste Attribut, nach dem gesucht werden soll.

Thread-sichere, aber nicht gleichzeitige Beispiele

Ändern Sie im obigen Code die LIST Deklaration in

```
public static final List<Integer> LIST = Collections.synchronizedList(new ArrayList<>());
```

Kann (und wird statistisch gesehen auf den meisten modernen Multi-CPU / Core-Architekturen) zu Ausnahmen führen.

Synchronisierte Auflistungen aus den Collections Utility-Methoden sind threadsicher für das Hinzufügen / Entfernen von Elementen, nicht jedoch für die Iteration (es sei denn, die zugrunde liegende Auflistung ist bereits übergeben).

Einfügen in ConcurrentHashMap

```
public class InsertIntoConcurrentHashMap
{

    public static void main(String[] args)
    {
        ConcurrentHashMap<Integer, SomeObject> concurrentHashMap = new ConcurrentHashMap<>();

        SomeObject value = new SomeObject();
        Integer key = 1;

        SomeObject previousValue = concurrentHashMap.putIfAbsent(1, value);
        if (previousValue != null)
        {
            //Then some other value was mapped to key = 1. 'value' that was passed to
            //putIfAbsent method is NOT inserted, hence, any other thread which calls
            //concurrentHashMap.get(1) would NOT receive a reference to the 'value'
            //that your thread attempted to insert. Decide how you wish to handle
            //this situation.
        }

       else
       {
            //'value' reference is mapped to key = 1.
       }
    }
}
```

Gleichzeitige Sammlungen online lesen: https://riptutorial.com/de/java/topic/8363/gleichzeitige-sammlungen

Kapitel 52: Grundlegende Kontrollstrukturen

Bemerkungen

Alle Kontrollstrukturen verwenden, sofern nicht anders angegeben, Blockanweisungen . Diese sind durch geschweifte Klammern {} .

Dies unterscheidet sich von normalen Aussagen , die geschweiften Klammern *nicht* erforderlich ist , sondern auch mit einem steifen Vorbehalt kommen , dass nur die Zeile *unmittelbar nach* der vorangegangenen Aussage betrachtet werden würde.

Daher ist es absolut zulässig, eine dieser Kontrollstrukturen ohne geschweifte Klammern zu schreiben, solange nur *eine* Anweisung dem Anfang folgt. Es wird jedoch ***dringend davon abgeraten*** , da dies zu fehlerhaften Implementierungen oder fehlerhaftem Code führen kann.

Beispiel:

```
// 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

If / Else If / Else Kontrolle

```
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");
}
```

Der if Block wird nur ausgeführt, wenn i 1 oder weniger ist.

Die else if Bedingung wird nur geprüft, wenn alle Bedingungen davor (in den vorherigen else if Konstrukten und die übergeordneten if Konstrukte) auf false getestet wurden. In diesem Beispiel wird die Bedingung else if nur geprüft, wenn i größer oder gleich 2 ist.

Wenn das Ergebnis true , wird der Block ausgeführt, und alle else if und else Konstrukte, die danach erstellt werden, werden übersprungen.

Wenn keiner der if und else if , true else else if Bedingungen zu getestet true , die else wird Block am Ende ausgeführt werden.

Für Loops

```
for (int i = 0; i < 100; i++) {
    System.out.println(i);
```

```
}
```

Die drei Komponenten der for Schleife (getrennt durch ;) sind Variablendeklaration / Initialisierung (hier int i = 0), die Bedingung (hier i < 100) und die Inkrement-Anweisung (hier i++). Die Variablendeklaration wird einmal ausgeführt, als würde sie beim ersten Durchlauf direkt innerhalb von { platziert. Dann wird die Bedingung überprüft, wenn sie true ist true der Hauptteil der Schleife ausgeführt. Wenn sie false ist, stoppt die Schleife. Unter der Annahme, dass die Schleife fortgesetzt wird, wird der Körper ausgeführt, und wenn } erreicht wird, wird die Inkrementierungsanweisung ausgeführt, bevor die Bedingung erneut geprüft wird.

Die geschweiften Klammern sind optional (Sie können eine Zeile mit einem Semikolon verwenden), wenn die Schleife nur eine Anweisung enthält. Es wird jedoch immer empfohlen, Zahnspangen zu verwenden, um Missverständnisse und Fehler zu vermeiden.

Die for Loop-Komponenten sind optional. Wenn Ihre Geschäftslogik einen dieser Teile enthält, können Sie die entsprechende Komponente in Ihrer for Schleife auslassen.

```
int i = obj.getLastestValue(); // i value is fetched from a method

for (; i < 100; i++) { // here initialization is not done
    System.out.println(i);
}
```

Die for (;;) { function-body } -Struktur entspricht einer while (true) -Schleife.

Nested For Loops

Jede Schleifenanweisung mit einer anderen Schleifenanweisung innerhalb einer geschachtelten Schleife. Der gleiche Weg für das Looping mit mehr innerer Schleife wird als "verschachtelte for-Schleife" bezeichnet.

```
    for(;;){
        //Outer Loop Statements
        for(;;){
            //Inner Loop Statements
        }
        //Outer Loop Statements
    }
```

Verschachtelte for-Schleife kann demonstriert werden, um dreieckige Zahlen zu drucken.

```
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);
    }
 }
```

Während Schleifen

```
int i = 0;
while (i < 100) { // condition gets checked BEFORE the loop body executes
    System.out.println(i);
    i++;
}
```

A while Schleife läuft solange die Bedingung innerhalb der Klammern ist true . Dies wird auch

als "Pre-Test-Loop" -Struktur bezeichnet, da die Bedingungsanweisung erfüllt werden muss, bevor der Hauptschleifenkörper jedes Mal ausgeführt wird.

Die geschweiften Klammern sind optional, wenn die Schleife nur eine Anweisung enthält. Einige Codierungsstilkonventionen bevorzugen jedoch die geschweiften Klammern.

mache ... während der Schleife

Die do...while Schleife unterscheidet sich von anderen Schleifen dadurch, dass sie **mindestens einmal ausgeführt wird** . Sie wird auch als "Post-Test-Loop" -Struktur bezeichnet, da die Bedingungsanweisung nach dem Hauptschleifenrumpf ausgeführt wird.

```
int i = 0;
do {
    i++;
    System.out.println(i);
} while (i < 100); // Condition gets checked AFTER the content of the loop executes.
```

In diesem Beispiel wird die Schleife ausgeführt, bis die Zahl 100 gedruckt ist (obwohl die Bedingung i < 100 und nicht i <= 100), da die Schleifenbedingung *nach* Ausführung der Schleife ausgewertet wird.

Mit der Garantie von mindestens einer Ausführung ist es möglich, Variablen außerhalb der Schleife zu deklarieren und sie innerhalb zu initialisieren.

```
String theWord;
Scanner scan = new Scanner(System.in);
do {
    theWord = scan.nextLine();
} while (!theWord.equals("Bird"));

System.out.println(theWord);
```

In diesem Zusammenhang wird theWord außerhalb der Schleife definiert. Da es jedoch garantiert ist, dass es einen Wert auf der Grundlage seines natürlichen Flusses hat, wird das theWord initialisiert.

Für jeden

Java SE 5

Mit Java 5 und höher können For-Each-Loops, auch als erweiterte For-Loops bezeichnet, verwendet werden:

```
List strings = new ArrayList();

strings.add("This");
strings.add("is");
strings.add("a for-each loop");

for (String string : strings) {
    System.out.println(string);
}
```

Für jede Schleife können Arrays und Implementierungen der Iterable Schnittstelle Iterable werden, die spätere enthält Collections- Klassen wie List oder Set .

Die Schleifenvariable kann von einem beliebigen Typ sein, der vom Quelltyp zuweisbar ist.

Die Schleifenvariable für eine erweiterte for-Schleife für Iterable<T> oder T[] kann vom Typ S ,

wenn

- T extends S
- Sowohl T als auch S sind primitive Typen und können ohne Besetzung zugewiesen werden
- S ist ein primitiver Typ, und T kann nach der Unboxing-Konvertierung in einen Typ konvertiert werden, der S .
- T ist ein primitiver Typ und kann durch automatische Konvertierung in S umgewandelt werden.

Beispiele:

```
T elements = ...
for (S s : elements) {
}
```

T	S	Kompiliert
int []	lange	Ja
lange[]	int	Nein
Iterable<Byte>	lange	Ja
Iterable<String>	Zeichenfolge	Ja
Iterable<CharSequence>	String	Nein
int []	Lange	Nein
int []	Ganze Zahl	Ja

Ansonsten

```
int i = 2;
if (i < 2) {
  System.out.println("i is less than 2");
} else {
  System.out.println("i is greater than 2");
}
```

Eine if Anweisung führt den Code abhängig vom Ergebnis der Bedingung in Klammern aus. Wenn die Bedingung in Klammern wahr ist, wird der Block der if –Anweisung eingegeben, der durch geschweifte Klammern wie { und } . Das Öffnen der Klammer bis zur schließenden Klammer ist der Geltungsbereich der if-Anweisung.

Der else Block ist optional und kann weggelassen werden. Sie wird ausgeführt, wenn die if Anweisung false und nicht, wenn die if Anweisung true ist. In diesem Fall wird die if Anweisung ausgeführt.

Siehe auch: Ternary If

Anweisung wechseln

Die switch Anweisung ist die Mehrfachzweig-Anweisung von Java. Es wird verwendet, um lange if - else if - else Ketten zu ersetzen und sie lesbarer zu machen. Im Gegensatz zu if Anweisungen darf man jedoch keine Ungleichungen verwenden. Jeder Wert muss konkret definiert werden.

Die switch Anweisung besteht aus drei wichtigen Komponenten:

- case : Dies ist der Wert, der auf Äquivalenz mit dem Argument der switch Anweisung ausgewertet wird.

- default : Dies ist ein optionaler Ausdruck für alle case , falls keine der case Anweisungen den Wert true ergibt.
- Abrupte Beendigung der case - Anweisung; in der Regel break : Dies ist erforderlich, um die unerwünschte Auswertung weiterer case Anweisungen zu verhindern.

Mit Ausnahme von continue kann eine beliebige Anweisung verwendet werden, die den abrupten Abschluss einer Anweisung verursachen würde . Das beinhaltet:

- break
  - return
  - throw

In dem folgenden Beispiel wird eine typische switch Anweisung mit vier möglichen Fällen einschließlich 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");
}
```

Durch das Weglassen von break oder einer Anweisung, die zu einem abrupten Abschluss führen würde, können wir sogenannte "Fall-Through" -Fälle nutzen, die verschiedene Werte auswerten. Dies kann zum Erstellen von Bereichen verwendet werden, für die ein Wert erfolgreich ist, ist aber noch nicht so flexibel wie Ungleichungen.

```
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");
}
```

Im Falle von foo == 1 die Ausgabe:

```
I'm equal or greater than one
I'm one, two, or three
```

Im Falle von foo == 3 die Ausgabe:

```
I'm one, two, or three
```

Java SE 5

Die switch-Anweisung kann auch mit enum s verwendet werden.

```
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

Die switch Anweisung kann auch mit String s verwendet werden.

```
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");
    }
}
```

Ternärer Betreiber

Manchmal müssen Sie nach einer Bedingung suchen und den Wert einer Variablen festlegen.

Für ex.

```
String name;

if (A > B) {
    name = "Billy";
} else {
    name = "Jimmy";
}
```

Dies kann leicht in eine Zeile geschrieben werden als

```
String name = A > B ? "Billy" : "Jimmy";
```

Der Wert der Variablen wird unmittelbar nach der Bedingung auf den Wert gesetzt, wenn die Bedingung erfüllt ist. Wenn die Bedingung falsch ist, wird der Variable der zweite Wert zugewiesen.

Brechen

Die break Anweisung beendet eine Schleife (wie for , while) oder die Auswertung einer switch-

Anweisung .

Schleife:

```
while(true) {
   if(someCondition == 5) {
       break;
   }
}
```

Die Schleife im Beispiel würde für immer laufen. Wenn jedoch someCondition an einem Punkt der Ausführung gleich 5 , endet die Schleife.

Wenn mehrere Schleifen kaskadiert sind, endet nur die innerste Schleife mit break .

Versuchen Sie ... Fang ... Endlich

Die Kontrollstruktur try { ... } catch (...) { ... } wird für die Behandlung von Ausnahmen verwendet .

```
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.");
}
```

Dies würde drucken:

```
Ungültige Eingabe. 'abc' ist keine gültige ganze Zahl.
```

Eine finally Klausel kann nach dem hinzugefügt werden , catch . Die finally Klausel wird immer ausgeführt, unabhängig davon, ob eine Ausnahme ausgelöst wurde.

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");
}
```

Dies würde drucken:

```
Ungültige Eingabe. 'abc' ist keine gültige ganze Zahl.
Dieser Code wird immer ausgeführt, auch wenn eine Ausnahme ausgelöst wird
```

Verschachtelte Pause / weiter

Es ist möglich, eine äußere Schleife mit den Label-Anweisungen zu break / continue :

```
outerloop:
for(...) {
    innerloop:
    for(...) {
        if(condition1)
            break outerloop;

        if(condition2)
            continue innerloop; // equivalent to: continue;
    }
}
```

Es gibt keine andere Verwendung für Etiketten in Java.

Continue-Anweisung in Java

Die continue-Anweisung wird verwendet, um die verbleibenden Schritte in der aktuellen Iteration zu überspringen und mit der nächsten Schleifeniteration zu beginnen. Die Steuerung wechselt von der continue Anweisung zum Schrittwert (Inkrement oder Dekrement), falls vorhanden.

```
String[] programmers = {"Adrian", "Paul", "John", "Harry"};

    //john is not printed out
    for (String name : programmers) {
        if (name.equals("John"))
            continue;
        System.out.println(name);
    }
```

Die continue Anweisung kann auch die Steuerung des Programms zum Schrittwert (falls vorhanden) einer benannten Schleife verschieben:

```
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;
    }
}
```

Grundlegende Kontrollstrukturen online lesen:
https://riptutorial.com/de/java/topic/118/grundlegende-kontrollstrukturen

Kapitel 53: Hash-tabelle

Einführung

Hashtable ist eine Klasse in Java-Sammlungen, die die Map-Schnittstelle implementiert und die Dictionary-Klasse erweitert

Enthält nur eindeutige Elemente und deren synchronisierte Elemente

Examples

Hash-tabelle

```
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);
   }
}
```

Hash-tabelle online lesen: https://riptutorial.com/de/java/topic/10709/hash-tabelle

Kapitel 54: Häufige Java-Fallstricke

Einführung

In diesem Thema werden einige der häufigsten Fehler beschrieben, die von Anfängern in Java gemacht wurden.

Dies schließt etwaige häufige Fehler bei der Verwendung der Java-Sprache oder das Verständnis der Laufzeitumgebung ein.

Fehler, die mit bestimmten APIs verbunden sind, können in den für diese APIs spezifischen Themen beschrieben werden. Strings sind ein Sonderfall. Sie werden in der Java-Sprachspezifikation behandelt. Andere Details als häufig vorkommende Fehler können in diesem Thema unter Strings beschrieben werden .

Examples

Fallstricke: Verwenden Sie ==, um primitive Wrapper-Objekte wie Integer zu vergleichen

(Diese Fallstricke gilt für alle primitiven Wrapper-Typen, wir werden sie jedoch für Integer und int illustrieren.)

Wenn Sie mit Integer Objekten arbeiten, ist es verführerisch, == zum Vergleichen von Werten zu verwenden, da Sie dies mit int Werten tun würden. Und in manchen Fällen scheint das zu funktionieren:

```
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
```

Hier haben wir zwei Integer Objekte mit dem Wert 1 und vergleichen (in diesem Fall haben wir eines aus einem String und eines aus einem int Literal erstellt. Es gibt andere Alternativen). Wir beobachten auch, dass die beiden Vergleichsmethoden (== und equals) beide true .

Dieses Verhalten ändert sich, wenn wir andere Werte wählen:

```
Integer int2_1 = Integer.valueOf("1000");
Integer int2_2 = Integer.valueOf(1000);

System.out.println("int2_1 == int2_2: " + (int2_1 == int2_2));          // false
System.out.println("int2_1 equals int2_2: " + int2_1.equals(int2_2));   // true
```

In diesem Fall liefert nur der equals das korrekte Ergebnis.

Der Grund für diesen Verhaltensunterschied ist, dass die JVM einen Cache mit Integer Objekten für den Bereich von -128 bis 127 verwaltet. (Der obere Wert kann mit der Systemeigenschaft "java.lang.Integer.IntegerCache.high" oder ") überschrieben werden JVM-Argument "-XX: AutoBoxCacheMax = Größe"). Bei Werten in diesem Bereich gibt Integer.valueOf() den zwischengespeicherten Wert zurück, anstatt einen neuen zu erstellen.

Daher haben im ersten Beispiel die Integer.valueOf(1) und Integer.valueOf("1") die gleiche zwischengespeicherte Integer Instanz zurückgegeben. Im zweiten Beispiel dagegen haben Integer.valueOf(1000) und Integer.valueOf("1000") sowohl neue Integer Objekte erstellt als auch zurückgegeben.

Der Operator == für Referenztypen prüft auf Referenzgleichheit (dh dasselbe Objekt). Daher ist im ersten Beispiel int1_1 == int1_2 true da die Referenzen gleich sind. Im zweiten Beispiel ist int2_1 == int2_2 falsch, da die Referenzen unterschiedlich sind.

Fallstricke: Vergessen, Ressourcen freizusetzen

Jedes Mal, wenn ein Programm eine Ressource öffnet, z. B. eine Datei oder eine Netzwerkverbindung, ist es wichtig, die Ressource freizugeben, sobald Sie mit der Verwendung fertig sind. Ähnliche Vorsicht ist geboten, wenn während des Betriebs solcher Ressourcen eine Ausnahme ausgelöst wird. Man könnte argumentieren, dass FileInputStream über einen Finalizer verfügt , der die close() -Methode für ein Garbage Collection-Ereignis aufruft. Da wir jedoch nicht sicher sind, wann ein Speicherbereinigungszyklus beginnt, kann der Eingabestrom Computerressourcen auf unbestimmte Zeit beanspruchen. Die Ressource muss in einem finally Abschnitt eines try-catch-Blocks geschlossen werden:

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();
        }
    }
}
```

Seit Java 7 gibt es eine wirklich nützliche und übersichtliche Anweisung, die speziell für diesen Fall in Java 7 eingeführt wird und Try-with-resources heißt:

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();
        }
    }
}
```

Die *try-with-resources*- Anweisung kann mit jedem Objekt verwendet werden, das die Schnittstelle Closeable oder AutoCloseable implementiert. Dadurch wird sichergestellt, dass jede Ressource am Ende der Anweisung geschlossen wird. Der Unterschied zwischen den beiden Schnittstellen besteht darin, dass die close() Methode von Closeable eine IOException die auf irgendeine Weise behandelt werden muss.

In Fällen, in denen die Ressource bereits geöffnet wurde, aber nach der Verwendung sicher geschlossen werden soll, kann sie einer lokalen Variablen innerhalb der try-with-resources zugewiesen werden

Java SE 7

```
private static void printFileJava7(InputStream extResource) throws IOException {
    try (InputStream input = extResource) {
        ... //access resource
    }
}
```

Die lokale Ressourcenvariable, die im try-with-resources-Konstruktor erstellt wird, ist effektiv final.

Pitfall: Speicherlecks

Java verwaltet den Speicher automatisch. Sie müssen den Speicher nicht manuell freigeben. Der Speicher eines Objekts auf dem Heap kann von einem Speicherbereiniger freigegeben werden, wenn das Objekt von einem aktiven Thread nicht mehr *erreichbar ist* .

Sie können jedoch verhindern, dass Speicher freigegeben wird, indem Sie zulassen, dass nicht mehr benötigte Objekte erreichbar sind. Unabhängig davon, ob Sie dies als Speicherverlust oder als Speicherverpackung bezeichnen, das Ergebnis ist das gleiche - eine unnötige Erhöhung des zugewiesenen Speichers.

Speicherverluste in Java können auf verschiedene Arten auftreten. Der häufigste Grund sind jedoch permanente Objektverweise, da der Garbage Collector Objekte nicht aus dem Heap entfernen kann, solange noch Verweise darauf vorhanden sind.

Statische Felder

Sie können einen solchen Verweis erstellen, indem Sie eine Klasse mit einem static Feld definieren, das eine Sammlung von Objekten enthält, und vergessen, das static Feld auf null zu setzen, nachdem die Sammlung nicht mehr benötigt wird. static Felder werden als GC-Roots betrachtet und niemals gesammelt. Ein weiteres Problem sind Lecks im Nicht-Heap-Speicher, wenn JNI verwendet wird.

Classloader-Leck

Der heimtückischste Speicherleck ist jedoch das Classloader-Leck . Ein Classloader enthält einen Verweis auf jede geladene Klasse, und jede Klasse enthält einen Verweis auf ihren Classloader. Jedes Objekt enthält auch einen Verweis auf seine Klasse. Wenn also selbst ein *einzelnes* Objekt einer von einem Classloader geladenen Klasse kein Müll ist, kann keine einzige Klasse abgerufen werden, die dieser Classloader geladen hat. Da jede Klasse auch auf ihre statischen Felder verweist, können sie auch nicht erfasst werden.

Akkumulationsleck Das Beispiel für Akkumulationsleck könnte folgendermaßen aussehen:

```
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();
}
```

In diesem Beispiel werden zwei geplante Aufgaben erstellt. Die erste Aufgabe übernimmt die letzte Nummer von einer Deque- numbers , und wenn die Nummer durch 51 teilbar ist, werden die Nummer und die Deque-Größe gedruckt. Die zweite Aufgabe setzt Zahlen in den Deque. Beide Tasks

werden mit einer festen Rate geplant und alle 10 ms ausgeführt.

Wenn der Code ausgeführt wird, werden Sie feststellen, dass die Größe des Deque ständig zunimmt. Dies führt letztendlich dazu, dass der Deque mit Objekten gefüllt wird, die den gesamten verfügbaren Heapspeicher beanspruchen.

Um dies zu verhindern und gleichzeitig die Semantik dieses Programms zu erhalten, können wir eine andere Methode verwenden, um Zahlen aus dem Deque zu pollLast : pollLast . Im Gegensatz zur Methode peekLast gibt pollLast das Element zurück und entfernt es aus dem Deque, während peekLast nur das letzte Element zurückgibt.

Fallstricke: Verwenden von == zum Vergleichen von Zeichenketten

Ein häufiger Fehler für Java-Anfänger ist die Verwendung des Operators == , um zu testen, ob zwei Zeichenfolgen gleich sind. Zum Beispiel:

```
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?");
            }
        }
    }
}
```

Das obige Programm soll das erste Befehlszeilenargument testen und verschiedene Meldungen drucken, wenn es nicht das Wort "Hallo" ist. Aber das Problem ist, dass es nicht funktioniert. Dieses Programm wird "Fühlen Sie sich heute mürrisch?" egal, was das erste Kommandozeilenargument ist.

In diesem Fall wird der String "Hallo" in den String-Pool eingefügt, während der String args [0] sich auf dem Heap befindet. Das bedeutet, dass es zwei Objekte gibt, die dasselbe Literal darstellen, und jedes mit seiner Referenz. Da == auf Referenzen prüft und nicht auf tatsächliche Gleichheit, führt der Vergleich meistens zu einem falschen Ergebnis. Dies bedeutet nicht, dass es immer so ist.

Wenn Sie == zum Testen von Strings verwenden, testen Sie tatsächlich, ob zwei String Objekte dasselbe Java-Objekt sind. Leider bedeutet dies nicht die Gleichheit von Zeichenfolgen in Java. In der Tat ist die korrekte Methode zum Testen von Zeichenfolgen die Verwendung der Methode equals(Object) . Für ein Paar von Strings möchten wir normalerweise testen, ob sie aus denselben Zeichen in derselben Reihenfolge bestehen.

```
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?");
            }
        }
    }
}
```

Aber es wird tatsächlich schlimmer. Das Problem ist, dass == *unter* Umständen die erwartete Antwort gibt. Zum Beispiel

```
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");
        }
    }
}
```

Interessanterweise wird dies "gleich" gedruckt, auch wenn wir die Zeichenketten falsch testen. Warum das? Da in der Java-Sprachspezifikation (Abschnitt 3.10.5: String-Literale) festgelegt ist, dass zwei aus denselben Zeichen bestehende Zeichenketten >> Literale << tatsächlich von demselben Java-Objekt dargestellt werden. Daher ist der Test == für gleiche Literale wahr. (Die String-Literale werden "interniert" und beim Laden Ihres Codes zu einem gemeinsam genutzten "String-Pool" hinzugefügt. Dies ist jedoch tatsächlich ein Implementierungsdetail.)

Die Java-Sprachspezifikation schreibt außerdem Folgendes vor: Wenn Sie einen Konstantenausdruck zur Kompilierungszeit haben, der zwei String-Literale verkettet, entspricht dies einem einzelnen Literal. Somit:

```
    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");
        }
    }
}
```

Dies gibt "1. gleich" und "2. anders" aus. Im ersten Fall wird der Ausdruck + zur Kompilierzeit ausgewertet und wir vergleichen ein String Objekt mit sich selbst. Im zweiten Fall wird es zur Laufzeit ausgewertet und wir vergleichen zwei verschiedene String Objekte

Zusammenfassend ist die Verwendung von == zum Testen von Zeichenfolgen in Java fast immer falsch, es kann jedoch nicht garantiert werden, dass sie die falsche Antwort gibt.

Fallstricke: Testen Sie eine Datei, bevor Sie versuchen, sie zu öffnen.

Einige Personen empfehlen, dass Sie verschiedene Tests auf eine Datei anwenden, bevor Sie versuchen, sie zu öffnen, um eine bessere Diagnose zu bieten oder um Ausnahmen zu vermeiden. Diese Methode versucht beispielsweise zu prüfen, ob der path einer lesbaren Datei entspricht:

```
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;
}
```

Sie können die obige Methode folgendermaßen verwenden:

```
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.
}
```

Das erste Problem liegt in der Signatur für FileInputStream(File) da der Compiler weiterhin darauf besteht, IOException hier zu fangen, oder weiter oben im Stack.

Das zweite Problem besteht darin, dass durch getValidatedFile durchgeführte getValidatedFile nicht garantieren, dass FileInputStream erfolgreich ist.

- Racebedingungen: Ein anderer Thread oder ein separater Prozess könnte die Datei umbenennen, die Datei löschen oder den Lesezugriff entfernen, nachdem die getValidatedFile zurückgegeben wurde. IOException würde zu einer "einfachen" IOException ohne die benutzerdefinierte Nachricht führen.
- Es gibt Randfälle, die von diesen Tests nicht abgedeckt werden. Auf einem System mit SELinux im Modus "Erzwingen" kann der Versuch, eine Datei zu lesen, zum Beispiel fehlschlagen, obwohl canRead() true canRead() .

Das dritte Problem ist, dass die Tests ineffizient sind. Zum Beispiel führen die Aufrufe exists , isFile und canRead jeweils einen Syscall aus , um die erforderliche Prüfung durchzuführen. Anschließend wird ein weiterer Systemaufruf ausgeführt, um die Datei zu öffnen. Dabei werden die gleichen Überprüfungen im Hintergrund wiederholt.

Kurz gesagt, Methoden wie getValidatedFile sind fehlgeleitet. Versuchen Sie einfach, die Datei zu öffnen und die Ausnahme zu behandeln:

```
try (InputStream is = new FileInputStream("somefile")) {
    // Read data etc.
} catch (IOException ex) {
    System.err.println("IO Error processing 'somefile': " + ex.getMessage());
    return;
}
```

Wenn Sie IO-Fehler beim Öffnen und Lesen unterscheiden möchten, können Sie ein verschachteltes Try / Catch verwenden. Wenn Sie eine bessere Diagnostik für offene Ausfälle produzieren wollte, könnte führen Sie die exists , isFile und canRead Kontrollen im Handler.

Fallstricke: Variablen als Objekte betrachten

Keine Java-Variable repräsentiert ein Objekt.

```
String foo;   // NOT AN OBJECT
```

Kein Java-Array enthält Objekte.

```
String bar[] = new String[100];  // No member is an object.
```

Wenn Sie Variablen fälschlicherweise als Objekte betrachten, wird Sie das tatsächliche Verhalten der Java-Sprache überraschen.

- Bei Java-Variablen, die einen primitiven Typ haben (z. B. int oder float), enthält die

Variable eine Kopie des Werts. Alle Kopien eines primitiven Wertes sind nicht unterscheidbar. dh es gibt nur einen int Wert für die Nummer eins. Primitive Werte sind keine Objekte und sie verhalten sich nicht wie Objekte.

- Bei Java-Variablen, die einen Referenztyp (entweder einen Klassen- oder einen Arraytyp) haben, enthält die Variable eine Referenz. Alle Exemplare einer Referenz sind nicht unterscheidbar. Verweise können auf Objekte zeigen oder sie können null was bedeutet, dass sie auf kein Objekt zeigen. Sie sind jedoch keine Objekte und verhalten sich nicht wie Objekte.

Variablen sind in beiden Fällen keine Objekte und enthalten in beiden Fällen keine Objekte. Sie können *Verweise auf Objekte* enthalten, sagen aber etwas anderes aus.

Beispielklasse

In den folgenden Beispielen wird diese Klasse verwendet, die einen Punkt im 2D-Raum darstellt.

```
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;
   }
}
```

Eine Instanz dieser Klasse ist ein Objekt, das zwei Felder x und y , die den Typ int .

Wir können viele Instanzen der MutableLocation Klasse haben. Einige stellen die gleichen Positionen im 2D-Raum dar; dh die jeweiligen Werte von x und y stimmen überein. Andere repräsentieren verschiedene Standorte.

Mehrere Variablen können auf dasselbe Objekt zeigen

```
MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);
```

In der oben haben wir drei Variablen erklärt here , there und elsewhere , die Verweise auf halten können MutableLocation Objekte.

Wenn Sie diese Variablen (falsch) als Objekte betrachten, werden Sie die Anweisungen wahrscheinlich falsch interpretieren:

1. Kopieren Sie den Ort "[1, 2]" here
2. Kopieren Sie den Ort "[1, 2]" there
3. Kopieren Sie den Ort "[1, 2]" an eine elsewhere

Daraus lässt sich wahrscheinlich schließen, dass wir in den drei Variablen drei unabhängige Objekte haben. In der Tat gibt es *nur zwei Objekte, die* von den oben genannten erstellt wurden. Die Variablen here und there beziehen sich tatsächlich auf dasselbe Objekt.

Wir können das demonstrieren. Angenommen die Variablendeklarationen wie oben:

```
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);
```

Dadurch wird Folgendes ausgegeben:

```
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
```

Wir haben here.x einen neuen Wert here.x und der Wert, den wir über dort sehen, wurde there.x . Sie beziehen sich auf dasselbe Objekt. Der Wert, den wir über elsewhere.x hat sich jedoch nicht geändert, weshalb elsewhere auf ein anderes Objekt verweisen muss.

Wenn eine Variable ein Objekt ist, dann ist die Zuordnung here.x = 42 würde mich nicht ändern there.x .

Der Gleichheitsoperator testet NICHT, ob zwei Objekte gleich sind

Wenn Sie den Gleichheitsoperator (==) auf Referenzwerte anwenden, wird geprüft, ob sich die Werte auf dasselbe Objekt beziehen. Es wird *nicht* getestet, ob zwei (verschiedene) Objekte im intuitiven Sinne "gleich" sind.

```
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");
}
```

Dies wird gedruckt "Hier ist da", aber es wird nicht gedruckt "Hier ist woanders". (Die Referenzen here und elsewhere beziehen sich auf zwei verschiedene Objekte.)

Wenn wir dagegen die equals(Object) implementierte equals(Object) -Methode aufrufen, testen wir, ob zwei MutableLocation Instanzen MutableLocation Position haben.

```
if (here.equals(there)) {
    System.out.println("here equals there");
}
if (here.equals(elsewhere)) {
    System.out.println("here equals elsewhere");
}
```

Dadurch werden beide Meldungen gedruckt. Insbesondere gibt here.equals(elsewhere) true zurück true da die semantischen Kriterien, die wir für die Gleichheit zweier MutableLocation Objekte ausgewählt haben, erfüllt wurden.

Methodenaufrufe übergeben KEINE Objekte

Java-Methodenaufrufe verwenden *Übergabe nach Wert* [1] , um Argumente zu übergeben und ein Ergebnis zurückzugeben.

Wenn Sie einen Verweiswert an eine Methode übergeben, übergeben Sie tatsächlich einen Verweis auf ein Objekt *anhand des Werts* . Dies bedeutet, dass eine Kopie des Objektverweises erstellt wird.

Solange beide Objektverweise immer noch auf dasselbe Objekt zeigen, können Sie das Objekt von einem der beiden Verweise aus ändern. Dies führt bei einigen zu Verwirrung.

Allerdings sind vorbei Sie *nicht* ein Objekt als Verweis [2]. Der Unterschied ist, dass, wenn die Objektreferenzkopie so geändert wird, dass sie auf ein anderes Objekt zeigt, die ursprüngliche Objektreferenz weiterhin auf das Originalobjekt zeigt.

```
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".
}
```

Sie übergeben auch keine Kopie des Objekts.

```
void f(MutableLocation foo) {
    foo.x = 42;
}

void g() {
    MutableLocation foo = new MutableLocation(0, 0);
    f(foo);
    System.out.println("foo.x is " + foo.x); // Prints "foo.x is 42"
}
```

---

1 - In Sprachen wie Python und Ruby wird der Begriff "Weitergabe durch Freigeben" für "Wertübergabe" eines Objekts / einer Referenz bevorzugt.

2 - Der Begriff "Referenzübergabe" oder "Referenzabruf" hat in der Programmiersprache eine ganz bestimmte Bedeutung. In der Tat bedeutet dies, dass Sie die Adresse *einer Variablen oder eines Array-Elements übergeben* . Wenn die aufgerufene Methode dem Formalargument einen neuen Wert zuweist, ändert sie den Wert in der ursprünglichen Variablen. Java unterstützt dies nicht. Eine ausführlichere Beschreibung der verschiedenen Mechanismen für die Übergabe von Parametern finden Sie unter https://en.wikipedia.org/wiki/Evaluation_strategy .

Pitfall: Kombination von Zuordnung und Nebenwirkungen

Gelegentlich sehen wir StackOverflow Java-Fragen (und C- oder C ++ - Fragen), in denen Folgendes gefragt wird:

```
i += a[i++] + b[i--];
```

wird für einige bekannte Anfangszustände von i , a und b zu ... ausgewertet.

Allgemein gesagt:

- Für Java ist die Antwort immer [1] angegeben, aber nicht naheliegend und oft schwer zu verstehen
- Für C und C ++ ist die Antwort oft nicht spezifiziert.

Solche Beispiele werden häufig in Prüfungen oder Vorstellungsgesprächen verwendet, um zu sehen, ob der Student oder der Interviewpartner versteht, wie die Ausdrucksbewertung in der Java-Programmiersprache wirklich funktioniert. Dies ist wohl als "Test des Wissens" legitim, aber das bedeutet nicht, dass Sie dies in einem echten Programm tun sollten.

Zur Veranschaulichung ist das folgende, scheinbar einfache Beispiel in StackOverflow-Fragen (wie dieser) einige Male erschienen. In manchen Fällen erscheint dies als echter Fehler in einem Code.

```
int a = 1;
a = a++;
System.out.println(a);    // What does this print.
```

Die meisten Programmierer (einschließlich Java-Experten), die diese Anweisungen *schnell* lesen, würden sagen, dass sie 2 ausgeben. Tatsächlich gibt es 1 . Für eine detaillierte Erklärung der Gründe lesen Sie bitte diese Antwort .

Der wahre Einstieg aus diesem und ähnlichen Beispielen ist jedoch, dass *jede* Java-Anweisung, die dieselbe Variable *sowohl als auch* Nebeneffekte zuordnet, *im besten* Fall schwer zu verstehen ist und *im schlimmsten Fall* irreführend ist. Sie sollten das Schreiben von Code vermeiden.

---

1 - mögliche Probleme mit dem Java-Speichermodell, falls die Variablen oder Objekte für andere Threads sichtbar sind.

Pitfall: Dass String nicht eine unveränderliche Klasse ist, ist nicht bekannt

Neue Java-Programmierer vergessen häufig, dass die Java- String Klasse nicht veränderbar ist, oder sie verstehen nicht ganz. Dies führt zu Problemen wie im folgenden Beispiel:

```
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();
    }
}
```

Der obige Code soll Befehlszeilenargumente in Großbuchstaben drucken. Leider funktioniert es nicht, der Fall der Argumente wird nicht geändert. Das Problem ist diese Aussage:

```
s.toUpperCase();
```

Möglicherweise denken Sie, dass der Aufruf von toUpperCase() s in eine Großbuchstabenzeichenfolge ändert. Tut es nicht Es kann nicht! String Objekte sind unveränderlich. Sie können nicht geändert werden.

In Wirklichkeit ist die toUpperCase() *zurückgibt* Methode ein String - Objekt , das eine Großversion der ist String , die Sie nennen es auf. Dies wird wahrscheinlich ein neues String Objekt sein, aber wenn s bereits alle Großbuchstaben war, könnte das Ergebnis die vorhandene Zeichenfolge sein.

Um diese Methode effektiv verwenden zu können, müssen Sie das vom Methodenaufruf zurückgegebene Objekt verwenden. zum Beispiel:

```
s = s.toUpperCase();
```

Tatsächlich gilt die Regel "Zeichenfolgen ändert sich nicht" für alle String Methoden. Wenn Sie sich daran erinnern, können Sie eine ganze Kategorie von Anfängerfehlern vermeiden.

Häufige Java-Fallstricke online lesen: https://riptutorial.com/de/java/topic/4388/haufige-java-fallstricke

Kapitel 55: HTTP-Verbindung

Bemerkungen

- Wenn Sie HttpUrlConnection unter Android verwenden, müssen Sie Ihrer App die Internet-Berechtigung hinzufügen (in AndroidManifest.xml).
- Es gibt auch andere Java-HTTP-Clients und -Bibliotheken, z. B. OkHttp von Square, die einfacher zu verwenden sind und möglicherweise eine bessere Leistung oder mehr Funktionen bieten.

Examples

Rufen Sie den Antworttext aus einer URL als String ab

```
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();
}
```

Dadurch werden Textdaten von der angegebenen URL heruntergeladen und als String zurückgegeben.

Wie das funktioniert:

- Zuerst erstellen wir eine HttpUrlConnection aus unserer URL mit der new URL(url).openConnection() . Wir UrlConnection die UrlConnection um, UrlConnection diese in eine HttpUrlConnection , sodass wir auf Dinge wie das Hinzufügen von Headern (z. B. User Agent) oder das Überprüfen des Antwortcodes zugreifen können. (Dieses Beispiel macht das nicht, aber es ist leicht hinzuzufügen.)
- Erstellen Sie dann InputStream basierend auf dem Antwortcode (zur Fehlerbehandlung).
- Erstellen Sie dann einen BufferedReader , mit dem wir Text aus InputStream lesen können, den wir aus der Verbindung erhalten.
- Jetzt hängen wir den Text Zeile für Zeile an einen StringBuilder an.
- Schließen Sie den InputStream und geben Sie den jetzt vorhandenen String zurück.

Anmerkungen:

- Diese Methode IoException eine IoException aus, falls ein Fehler IoException (z. B. ein Netzwerkfehler oder keine Internetverbindung), und es wird auch eine *ungeprüfte* MalformedUrlException wenn die angegebene URL nicht gültig ist.
- Es kann zum Lesen von URLs verwendet werden, die Text zurückgeben, z. B. Webseiten (HTML), REST-APIs, die JSON oder XML zurückgeben, usw.
- Siehe auch: Lese URL to String in wenigen Zeilen Java-Code .

Verwendungszweck:

Ist sehr einfach:

```
String text = getText("http://example.com");
//Do something with the text from example.com, in this case the HTML.
```

Post-Daten

```
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();
    }
}
```

Dadurch werden POST-Daten an die angegebene URL gesendet, und die Antwort wird Zeile für Zeile gelesen.

Wie es funktioniert

- Wie üblich erhalten wir die HttpURLConnection von einer URL .
- setRequestProperty Sie den Inhaltstyp mit setRequestProperty . Standardmäßig ist es application/x-www-form-urlencoded
- setDoOutput(true) teilt der Verbindung mit, dass wir Daten senden werden.
- Dann erhalten wir den OutputStream indem wir getOutputStream() aufrufen und Daten in ihn schreiben. Vergiss nicht, es zu schließen, wenn du fertig bist.
- Zum Schluss lesen wir die Serverantwort.

Ressource löschen

```
   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();
        }
    }
```

Dies löscht die Ressource in der angegebenen URL und druckt dann den Antwortheader.

Wie es funktioniert

- Wir erhalten die HttpURLConnection von einer URL .
- setRequestProperty Sie den Inhaltstyp mit setRequestProperty . Standardmäßig ist es application/x-www-form-urlencoded
- setDoInput(true) teilt der Verbindung mit, dass die URL-Verbindung für die Eingabe verwendet werden soll.
- setRequestMethod("DELETE") zum Ausführen von HTTP DELETE

Zum Schluss drucken wir den Server-Antwortheader.

Überprüfen Sie, ob eine Ressource vorhanden ist

```
/**
 * 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;
}
```

Erläuterung:

Wenn Sie nur prüfen, ob eine Ressource vorhanden ist, ist es besser, eine HEAD-Anforderung als eine GET zu verwenden. Dadurch wird der Aufwand für die Übertragung der Ressource vermieden.

Beachten Sie, dass die Methode nur dann true zurückgibt true wenn der Antwortcode 200 lautet. Wenn Sie Umleitungsantworten (z. B. 3XX) erwarten, muss die Methode möglicherweise verbessert werden, um sie zu berücksichtigen.

Beispiel:

```
checkIfResourceExists(new URL("http://images.google.com/")); // true
checkIfResourceExists(new URL("http://pictures.google.com/")); // false
```

HTTP-Verbindung online lesen: https://riptutorial.com/de/java/topic/156/http-verbindung

Kapitel 56: InputStreams und OutputStreams

Syntax

- int read (byte [] b) löst IOException aus

Bemerkungen

Beachten Sie, dass Sie InputStream meistens NICHT direkt, sondern BufferedStream oder ähnliches verwenden. Dies liegt daran, dass InputStream bei jedem Aufruf der InputStream aus der Quelle liest. Dies kann zu erheblicher CPU-Auslastung bei Kontextwechseln in den Kernel und aus diesem heraus führen.

Examples

InputStream in einen String einlesen

Manchmal möchten Sie die Byte-Eingabe in einen String einlesen. Dazu müssen Sie etwas finden, das zwischen byte und den als char verwendeten UTF-16-Codepunkten "native Java" konvertiert. InputStreamReader geschieht mit einem InputStreamReader .

Um den Vorgang etwas zu beschleunigen, ist es "üblich", einen Puffer zuzuweisen, damit beim Lesen von Input nicht zu viel Aufwand entsteht.

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();
}
```

Die Umwandlung dieses Beispiels in Java SE 6 (und niedriger) kompatiblen Code wird für den Leser als Übung ausgelassen.

Schreiben von Bytes in einen OutputStream

Schreiben von Bytes in einen OutputStream zu einem Zeitpunkt

```
OutputStream stream = object.getOutputStream();

byte b = 0x00;
stream.write( b );
```

Schreiben eines Byte-Arrays

```
byte[] bytes = new byte[] { 0x00, 0x00 };

stream.write( bytes );
```

Einen Abschnitt eines Byte-Arrays schreiben

```
int offset = 1;
int length = 2;
byte[] bytes = new byte[] { 0xFF, 0x00, 0x00, 0xFF };

stream.write( bytes, offset, length );
```

Streams schließen

Die meisten Streams müssen geschlossen werden, wenn Sie damit fertig sind. Andernfalls können Sie einen Speicherverlust verursachen oder eine Datei geöffnet lassen. Es ist wichtig, dass Streams auch dann geschlossen werden, wenn eine Ausnahme ausgelöst wird.

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
}
```

Denken Sie daran: Try-with-Resources garantiert, dass die Ressourcen beim Beenden der Blockierung geschlossen wurden, unabhängig davon, ob dies mit dem üblichen Kontrollfluss oder aufgrund einer Ausnahme geschieht.

Java SE 6

Try-with-resources ist manchmal keine Option, oder Sie unterstützen ältere Versionen von Java 6 oder früher. In diesem Fall ist eine ordnungsgemäße Handhabung die Verwendung eines finally Blocks:

```
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...
    }
}
```

Beachten Sie, dass beim Schließen eines Wrapper-Streams auch der zugrunde liegende Stream geschlossen wird. Dies bedeutet, dass Sie einen Stream nicht umbrechen können, den Wrapper

schließen und dann den ursprünglichen Stream weiterhin verwenden können.

Eingabestrom in Ausgabestrom kopieren

Diese Funktion kopiert Daten zwischen zwei Streams -

```
void copy(InputStream in, OutputStream out) throws IOException {
    byte[] buffer = new byte[8192];
    while ((bytesRead = in.read(buffer)) > 0) {
        out.write(buffer, 0, bytesRead);
    }
}
```

Beispiel -

```
 // reading from System.in and writing to System.out
copy(System.in, System.out);
```

Eingabe / Ausgabe-Streams umschließen

OutputStream und InputStream haben viele verschiedene Klassen, jede mit einer einzigartigen Funktionalität. Wenn Sie einen Stream um einen anderen wickeln, erhalten Sie die Funktionalität beider Streams.

Sie können einen Stream beliebig oft bündeln. Notieren Sie sich einfach die Bestellung.

Nützliche Kombinationen

Zeichen in eine Datei schreiben, während ein Puffer verwendet wird

```
File myFile = new File("targetFile.txt");
PrintWriter writer = new PrintWriter(new BufferedOutputStream(new FileOutputStream(myFile)));
```

Daten vor dem Schreiben in eine Datei komprimieren und verschlüsseln, während ein Puffer verwendet wird

```
Cipher cipher = ... // Initialize cipher
File myFile = new File("targetFile.enc");
BufferedOutputStream outputStream = new BufferedOutputStream(new DeflaterOutputStream(new
CipherOutputStream(new FileOutputStream(myFile), cipher)));
```

Liste der Input / Output Stream-Wrapper

Verpackung	Beschreibung
BufferedOutputStream / BufferedInputStream	Während OutputStream ein Byte zu einem Zeitpunkt schreibt, BufferedOutputStream schreibt Daten in Stücke schneiden. Dies reduziert die Anzahl der Systemaufrufe und verbessert so die Leistung.
DeflaterOutputStream / DeflaterInputStream	Führt eine Datenkomprimierung durch.
InflaterOutputStream / InflaterInputStream	Führt die Dekomprimierung von Daten durch.
CipherOutputStream /	Verschlüsselt / entschlüsselt Daten.

Verpackung	Beschreibung
CipherInputStream	
DigestOutputStream / DigestInputStream	Erzeugt Message Digest, um die Datenintegrität zu überprüfen.
CheckedOutputStream / CheckedInputStream	Erzeugt eine CheckSum. CheckSum ist eine trivialere Version von Message Digest.
DataOutputStream / DataInputStream	Ermöglicht das Schreiben von primitiven Datentypen und Strings. Bezeichnet das Schreiben von Bytes. Plattformunabhängig.
PrintStream	Ermöglicht das Schreiben von primitiven Datentypen und Strings. Bezeichnet das Schreiben von Bytes. Plattformabhängig
OutputStreamWriter	Konvertiert einen OutputStream in einen Writer. Ein OutputStream behandelt Bytes, während Writers sich mit Zeichen befassen
PrintWriter	Ruft automatisch OutputStreamWriter auf. Ermöglicht das Schreiben von primitiven Datentypen und Strings. Nur für das Schreiben von Zeichen und am besten für das Schreiben von Zeichen

DataInputStream-Beispiel

```
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+"-");
    }
  }
}
```

InputStreams und OutputStreams online lesen:
https://riptutorial.com/de/java/topic/110/inputstreams-und-outputstreams

Kapitel 57: Iterator und Iterable

Einführung

Der java.util.Iterator ist die Standard-Java SE-Schnittstelle für Objekte, die das Iterator-Entwurfsmuster implementieren. Die Schnittstelle java.lang.Iterable ist für Objekte java.lang.Iterable , *die* einen Iterator *bereitstellen* können.

Bemerkungen

Es ist möglich, ein Array mithilfe der for -each-Schleife for durchlaufen, obwohl Java-Arrays keine Iterable-Funktion implementieren. Das Iterieren wird von JVM mit einem nicht zugänglichen Index im Hintergrund ausgeführt.

Examples

Iterable in for-Schleife verwenden

Klassen, die die Iterable<> -Schnittstelle implementieren, können in for Schleifen verwendet werden. Dies ist eigentlich nur syntaktischer Zucker , um einen Iterator vom Objekt zu erhalten und ihn zu verwenden, um alle Elemente nacheinander zu erhalten. Es macht den Code klarer, schneller zu schreiben und weniger fehleranfällig.

```
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);
        }
    }
}
```

Verwenden des rohen Iterators

Während die Verwendung der foreach-Schleife (oder "extended for loop") einfach ist, ist es manchmal von Vorteil, den Iterator direkt zu verwenden. Wenn Sie beispielsweise eine Reihe von durch Kommas getrennten Werten ausgeben möchten, das letzte Element jedoch kein Komma enthalten soll:

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

Dies ist viel einfacher und übersichtlicher als eine isLastEntry Variable oder Berechnungen mit dem Schleifenindex.

Erstellen Sie Ihre eigenen Iterable.

Um Ihre eigene Iterable wie mit jeder Schnittstelle zu erstellen, implementieren Sie einfach die abstrakten Methoden in der Schnittstelle. Für Iterable gibt es nur einen, der iterator() . Der Iterator Rückgabetyp ist selbst eine Schnittstelle mit drei abstrakten Methoden. Sie können einen mit einer Sammlung verknüpften Iterator zurückgeben oder Ihre eigene benutzerdefinierte Implementierung erstellen:

```
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");
            }
        };
    }
}
```

Benutzen:

```
public static void main(String[] args) {
    for(char c : new Alphabet()) {
        System.out.println("c = " + c);
    }
}
```

Der neue Iterator sollte einen Status aufweisen, der auf das erste Element verweist. Jeder Aufruf von next aktualisiert seinen Status so, dass er auf das nächste Element verweist. Das hasNext() prüft, ob der Iterator am Ende ist. Wenn der Iterator mit einer modifizierbaren Auflistung verbunden war, könnte die optionale remove() Methode des Iterators implementiert werden, um das aktuell aufgerufene Element aus der zugrunde liegenden Auflistung zu entfernen.

Elemente mit einem Iterator entfernen

Die Iterator.remove() -Methode ist eine optionale Methode, die das vom vorherigen Aufruf von Iterator.next() Element entfernt. Mit dem folgenden Code wird beispielsweise eine Liste von

Zeichenfolgen aufgefüllt und anschließend alle leeren Zeichenfolgen entfernt.

```
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());
```

Ausgabe :

```
Old Size : 5
New Size : 3
```

Beachten Sie, dass der obige Code der sichere Weg ist, Elemente zu entfernen, während Sie eine typische Sammlung durchlaufen. Wenn Sie stattdessen versuchen, Elemente aus einer Sammlung wie folgt zu entfernen:

```
for (String el: names) {
    if (el.equals("")) {
        names.remove(el); // WRONG!
    }
}
```

Eine typische Auflistung (z. B. ArrayList), die Iteratoren mit *Fail-Fast*- Iteratorsemantik bereitstellt, löst eine ConcurrentModificationException .

Die remove() Methode kann nur (einmal) nach einem next() Aufruf aufgerufen werden. Wenn es vor dem Aufruf von next() aufgerufen wird oder zweimal nach einem next() Aufruf aufgerufen wird, IllegalStateException der remove() Aufruf eine IllegalStateException .

Der Vorgang zum remove wird als *optionaler* Vorgang beschrieben. dh nicht alle Iteratoren werden es zulassen. Beispiele, bei denen dies nicht unterstützt wird, sind Iteratoren für unveränderliche Sammlungen, schreibgeschützte Ansichten von Sammlungen oder Sammlungen mit fester Größe. Wenn remove() aufgerufen wird, wenn der Iterator das Entfernen nicht unterstützt, wird eine UnsupportedOperationException .

Iterator und Iterable online lesen: https://riptutorial.com/de/java/topic/172/iterator-und-iterable

Kapitel 58: JAR-Dateien mit mehreren Versionen

Einführung

Eine der in Java 9 eingeführten Funktionen ist die Multi-Release-Jar (MRJAR), die die Bündelung von Code für mehrere Java-Releases in derselben Jar-Datei ermöglicht. Die Funktion ist in JEP 238 angegeben .

Examples

Beispiel für den Inhalt einer Jar-Datei mit mehreren Versionen

Durch Festlegen von Multi-Release: true in der MANIFEST.MF-Datei wird die Jar-Datei zu einer Multi-Release-Jar und die Java-Laufzeitumgebung (sofern das MRJAR-Format unterstützt wird) wählt die entsprechenden Klassenversionen abhängig von der aktuellen Hauptversion aus .

Die Struktur eines solchen Bechers ist wie folgt:

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

- Bei JDKs <9 sind nur die Klassen im Root-Eintrag für die Java-Laufzeitumgebung sichtbar.
- Auf einem JDK 9 werden die Klassen A und B aus dem Verzeichnis root/META-INF/versions/9 geladen, während C und D aus dem Basiseintrag geladen werden.
- Auf einem JDK 10 wird Klasse A aus dem Verzeichnis root/META-INF/versions/10 geladen.

Ein Mehrfach-Release-Jar mit dem Jar-Tool erstellen

Mit dem Befehl jar können Sie eine Jar-Version mit mehreren Versionen erstellen, die zwei Versionen derselben Klasse enthält, die sowohl für Java 8 als auch für Java 9 kompiliert wurden. Die Warnung gibt jedoch an, dass die Klassen identisch sind:

```
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
```

Die Option --release 9 weist jar an, alles, was folgt (das demo Paket im sampleproject-9), in einen versionierten Eintrag in der MRJAR aufzunehmen, und zwar unter root/META-INF/versions/9 . Das Ergebnis ist folgender Inhalt:

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

```
              - SampleClass.class
```

Lassen Sie uns nun eine Klasse namens Main erstellen, die die URL der SampleClass , und fügen Sie sie für die Java 9-Version hinzu:

```
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);
    }
}
```

Wenn wir diese Klasse kompilieren und den JAR-Befehl erneut ausführen, wird ein Fehler angezeigt:

```
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
```

Der Grund ist, dass das jar Tool das Hinzufügen öffentlicher Klassen zu versionierten Einträgen verhindert, wenn diese nicht ebenfalls zu den Basiseinträgen hinzugefügt werden. Dies geschieht, damit der MRJAR dieselbe öffentliche API für die verschiedenen Java-Versionen verfügbar macht. Beachten Sie, dass diese Regel zur Laufzeit nicht erforderlich ist. Es kann nur von Werkzeugen wie dem jar angewendet werden. In diesem speziellen Fall besteht der Zweck von Main darin, Beispielcode auszuführen, sodass wir einfach eine Kopie im Basiseintrag hinzufügen können. Wenn die Klasse Teil einer neueren Implementierung war, die wir nur für Java 9 benötigen, könnte sie nicht öffentlich gemacht werden.

Um Main zum Root-Eintrag hinzuzufügen, müssen wir ihn zunächst für ein Pre-Java 9-Release kompilieren. Dies kann mit der neuen Option --release von 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
```

Das Ausführen der Main-Klasse zeigt, dass die SampleClass aus dem versionierten Verzeichnis geladen wird:

```
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 einer geladenen Klasse in einer Jar mit mehreren Versionen

Angesichts der folgenden Multi-Release-Jar:

```
jar root
  - demo
     - SampleClass.class
```

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

Die folgende Klasse druckt die URL der 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);
    }
}
```

Wenn die Klasse kompiliert und zu dem versionierten Eintrag für Java 9 in der MRJAR hinzugefügt wird, führt das Ausführen zu folgendem Ergebnis:

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

JAR-Dateien mit mehreren Versionen online lesen: https://riptutorial.com/de/java/topic/9866/jar-dateien-mit-mehreren-versionen

Kapitel 59: Java Compiler - "Javac"

Bemerkungen

Der Befehl javac wird zum Kompilieren von Java-Quelldateien zu Bytecode-Dateien verwendet. Bytecode-Dateien sind plattformunabhängig. Dies bedeutet, dass Sie Ihren Code auf einer Art von Hardware und Betriebssystem kompilieren und dann auf einer anderen Plattform ausführen können, die Java unterstützt.

Der Befehl javac ist in den JDK-Distributionen (Java Development Kit) enthalten.

Der Java-Compiler und der Rest der Standard-Java-Toolchain unterliegen folgenden Einschränkungen für den Code:

- Der Quellcode wird in Dateien mit dem Suffix ".java" gespeichert.
- Bytecodes werden in Dateien mit dem Suffix ".class" gespeichert.
- Bei Quell- und Bytecode-Dateien im Dateisystem müssen die Dateipfadnamen die Paket- und Klassennamen angeben.

Hinweis: Der javac Compiler sollte nicht mit dem Just In Time-Compiler (JIT) verwechselt werden, der Bytecodes in nativen Code kompiliert.

Examples

Der Befehl 'Javac' - Erste Schritte

Einfaches Beispiel

Angenommen, die "HelloWorld.java" enthält die folgende Java-Quelle:

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

(Eine Erläuterung des obigen Codes finden Sie unter Erste Schritte mit Java Language .)

Wir können die obige Datei mit diesem Befehl kompilieren:

```
$ javac HelloWorld.java
```

Dies erzeugt eine Datei namens "HelloWorld.class", die wir dann wie folgt ausführen können:

```
$ java HelloWorld
Hello world!
```

Die wichtigsten Punkte aus diesem Beispiel sind:

1. Der Quelldateiname "HelloWorld.java" muss mit dem Klassennamen in der Quelldatei übereinstimmen. HelloWorld ist HelloWorld . Wenn sie nicht übereinstimmen, wird ein Kompilierungsfehler angezeigt.
2. Der Bytecode-Dateiname "HelloWorld.class" entspricht dem Klassennamen. Wenn Sie die "HelloWorld.class" umbenennen, erhalten Sie eine Fehlermeldung, wenn Sie versuchen, sie auszuführen.
3. Wenn Sie eine Java-Anwendung mit java ausführen, geben Sie den Klassennamen NICHT den Bytecode-Dateinamen an.

Beispiel mit Paketen

Die meisten praktischen Java-Codes verwenden Pakete, um den Namespace für Klassen zu organisieren und das Risiko einer versehentlichen Klassennamenkollision zu reduzieren.

Wenn wir das erklären wollte HelloWorld Klasse in einem Paket Anruf com.example , die „HelloWorld.java" würde folgende Komponenten enthalten Java Quelle:

```
package com.example;

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

Diese Quellcodedatei muss in einer Verzeichnisstruktur gespeichert werden, deren Struktur der Paketnamen entspricht.

```
.    # the current directory (for this example)
|
 ----com
     |
      ----example
          |
           ----HelloWorld.java
```

Wir können die obige Datei mit diesem Befehl kompilieren:

```
$ javac com/example/HelloWorld.java
```

Dadurch wird eine Datei mit dem Namen "com / example / HelloWorld.class" erzeugt. dh nach dem Kompilieren sollte die Dateistruktur so aussehen:

```
.    # the current directory (for this example)
|
 ----com
     |
      ----example
          |
           ----HelloWorld.java
           ----HelloWorld.class
```

Wir können die Anwendung dann wie folgt ausführen:

```
$ java com.example.HelloWorld
Hello world!
```

Zusätzliche Punkte aus diesem Beispiel sind zu beachten:

1. Die Verzeichnisstruktur muss mit der Paketnamenstruktur übereinstimmen.
2. Wenn Sie die Klasse ausführen, muss der vollständige Klassenname angegeben werden. dh "com.example.HelloWorld" nicht "HelloWorld".
3. Sie müssen Java-Code nicht aus dem aktuellen Verzeichnis kompilieren und ausführen. Wir machen es hier nur zur Veranschaulichung.

Mehrere Dateien auf einmal mit 'javac' kompilieren.

Wenn Ihre Anwendung aus mehreren Quellcodedateien besteht (und die meisten tun es!), Können Sie sie einzeln kompilieren. Alternativ können Sie mehrere Dateien gleichzeitig kompilieren, indem Sie die Pfadnamen auflisten:

```
$ javac Foo.java Bar.java
```

oder verwenden Sie die Dateinamen-Platzhalterfunktionalität Ihrer Befehls-Shell ....

```
$ javac *.java
$ javac com/example/*.java
$ javac */**/*.java #Only works on Zsh or with globstar enabled on your shell
```

Dadurch werden alle Java-Quelldateien im aktuellen Verzeichnis, im Verzeichnis "com / example" und rekursiv in untergeordneten Verzeichnissen kompiliert. Eine dritte Alternative besteht darin, eine Liste von Quelldateinamen (und Compileroptionen) als Datei bereitzustellen. Zum Beispiel:

```
$ javac @sourcefiles
```

wo die sourcefiles enthält:

```
Foo.java
Bar.java
com/example/HelloWorld.java
```

Hinweis: Das Kompilieren von Code eignet sich für kleine Ein-Personen-Projekte und einmalige Programme. Darüber hinaus ist es ratsam, ein Java-Build-Tool auszuwählen und zu verwenden. Alternativ verwenden die meisten Programmierer eine Java IDE (z. B. NetBeans , Eclipse , IntelliJ IDEA), die einen eingebetteten Compiler und die inkrementelle Erstellung von "Projekten" bietet.

Häufig verwendete "Javac" -Optionen

Hier sind ein paar Optionen für den Befehl javac , die für Sie wahrscheinlich nützlich sind

- Die Option -d legt ein Zielverzeichnis für das Schreiben der ".class" -Dateien fest.
- Die Option -sourcepath legt einen Quellcode-Suchpfad fest.
- Die Option -cp oder -classpath legt den Suchpfad zum Suchen externer und zuvor kompilierter Klassen fest. Weitere Informationen zum Klassenpfad und seiner Angabe finden Sie im Thema Der Klassenpfad.
- Die Option -version die Versionsinformationen des Compilers aus.

Eine vollständigere Liste der Compileroptionen wird in einem separaten Beispiel beschrieben.

Verweise

Die endgültige Referenz für den Befehl javac ist die Oracle-Handbuchseite für javac .

Kompilieren für eine andere Java-Version

Die Java-Programmiersprache (und ihre Laufzeit) hat seit ihrer Veröffentlichung seit ihrer Veröffentlichung zahlreiche Änderungen erfahren. Diese Änderungen umfassen:

- Änderungen in der Syntax und Semantik der Java-Programmiersprache
- Änderungen in den APIs, die von den Java-Standardklassenbibliotheken bereitgestellt werden.
- Änderungen im Java-Befehlssatz (Bytecode) und im Format der Klassendatei.

Mit wenigen Ausnahmen (z. B. das Schlüsselwort enum , Änderungen an einigen "internen" Klassen usw.) sind diese Änderungen abwärtskompatibel.

- Ein Java-Programm, das mit einer älteren Version der Java-Toolchain kompiliert wurde, wird auf einer neueren Version der Java-Plattform ohne Neukompilierung ausgeführt.
- Ein Java-Programm, das in einer älteren Java-Version geschrieben wurde, wird erfolgreich mit einem neuen Java-Compiler kompiliert.

Kompilieren von altem Java mit einem neueren Compiler

Wenn Sie älteren Java-Code auf einer neueren Java-Plattform (neu) kompilieren müssen, um auf der neueren Plattform ausgeführt zu werden, müssen Sie im Allgemeinen keine speziellen Kompilierungsflags angeben. In einigen Fällen (z. B. wenn Sie enum als Bezeichner verwendet haben), können Sie die Option -source , um die neue Syntax zu deaktivieren. Zum Beispiel die folgende Klasse gegeben:

```
public class OldSyntax {
    private static int enum;  // invalid in Java 5 or later
}
```

Zum Kompilieren der Klasse mit einem Java 5-Compiler (oder höher) ist Folgendes erforderlich:

```
$ javac -source 1.4 OldSyntax.java
```

Kompilieren für eine ältere Ausführungsplattform

Wenn Sie Java für die Ausführung auf älteren Java-Plattformen kompilieren müssen, installieren Sie am einfachsten ein JDK für die älteste Version, die Sie unterstützen müssen, und verwenden Sie den Compiler dieses JDK in Ihren Builds.

Sie können auch mit einem neueren Java-Compiler kompilieren, dies ist jedoch kompliziert. Zunächst einige wichtige Voraussetzungen, die erfüllt sein müssen:

- Der Code, den Sie kompilieren, darf keine Java-Sprachkonstrukte verwenden, die in der Java-Version, auf die Sie abzielen, nicht verfügbar waren.
- Der Code darf nicht von Standard-Java-Klassen, -Feldern, -Methoden usw. abhängig sein, die auf älteren Plattformen nicht verfügbar waren.
- Bibliotheken von Drittanbietern, von denen der Code abhängig ist, müssen auch für die ältere Plattform erstellt werden und zur Kompilierungszeit und zur Laufzeit verfügbar sein.

Wenn die Voraussetzungen erfüllt sind, können Sie den Code für eine ältere Plattform mit der Option -target kompilieren. Zum Beispiel,

```
$ javac -target 1.4 SomeClass.java
```

kompiliert die obige Klasse, um Bytecodes zu erzeugen, die mit JVM von Java 1.4 oder höher kompatibel sind. (Tatsächlich impliziert die Option -source ein kompatibles -target , sodass javac -source 1.4 ... die gleiche Wirkung hätte. Die Beziehung zwischen -source und -target ist in der Oracle-Dokumentation beschrieben.)

Wenn Sie jedoch -target oder -source , kompilieren Sie immer noch mit den Standardklassenbibliotheken, die vom JDK des Compilers bereitgestellt werden. Wenn Sie nicht vorsichtig sind, können Sie Klassen mit der korrekten Bytecode-Version erhalten, jedoch mit Abhängigkeiten von nicht verfügbaren APIs. Die Lösung besteht darin, die Option -bootclasspath zu verwenden. Zum Beispiel:

```
$ javac -target 1.4 --bootclasspath path/to/java1.4/rt.jar SomeClass.java
```

wird mit einem alternativen Satz von Laufzeitbibliotheken kompiliert. Wenn die zu kompilierende Klasse (versehentlich) von neueren Bibliotheken abhängig ist, werden Kompilierungsfehler angezeigt.

Java Compiler - "Javac" online lesen: https://riptutorial.com/de/java/topic/4478/java-compiler---javac-

Kapitel 60: Java installieren (Standard Edition)

Einführung

Diese Dokumentationsseite bietet Zugriff auf Anweisungen zum Installieren der java standard edition auf Windows , Linux und macOS Computern.

Examples

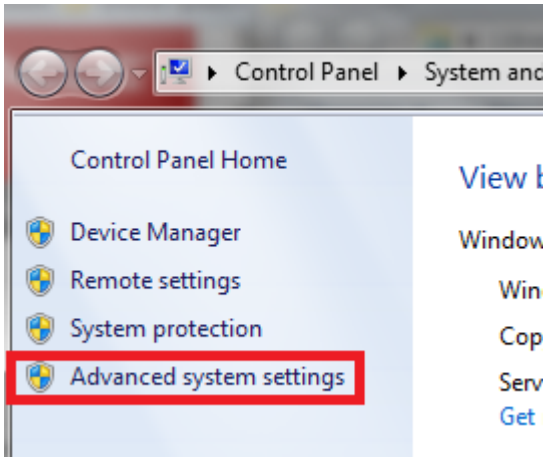
Einstellung von% PATH% und% JAVA_HOME% nach der Installation unter Windows

Annahmen:

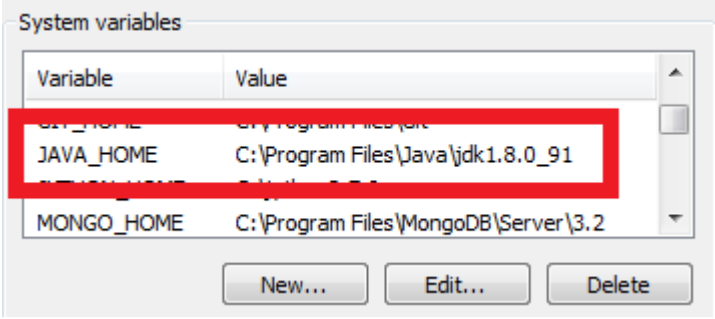
- Ein Oracle-JDK wurde installiert.
- Das JDK wurde im Standardverzeichnis installiert.

Setup-Schritte

1. Öffnen Sie den Windows Explorer.
2. Klicken Sie im Navigationsbereich links mit der rechten Maustaste auf *Diesen PC* (oder *Computer* für ältere Windows-Versionen). Es gibt einen kürzeren Weg, ohne den Explorer in aktuellen Windows-Versionen zu verwenden: Drücken Sie einfach Win + Pause
3. Klicken Sie im neu geöffneten Fenster der Systemsteuerung mit der linken Maustaste auf *Erweiterte Systemeinstellungen*, die sich in der oberen linken Ecke befinden sollte. Dies öffnet das Fenster *Systemeigenschaften* .



   Alternativ können Sie SystemPropertiesAdvanced (Groß- und Kleinschreibung beachten) in *Run* (Win + R) eingeben und die Eingabetaste drücken .
4. Wählen Sie auf der Registerkarte *Erweitert* der *Systemeigenschaften* die Schaltfläche Umgebungsvariablen ... in der rechten unteren Ecke des Fensters.
5. Fügen Sie eine **Variable New Systems** durch die Schaltfläche Neu ... in *Systemvariablen* mit dem Namen klicken JAVA_HOME und dessen Wert ist der Pfad zu dem Verzeichnis , in dem das JDK installiert wurde. Nachdem Sie diese Werte eingegeben haben, drücken Sie OK .



6. Blättern Sie durch die Liste der *Systemvariablen* und wählen Sie die Path .

7. **ACHTUNG:** Windows verwendet Path , um wichtige Programme zu finden. Wenn eine oder alle davon entfernt werden, kann Windows möglicherweise nicht ordnungsgemäß funktionieren. Es muss geändert werden, damit Windows das JDK ausführen kann. Klicken Sie dazu auf die Schaltfläche "Bearbeiten", und wählen Sie die Path Variable aus. %JAVA_HOME%\bin; an den Anfang der Path .

Es ist besser, am Anfang der Zeile anzuhängen, da die Software von Oracle zum Registrieren der eigenen Java-Version in Path Dadurch wird Ihre Version ignoriert, wenn sie nach der Oracle-Deklaration auftritt.

Überprüfe deine Arbeit

1. Öffnen Sie die Eingabeaufforderung, indem Sie auf Start klicken, dann cmd eingeben und die Enter drücken.
2. Geben Sie javac -version in die Eingabeaufforderung ein. Wenn dies erfolgreich war, wird die Version des JDK auf dem Bildschirm gedruckt.

Hinweis: Wenn Sie es erneut versuchen müssen, schließen Sie die Aufforderung, bevor Sie Ihre Arbeit überprüfen. Dadurch werden Fenster gezwungen, die neue Version von Path .

Auswahl einer geeigneten Java SE-Version

Es gibt viele Java-Versionen seit der ursprünglichen Java 1.0-Version im Jahr 1995. (Eine Übersicht finden Sie in der Java-Versionsgeschichte .) Die meisten Versionen haben jedoch ihre offiziellen End Of Life-Daten überschritten. Dies bedeutet, dass der Hersteller (in der Regel Oracle) die Neuentwicklung für das Release eingestellt hat und keine öffentlichen / kostenlosen Patches mehr für Fehler oder Sicherheitsprobleme bereitstellt. (Private Patch-Versionen sind normalerweise für Personen / Organisationen mit Supportvertrag verfügbar; wenden Sie sich an das Vertriebsbüro Ihres Lieferanten.)

Im Allgemeinen ist die empfohlene Java SE-Version das neueste Update für die neueste öffentliche Version. Derzeit ist dies die neueste verfügbare Java 8-Version. Java 9 soll 2017 veröffentlicht werden. (Java 7 hat sein End Of Life bestanden. Die letzte öffentliche Veröffentlichung war April 2015. Java 7 und frühere Versionen werden nicht empfohlen.)

Diese Empfehlung gilt für alle neuen Java-Entwicklungen und für alle, die Java lernen. Dies gilt auch für Benutzer, die nur Java-Software von Drittanbietern verwenden möchten. Im Allgemeinen funktioniert gut geschriebener Java-Code in einer neueren Version von Java. (Überprüfen Sie jedoch die Versionshinweise der Software und wenden Sie sich an den Autor / Anbieter / Verkäufer, wenn Sie Zweifel haben.)

Wenn Sie mit einer älteren Java-Codebase arbeiten, sollten Sie sicherstellen, dass Ihr Code auf der neuesten Version von Java ausgeführt wird. Die Entscheidung, wann die Funktionen neuerer Java-Versionen verwendet werden sollen, ist schwieriger, da dies die Möglichkeit beeinträchtigt, Kunden zu unterstützen, die ihre Java-Installation nicht durchführen können oder nicht.

Java Release und Versionsbenennung

Die Benennung von Java-Versionen ist etwas verwirrend. Wie in dieser Tabelle gezeigt, gibt es zwei Systeme zur Benennung und Nummerierung:

JDK-Version	Marketingname
jdk-1,0	JDK 1,0
jdk-1,1	JDK 1.1
jdk-1,2	J2SE 1.2
...	...
jdk-1,5	J2SE 1.5 wurde in Java SE 5 umbenannt
jdk-1,6	Java SE 6
jdk-1.7	Java SE 7
jdk-1,8	Java SE 8
jdk-9 [1]	Java SE 9 (noch nicht veröffentlicht)

1 - Es scheint, dass Oracle beabsichtigt, die bisherige Praxis der Verwendung eines "semantischen Versionsnummer" - Schemas in den Java-Versionszeichenfolgen zu durchbrechen. Es bleibt abzuwarten, ob sie dies durchsetzen werden.

Das "SE" in den Marketingnamen bezieht sich auf die Standard Edition. Dies ist die Basisversion für die Ausführung von Java auf den meisten Laptops, PCs und Servern (außer Android).

Es gibt zwei weitere offizielle Ausgaben von Java: "Java ME" ist die Micro Edition und "Java EE" die Enterprise Edition. Die Android-Variante von Java unterscheidet sich auch erheblich von Java SE. Java ME, Java EE und Android Java sind nicht Gegenstand dieses Themas.

Die vollständige Versionsnummer für eine Java-Version sieht folgendermaßen aus:

```
1.8.0_101-b13
```

Dies besagt JDK 1.8.0, Update 101, Build # 13. Oracle bezeichnet dies in den Versionshinweisen als:

```
Java™ SE Development Kit 8, Update 101 (JDK 8u101)
```

Die Update-Nummer ist wichtig - Oracle veröffentlicht regelmäßig Updates für eine Hauptversion mit Sicherheitspatches, Fehlerbehebungen und (in einigen Fällen) neuen Funktionen. Die Buildnummer ist normalerweise irrelevant. Beachten Sie, dass sich Java 8 und Java 1.8 *auf dasselbe beziehen* . Java 8 ist nur der "Marketing" -Name für Java 1.8.

Was brauche ich für die Java-Entwicklung?

Eine JDK-Installation und ein Texteditor sind das Minimum für die Java-Entwicklung. (Es ist schön, einen Texteditor zu haben, der Java-Syntax-Highlighting ermöglicht, aber Sie können auch darauf verzichten.)

Für ernsthafte Entwicklungsarbeit wird jedoch empfohlen, dass Sie auch Folgendes verwenden:

- Eine Java-IDE wie Eclipse, Intellij IDEA oder NetBeans

- Ein Java-Build-Tool wie Ant, Gradle oder Maven
- Ein Versionskontrollsystem zur Verwaltung Ihrer Codebasis (mit geeigneten Sicherungen und Off-Site-Replikation)
- Testwerkzeuge und CI-Tools (Continuous Integration)

Installieren eines Java-JDK unter Linux

Verwenden des Package Managers

JDK- und / oder JRE-Versionen für OpenJDK oder Oracle können mit dem Paketmanager der meisten Mainstream-Linux-Distributionen installiert werden. (Die Auswahlmöglichkeiten, die Ihnen zur Verfügung stehen, hängen von der Distribution ab.)

In der Regel öffnen Sie das Terminalfenster und führen die folgenden Befehle aus. (Es wird davon ausgegangen, dass Sie über ausreichende Zugriffsrechte verfügen, um als "root" -Benutzer Befehle auszuführen. sudo ist die Funktion des Befehls sudo . Wenn Sie dies nicht tun, wenden Sie sich an die Administratoren Ihres Systems.)

Es wird empfohlen, den Paket-Manager zu verwenden, da er (in der Regel) die Java-Installation auf dem neuesten Stand hält.

apt-get , Debian-basierte Linux-Distributionen (Ubuntu usw.)

Die folgenden Anweisungen installieren Oracle Java 8:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Hinweis: Um die Java 8-Umgebungsvariablen automatisch einzurichten, können Sie das folgende Paket installieren:

```
$ sudo apt-get install oracle-java8-set-default
```

Erstellen einer .deb Datei

Wenn Sie es vorziehen , die erstellen .deb sich aus der Datei .tar.gz - Datei von Oracle heruntergeladen haben , gehen Sie wie folgt (vorausgesetzt , Sie haben die heruntergeladene .tar.gz ./<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
```

Hinweis : Dies erwartet, dass die Eingabe als ".tar.gz" -Datei bereitgestellt wird.

slackpkg , Slackware-basierte Linux-Distributionen

```
sudo slapt-get install default-jdk
```

yum , RedHat, CentOS usw

```
sudo yum install java-1.8.0-openjdk-devel.x86_64
```

dnf , Fedora

In den letzten Fedora-Veröffentlichungen wurde yum von dnf .

```
sudo dnf install java-1.8.0-openjdk-devel.x86_64
```

In den letzten Fedora-Versionen gibt es keine Pakete für die Installation von Java 7 und früheren Versionen.

pacman , Arch-basierte Linux-Distributionen

```
sudo pacman -S jdk8-openjdk
```

sudo ist nicht erforderlich, wenn Sie als Rootbenutzer ausgeführt werden.

Gentoo Linux

Der Gentoo Java-Guide wird vom Gentoo Java-Team gepflegt und enthält eine aktualisierte Wiki-Seite mit den korrekten Portage-Paketen und erforderlichen USE-Flags.

Installieren von Oracle-JDKs auf Redhat, CentOS, Fedora

Installation des JDK von einer Oracle tar.gz oder JRE-Datei tar.gz

1. Laden Sie die entsprechende Oracle-Archivdatei ("tar.gz") für die gewünschte Version von der Oracle Java-Downloadseite herunter .
2. Wechseln Sie in das Verzeichnis, in das Sie die Installation einfügen möchten.
3. Dekomprimieren Sie die Archivdatei. z.B

```
tar xzvf jdk-8u67-linux-x64.tar.gz
```

Installation aus einer Oracle Java RPM-Datei.

1. Rufen Sie die erforderliche RPM-Datei für die gewünschte Version von der Oracle Java-Downloadseite ab .
2. Mit dem rpm Befehl installieren. Zum Beispiel:

```
$ sudo rpm -ivh jdk-8u67-linux-x644.rpm
```

Java JDK oder JRE unter Windows installieren

Für Windows-Plattformen sind nur Oracle JDKs und JREs verfügbar. Das Installationsverfahren ist einfach:

1. Besuchen Sie die Oracle Java- Downloadseite :
2. Klicken Sie entweder auf die Schaltfläche JDK, JRE oder Server JRE. Beachten Sie, dass Sie für die Entwicklung mit Java ein JDK benötigen. Den Unterschied zwischen JDK und JRE finden Sie hier
3. Blättern Sie nach unten zu der Version, die Sie herunterladen möchten. (Im Allgemeinen wird die neueste Version empfohlen.)
4. Aktivieren Sie das Optionsfeld "Lizenzvereinbarung akzeptieren".
5. Laden Sie das Installationsprogramm für Windows x86 (32 Bit) oder Windows x64 (64 Bit) herunter.
6. Führen Sie das Installationsprogramm wie üblich für Ihre Windows-Version aus.

Eine alternative Methode zur Installation von Java unter Windows mithilfe der Eingabeaufforderung ist die Verwendung von Chocolately:

1. Installieren Sie Chocolately von https://chocolatey.org/
2. Öffnen Sie eine cmd-Instanz, drücken Sie beispielsweise Win + R und geben Sie "cmd" in das "Run" -Fenster ein, gefolgt von einem Enter.
3. Führen Sie in Ihrer cmd-Instanz den folgenden Befehl aus, um ein Java 8-JDK herunterzuladen und zu installieren:

```
C:\> choco install jdk8
```

Erste Schritte mit tragbaren Versionen

Es gibt Fälle, in denen Sie JDK / JRE auf einem System mit eingeschränkten Berechtigungen wie einer VM installieren möchten, oder Sie möchten mehrere Versionen oder Architekturen (x64 / x86) von JDK / JRE installieren und verwenden. Die Schritte bleiben bis zu dem Punkt erhalten, an dem Sie das Installationsprogramm (.EXE) herunterladen. Die folgenden Schritte sind wie folgt (Die Schritte gelten für JDK / JRE 7 und höher, bei älteren Versionen unterscheiden sie sich geringfügig in den Namen von Ordnern und Dateien):

1. Verschieben Sie die Datei an einen geeigneten Ort, an dem Ihre Java-Binärdateien dauerhaft gespeichert werden sollen.
2. Installieren Sie 7-Zip oder seine portable Version, wenn Sie über eingeschränkte Berechtigungen verfügen.
3. Extrahieren Sie mit 7-Zip die Dateien aus dem Java-Installationsprogramm EXE an den Speicherort.
4. Öffnen Sie die Eingabeaufforderung, indem Sie die Shift und Right-Click im Explorer-Ordner gedrückt halten oder von einem beliebigen Ort zu diesem Speicherort navigieren.
5. Navigieren Sie zu dem neu erstellten Ordner. jdk-7u25-windows-x64 lautet jdk-7u25-windows-x64 . cd jdk-7u25-windows-x64 also cd jdk-7u25-windows-x64 . Geben Sie dann die folgenden Befehle in der angegebenen Reihenfolge ein:

   ```
   cd .rsrc\JAVA_CAB10

   extrac32 111
   ```

6. Dadurch wird eine tools.zip Datei an diesem Ort erstellt. Extrahieren Sie die tools.zip mit 7-Zip, sodass die darin enthaltenen Dateien jetzt unter tools im selben Verzeichnis erstellt werden.
7. Führen Sie nun diese Befehle an der bereits geöffneten Eingabeaufforderung aus:

   ```
   cd tools

   for /r %x in (*.pack) do .\bin\unpack200 -r "%x" "%~dx%~px%~nx.jar"
   ```

8. Warten Sie, bis der Befehl abgeschlossen ist. Kopieren Sie den Inhalt der tools an den Ort, an dem sich Ihre Binärdateien befinden sollen.

Auf diese Weise können Sie alle Versionen von JDK / JRE installieren, die gleichzeitig installiert werden müssen.

Originaler Beitrag: http://stackoverflow.com/a/6571736/1448252

Installieren eines Java-JDK unter macOS

Oracle Java 7 und Java 8

Java 7 und Java 8 für macOS sind bei Oracle erhältlich. Diese Oracle-Seite beantwortet viele Fragen zu Java für Mac. Beachten Sie, dass Java 7 vor 7u25 aus Sicherheitsgründen von Apple deaktiviert wurde.

Im Allgemeinen erfordert Oracle Java (Version 7 und höher) einen Intel-basierten Mac, auf dem macOS 10.7.3 oder höher ausgeführt wird.

Installation von Oracle Java

Java 7 & 8 JDK- und JRE-Installationsprogramme für macOS können von der Oracle-Website

heruntergeladen werden:

- Java 8 - Java SE-Downloads
- Java 7 - Oracle Java-Archiv.

Nachdem Sie das entsprechende Paket heruntergeladen haben, doppelklicken Sie auf das Paket und führen Sie den normalen Installationsvorgang durch. Ein JDK sollte hier installiert werden:

```
/Library/Java/JavaVirtualMachines/<version>.jdk/Contents/Home
```

wo entspricht der installierten Version.

Befehlszeilenumschaltung

Wenn Java installiert ist, wird die installierte Version automatisch als Standard festgelegt. Um zwischen verschiedenen zu wechseln, verwenden Sie:

```
export JAVA_HOME=/usr/libexec/java_home -v 1.6 #Or 1.7 or 1.8
```

Die folgenden Funktionen können dem ~/.bash_profile (wenn Sie die Standard-Bash-Shell verwenden), um die Verwendung zu vereinfachen:

```
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 unter macOS

Bei älteren Versionen von macOS (10.11 El Capitan und früher) ist Apples Release von Java 6 vorinstalliert. Wenn installiert, kann es an dieser Stelle gefunden werden:

```
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

Beachten Sie, dass Java 6 schon lange nicht mehr verfügbar ist. Daher wird ein Upgrade auf eine neuere Version empfohlen. Weitere Informationen zur Neuinstallation von Apple Java 6 finden Sie auf der Oracle-Website.

Konfigurieren und Wechseln von Java-Versionen unter Linux mithilfe von Alternativen

Verwenden von Alternativen

Viele Linux-Distributionen verwenden den Befehl alternatives zum Wechseln zwischen verschiedenen Befehlsversionen. Sie können dies zum Wechseln zwischen verschiedenen auf einem Computer installierten Java-Versionen verwenden.

1. Setzen Sie in einer Befehlsshell $ JDK auf den Pfadnamen eines neu installierten JDK. z.B

   ```
   $ JDK=/Data/jdk1.8.0_67
   ```

2. Verwenden Sie alternatives --install , um die Befehle im Java SDK zu Alternativen hinzuzufügen:

   ```
   $ 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
   ```

   Und so weiter.

Nun können Sie wie folgt zwischen verschiedenen Versionen eines Java-Befehls wechseln:

```
$ 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
$
```

Weitere Informationen zur Verwendung von alternatives finden Sie in der manuellen Eingabe von Alternativen (8) .

Arch-basierte Installationen

Arch Linux-basierte Installationen enthalten den Befehl archlinux-java , um die Java-Versionen zu wechseln.

Installierte Umgebungen auflisten

```
$ archlinux-java status
Available Java environments:
  java-7-openjdk (default)
  java-8-openjdk/jre
```

Umschalten der aktuellen Umgebung

```
# archlinux-java set <JAVA_ENV_NAME>
```

Z.B:

```
# archlinux-java set java-8-openjdk/jre
```

Weitere Informationen finden Sie im Arch Linux Wiki

Überprüfung und Konfiguration nach der Installation unter Linux

Nach der Installation eines Java SDK sollten Sie prüfen, ob es einsatzbereit ist. Sie können dies tun, indem Sie diese beiden Befehle mit Ihrem normalen Benutzerkonto ausführen:

```
$ java -version
$ javac -version
```

Diese Befehle geben die Versionsinformationen für JRE und JDK (jeweils) aus, die sich im Befehlssuchpfad Ihrer Shell befinden. Suchen Sie nach der JDK / JRE-Versionszeichenfolge.

- Wenn einer der obigen Befehle fehlschlägt und "Befehl nicht gefunden" lautet, befindet sich die JRE oder das JDK überhaupt nicht im Suchpfad. Gehe **direkt** unter **PATH konfigurieren** .
- Wenn einer der obigen Befehle eine andere Versionszeichenfolge als erwartet anzeigt, müssen Sie entweder Ihren Suchpfad oder das "Alternative" -System anpassen. Gehen Sie zu **Alternativen prüfen**
- Wenn die richtigen Versionszeichenfolgen angezeigt werden, sind Sie fast fertig. **Fahren Sie mit Prüfen von JAVA_HOME fort**

PATH direkt konfigurieren

Wenn sich im Suchpfad momentan kein java oder javac befindet, können Sie dies einfach dem Suchpfad hinzufügen.

Finden Sie zunächst heraus, wo Sie Java installiert haben. siehe **Wo wurde Java installiert?** unten, wenn Sie Zweifel haben.

Wenn Sie davon ausgehen, dass bash Ihre Befehls-Shell ist, fügen Sie am Ende von ~/.bash_profile oder ~/.bashrc die folgenden Zeilen ein (wenn Sie Bash als Shell verwenden).

```
JAVA_HOME=<installation directory>
PATH=$JAVA_HOME/bin:$PATH

export JAVA_HOME
export PATH
```

... ersetzt <installation directory> durch den Pfadnamen für Ihr Java-Installationsverzeichnis. Beachten Sie, dass im obigen Verzeichnis davon ausgegangen wird, dass das Installationsverzeichnis ein bin Verzeichnis enthält und das bin Verzeichnis die Befehle java und javac enthält, die Sie verwenden java .

Als nächstes müssen Sie die Datei bearbeiten, die Sie gerade bearbeitet haben, sodass die Umgebungsvariablen für Ihre aktuelle Shell aktualisiert werden.

```
$ source ~/.bash_profile
```

Als nächstes wiederholen Sie die java und javac Versionsprüfungen. Wenn immer noch Probleme auftreten, verwenden Sie which java und which javac , um zu überprüfen, ob Sie die Umgebungsvariablen korrekt aktualisiert haben.

Abmelden und erneut anmelden, damit die aktualisierten Umgebungsvariablen für alle Shells ptopagiert werden. Sie sollten jetzt fertig sein.

Alternativen prüfen

Wenn java -version oder javac -version funktioniert hat, aber eine unerwartete Versionsnummer angegeben hat, müssen Sie überprüfen, woher die Befehle kommen. Verwenden Sie which und ls -l , um dies wie folgt herauszufinden:

```
$ ls -l `which java`
```

Wenn die Ausgabe folgendermaßen aussieht:

```
lrwxrwxrwx. 1 root root 22 Jul 30 22:18 /usr/bin/java -> /etc/alternatives/java
```

dann wird die alternatives Versionsumschaltung verwendet. Sie müssen entscheiden, ob Sie es weiterhin verwenden möchten, oder es einfach überschreiben, indem Sie den PATH direkt setzen.

- Konfigurieren und Wechseln von Java-Versionen unter Linux mithilfe von Alternativen
- Siehe "PATH direkt konfigurieren" oben.

Wo wurde Java installiert?

Java kann je nach Installationsmethode an verschiedenen Orten installiert werden.

- Die Oracle-RPMs setzen die Java-Installation in "/ usr / java".
- Auf Fedora lautet der Standardspeicherort "/ usr / lib / jvm".
- Wenn Java manuell aus ZIP- oder JAR-Dateien installiert wurde, kann die Installation an einem beliebigen Ort erfolgen.

Wenn Sie das Installationsverzeichnis nur schwer finden können, empfehlen wir, den Befehl find (oder slocate) zu finden. Zum Beispiel:

```
$ find / -name java -type f 2> /dev/null
```

Dies gibt Ihnen die Pfadnamen für alle Dateien namens java auf Ihrem System. (Die Umleitung des Standardfehlers zu "/ dev / null" unterdrückt Meldungen zu Dateien und Verzeichnissen, auf die Sie nicht zugreifen können.)

Installation von Oracle Java unter Linux mit der neuesten TAR-Datei

Führen Sie die folgenden Schritte aus, um Oracle JDK von der neuesten TAR-Datei zu installieren:

1. Laden Sie die neueste TAR-Datei hier herunter - Aktuell ist das Java SE Development Kit 8u112.
2. Sie benötigen Sudo-Privilegien:

   ```
   sudo su
   ```

3. Erstellen Sie ein Verzeichnis für die Jdk-Installation:

   ```
   mkdir /opt/jdk
   ```

4. Heruntergeladene Tar extrahieren:

   ```
   tar -zxf jdk-8u5-linux-x64.tar.gz -C /opt/jdk
   ```

5. Überprüfen Sie, ob die Dateien extrahiert wurden:

   ```
   ls /opt/jdk
   ```

6. Festlegen von Oracle JDK als Standard-JVM:

   ```
   update-alternatives --install /usr/bin/java java /opt/jdk/jdk1.8.0_05/bin/java 100
   ```

   und

   ```
   update-alternatives --install /usr/bin/javac javac /opt/jdk/jdk1.8.0_05/bin/javac 100
   ```

7. Überprüfen Sie die Java-Version:

   ```
   Java-Version
   ```

Erwartete Ausgabe:

```
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)
```

Java installieren (Standard Edition) online lesen:
https://riptutorial.com/de/java/topic/4754/java-installieren--standard-edition-

Kapitel 61: Java Native Access

Examples

Einführung in JNA

Was ist JNA?

Java Native Access (JNA) ist eine von der Community entwickelte Bibliothek, die Java-Programmen den einfachen Zugriff auf native freigegebene Bibliotheken (.dll Dateien unter Windows, .so Dateien unter Unix) ermöglicht.

Wie kann ich es benutzen?

- Laden Sie zunächst die neueste Version von JNA herunter und verweisen Sie auf jna.jar im CLASSPATH Ihres Projekts.
- Zweitens kopieren, kompilieren und führen Sie den folgenden Java-Code aus

  In dieser Einführung nehmen wir an, dass die native Plattform Windows ist. Wenn Sie auf einer anderen Plattform laufen, ersetzen Sie einfach die Zeichenfolge "msvcrt" durch die Zeichenfolge "c" im nachstehenden Code.

Das kleine Java-Programm unten druckt eine Nachricht auf der Konsole, indem die C printf Funktion printf wird.

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 !");
    }
}
```

Wohin jetzt?

Springen Sie in ein anderes Thema oder springen Sie auf die offizielle Seite .

Java Native Access online lesen: https://riptutorial.com/de/java/topic/5244/java-native-access

Kapitel 62: Java SE 7-Funktionen

Einführung

In diesem Thema finden Sie eine Zusammenfassung der neuen Funktionen, die der Java-Programmiersprache in Java SE 7 hinzugefügt wurden. Es gibt viele andere neue Funktionen in anderen Bereichen wie JDBC und Java Virtual Machine (JVM), die nicht behandelt werden in diesem Punkt.

Bemerkungen

Verbesserungen in Java SE 7

Examples

Neue Funktionen der Programmiersprache Java SE 7

- Binäre Literale : Die Integraltypen (Byte, Kurz, Int und Lang) können auch mit dem Binärzahlensystem ausgedrückt werden. Um ein binäres Literal anzugeben, fügen Sie der Zahl das Präfix 0b oder 0B hinzu.
- Zeichenfolgen in switch-Anweisungen : Sie können ein String-Objekt im Ausdruck einer switch-Anweisung verwenden
- Die try-with-resources-Anweisung : Die try-with-resources-Anweisung ist eine try-Anweisung, die eine oder mehrere Ressourcen deklariert. Eine Ressource ist ein Objekt, das geschlossen werden muss, nachdem das Programm damit fertig ist. Die try-with-resources-Anweisung stellt sicher, dass jede Ressource am Ende der Anweisung geschlossen wird. Jedes Objekt, das java.lang.AutoCloseable implementiert, das alle Objekte enthält, die java.io.Closeable implementieren, kann als Ressource verwendet werden.
- Mehrere Ausnahmetypen abfangen und Ausnahmen mit verbesserter Typprüfung erneut auslösen : Ein einzelner Fangblock kann mehrere Ausnahmetypen behandeln. Diese Funktion kann die Duplizierung von Code reduzieren und die Versuchung verringern, eine übermäßig breite Ausnahme abzufangen.
- Unterstriche in numerischen Literalen : In einem numerischen Literal können beliebig viele Unterstriche (_) an einer beliebigen Stelle zwischen den Ziffern stehen. Mit dieser Funktion können Sie z. B. Zifferngruppen in numerischen Literalen voneinander trennen, wodurch die Lesbarkeit Ihres Codes verbessert wird.
- Typinferenz für die Erstellung generischer Instanzen : Sie können die zum Aufrufen des Konstruktors einer generischen Klasse erforderlichen Typargumente durch einen leeren Satz von Typparametern (<>) ersetzen, sofern der Compiler die Typargumente aus dem Kontext ableiten kann. Dieses Winkelpaar wird informell als Diamant bezeichnet.
- Verbesserte Compiler-Warnungen und -Fehler bei der Verwendung nichtformbarer Parameter mit Varargs-Methoden

Binäre Literale

```
// 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;
```

Die try-with-resources-Anweisung

Das Beispiel liest die erste Zeile aus einer Datei. Es verwendet eine Instanz von BufferedReader , um Daten aus der Datei zu lesen. BufferedReader ist eine Ressource, die geschlossen werden muss, nachdem das Programm damit beendet wurde:

```
static String readFirstLineFromFile(String path) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

In diesem Beispiel ist die in der try-with-resources-Anweisung deklarierte Ressource ein BufferedReader . Die Deklarationsanweisung wird unmittelbar nach dem Schlüsselwort try in Klammern angezeigt. Die Klasse BufferedReader in Java SE 7 und höher implementiert die Schnittstelle java.lang.AutoCloseable . Da die BufferedReader Instanz in einer try-with-resource-Anweisung deklariert ist, wird sie unabhängig davon geschlossen, ob die try-Anweisung normal oder abrupt abgeschlossen wird (als Ergebnis der Methode BufferedReader.readLine , die eine IOException).

Unterstriche in numerischen Literalen

Das folgende Beispiel zeigt andere Möglichkeiten, wie Sie den Unterstrich in numerischen Literalen verwenden können:

```
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;
```

Sie können Unterstriche nur zwischen Ziffern setzen. Unterstriche können an folgenden Stellen nicht gesetzt werden:

- Am Anfang oder Ende einer Nummer
- Angrenzend an einen Dezimalpunkt in einem Fließkomma-Literal
- Vor einem Suffix von F oder L
- An Stellen, an denen eine Ziffernfolge erwartet wird

Typinferenz für die Generierung einer generischen Instanz

Sie können verwenden

```
Map<String, List<String>> myMap = new HashMap<>();
```

anstatt

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

Sie können jedoch nicht verwenden

```
List<String> list = new ArrayList<>();
list.add("A");

  // The following statement should fail since addAll expects
  // Collection<? extends String>
```

```
list.addAll(new ArrayList<>());
```

weil es nicht kompilieren kann. Beachten Sie, dass der Diamant häufig in Methodenaufrufen funktioniert. Es wird jedoch empfohlen, den Diamanten hauptsächlich für Variablendeklarationen zu verwenden.

Zeichenfolgen in switch-Anweisungen

```
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;
}
```

Java SE 7-Funktionen online lesen: https://riptutorial.com/de/java/topic/8272/java-se-7-funktionen

Kapitel 63: Java SE 8-Funktionen

Einführung

In diesem Thema finden Sie eine Zusammenfassung der neuen Funktionen, die der Java-Programmiersprache in Java SE 8 hinzugefügt wurden. Es gibt viele andere neue Funktionen in anderen Bereichen wie JDBC und Java Virtual Machine (JVM), die nicht behandelt werden in diesem Punkt.

Bemerkungen

Referenz: Verbesserungen in Java SE 8

Examples

Neue Funktionen der Programmiersprache Java SE 8

- Lambda Expressions , eine neue Sprachfunktion, wurde in dieser Version eingeführt. Sie ermöglichen es Ihnen, Funktionalität als Methodenargument oder Code als Daten zu behandeln. Mit Lambda-Ausdrücken können Sie Instanzen von Schnittstellen mit einer Methode (als funktionale Schnittstellen bezeichnet) kompakter ausdrücken.
    - Methodenreferenzen bieten leicht lesbare Lambda-Ausdrücke für Methoden, die bereits einen Namen haben.
    - Standardmethoden ermöglichen das Hinzufügen neuer Funktionen zu den Schnittstellen von Bibliotheken und gewährleisten die binäre Kompatibilität mit Code, der für ältere Versionen dieser Schnittstellen geschrieben wurde.
    - Neue und erweiterte APIs, die Lambda-Ausdrücke und Streams in Java SE 8 nutzen, beschreiben neue und erweiterte Klassen, die Lambda-Ausdrücke und Streams nutzen.
- Verbesserte Typinferenz - Der Java-Compiler nutzt die Zieltypisierung, um die Typparameter eines generischen Methodenaufrufs abzuleiten. Der Zieltyp eines Ausdrucks ist der Datentyp, den der Java-Compiler erwartet, abhängig davon, wo der Ausdruck angezeigt wird. Sie können beispielsweise den Zieltyp einer Zuweisungsanweisung für die Typinferenz in Java SE 7 verwenden. In Java SE 8 können Sie jedoch den Zieltyp für die Typinferenz in mehr Kontexten verwenden.
    - Targeting in Lambda-Ausdrücken
    - Typ Inferenz
- Wiederholende Annotationen bieten die Möglichkeit, denselben Annotationstyp mehr als einmal auf dieselbe Deklaration oder Typverwendung anzuwenden.
- Typanmerkungen bieten die Möglichkeit, eine Anmerkung überall dort anzuwenden, wo ein Typ verwendet wird, nicht nur bei einer Deklaration. In Verbindung mit einem steckbaren Typsystem ermöglicht diese Funktion eine verbesserte Typprüfung Ihres Codes.
- Reflexion von Methodenparametern - Sie können die Namen der Formalparameter einer beliebigen Methode oder eines Konstruktors mit der Methode java.lang.reflect.Executable.getParameters . (Die Klassen Method und Constructor erweitern die Klasse Executable und erben daher die Methode Executable.getParameters) .class Dateien speichern jedoch standardmäßig keine formalen Parameternamen. Um formale Parameternamen in einer bestimmten .class Datei zu speichern und dadurch der Reflection-API das Abrufen von formalen Parameternamen zu ermöglichen, kompilieren Sie die Quelldatei mit der Option -parameters des Javac-Compilers.
- Date-Time-API - Neue Zeit-API in java.time . In diesem Fall müssen Sie keine Zeitzone festlegen.

Java SE 8-Funktionen online lesen: https://riptutorial.com/de/java/topic/8267/java-se-8-funktionen

Kapitel 64: Java Virtual Machine (JVM)

Examples

Das sind die Grundlagen.

JVM ist eine **abstrakte Computer-** oder **virtuelle Maschine** , die sich in Ihrem RAM befindet. Es verfügt über eine plattformunabhängige Ausführungsumgebung, die Java-Bytecode in nativen Maschinencode interpretiert. (Javac ist Java Compiler, der Ihren Java-Code in Bytecode kompiliert.)

In der JVM wird ein Java-Programm ausgeführt, das auf die zugrunde liegende physische Maschine abgebildet wird. Es ist ein Programmierwerkzeug in JDK.

(*Byte code* ist plattformunabhängiger Code, der auf jeder Plattform ausgeführt wird, und *Machine code* ist plattformspezifischer Code, der nur auf bestimmten Plattformen wie Windows oder Linux ausgeführt wird; er hängt von der Ausführung ab.)

Einige der Komponenten: -

- Class Loder - Laden Sie die .class-Datei in den Arbeitsspeicher.
- Bytecode Verifier - Überprüfen Sie, ob in Ihrem Code Verstöße gegen Zugriffsbeschränkungen vorliegen.
- Execution Engine - konvertiert den Bytecode in ausführbaren Maschinencode.
- JIT (just in time) - JIT ist Teil der JVM, die zur Verbesserung der Leistung von JVM verwendet wurde. Sie wird Java-Bytecode während der Ausführungszeit dynamisch kompilieren oder in nativen Maschinencode übersetzen.

(Bearbeitet)

Java Virtual Machine (JVM) online lesen: https://riptutorial.com/de/java/topic/8110/java-virtual-machine--jvm-

Kapitel 65: Java-Agenten

Examples

Klassen mit Agenten ändern

Stellen Sie zunächst sicher, dass der verwendete Agent in der Manifest.mf die folgenden Attribute aufweist:

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

Beim Starten eines Java-Agenten kann der Agent auf die Klasse Instrumentation zugreifen. Mit Instrumentation können Sie *addTransformer (ClassFileTransformer Transformator) aufrufen* . Mit ClassFileTransformers können Sie die Bytes von Klassen neu schreiben. Die Klasse verfügt nur über eine einzige Methode, die den ClassLoader bereitstellt, der die Klasse, den Namen der Klasse, eine Instanz von java.lang.Class, die ProtectionDomain-Instanz und schließlich die Bytes der Klasse selbst lädt.

Es sieht aus wie das:

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

Das Ändern einer Klasse nur aus Bytes kann sehr lange dauern. Um dies zu beheben, gibt es Bibliotheken, mit denen die Klassenbytes in etwas Nutzbareres umgewandelt werden können.

In diesem Beispiel verwende ich ASM, aber andere Alternativen wie Javassist und BCEL weisen ähnliche Funktionen auf.

```
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;
}
```

Von hier aus können Änderungen am ClassNode-Objekt vorgenommen werden. Dies macht das Ändern des Feld- / Methodenzugriffs unglaublich einfach. Mit der Tree-API von ASM ist das Ändern des Bytecodes von Methoden ein Kinderspiel.

Sobald die Änderungen abgeschlossen sind, können Sie den ClassNode mit der folgenden Methode wieder in Byte konvertieren und in der *transform*- Methode zurückgeben:

```
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;
}
```

Hinzufügen eines Agenten zur Laufzeit

Agenten können zur Laufzeit einer JVM hinzugefügt werden. Um einen Agenten zu laden, müssen Sie die *VirtualMachine.attatch (String-ID)* der Attach-API verwenden. Sie können dann eine kompilierte Agenten-Dose mit der folgenden Methode laden:

```
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);
    }
}
```

Dies ruft im geladenen Agenten nicht *premain ((String agentArgs, Instrumentation inst)* auf, sondern stattdessen *agentmain (String agentArgs, Instrumentation inst)* . Dazu muss *Agent-Class* im Agenten Manifest.mf festgelegt werden.

Einrichten eines Basisagenten

Die Premain-Klasse enthält die Methode *"premain (String agentArgs Instrumentation inst)".*

Hier ist ein Beispiel:

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

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

Wenn Sie es in eine JAR-Datei kompilieren, öffnen Sie das Manifest und stellen Sie sicher, dass es über das Attribut "Premain-Class" verfügt.

Hier ist ein Beispiel:

```
Premain-Class: PremainExample
```

Um den Agenten mit einem anderen Java-Programm "myProgram" zu verwenden, müssen Sie den Agenten in den JVM-Argumenten definieren:

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

Java-Agenten online lesen: https://riptutorial.com/de/java/topic/1265/java-agenten

Kapitel 66: JavaBean

Einführung

JavaBeans (TM) ist ein Muster zum Entwerfen von Java-Klassen-APIs, mit dem Instanzen (Beans) in verschiedenen Kontexten und mit verschiedenen Tools verwendet werden können, *ohne* Java-Code explizit zu schreiben. Die Muster bestehen aus Konventionen zur Definition von Getern und Setters für *Eigenschaften* , zur Definition von Konstruktoren und zur Definition von Ereignis-Listener-APIs.

Syntax

- **JavaBean-Eigenschaftsbenennungsregeln**
- Wenn die Eigenschaft nicht boolean ist, muss das Präfix der Getter-Methode get sein. GetSize () ist beispielsweise ein gültiger JavaBeans-Getter-Name für eine Eigenschaft mit dem Namen "size". Beachten Sie, dass Sie keine Variable mit dem Namen size benötigen. Der Name der Eigenschaft wird von den Get- und Setzern abgeleitet, nicht durch Variablen in Ihrer Klasse. Was Sie von getSize () zurückgeben, liegt bei Ihnen.
- Wenn die Eigenschaft boolean ist, lautet das Präfix der Getter-Methode entweder get oder is. Zum Beispiel sind getStopped () oder isStopped () beide gültige JavaBeans-Namen für eine boolesche Eigenschaft.
- Das Präfix der Setter-Methode muss festgelegt werden. SetSize () ist beispielsweise der gültige JavaBean-Name für eine Eigenschaft mit dem Namen size.
- Um den Namen einer Getter- oder Setter-Methode zu vervollständigen, ändern Sie den ersten Buchstaben des Eigenschaftennamens in Großbuchstaben und hängen Sie ihn an das entsprechende Präfix an (Get, Is oder Set).
- Setter-Methodensignaturen müssen als public markiert werden, mit einem ungültigen Rückgabetyp und einem Argument, das den Eigenschaftstyp darstellt.
- Getter-Methodensignaturen müssen als public markiert werden, ohne Argumente und mit einem Rückgabetyp, der dem Argumenttyp der Setter-Methode für diese Eigenschaft entspricht.
- **Namensregeln für den JavaBean-Listener**
- Listener-Methodennamen, die zum "Registrieren" eines Listeners bei einer Ereignisquelle verwendet werden, müssen das Präfix add gefolgt vom Listener-Typ verwenden. Beispielsweise ist addActionListener () ein gültiger Name für eine Methode, die eine Ereignisquelle zulassen muss, damit andere sich für Action-Ereignisse registrieren können.
- Listener-Methodennamen, die zum Entfernen ("Aufheben der Registrierung") eines Listeners verwendet werden, müssen das Präfix remove und dann den Listener-Typ verwenden (unter Verwendung derselben Regeln wie die Registrierungs-Add-Methode).
- Der Typ des Listeners, der hinzugefügt oder entfernt werden soll, muss als Argument an die Methode übergeben werden.
- Listener-Methodennamen müssen mit dem Wort "Listener" enden.

Bemerkungen

Damit eine Klasse ein Java-Bean sein kann, muss dieser Standard eingehalten werden - zusammenfassend:

- Alle seine Eigenschaften müssen privat sein und nur über Getter und Setter zugänglich sein.
- Es muss einen öffentlichen Konstruktor ohne Argumente enthalten.
- Muss die java.io.Serializable Schnittstelle implementieren.

Examples

Grundlegende Java Bean

```
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;
    }
}
```

JavaBean online lesen: https://riptutorial.com/de/java/topic/8157/javabean

Kapitel 67: Java-Bereitstellung

Einführung

Es gibt eine Vielzahl von Technologien zum "Packen" von Java-Anwendungen, Webanwendungen usw. zur Bereitstellung auf der Plattform, auf der sie ausgeführt werden. Sie reichen von einfachen Bibliotheken oder ausführbaren JAR Dateien, WAR und EAR Dateien bis zu Installationsprogrammen und eigenständigen ausführbaren Dateien.

Bemerkungen

Grundsätzlich kann ein Java-Programm durch Kopieren einer kompilierten Klasse (z. B. einer ".class" -Datei) oder einer Verzeichnisstruktur mit kompilierten Klassen implementiert werden. Java wird jedoch normalerweise auf eine der folgenden Arten bereitgestellt:

- Durch Kopieren einer JAR-Datei oder Sammlung von JAR-Dateien in das System, auf dem sie ausgeführt werden; zB mit javac .
- Durch Kopieren oder Hochladen einer WAR-, EAR- oder ähnlichen Datei in einen "Servlet-Container" oder "Anwendungsserver".
- Durch Ausführen eines Anwendungsinstallationsprogramms, das das obige automatisiert. Das Installationsprogramm installiert möglicherweise auch eine eingebettete JRE.
- Legen Sie die JAR-Dateien für die Anwendung auf einem Webserver ab, damit sie mit Java WebStart gestartet werden können.

Das Beispiel zum Erstellen von JAR-, WAR- und EAR-Dateien fasst die verschiedenen Möglichkeiten zum Erstellen dieser Dateien zusammen.

Es gibt zahlreiche Open-Source- und kommerzielle Tools für "Installer-Generator" und "EXE-Generator" für Java. Ebenso gibt es Tools zum Verschleiern von Java-Klassendateien (um das Reverse Engineering schwieriger zu machen) und zum Hinzufügen von Laufzeitlizenzen. Dies ist alles außerhalb der Dokumentation für "Java Programming Language".

Examples

Erstellen einer ausführbaren JAR-Datei über die Befehlszeile

Um ein Glas zu machen, benötigen Sie eine oder mehrere Klassendateien. Dies sollte eine Hauptmethode haben, wenn es mit einem Doppelklick ausgeführt werden soll.

Für dieses Beispiel verwenden wir:

```
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);
    }

}
```

Es wurde HelloWorld.java genannt

Als nächstes wollen wir dieses Programm kompilieren.

Sie können jedes Programm verwenden, das Sie dazu möchten. Informationen zum Ausführen über die Befehlszeile finden Sie in der Dokumentation zum Kompilieren und Ausführen Ihres ersten Java-Programms.

Sobald Sie HelloWorld.class haben, erstellen Sie einen neuen Ordner und rufen Sie ihn an, wie Sie möchten.

Erstellen Sie eine weitere Datei namens manifest.txt und fügen Sie sie ein

```
Main-Class: HelloWorld
Class-Path: HelloWorld.jar
```

Legen Sie es mit HelloWorld.class in demselben Ordner ab
Verwenden Sie die Befehlszeile, um Ihr aktuelles Verzeichnis (cd C:\Your\Folder\Path\Here unter Windows) zu Ihrem Ordner zu machen.

Verwenden Sie Terminal und wechseln Sie in das Verzeichnis (cd /Users/user/Documents/Java/jarfolder auf Mac) Ihres Ordners

Wenn dies abgeschlossen ist, geben Sie jar -cvfm HelloWorld.jar manifest.txt HelloWorld.class und drücken Sie die Eingabetaste. Dadurch wird eine JAR-Datei (im Ordner mit Ihrem Manifest und HelloWorld.class) erstellt. Dabei werden die angegebenen .class-Dateien mit dem Namen HelloWorld.jar verwendet. Informationen zu den Optionen (wie -m und -v) finden Sie im Abschnitt Syntax.
Wechseln Sie nach diesen Schritten in Ihr Verzeichnis mit der Manifestdatei, und Sie sollten HelloWorld.jar finden
Wenn Sie darauf klicken, sollte Hello, World in einem Textfeld angezeigt werden.

Erstellen von JAR-, WAR- und EAR-Dateien

Die Dateitypen JAR, WAR und EAR sind im Wesentlichen ZIP-Dateien mit einer "Manifest" -Datei und (für WAR- und EAR-Dateien) eine bestimmte interne Verzeichnis- / Dateistruktur.

Die empfohlene Methode zum Erstellen dieser Dateien ist die Verwendung eines Java-spezifischen Build-Tools, das die Anforderungen für die jeweiligen Dateitypen "versteht". Wenn Sie kein Build-Tool verwenden, ist die IDE-Option "Export" die nächste Option.

(*Anmerkung der Redaktion: Die Beschreibungen zum Erstellen dieser Dateien finden Sie am besten in der Dokumentation der jeweiligen Werkzeuge. Fügen Sie sie dort ein. Bitte zeigen Sie etwas Selbstbeherrschung und KEINEN Schuhhorn in diesem Beispiel!*)

Erstellen von JAR- und WAR-Dateien mit Maven

Beim Erstellen eines JAR oder WAR mit Maven müssen Sie nur das richtige <packaging> -Element in die POM-Datei einfügen. z.B,

```
<packaging>jar</packaging>
```

oder

```
<packaging>war</packaging>
```

Für mehr Details. Maven kann so konfiguriert werden, dass "ausführbare" JAR-Dateien erstellt werden, indem die erforderlichen Informationen zur Einstiegspunktklasse und zu externen Abhängigkeiten als Plugin-Eigenschaften für das Maven-Jar-Plugin hinzugefügt werden. Es gibt sogar ein Plugin zum Erstellen von "uberJAR" -Dateien, die eine Anwendung und ihre Abhängigkeiten in einer einzigen JAR-Datei kombinieren.

Weitere Informationen finden Sie in der Maven-Dokumentation (http://www.riptutorial.com/topic/898).

Erstellen von JAR-, WAR- und EAR-Dateien mit Ant

Das Ant-Build-Tool hat separate "Aufgaben" zum Erstellen von JAR, WAR und EAR. Weitere Informationen finden Sie in der Ant-Dokumentation (http://www.riptutorial.com/topic/4223).

Erstellen von JAR-, WAR- und EAR-Dateien mithilfe einer IDE

Die drei beliebtesten Java-IDEs verfügen alle über eine integrierte Unterstützung für die Erstellung von Implementierungsdateien. Die Funktionalität wird oft als "Exportieren" bezeichnet.

- Eclipse - http://www.riptutorial.com/topic/1143
- NetBeans - http://www.riptutorial.com/topic/5438
- Intellij-IDEA - Exportieren

Erstellen von JAR-, WAR- und EAR-Dateien mit dem Befehl `jar` .

Es ist auch möglich, diese Dateien mit dem Befehl jar "von Hand" zu erstellen. Es muss lediglich ein Dateibaum mit den richtigen Komponentendateien an der richtigen Stelle zusammengefügt werden, eine Manifestdatei erstellt und jar , um die JAR-Datei zu erstellen.

Weitere Informationen finden Sie im jar Befehlsthema (JAR-Dateien erstellen und ändern)

Einführung in Java Web Start

Die Oracle Java-Tutorials fassen Web Start wie folgt zusammen:

> Die Java Web Start-Software ermöglicht das Starten von Anwendungen mit vollem Funktionsumfang mit einem einzigen Klick. Benutzer können Anwendungen herunterladen und starten, z. B. ein komplettes Tabellenkalkulationsprogramm oder einen Internet-Chat-Client, ohne lange Installationsvorgänge durchzuführen.

Weitere Vorteile von Java Web Start sind Unterstützung für signierten Code und explizite Deklaration von Plattformabhängigkeiten sowie Unterstützung für die Zwischenspeicherung von Code und die Bereitstellung von Anwendungsupdates.

Java Web Start wird auch als JavaWS und JAWS bezeichnet. Die wichtigsten Informationsquellen sind:

- Die Java-Tutorials - Lektion: Java Web Start
- Java Web Start Guide
- Java Web Start-FAQ
- JNLP-Spezifikation
- javax.jnlp API-Dokumentation
- Java Web Start-Entwickler-Site

Voraussetzungen

Auf hoher Ebene verteilt Web Start Java-Anwendungen, die als JAR-Dateien gepackt sind, von einem Remote-Webserver aus. Die Voraussetzungen sind:

- Eine bereits vorhandene Java-Installation (JRE oder JDK) auf dem Zielcomputer, auf dem die Anwendung ausgeführt werden soll. Java 1.2.2 oder höher ist erforderlich:
  - Ab Java 5.0 ist die Web Start-Unterstützung im JRE / JDK enthalten.
  - Bei früheren Versionen wird die Web Start-Unterstützung separat installiert.
  - Die Web Start-Infrastruktur enthält Javascript, das auf einer Webseite enthalten sein kann, um den Benutzer bei der Installation der erforderlichen Software zu unterstützen.

- Der Webserver, der die Software hostet, muss für den Zielcomputer zugänglich sein.
- Wenn der Benutzer eine Web Start-Anwendung mit einem Link auf einer Webseite startet, gilt Folgendes:
    - Sie benötigen einen kompatiblen Webbrowser und
    - Für moderne (sichere) Browser müssen sie wissen, wie sie dem Browser mitteilen sollen, dass Java ausgeführt werden soll, ohne die Sicherheit des Webbrowsers zu beeinträchtigen.

Eine Beispiel-JNLP-Datei

Das folgende Beispiel soll die grundlegende Funktionalität von JNLP veranschaulichen.

```
<?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>
```

Wie Sie sehen, ist eine JNLP-Datei XML-basiert, und die Informationen sind alle im <jnlp> -Element enthalten.

- Das spec Attribut gibt die Version der JNPL-Spezifikation an, mit der diese Datei übereinstimmt.
- Das codebase Attribut gibt die Basis-URL zum Auflösen relativer href URLs im Rest der Datei an.
- Das href Attribut gibt die endgültige URL für diese JNLP-Datei an.
- Das Element <information> enthält Metadaten der Anwendung einschließlich Titel, Autoren, Beschreibung und Hilfeseite.
- Das <resources> -Element beschreibt die Abhängigkeiten für die Anwendung, einschließlich der erforderlichen Java-Version, Betriebssystemplattform und JAR-Dateien.
- Das Element <application-desc> (oder <applet-desc>) enthält Informationen, die zum Starten der Anwendung erforderlich sind.

Webserver einrichten

Der Webserver muss so konfiguriert sein, dass application/x-java-jnlp-file als MIME-Typ für .jnlp Dateien verwendet wird.

Die JNLP-Datei und die JAR-Dateien der Anwendung müssen auf dem Webserver installiert sein, damit sie unter Verwendung der in der JNLP-Datei angegebenen URLs verfügbar sind.

Aktivieren des Starts über eine Webseite

Wenn die Anwendung über einen Weblink gestartet werden soll, muss die Seite, die den Link enthält, auf dem Webserver erstellt werden.

- Wenn Sie davon ausgehen können, dass Java Web Start bereits auf dem Computer des Benutzers installiert ist, muss die Webseite lediglich einen Link zum Starten der Anwendung enthalten. Zum Beispiel.

  ```
  <a href="https://www.example.com/demo_webstart.jnlp">Launch the application</a>
  ```

- Andernfalls sollte die Seite auch Skripts enthalten, um die Art des von dem Benutzer verwendeten Browsers zu ermitteln und die erforderliche Version von Java herunterzuladen und zu installieren.

HINWEIS: Es ist keine gute Idee, Benutzer zu ermutigen, Java auf diese Weise zu installieren, oder sogar Java in ihren Webbrowsern zu aktivieren, damit der Start von JNLP-Webseiten funktioniert.

Starten Sie Web Start-Anwendungen über die Befehlszeile

Die Anweisungen zum Starten einer Web Start-Anwendung über die Befehlszeile sind einfach. Wenn der Benutzer eine Java 5.0-JRE oder ein JDK installiert hat, müssen Sie einfach Folgendes ausführen:

```
$ javaws <url>
```

Dabei ist <url> die URL für die JNLP-Datei auf dem Remote-Server.

Erstellen eines UberJAR für eine Anwendung und ihre Abhängigkeiten

Eine allgemeine Anforderung an eine Java-Anwendung ist, dass sie durch Kopieren einer einzelnen Datei implementiert werden kann. Für einfache Anwendungen, die nur von den Standard-Java SE-Klassenbibliotheken abhängig sind, wird diese Anforderung erfüllt, indem eine JAR-Datei erstellt wird, die alle (kompilierten) Anwendungsklassen enthält.

Die Dinge sind nicht so einfach, wenn die Anwendung von Bibliotheken von Drittanbietern abhängt. Wenn Sie einfach Abhängigkeits-JAR-Dateien in eine Anwendungs-JAR einfügen, kann der Standard-Java-Klassenlader die Bibliotheksklassen nicht finden und Ihre Anwendung wird nicht gestartet. Stattdessen müssen Sie eine einzelne JAR-Datei erstellen, die die Anwendungsklassen und zugehörigen Ressourcen zusammen mit den Abhängigkeitsklassen und -ressourcen enthält. Diese müssen als einzelner Namespace organisiert werden, damit der Classloader durchsucht werden kann.

Eine solche JAR-Datei wird häufig als UberJAR bezeichnet.

Erstellen eines UberJAR mit dem Befehl "jar"

Das Verfahren zum Erstellen eines UberJAR ist unkompliziert. (Ich werde zur Vereinfachung Linux-Befehle verwenden. Die Befehle sollten für Mac OS und für Windows ähnlich sein.)

1. Erstellen Sie ein temporäres Verzeichnis und wechseln Sie in das Verzeichnis.

   ```
   $ mkdir tempDir
   $ cd tempDir
   ```

2. Verwenden Sie für jede abhängige JAR-Datei *in der umgekehrten Reihenfolge* , in der sie im Klassenpfad der Anwendung angezeigt werden müssen, den jar Befehl, um die JAR-Datei in das temporäre Verzeichnis zu entpacken.

   ```
   $ jar -xf <path/to/file.jar>
   ```

   Dadurch für mehrere JAR - Dateien werden Inhalte der JAR - Dateien *überlagern.*

3. Kopieren Sie die Anwendungsklassen aus der Build-Struktur in das temporäre Verzeichnis

```
$ cp -r path/to/classes .
```

4. Erstellen Sie den UberJAR aus dem Inhalt des temporären Verzeichnisses:

```
$ jar -cf ../myApplication.jar
```

Wenn Sie eine ausführbare JAR-Datei erstellen, fügen Sie wie hier beschrieben ein entsprechendes MANIFEST.MF hinzu.

Erstellen eines UberJAR mit Maven

Wenn Ihr Projekt mit Maven erstellt wurde, können Sie mit den Plugins "maven-assembly" oder "maven-shade" ein UberJAR erstellen. Weitere Informationen finden Sie im Thema Maven Assembly (in der Maven- Dokumentation).

Die Vor- und Nachteile von UberJARs

Einige Vorteile von UberJARs sind selbstverständlich:

- Ein UberJAR ist leicht zu verteilen.
- Sie können die Bibliotheksabhängigkeiten für eine UberJAR nicht aufheben, da die Bibliotheken eigenständig sind.

Wenn Sie zum Erstellen des UberJAR ein geeignetes Werkzeug verwenden, haben Sie außerdem die Möglichkeit, nicht verwendete Bibliotheksklassen aus der JAR-Datei auszuschließen. Dies geschieht jedoch normalerweise durch statische Analyse der Klassen. Wenn in Ihrer Anwendung Reflexions-, Anmerkungsverarbeitung und ähnliche Techniken verwendet werden, müssen Sie darauf achten, dass Klassen nicht falsch ausgeschlossen werden.

UberJARs haben auch einige Nachteile:

- Wenn Sie viele UberJARs mit den gleichen Abhängigkeiten haben, enthält jeder eine Kopie der Abhängigkeiten.
- Einige Open - Source - Bibliotheken haben Lizenzen , die [1] ihre Verwendung in einem UberJAR ausschließen *kann*.

---

1 - Bei einigen Open-Source-Bibliothekslizenzen können Sie die Bibliothek nur verwenden, wenn der Endbenutzer eine Version der Bibliothek durch eine andere ersetzen kann. UberJARs können das Ersetzen von Versionsabhängigkeiten erschweren.

Java-Bereitstellung online lesen: https://riptutorial.com/de/java/topic/6840/java-bereitstellung

Kapitel 68: Java-Code dokumentieren

Einführung

Die Dokumentation für Java-Code wird häufig mit Javadoc erstellt . Javadoc wurde von Sun Microsystems erstellt, um API-Dokumentation im HTML-Format aus Java-Quellcode zu erstellen. Die Verwendung des HTML-Formats bietet die Möglichkeit, verwandte Dokumente miteinander zu verknüpfen.

Syntax

- / ** - Start von JavaDoc für eine Klasse, ein Feld, eine Methode oder ein Paket
- @author // Um dem Autor der Klasse, des Interfaces oder der Aufzählung einen Namen zu geben. Es ist notwendig.
- @version // Die Version dieser Klasse, Schnittstelle oder Enumeration. Es ist notwendig. Sie können Makros wie% I% oder% G% für Ihre Quellcodeverwaltungssoftware verwenden, um an der Kasse auszufüllen.
- @param // Zeigt die Argumente (Parameter) einer Methode oder eines Konstruktors an. Geben Sie für jeden Parameter ein @param-Tag an.
- @return // Um die Rückgabetypen für nicht-ungültige Methoden anzuzeigen.
- @exception // Zeigt an, welche Ausnahmen von der Methode oder vom Konstruktor ausgelöst werden könnten. Ausnahmen , die gefangen werden **muss** sollte hier aufgeführt werden. Wenn Sie möchten, können Sie auch diejenigen hinzufügen, die nicht abgefangen werden müssen, wie ArrayIndexOutOfBoundsException. Geben Sie eine @ Ausnahme für jede Ausnahme an, die ausgelöst werden kann.
- @throws // Wie @exception.
- @see // Links zu einer Methode, einem Feld, einer Klasse oder einem Paket. Verwenden Sie in Form von package.Class # etwas.
- @since // Bei dieser Methode wurde ein Feld oder eine Klasse hinzugefügt. Zum Beispiel JDK-8 für eine Klasse wie java.util.Optional <T> .
- @serial, @serialField, @serialData // Wird verwendet, um die serialVersionUID anzuzeigen.
- @deprecated // Eine Klasse, eine Methode oder ein Feld als veraltet markieren. Ein Beispiel wäre java.io.StringBufferInputStream . Eine vollständige Liste der vorhandenen veralteten Klassen finden Sie hier .
- {@link} // Ähnlich wie @see, kann jedoch mit benutzerdefiniertem Text verwendet werden: {@link #setDefaultCloseOperation (int closeOperation). Weitere Informationen finden Sie unter JFrame # setDefaultCloseOperation.
- {@linkplain} // Ähnlich wie {@link}, jedoch ohne Code-Schriftart.
- {@code} // Für wörtlichen Code, z. B. HTML-Tags. Zum Beispiel: {@code <html> </ html>}. Dies wird jedoch eine Monospace-Schriftart verwenden. Verwenden Sie {@literal}, um dasselbe Ergebnis ohne Monospace-Schriftart zu erhalten.
- {@literal} // Wie {@code}, jedoch ohne die Monospaced-Schriftart.
- {@value} // Zeigt den Wert eines statischen Feldes an: Der Wert von JFrame # EXIT_ON_CLOSE ist {@value}. Sie können auch mit einem bestimmten Feld verknüpfen: Verwendet den App-Namen {@value AppConstants # APP_NAME}.
- {@docRoot} // Der Stammordner des JavaDoc-HTML-Objekts relativ zur aktuellen Datei. Beispiel: Credits .
- HTML ist erlaubt: <code> "Hi Cookies" .Substring (3) </ code>.
- * / - Ende der JavaDoc-Deklaration

Bemerkungen

Javadoc ist ein im JDK enthaltenes Werkzeug, mit dem Kommentare im Code in eine HTML-Dokumentation konvertiert werden können. Die Java-API-Spezifikation wurde mit Javadoc erstellt. Dasselbe gilt für einen Großteil der Dokumentation von Drittanbieter-Bibliotheken.

Examples

Klassendokumentation

Alle Javadoc-Kommentare beginnen mit einem Blockkommentar gefolgt von einem Sternchen (/**)

und enden, wenn der Blockkommentar dies tut (*/). Optional kann jede Zeile mit einem beliebigen Leerzeichen und einem einzelnen Stern beginnen. Diese werden bei der Erstellung der Dokumentationsdateien ignoriert.

```
/**
 * 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 {
}
```

Die gleichen Classes und Formate, die für Classes verwendet werden, können auch für Enums und Interfaces werden.

Methodendokumentation

Alle Javadoc-Kommentare beginnen mit einem Blockkommentar gefolgt von einem Sternchen (/**) und enden, wenn der Blockkommentar dies tut (*/). Optional kann jede Zeile mit einem beliebigen Leerzeichen und einem einzelnen Stern beginnen. Diese werden bei der Erstellung der Dokumentationsdateien ignoriert.

```
/**
 * 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;
}
```

Felddokumentation

Alle Javadoc-Kommentare beginnen mit einem Blockkommentar gefolgt von einem Sternchen (/**) und enden, wenn der Blockkommentar dies tut (*/). Optional kann jede Zeile mit einem beliebigen Leerzeichen und einem einzelnen Stern beginnen. Diese werden bei der Erstellung der Dokumentationsdateien ignoriert.

```
/**
 * 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";
```

Paketdokumentation

Java SE 5

Es ist möglich, eine Dokumentation auf Paketebene in Javadocs mit einer Datei namens package-info.java . Diese Datei muss wie folgt formatiert sein. Führende Leerzeichen und Sternchen

optional, normalerweise in jeder Zeile aus Gründen der Formatierung

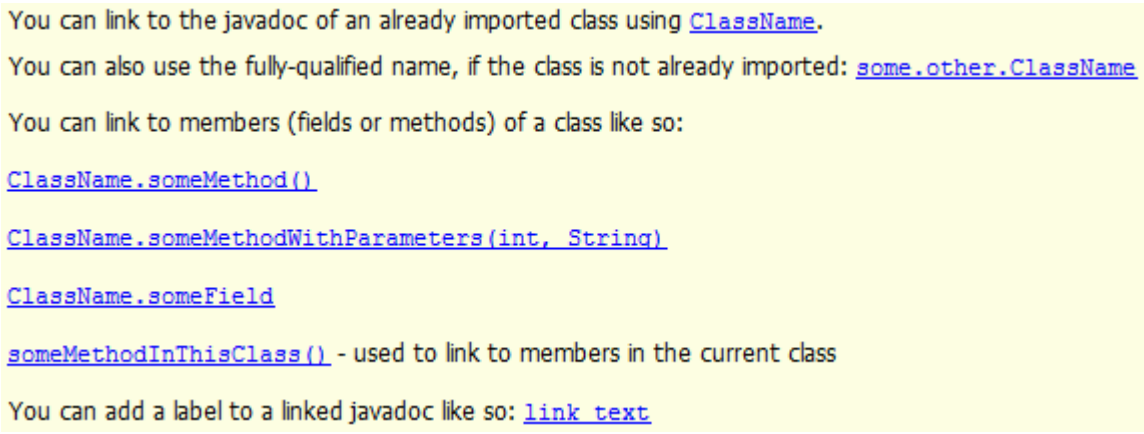
```
/**
 * Package documentation goes here; any documentation before the first period will
 * be used as a summary.
 *
 * It is common practice to leave a blank line between the summary and the rest
 * of the documentation; use this space to describe the package in as much detail
 * as is appropriate.
 *
 * 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.
```

In diesem Fall müssen Sie diese Datei package-info.java im Ordner des Java-Pakets com.example.foo .

Links

Das Verknüpfen mit anderen Javadocs erfolgt mit dem @link Tag:

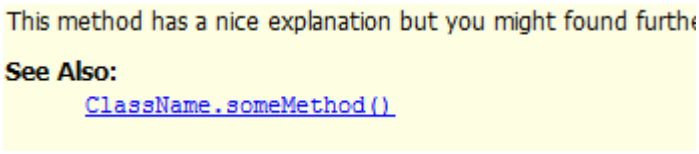
```
/**
 * You can link to the javadoc of an already imported class using {@link ClassName}.
 *
 * You can also use the fully-qualified name, if the class is not already imported:
 *  {@link some.other.ClassName}
 *
 * You can link to members (fields or methods) of a class like so:
 *  {@link ClassName#someMethod()}
 *  {@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}
 */
```



Mit dem Tag @see können Sie dem Abschnitt *Siehe auch* Elemente hinzufügen. Wie @param oder @return der Ort, an dem sie erscheinen, nicht relevant. Die Spezifikation sagt, Sie sollten es nach @return schreiben.

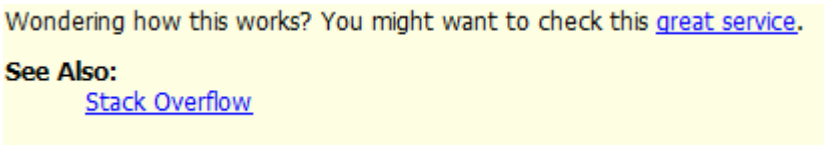
```
/**
 * This method has a nice explanation but you might found further
```

```
 * information at the bottom.
 *
 * @see ClassName#someMethod()
 */
```



Wenn Sie **Links zu externen Ressourcen** hinzufügen möchten, **können** Sie einfach das HTML-Tag <a> . Sie können es überall oder innerhalb von @link und @see Tags verwenden.

```
/**
 * 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>
 */
```



Javadocs über die Befehlszeile erstellen

Viele IDEs unterstützen das automatische Generieren von HTML aus Javadocs. Einige Build-Tools (z. B. Maven und Gradle) verfügen auch über Plugins, die die HTML-Erstellung unterstützen.

Diese Tools sind jedoch nicht erforderlich, um den Javadoc-HTML-Code zu generieren. Dies kann mit dem Befehlszeilen- javadoc Tool ausgeführt werden.

Die grundlegendste Verwendung des Tools ist:

```
javadoc JavaFile.java
```

JavaFile.java wird HTML aus den Javadoc-Kommentaren in JavaFile.java .

Eine praktischere Anwendung des Befehlszeilentools, das rekursiv alle Java-Dateien in [source-directory] [package.name] , Dokumentation für [package.name] und alle [package.name] erstellt und den generierten HTML- [package.name] in das [docs-directory] [package.name] ist:

```
javadoc -d [docs-directory] -subpackages -sourcepath [source-directory] [package.name]
```

Inline-Code-Dokumentation

Neben der Javadoc-Dokumentation kann der Code auch inline dokumentiert werden.

Einzeilige Kommentare werden von // gestartet und dürfen nach einer Anweisung in derselben Zeile stehen, jedoch nicht vorher.

```
public void method() {

  //single line comment
  someMethodCall(); //single line comment after statement

}
```

Mehrzeilige Kommentare werden zwischen /* und */ . Sie können sich über mehrere Zeilen erstrecken und wurden sogar zwischen Anweisungen positioniert.

```
public void method(Object object) {

  /*
    multi
    line
    comment
  */
  object/*inner-line-comment*/.method();
}
```

JavaDocs sind eine spezielle Form von mehrzeiligen Kommentaren, beginnend mit /** .

Da zu viele Inline-Kommentare die Lesbarkeit von Code beeinträchtigen, sollten sie sparsam verwendet werden, falls der Code nicht selbsterklärend genug ist oder die Entwurfsentscheidung nicht offensichtlich ist.

Ein zusätzlicher Anwendungsfall für einzeilige Kommentare ist die Verwendung von TAGs, bei denen es sich um kurze, auf Konventionen basierende Schlüsselwörter handelt. Einige Entwicklungsumgebungen erkennen bestimmte Konventionen für solche Einzelkommentare an. Häufige Beispiele sind

- //TODO
- //FIXME

Oder geben Sie Referenzen aus, zB für Jira

- //PRJ-1234

Codeausschnitte in der Dokumentation

Die kanonische Art, Code in der Dokumentation zu schreiben, ist mit dem {@code } . Wenn innerhalb von <pre></pre> mehrzeiliger Code-Wrap vorhanden ist.

```
/**
 * 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 {
```

Manchmal müssen Sie in den Javadoc-Kommentar etwas komplexen Code einfügen. Das @ -Zeichen ist besonders problematisch. Die Verwendung des alten <code> -Tags neben dem {@literal } löst das Problem.

```
/**
 * 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 { }
```

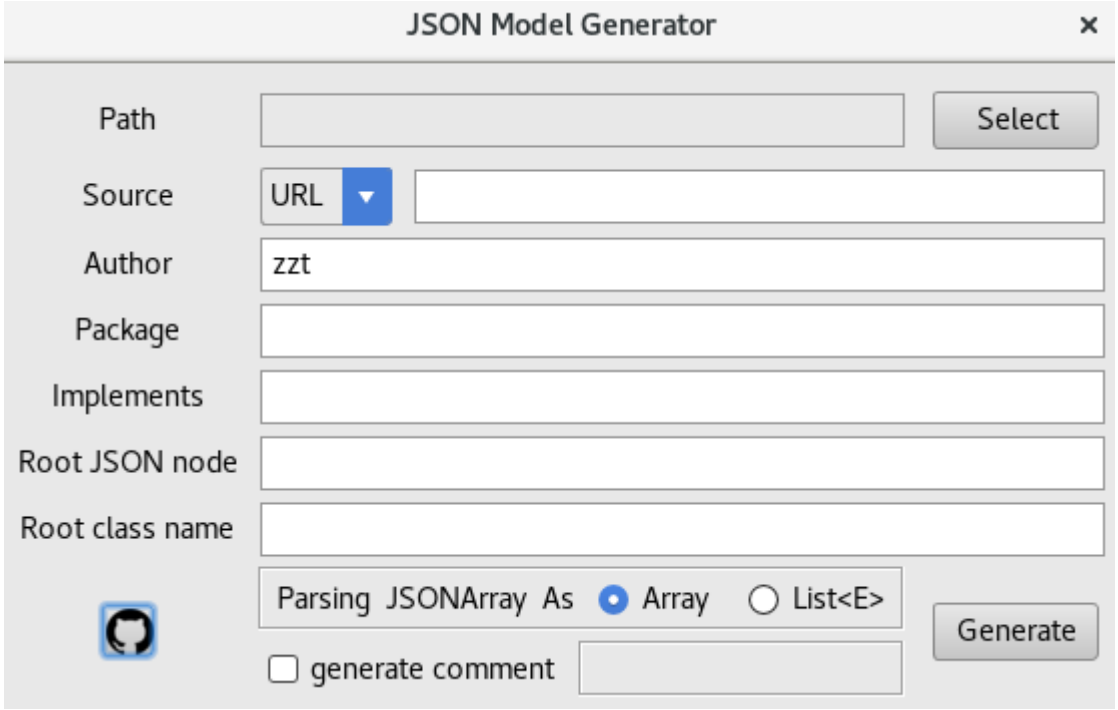
Java-Code dokumentieren online lesen: https://riptutorial.com/de/java/topic/140/java-code-dokumentieren

Kapitel 69: Java-Code generieren

Examples

Generiere POJO aus JSON

- Installieren Sie das JSON Model Genrator-Plugin von Intellij, indem Sie in der Einstellung "Intellij" suchen.
- Starten Sie das Plugin über 'Tools'
- Geben Sie das Feld der Benutzeroberfläche wie folgt ein ('Pfad' 、 'Quelle' 、 'Paket' ist erforderlich):



- Klicken Sie auf die Schaltfläche "Generieren" und Sie sind fertig.

Java-Code generieren online lesen: https://riptutorial.com/de/java/topic/9400/java-code-generieren

Kapitel 70: Java-Druckdienst

Einführung

Die Java- Druckdienst- API bietet Funktionen zum Erkennen von Druckdiensten und zum Senden von Druckanforderungen für diese Dienste.

Es enthält erweiterbare Druckattribute, die auf den im Internet Printing Protocol (IPP) 1.1 angegebenen Standardattributen der IETF-Spezifikation RFC 2911 basieren.

Examples

Entdecken Sie die verfügbaren Druckdienste

Um alle verfügbaren Druckdienste zu ermitteln, können wir die PrintServiceLookup Klasse verwenden. Mal sehen wie:

```
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());
        }
    }

}
```

Wenn dieses Programm in einer Windows-Umgebung ausgeführt wird, wird Folgendes gedruckt:

```
Print service name: Fax
Print service name: Microsoft Print to PDF
Print service name: Microsoft XPS Document Viewer
```

Ermitteln des Standarddruckdienstes

Um den Standarddruckdienst zu ermitteln, können wir die PrintServiceLookup Klasse verwenden. Mal sehen wie:

```
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());
    }

}
```

Erstellen eines Druckauftrags von einem Druckdienst aus

Ein Druckauftrag ist eine Aufforderung, etwas in einem bestimmten Druckdienst zu drucken. Es besteht im Wesentlichen aus:

- die Daten, die gedruckt werden sollen (siehe Erstellen des Dokuments, das gedruckt werden soll)
- eine Reihe von Attributen

Nach der Abholung der richtigen Druckdienstinstanz können wir die Erstellung eines Druckauftrags anfordern:

```
DocPrintJob printJob = printService.createPrintJob();
```

Die DocPrintJob Schnittstelle uns das bieten print

```
printJob.print(doc, pras);
```

Das doc - Argument ist ein Doc : die Daten , die gedruckt werden.

Und das Argument pras ist eine PrintRequestAttributeSet Schnittstelle: eine Gruppe von PrintRequestAttribute . Sind Beispiele für Druckanforderungsattribute:

- Anzahl der Exemplare (1, 2 usw.),
- Orientierung (Hochformat oder Landschaft)
- Farbstärke (Monochrom, Farbe)
- Qualität (Entwurf, normal, hoch)
- Seiten (einseitig, zweiseitig usw.)
- und so weiter...

Die PrintException kann eine PrintException .

Erstellen des Dokuments, das gedruckt werden soll

Doc ist eine Schnittstelle und die Java Print Service API bietet eine einfache Implementierung namens SimpleDoc .

Jede Doc Instanz besteht im Wesentlichen aus zwei Aspekten:

- den Inhalt der Druckdaten selbst (eine E-Mail, ein Bild, ein Dokument usw.)
- das Druckdatenformat, DocFlavor (MIME-Typ + Repräsentationsklasse).

Bevor Sie das Doc Objekt erstellen, müssen Sie unser Dokument von irgendwoher laden. Im Beispiel laden wir eine bestimmte Datei von der Festplatte:

```
FileInputStream pdfFileInputStream = new FileInputStream("something.pdf");
```

Wir müssen also jetzt einen DocFlavor auswählen, der zu unserem Inhalt passt. Die DocFlavor Klasse verfügt über eine Reihe von Konstanten, um die häufigsten Datentypen darzustellen. Lass uns die INPUT_STREAM.PDF :

```
DocFlavor pdfDocFlavor = DocFlavor.INPUT_STREAM.PDF;
```

Jetzt können wir eine neue Instanz von SimpleDoc :

```
Doc doc = new SimpleDoc(pdfFileInputStream, pdfDocFlavor , null);
```

Das doc kann jetzt an die Druckauftragsanfrage gesendet werden (siehe Erstellen eines Druckauftrags über einen Druckservice).

Druckauftragsattribute definieren

Manchmal müssen wir einige Aspekte der Druckanforderung ermitteln. Wir nennen sie *Attribut* .

Sind Beispiele für Druckanforderungsattribute:

- Anzahl der Exemplare (1, 2 usw.),
- Orientierung (Hochformat oder Landschaft)
- Farbstärke (Monochrom, Farbe)
- Qualität (Entwurf, normal, hoch)
- Seiten (einseitig, zweiseitig usw.)
- und so weiter...

Bevor Sie einen von ihnen auswählen und welchen Wert jeder von ihnen haben wird, müssen wir zunächst eine Reihe von Attributen erstellen:

```
PrintRequestAttributeSet pras = new HashPrintRequestAttributeSet();
```

Jetzt können wir sie hinzufügen. Einige Beispiele sind:

```
pras.add(new Copies(5));
pras.add(MediaSize.ISO_A4);
pras.add(OrientationRequested.PORTRAIT);
pras.add(PrintQuality.NORMAL);
```

Das pras Objekt kann jetzt an die Druckauftragsanfrage gesendet werden (siehe Erstellen eines Druckauftrags über einen Druckdienst).

Statusänderung des Druckauftrags abhören

Für die meisten Druck-Clients ist es äußerst nützlich zu wissen, ob ein Druckauftrag abgeschlossen wurde oder fehlgeschlagen ist.

Die Java-Druckdienst-API bietet einige Funktionen, um sich über diese Szenarien zu informieren. Alles was wir tun müssen ist:

- Bereitstellung einer Implementierung für die PrintJobListener Schnittstelle und
- Registrieren Sie diese Implementierung beim Druckauftrag.

Wenn sich der Status des Druckauftrags ändert, werden wir benachrichtigt. Wir können alles tun, zum Beispiel:

- eine Benutzeroberfläche aktualisieren,
- einen anderen Geschäftsprozess starten,
- etwas in die Datenbank aufnehmen,
- oder einfach protokollieren.

Im folgenden Beispiel protokollieren wir jede Statusänderung des Druckauftrags:

```
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 ");
    }
}
```

Schließlich können wir unsere Druckauftragslistenerimplementierung vor der Druckanforderung selbst in den Druckauftrag einfügen:

```
DocPrintJob printJob = printService.createPrintJob();

printJob.addPrintJobListener(new LoggerPrintJobListener());

printJob.print(doc, pras);
```

Das Argument " *PrintJobEvent pje*"

Beachten Sie, dass jede Methode ein PrintJobEvent pje Argument PrintJobEvent pje . Wir verwenden es in diesem Beispiel nicht der Einfachheit halber, aber Sie können es verwenden, um den Status zu untersuchen. Zum Beispiel:

```
pje.getPrintJob().getAttributes();
```

PrintJobAttributeSet eine PrintJobAttributeSet Objektinstanz zurück, und Sie können sie for-each ausführen.

Ein anderer Weg, um das gleiche Ziel zu erreichen

Eine weitere Option, um dasselbe Ziel zu erreichen, ist die Erweiterung der PrintJobAdapter Klasse, wie der Name sagt, ein Adapter für PrintJobListener . Durch die Implementierung der Schnittstelle müssen wir alle zwingend implementieren. Der Vorteil ist, dass wir nur die Methoden überschreiben müssen, die wir wollen. Mal sehen, wie es funktioniert:

```
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 =( ");
    }
}
```

Beachten Sie, dass wir nur einige bestimmte Methoden überschreiben.

In dem Beispiel, in dem die Schnittstelle PrintJobListener implementiert ist, fügen Sie dem Druckauftrag den Listener hinzu, bevor Sie ihn zum Drucken senden:

```
printJob.addPrintJobListener(new LoggerPrintJobAdapter());

printJob.print(doc, pras);
```

Java-Druckdienst online lesen: https://riptutorial.com/de/java/topic/10178/java-druckdienst

Kapitel 71: Java-Editionen, Versionen, Releases und Distributionen

Examples

Unterschiede zwischen Java SE JRE- oder Java SE JDK-Distributionen

Sun / Oracle-Versionen von Java SE gibt es in zwei Formen: JRE und JDK. Einfach ausgedrückt unterstützen JREs das Ausführen von Java-Anwendungen, und JDKs unterstützen auch die Java-Entwicklung.

Java-Laufzeitumgebung

Java Runtime Environment- oder JRE-Distributionen bestehen aus einer Reihe von Bibliotheken und Tools, die zum Ausführen und Verwalten von Java-Anwendungen erforderlich sind. Zu den Tools in einer typischen modernen JRE gehören:

- Der java Befehl zum Ausführen eines Java-Programms in einer JVM (Java Virtual Machine)
- Der Befehl jjs zum Ausführen der Nashorn-JavaScript-Engine.
- Der Befehl keytool zum Bearbeiten von Java-Keystores.
- Der policytool Befehl zum Bearbeiten von Sicherheits-Sandbox-Sicherheitsrichtlinien.
- Die Tools pack200 und unpack200 zum Packen und Entpacken der "pack200" -Datei für die Webbereitstellung.
- Die orbd , rmid , rmiregistry und tnameserv unterstützen Java tnameserv und RMI-Anwendungen.

"Desktop JRE" -Installer enthalten ein Java-Plugin, das für einige Webbrowser geeignet ist. Dies wird bewusst aus "Server JRE" installers.linux syscall read benchmarku ausgelassen

Seit Java 7 Update 6 enthalten JRE-Installationsprogramme JavaFX (Version 2.2 oder höher).

Java Entwickler-Kit

Ein Java Development Kit oder eine JDK-Distribution enthält die JRE-Tools und zusätzliche Tools zum Entwickeln von Java-Software. Zu den zusätzlichen Tools gehören normalerweise:

- Der Befehl javac , der Java-Quellcode (".java") in Bytecode-Dateien (".class") übersetzt.
- Die Tools zum Erstellen von JAR-Dateien wie jar und jarsigner
- Entwicklungswerkzeuge wie:
    - appletviewer zum Ausführen von Applets
    - idlj der CORBA-IDL-zu-Java-Compiler
    - javah der JNI-Stub-Generator
    - native2ascii zur Zeichensatzkonvertierung von Java-Quellcode
    - schemagen den Java-zu-XML-Schema-Generator (Teil von JAXB).
    - serialver generiert eine Java Object Serialization-Versionszeichenfolge.
    - Die Support-Tools für wsgen und wsimport für JAX-WS
- Diagnosewerkzeuge wie:
    - jdb der grundlegende Java-Debugger
    - jmap und jhat zum Ablegen und Analysieren eines Java- jhat .
    - jstack zum jstack eines Thread-Stack-Dumps.
    - javap zum Untersuchen von ".class" -Dateien.
- Anwendungsmanagement- und Überwachungstools wie:
    - jconsole eine Managementkonsole,
    - jstat , jstatd , jinfo und jps für die Anwendungsüberwachung

Eine typische Sun / Oracle-JDK-Installation enthält auch eine ZIP-Datei mit dem Quellcode der Java-Bibliotheken. Vor Java 6 war dies der einzige öffentlich verfügbare Java-Quellcode.

Ab Java 6 steht der vollständige Quellcode für OpenJDK auf der OpenJDK-Site zum Download zur Verfügung. Es ist normalerweise nicht in (Linux) JDK-Paketen enthalten, es ist jedoch als separates Paket verfügbar.

Was ist der Unterschied zwischen Oracle Hotspot und OpenJDK?

Orthogonal zur JRE- und JDK-Dichotomie gibt es zwei Arten von Java-Releases, die allgemein verfügbar sind:

- Die Oracle Hotspot-Versionen sind diejenigen, die Sie von den Oracle-Download-Sites herunterladen.
- Die OpenJDK-Versionen sind diejenigen, die (normalerweise von Drittanbietern) aus den OpenJDK-Quell-Repositorys erstellt werden.

In funktionaler Hinsicht gibt es wenig Unterschiede zwischen einer Hotspot-Version und einer OpenJDK-Version. Es gibt einige zusätzliche "Enterprise" -Funktionen in Hotspot, die Oracle (zahlende) Java-Kunden aktivieren können. Abgesehen davon gibt es in Hotspot und OpenJDK dieselbe Technologie.

Ein weiterer Vorteil von Hotspot gegenüber OpenJDK ist, dass Patch-Releases für Hotspot etwas früher verfügbar sind. Dies hängt auch davon ab, wie flexibel Ihr OpenJDK-Provider ist. Wie lange dauert beispielsweise das Build-Team einer Linux-Distribution, um einen neuen OpenJDK-Build vorzubereiten und zu prüfen und in seine öffentlichen Repositorys zu bringen.

Die andere Seite ist, dass die Hotspot-Versionen für die meisten Linux-Distributionen nicht in den Paket-Repositorys verfügbar sind. Das bedeutet, dass das Aufrechterhalten der Java-Software auf einem Linux-Computer normalerweise mehr Arbeit erfordert, wenn Sie Hotspot verwenden.

Unterschiede zwischen Java EE, Java SE, Java ME und JavaFX

Java-Technologie ist sowohl eine Programmiersprache als auch eine Plattform. Die Java-Programmiersprache ist eine objektorientierte Hochsprache mit einer bestimmten Syntax und einem bestimmten Stil. Eine Java-Plattform ist eine bestimmte Umgebung, in der Java-Programmiersprachenanwendungen ausgeführt werden.

Es gibt verschiedene Java-Plattformen. Viele Entwickler, selbst langjährige Entwickler von Java-Programmiersprachen, verstehen nicht, wie die verschiedenen Plattformen miteinander zusammenhängen.

Die Java-Programmiersprachenplattformen

Es gibt vier Plattformen der Java-Programmiersprache:

- Java-Plattform, Standard Edition (Java SE)
- Java-Plattform, Enterprise Edition (Java EE)
- Java-Plattform, Micro Edition (Java ME)
- Java FX

Alle Java-Plattformen bestehen aus einer Java Virtual Machine (VM) und einer Anwendungsprogrammierschnittstelle (API). Die Java Virtual Machine ist ein Programm für eine bestimmte Hardware- und Softwareplattform, das Java-Technologieanwendungen ausführt. Eine API ist eine Sammlung von Softwarekomponenten, die Sie zum Erstellen anderer Softwarekomponenten oder Anwendungen verwenden können. Jede Java-Plattform bietet eine virtuelle Maschine und eine API. Dadurch können Anwendungen, die für diese Plattform geschrieben wurden, auf jedem kompatiblen System mit allen Vorteilen der Java-Programmiersprache ausgeführt werden: Plattformunabhängigkeit, Leistung, Stabilität, Entwicklungsfreundlichkeit und Sicherheit.

Java SE

Wenn die meisten Leute an die Java-Programmiersprache denken, denken sie an die Java SE-API. Die API von Java SE bietet die Kernfunktionalität der Java-Programmiersprache. Es definiert alles von den grundlegenden Typen und Objekten der Java-Programmiersprache bis zu übergeordneten Klassen, die für Netzwerke, Sicherheit, Datenbankzugriff, Entwicklung grafischer

Benutzeroberflächen (GUI) und XML-Analyse verwendet werden.

Neben der Kern-API besteht die Java SE-Plattform aus einer virtuellen Maschine, Entwicklungstools, Bereitstellungstechnologien und anderen Klassenbibliotheken und Toolkits, die üblicherweise in Java-Technologieanwendungen verwendet werden.

Java EE

Die Java EE-Plattform basiert auf der Java SE-Plattform. Die Java EE-Plattform bietet eine API- und Laufzeitumgebung für die Entwicklung und Ausführung großer, mehrschichtiger, skalierbarer, zuverlässiger und sicherer Netzwerkanwendungen.

Java ME

Die Java ME-Plattform bietet eine API und eine virtuelle Maschine mit geringem Speicherbedarf für die Ausführung von Java-Programmiersprachenanwendungen auf kleinen Geräten wie Mobiltelefonen. Die API ist eine Teilmenge der Java SE-API zusammen mit speziellen Klassenbibliotheken, die für die Entwicklung von Anwendungen für kleine Geräte nützlich sind. Java ME-Anwendungen sind häufig Clients von Java EE-Plattformdiensten.

Java FX

Die Java FX-Technologie ist eine Plattform zum Erstellen von Rich-Internet-Anwendungen, die in Java FX ScriptTM geschrieben sind. Java FX Script ist eine statisch typisierte deklarative Sprache, die zu Bytecode der Java-Technologie kompiliert wird, der dann auf einer Java-VM ausgeführt werden kann. Anwendungen, die für die Java FX-Plattform geschrieben wurden, können Java-Programmiersprachklassen enthalten und mit ihnen verknüpfen und können Clients von Java EE-Plattformdiensten sein.

- Entnommen aus der Oracle-Dokumentation

Java SE-Versionen

Java SE-Versionsverlauf

Die folgende Tabelle enthält die Zeitleiste für die wichtigen Hauptversionen der Java SE-Plattform.

Java SE Version [1]	Code Name	Ende des Lebens (kostenlos [2])	Veröffentlichungsdatum
Java SE 9 (früher Zugriff)	*Keiner*	Zukunft	2017-07-27 (geschätzt)
Java SE 8	*Keiner*	Zukunft	2014-03-18
Java SE 7	Delphin	2015-04-14	2011-07-28
Java SE 6	Mustang	2013-04-16	2006-12-23
Java SE 5	Tiger	2009-11-04	2004-10-04
Java SE 1.4.2	Gottesanbeterin	vor dem 2009-11-04	2003-06-26
Java SE 1.4.1	Trichter /	vor dem 2009-11-04	2002-09-16

Java SE Version [1]	Code Name	Ende des Lebens (kostenlos [2])	Veröffentlichungsdatum
	Grashüpfer		
Java SE 1.4	Merlin	vor dem 2009-11-04	2002-02-06
Java SE 1.3.1	Marienkäfer	vor dem 2009-11-04	2001-05-17
Java SE 1.3	Turmfalke	vor dem 2009-11-04	2000-05-08
Java SE 1.2	Spielplatz	vor dem 2009-11-04	1998-12-08
Java SE 1.1	Wunderkerze	vor dem 2009-11-04	1997-02-19
Java SE 1.0	Eiche	vor dem 2009-11-04	1996-01-21

Fußnoten:

1. Die Links verweisen auf Online-Kopien der jeweiligen Release-Dokumentation auf der Oracle-Website. Die Dokumentation für viele ältere Versionen ist nicht mehr online, obwohl sie normalerweise aus dem Oracle Java-Archiv heruntergeladen werden kann.
2. Die meisten historischen Versionen von Java SE haben ihre offiziellen End-of-Life-Daten überschritten. Wenn eine Java-Version diesen Meilenstein überschreitet, stellt Oracle keine kostenlosen Updates mehr bereit. Kunden mit Supportverträgen stehen weiterhin Updates zur Verfügung.

Quelle:

- JDK-Veröffentlichungsdaten von Roedy Green von Canadian Mind Products

Highlights der Java SE-Version

Java SE-Version	Höhepunkte
Java SE 8	Lambda-Ausdrücke und MapReduce-inspirierte Streams. Die Nashorn-Javascript-Engine. Anmerkungen zu Typen und sich wiederholende Anmerkungen. Vorzeichenlose arithmetische Erweiterungen. Neue Datums- und Uhrzeit-APIs. Statisch verknüpfte JNI-Bibliotheken. JavaFX Launcher. Entfernung von PermGen.
Java SE 7	String-Schalter, *Try-with-Ressource* , Raute (`<>`), Verbesserungen des numerischen Literal und Verbesserungen bei der Behandlung von Ausnahmen / Wiederherstellen. Parallele Bibliotheksverbesserungen. Erweiterte Unterstützung für native Dateisysteme. Timsort ECC-Kryptoalgorithmen. Verbesserte Unterstützung für 2D-Grafiken (GPU). Steckbare Anmerkungen.
Java SE 6	Erhebliche Leistungsverbesserungen für JVM-Plattform und Swing. Skriptsprache-API und Mozilla Rhino-Javascript-Engine. JDBC 4.0. Compiler-API JAXB 2.0. Web Services-Unterstützung (JAX-WS)

Java SE-Version	Höhepunkte
Java SE 5	Generics, Anmerkungen, Auto-Boxing, `enum` , Varargs, erweitert `for` Schleifen und statische Importe. Spezifikation des Java-Speichermodells. Swing- und RMI-Verbesserungen. Hinzufügen des Pakets `java.util.concurrent.*` Und des `Scanner` .
Java SE 1.4	Das `assert` Schlüsselwort. Klassen für reguläre Ausdrücke Ausnahme-Verkettung. NIO-APIs - nicht blockierende E / A, `Buffer` und `Channel` . `java.util.logging.*` API. Image I / O-API. Integriertes XML und XSLT (JAXP). Integrierte Sicherheit und Kryptographie (JCE, JSSE, JAAS). Integrierter Java Web Start. Einstellungen-API.
Java SE 1.3	HotSpot-JVM enthalten. CORBA / RMI-Integration. Java Naming und Directory Interface (JNDI). Debugger-Framework (JPDA). JavaSound-API. Proxy-API
Java SE 1.2	Das Keyword `strictfp` . Swing-APIs Das Java-Plugin (für Webbrowser). CORBA-Interoperabilität. Rahmen für Sammlungen.
Java SE 1.1	Innere Klassen. Reflexion. JDBC. RMI. Unicode / Zeichenströme. Unterstützung bei der Internationalisierung. Überholung des AWT-Ereignismodells. JavaBeans.

Quelle:

- Wikipedia: Java-Versionsgeschichte

Java-Editionen, Versionen, Releases und Distributionen online lesen: https://riptutorial.com/de/java/topic/8973/java-editionen--versionen--releases-und-distributionen

Kapitel 72: Java-Fallstricke - Leistungsprobleme

Einführung

In diesem Thema werden einige "Fallstricke" (dh Fehler, die Java-Programmierer anfangen) gemacht, die sich auf die Java-Anwendungsleistung beziehen.

Bemerkungen

In diesem Thema werden einige "mikro" Java-Codierungspraktiken beschrieben, die ineffizient sind. In den meisten Fällen sind die Ineffizienzen relativ gering, aber es lohnt sich, sie zu vermeiden.

Examples

Pitfall - Der Aufwand für das Erstellen von Protokollnachrichten

TRACE und DEBUG Protokollebenen dienen dazu, zur Laufzeit sehr detaillierte Informationen über den Betrieb des angegebenen Codes zu vermitteln. Es wird normalerweise empfohlen, den Log-Level über diese Werte zu setzen. Es muss jedoch darauf geachtet werden, dass diese Aussagen die Leistung nicht beeinträchtigen, selbst wenn sie scheinbar "ausgeschaltet" sind.

Betrachten Sie diese Protokollanweisung:

```
// Processing a request of some kind, logging the parameters
LOG.debug("Request coming from " + myInetAddress.toString()
          + " parameters: " + Arrays.toString(veryLongParamArray));
```

Selbst wenn die Protokollebene auf INFO , werden an debug() Argumente bei jeder Ausführung der Zeile ausgewertet. Dies macht es in mehrfacher Hinsicht unnötig aufwendig:

- String Verkettung: Es werden mehrere String Instanzen erstellt
- InetAddress möglicherweise sogar eine DNS-Suche durch.
- Der veryLongParamArray möglicherweise sehr lang - das Erstellen eines veryLongParamArray daraus verbraucht Speicher und nimmt Zeit in veryLongParamArray

Lösung

Die meisten Protokollierungsframeworks bieten die Möglichkeit, Protokollnachrichten mithilfe von Fix-Strings und Objektreferenzen zu erstellen. Die Protokollnachricht wird nur ausgewertet, wenn die Nachricht tatsächlich protokolliert wird. Beispiel:

```
// No toString() evaluation, no string concatenation if debug is disabled
LOG.debug("Request coming from {} parameters: {}", myInetAddress, parameters));
```

Dies funktioniert sehr gut, solange alle Parameter mit Hilfe von String.valueOf (Object) in Strings konvertiert werden können. Wenn die Protokollmeldungsberechnung komplexer ist, kann die Protokollebene vor der Protokollierung überprüft werden:

```
if (LOG.isDebugEnabled()) {
    // Argument expression evaluated only when DEBUG is enabled
    LOG.debug("Request coming from {}, parameters: {}", myInetAddress,
              Arrays.toString(veryLongParamArray);
}
```

Hier wird LOG.debug() mit der kostspieligen Arrays.toString(Obect[]) nur verarbeitet, wenn DEBUG tatsächlich aktiviert ist.

Pitfall - String-Verkettung in einer Schleife skaliert nicht

Betrachten Sie den folgenden Code als Illustration:

```
public String joinWords(List<String> words) {
    String message = "";
    for (String word : words) {
        message = message + " " + word;
    }
    return message;
}
```

Unglücklicherweise ist dieser Code ineffizient, wenn die words lang ist. Die Wurzel des Problems ist diese Aussage:

```
message = message + " " + word;
```

Für jede Schleifeniteration erstellt diese Anweisung eine neue message die eine Kopie aller Zeichen in der ursprünglichen message an die zusätzliche Zeichen angehängt werden. Dadurch werden viele temporäre Zeichenfolgen generiert und viel kopiert.

Wenn wir joinWords analysieren, unter der Annahme, dass es N Wörter mit einer durchschnittlichen Länge von M gibt, stellen wir fest, dass temporäre O (N) joinWords erstellt werden und O (MN^2) -Zeichen in den Prozess kopiert werden. Die N^2 -Komponente ist besonders beunruhigend.

Der empfohlene Ansatz für diese Art von Problem [1] ist die Verwendung eines StringBuilder anstelle der String-Verkettung wie folgt:

```
public String joinWords2(List<String> words) {
    StringBuilder message = new StringBuilder();
    for (String word : words) {
        message.append(" ").append(word);
    }
    return message.toString();
}
```

Bei der Analyse von joinWords2 muss der joinWords2 berücksichtigt werden, joinWords2 das StringBuilder Backing-Array mit den Zeichen des joinWords2 "vergrößert" wird. Es stellt sich jedoch heraus, dass die Anzahl der neu erstellten Objekte O (logN) und die Anzahl der kopierten Zeichen O (MN) -Zeichen ist. Letzteres enthält Zeichen, die im letzten Aufruf von toString() kopiert wurden.

(Möglicherweise können Sie dies weiter StringBuilder , indem Sie den StringBuilder mit der richtigen Kapazität erstellen, mit der Sie beginnen können. Die Gesamtkomplexität bleibt jedoch gleich.)

Bei der Rückkehr zur ursprünglichen joinWords Methode stellt sich heraus, dass die kritische Anweisung von einem typischen Java-Compiler auf joinWords optimiert wird:

```
  StringBuilder tmp = new StringBuilder();
  tmp.append(message).append(" ").append(word);
  message = tmp.toString();
```

Der Java-Compiler "hebt" den StringBuilder jedoch nicht aus der Schleife, wie wir es joinWords2 im Code für joinWords2 .

Referenz:

- Msgstr "Ist Javas String '+' Operator in einer Schleife langsam?"

---

1 - In Java 8 und höher kann die Joiner Klasse verwendet werden, um dieses bestimmte Problem zu lösen. Aber *darum* geht es in diesem Beispiel *eigentlich nicht* .

Fallstricke - Die Verwendung von "new" zum Erstellen von primitiven Wrapper-Instanzen ist ineffizient

Mit der Java-Sprache können Sie new , um Instanzen wie Integer , Boolean usw. zu erstellen. Integer ist jedoch im Allgemeinen eine schlechte Idee. Es ist besser, entweder Autoboxing (Java 5 und höher) oder die Methode valueOf verwenden.

```
Integer i1 = new Integer(1);      // BAD
Integer i2 = 2;                   // BEST (autoboxing)
Integer i3 = Integer.valueOf(3);  // OK
```

Die Verwendung von new Integer(int) ist eine schlechte Idee, weil sie ein neues Objekt erstellt (sofern nicht durch JIT-Compiler optimiert). Wenn dagegen valueOf oder ein expliziter valueOf Aufruf verwendet wird, versucht die Java-Laufzeitumgebung, ein Integer Objekt aus einem Cache mit bereits vorhandenen Objekten zu verwenden. Jedes Mal, wenn die Laufzeit einen Cache-Treffer hat, wird die Erstellung eines Objekts vermieden. Dies spart auch Heap-Speicher und verringert die durch Objektabwanderung verursachten GC-Overheads.

Anmerkungen:

1. In aktuellen Java-Implementierungen wird Autoboxing durch Aufrufen von valueOf implementiert, und es gibt Caches für Boolean , Byte , Short , Integer , Long und Character .
2. Das Caching-Verhalten für die Integraltypen wird von der Java-Sprachspezifikation vorgegeben.

Pitfall - Das Aufrufen von 'new String (String)' ist ineffizient

Die Verwendung eines new String(String) Strings new String(String) zum Duplizieren eines Strings ist ineffizient und fast immer unnötig.

- String-Objekte sind unveränderlich, so dass sie zum Schutz vor Änderungen nicht kopiert werden müssen.
- In einigen älteren Java-Versionen können String Objekte Sicherungsarrays mit anderen String Objekten gemeinsam nutzen. In diesen Versionen ist es möglich, Speicher zu verlieren, indem eine (kleine) Teilzeichenfolge einer (großen) Zeichenfolge erstellt und beibehalten wird. String Backing-Arrays werden jedoch ab Java 7 nicht freigegeben.

Wenn kein nennenswerter Vorteil besteht, ist das Aufrufen von new String(String) einfach verschwenderisch:

- Das Erstellen der Kopie erfordert CPU-Zeit.
- Die Kopie benötigt mehr Speicher, was den Speicherbedarf der Anwendung erhöht und / oder den GC-Aufwand erhöht.
- Operationen wie equals(Object) und hashCode() können langsamer sein, wenn String-Objekte kopiert werden.

Pitfall - Das Aufrufen von System.gc () ist ineffizient

Es ist (fast immer) eine schlechte Idee, System.gc() .

Der Javadoc für die Methode gc() gibt Folgendes an:

> Der Aufruf der gc Methode legt nahe, dass sich die Java Virtual Machine auf das Recycling nicht verwendeter Objekte konzentriert, um den aktuell gc Speicher für eine schnelle Wiederverwendung verfügbar zu machen. Wenn die Kontrolle vom Methodenaufruf zurückkehrt, hat sich die Java Virtual Machine nach besten Kräften bemüht, die gc Leerzeichen von allen ausrangierten Objekten. "

Es gibt einige wichtige Punkte, die daraus gezogen werden können:

1. Die Verwendung des Wortes "schlägt" anstelle von "sagen" bedeutet, dass die JVM den Vorschlag ignorieren kann. Das Standardverhalten der JVM (aktuelle Versionen) folgt dem Vorschlag. Dies kann jedoch durch Setzen von -XX:+DisableExplicitGC beim Starten der JVM überschrieben werden.

2. Der Ausdruck "Ein bestmöglicher Versuch, Speicherplatz von allen verworfenen Objekten zurückzugewinnen", impliziert, dass der Aufruf von gc eine "vollständige" Garbage Collection auslöst.

Warum ist das Aufrufen von System.gc() eine schlechte Idee?

Erstens ist das Ausführen einer vollständigen Speicherbereinigung teuer. Bei einer vollständigen GC werden alle noch erreichbaren Objekte besucht und "markiert". dh jedes Objekt, das kein Müll ist. Wenn Sie dies auslösen, wenn nicht viel Müll gesammelt werden muss, leistet der GC viel Arbeit für relativ wenig Nutzen.

Zweitens neigt eine vollständige Speicherbereinigung dazu, die "Lokalität" -Eigenschaften der Objekte zu stören, die nicht gesammelt werden. Objekte, die ungefähr zur gleichen Zeit von demselben Thread zugewiesen werden, neigen dazu, im Speicher nahe beieinander zu liegen. Das ist gut. Objekte, die gleichzeitig zugewiesen werden, sind wahrscheinlich miteinander verbunden. dh sich aufeinander beziehen. Wenn Ihre Anwendung diese Verweise verwendet, ist der Speicherzugriff möglicherweise aufgrund verschiedener Speicher- und Seiten-Caching-Effekte schneller. Leider neigt eine vollständige Speicherbereinigung dazu, Objekte zu verschieben, so dass Objekte, die sich einmal in der Nähe befanden, jetzt weiter voneinander entfernt sind.

Drittens kann die Ausführung einer vollständigen Speicherbereinigung dazu führen, dass Ihre Anwendung angehalten wird, bis die Sammlung abgeschlossen ist. Während dies geschieht, reagiert Ihre Anwendung nicht.

In der Tat ist es die beste Strategie, die JVM entscheiden zu lassen, wann der GC ausgeführt wird und welche Art von Sammlung ausgeführt werden soll. Wenn Sie sich nicht einmischen, wählt die JVM einen Zeit- und Erfassungstyp aus, der den Durchsatz optimiert oder die GC-Pausenzeiten minimiert.

Am Anfang haben wir gesagt "... (fast immer) eine schlechte Idee ...". In der Tat gibt es ein paar Szenarien , in denen es *vielleicht* eine gute Idee sein:

1. Wenn Sie einen System.gc() für einen Code implementieren, der die System.gc() z. B. Finalizer oder schwache / weiche / System.gc() möglicherweise ein Aufruf von System.gc() erforderlich.

2. In einigen interaktiven Anwendungen kann es bestimmte Zeitpunkte geben, an denen es dem Benutzer egal ist, ob eine Garbage Collection-Pause vorliegt. Ein Beispiel ist ein Spiel, bei dem im "Spiel" natürliche Pausen auftreten. zB beim Laden eines neuen Levels.

Fallstricke - Die übermäßige Verwendung von primitiven Wrapper-Typen ist ineffizient

Betrachten Sie diese beiden Teile des Codes:

```
int a = 1000;
int b = a + 1;
```

und

```
Integer a = 1000;
Integer b = a + 1;
```

Frage: Welche Version ist effizienter?

Antwort: Die beiden Versionen sehen fast identisch aus, aber die erste Version ist wesentlich

effizienter als die zweite.

Die zweite Version verwendet eine Darstellung für die Nummern, die mehr Platz beansprucht, und setzt im Hintergrund auf das automatische Boxen und das automatische Ausblenden der Boxen. In der Tat entspricht die zweite Version direkt dem folgenden Code:

```
Integer a = Integer.valueOf(1000);               // box 1000
Integer b = Integer.valueOf(a.intValue() + 1);   // unbox 1000, add 1, box 1001
```

Vergleicht man dies mit der anderen Version, die int , gibt es offensichtlich drei zusätzliche Methodenaufrufe, wenn Integer verwendet wird. Im Falle von valueOf erstellen und initialisieren die Aufrufe jeweils ein neues Integer Objekt. All diese zusätzlichen Boxen und Unboxing-Arbeiten werden die zweite Version wahrscheinlich um eine Größenordnung langsamer machen als die erste.

Darüber hinaus valueOf die zweite Version in jedem valueOf Aufruf Objekte auf dem Heap zu. Während die Speicherplatznutzung plattformspezifisch ist, liegt sie wahrscheinlich für jedes Integer Objekt im Bereich von 16 Byte. Im Gegensatz dazu benötigt die int Version a zusätzlichen Heap-Speicherplatz, vorausgesetzt, a und b sind lokale Variablen.

Ein weiterer wichtiger Grund dafür, dass Grundelemente schneller sind als ihre geschachtelten Entsprechungen, ist die Anordnung ihrer jeweiligen Array-Typen im Speicher.

Wenn Sie int[] und Integer[] als Beispiel nehmen, werden die int *Werte* im Fall von int[] zusammenhängend im Speicher abgelegt. Bei Integer[] jedoch nicht die Werte festgelegt, sondern Verweise (Zeiger) auf Integer Objekte, die wiederum die tatsächlichen int Werte enthalten.

Abgesehen davon, dass es eine zusätzliche Ebene der Indirektion ist, kann dies ein großer Panzer sein, wenn es um die Cache-Lokalität geht, wenn die Werte durchlaufen werden. Bei einem int[] die CPU alle Werte des Arrays auf einmal in den Cache abrufen, da sie im Speicher zusammenhängend sind. Bei einem Integer[] die CPU jedoch möglicherweise für jedes Element einen zusätzlichen Speicherabruf ausführen, da das Array nur Verweise auf die tatsächlichen Werte enthält.

Kurz gesagt, die Verwendung von primitiven Wrapper-Typen ist sowohl für CPU- als auch für Speicherressourcen relativ teuer. Sie unnötig zu verwenden, ist effizient.

Fallstricke - Das Durchlaufen der Schlüssel einer Karte kann ineffizient sein

Der folgende Beispielcode ist langsamer als er sein muss:

```
Map<String, String> map = new HashMap<>();
for (String key : map.keySet()) {
    String value = map.get(key);
    // Do something with key and value
}
```

Dies liegt daran, dass für jeden Schlüssel in der Karte ein Map-Lookup (die Methode get()) erforderlich ist. Diese Suche ist möglicherweise nicht effizient (in einer HashMap ist dies das Aufrufen von hashCode für den Schlüssel, das Nachschlagen des richtigen Buckets in internen Datenstrukturen und manchmal sogar das Aufrufen von equals). Auf einer großen Karte ist dies möglicherweise kein unbedeutender Aufwand.

Der richtige Weg, dies zu vermeiden, besteht darin, die Einträge der Karte zu iterieren, die im Thema Sammlungen detailliert beschrieben werden

Fallstricke - Die Verwendung von size () zum Testen, ob eine Sammlung leer ist, ist ineffizient.

Das Java Collections Framework bietet zwei verwandte Methoden für alle Collection Objekte:

- size() gibt die Anzahl der Einträge in einer Collection

- `isEmpty()` Methode `isEmpty()` gibt true zurück, wenn (und nur wenn) die Collection leer ist.

Beide Methoden können verwendet werden, um die Leere der Sammlung zu testen. Zum Beispiel:

```
Collection<String> strings = new ArrayList<>();
boolean isEmpty_wrong = strings.size() == 0; // Avoid this
boolean isEmpty = strings.isEmpty();         // Best
```

Während diese Ansätze gleich aussehen, speichern einige Erfassungsimplementierungen die Größe nicht. Für eine solche Sammlung muss die Implementierung von size() die Größe bei jedem Aufruf berechnen. Zum Beispiel:

- Möglicherweise muss eine einfache verknüpfte Listenklasse (aber nicht die java.util.LinkedList) die Liste durchlaufen, um die Elemente zu zählen.
- Die ConcurrentHashMap Klasse muss die Einträge in allen "Segmenten" der Karte summieren.
- Bei einer langsamen Implementierung einer Sammlung muss möglicherweise die gesamte Sammlung im Speicher implementiert werden, um die Elemente zählen zu können.

Im Gegensatz dazu muss eine isEmpty() -Methode nur testen, ob *mindestens ein* Element in der Auflistung vorhanden ist. Dies beinhaltet nicht das Zählen der Elemente.

Während size() == 0 nicht immer weniger effizient ist als isEmpty() , ist es nicht isEmpty() , dass ein ordnungsgemäß implementiertes isEmpty() weniger effizient ist als size() == 0 . Daher wird isEmpty() bevorzugt.

Pitfall - Effizienzprobleme bei regulären Ausdrücken

Das Abgleichen von regulären Ausdrücken ist ein leistungsfähiges Werkzeug (in Java und in anderen Zusammenhängen), weist jedoch einige Nachteile auf. Einer davon, dass reguläre Ausdrücke eher teuer sind.

Muster- und Matcher-Instanzen sollten wiederverwendet werden

Betrachten Sie das folgende Beispiel:

```
/**
 * 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;
}
```

Dieser Code ist korrekt, aber ineffizient. Das Problem liegt im matches(...) . Unter der Haube entspricht s.matches("[A-Za-z0-9]*") diesem:

```
Pattern.matches(s, "[A-Za-z0-9]*")
```

was wiederum entspricht

```
Pattern.compile("[A-Za-z0-9]*").matcher(s).matches()
```

Der Pattern.compile("[A-Za-z0-9]*") analysiert den regulären Ausdruck, analysiert ihn und

erstellt ein Pattern Objekt, das die Datenstruktur enthält, die von der Regex-Engine verwendet wird. Dies ist eine nicht triviale Berechnung. Dann wird ein Matcher Objekt erstellt, um das Argument s Matcher . Schließlich rufen wir match() auf, um den eigentlichen Musterabgleich durchzuführen.

Das Problem ist, dass diese Arbeit für jede Wiederholungsschleife wiederholt wird. Die Lösung besteht darin, den Code wie folgt umzustrukturieren:

```
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;
}
```

Beachten Sie, dass der Javadoc für Pattern lautet:

> Instanzen dieser Klasse sind unveränderlich und können von mehreren gleichzeitigen Threads verwendet werden. Instanzen der Matcher Klasse sind für eine solche Verwendung nicht sicher.

Verwenden Sie match () nicht, wenn Sie find () verwenden sollten.

Angenommen, Sie möchten testen, ob eine Zeichenfolge s drei oder mehr Ziffern in einer Zeile enthält. Sie können dies auf verschiedene Weise ausdrücken, einschließlich:

```
  if (s.matches(".*[0-9]{3}.*")) {
      System.out.println("matches");
  }
```

oder

```
  if (Pattern.compile("[0-9]{3}").matcher(s).find()) {
      System.out.println("matches");
  }
```

Die erste ist prägnanter, dürfte aber auch weniger effizient sein. Auf den ersten Blick versucht die erste Version, die gesamte Saite mit dem Muster abzugleichen. Da ". *" Ein "gieriges" Muster ist, wird der Mustervergleicher wahrscheinlich "eifrig" bis zum Ende der Zeichenfolge vorrücken und zurückgehen, bis er eine Übereinstimmung findet.

Im Gegensatz dazu sucht die zweite Version von links nach rechts und stoppt die Suche, sobald die 3 Ziffern in einer Reihe gefunden werden.

Verwenden Sie effizientere Alternativen zu regulären Ausdrücken

Reguläre Ausdrücke sind ein mächtiges Werkzeug, sie sollten jedoch nicht Ihr einziges Werkzeug sein. Viele Aufgaben können auf andere Weise effizienter erledigt werden. Zum Beispiel:

```
  Pattern.compile("ABC").matcher(s).find()
```

macht das gleiche wie:

```
  s.contains("ABC")
```

Abgesehen davon, dass letzteres viel effizienter ist. (Auch wenn Sie die Kosten für die Erstellung des regulären Ausdrucks amortisieren können.)

Oft ist die Nicht-Regex-Form komplizierter. Zum Beispiel kann der Test, der vom allAlplanumeric matches() Aufruf der früheren allAlplanumeric Methode ausgeführt wird, wie allAlplanumeric umgeschrieben werden:

```
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;
}
```

Das ist jetzt mehr Code als mit einem Matcher , aber es wird auch wesentlich schneller sein.

Katastrophales Backtracking

(Dies ist möglicherweise ein Problem bei allen Implementierungen regulärer Ausdrücke, wir werden es hier jedoch erwähnen, da dies eine Gefahr für die Verwendung von Pattern .)

Betrachten Sie dieses (erfundene) Beispiel:

```
Pattern pat = Pattern.compile("(A+)+B");
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAB").matches());
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAC").matches());
```

Der erste Aufruf von println wird schnell true gedruckt. Die zweite wird false gedruckt. Schließlich. Wenn Sie mit dem obigen Code experimentieren, werden Sie feststellen, dass sich die Zeit jedes Mal verdoppelt, wenn Sie vor dem C ein A hinzufügen.

Dieses Verhalten ist ein Beispiel für *katastrophales Backtracking* . Die Pattern - Matching - Engine, die die Regex Matching implementiert ist , alle *möglichen* Wege fruchtlos versucht , die das Muster *könnte* passen.

Schauen wir uns an, was (A+)+B tatsächlich bedeutet. Oberflächlich betrachtet scheint es "ein oder mehrere A Zeichen gefolgt von einem B Wert" zu sein, doch in Wirklichkeit heißt es eine oder mehrere Gruppen, von denen jede aus einem oder mehreren A Zeichen besteht. Also zum Beispiel:

- 'AB' gilt nur für eine Richtung: '(A) B'
- "AAB" bietet zwei Möglichkeiten: "(AA) B" oder "(A) (A) B"
- "AAAB" bietet vier Möglichkeiten: "(AAA) B" oder "(AA) (A) B or '(A)(AA)B oder "(A) (A) (A) B"
- und so weiter

Mit anderen Worten ist die Anzahl möglicher Übereinstimmungen 2^N, wobei N die Anzahl von A Zeichen ist.

Das obige Beispiel ist eindeutig durchdacht, aber Muster, die diese Art von Leistungsmerkmalen aufweisen (dh O(2^N) oder O(N^K) für ein großes K), treten häufig auf, wenn unüberlegte reguläre Ausdrücke verwendet werden. Es gibt einige Standardmittel:

- Vermeiden Sie das Verschachteln von sich wiederholenden Mustern in anderen sich wiederholenden Mustern.
- Vermeiden Sie zu viele sich wiederholende Muster.
- Verwenden Sie gegebenenfalls keine Rückverfolgungswiederholung.
- Verwenden Sie keine regulären Ausdrücke für komplizierte Parsing-Aufgaben. (Schreiben Sie stattdessen einen richtigen Parser.)

Achten Sie schließlich auf Situationen, in denen ein Benutzer oder ein API-Client eine reguläre Ausdrücke mit pathologischen Merkmalen angeben kann. Dies kann zu einer versehentlichen oder vorsätzlichen "Dienstverweigerung" führen.

Verweise:

- Der Tag für reguläre Ausdrücke , insbesondere http://www.riptutorial.com/regex/topic/259/getting-started-with-regular-expressions/977/backtracking#t=20161001033913136116 3 und http://www.riptutorial.com/ Regex / Topic / 259 / Erste Schritte mit regulären Ausdrücken / 4527 / Wenn Sie nicht verwendet werden sollten reguläre Ausdrücke # t = 20161001033959356491 3
- "Regex Performance" von Jeff Atwood.
- "So töten Sie Java mit einem regulären Ausdruck" von Andreas Haufler.

Pitfall - Interning von Strings, damit Sie == verwenden können, ist eine schlechte Idee

Wenn einige Programmierer diesen Hinweis sehen:

> "Das Testen von Strings mit == ist falsch (es sei denn, die Strings sind intern)."

Ihre erste Reaktion ist auf interne Zeichenfolgen, so dass sie == . (Schließlich ist == schneller als der Aufruf von String.equals(...) , oder String.equals(...))

Dies ist der falsche Ansatz aus verschiedenen Perspektiven:

Zerbrechlichkeit

Zunächst können Sie == nur sicher verwenden, wenn Sie wissen, dass *alle* von Ihnen getesteten String Objekte intern sind. Das JLS garantiert, dass String-Literale in Ihrem Quellcode intern sind. Abgesehen von String.intern(String) selbst garantiert jedoch keine der Standard-Java SE-APIs die Rückgabe String.intern(String) Zeichenfolgen. Wenn Sie nur eine Quelle von String Objekten vermissen, die noch nicht intern sind, ist Ihre Anwendung unzuverlässig. Diese Unzuverlässigkeit manifestiert sich als falsche Negative und nicht als Ausnahmen, die die Erkennung erschweren könnten.

Kosten für die Verwendung von 'intern ()'

Unter der Haube arbeitet das Internieren, indem es eine Hashtabelle verwaltet, die zuvor internierte String Objekte enthält. Eine Art schwacher Referenzmechanismus wird verwendet, damit die interne Hash-Tabelle nicht zu einem Speicherverlust wird. Während die Hashtabelle in nativem Code implementiert ist (im Gegensatz zu HashMap , HashTable usw.), sind die intern Aufrufe immer noch relativ teuer in Bezug auf die verwendete CPU und den verwendeten Speicher.

Diese Kosten müssen mit den Einsparungen verglichen werden, die wir erhalten, wenn wir == anstelle von equals . Tatsächlich werden wir nicht abbrechen, es sei denn, jeder internierte String wird "einige Male" mit anderen Strings verglichen.

(Abgesehen davon: In den wenigen Situationen, in denen ein Interning sinnvoll ist, geht es in der Regel darum, den Speicherbedarf einer Anwendung zu reduzieren, in der dieselben Zeichenfolgen häufig wiederkehren *und* diese Zeichenfolgen eine lange Lebensdauer haben.)

Die Auswirkungen auf die Müllsammlung

Zusätzlich zu den oben beschriebenen direkten CPU- und Speicherkosten wirken sich intern integrierte Strings auf die Leistung des Garbage Collectors aus.

Für Java-Versionen vor Java 7 werden internierte Zeichenfolgen im "PermGen" -Bereich gespeichert, der selten gesammelt wird. Wenn PermGen gesammelt werden muss, wird (in der Regel) eine vollständige Speicherbereinigung ausgelöst. Wenn der PermGen-Speicherplatz vollständig gefüllt ist, stürzt die JVM ab, auch wenn in den regulären Heap-Speicherbereichen Speicherplatz vorhanden ist.

In Java 7 wurde der String-Pool aus "PermGen" in den normalen Heap verschoben. Die Hash-Tabelle wird jedoch immer noch eine langlebige Datenstruktur sein, die dazu führt, dass alle internen Strings langlebig werden. (Selbst wenn die internierten String-Objekte im Eden-Space zugewiesen wurden, würden sie höchstwahrscheinlich befördert, bevor sie gesammelt wurden.)

In allen Fällen verlängert das Internieren einer Zeichenfolge ihre Lebensdauer im Vergleich zu einer normalen Zeichenfolge. Dies erhöht den Aufwand für die Speicherbereinigung über die Lebensdauer der JVM.

Das zweite Problem besteht darin, dass die Hashtabelle einen schwachen Referenzmechanismus verwenden muss, um zu verhindern, dass String interning Speicher verliert. Ein solcher Mechanismus ist jedoch mehr Arbeit für den Müllsammler.

Diese Müllsammel-Overheads sind schwer zu quantifizieren, aber es gibt wenig Zweifel, dass sie existieren. Wenn Sie intern viel Gebrauch machen, können sie erheblich sein.

Die Hash-Tabellengröße des String-Pools

Gemäß dieser Quelle wird der String-Pool ab Java 6 als Hash-Tabelle mit fester Größe und Ketten implementiert, um Strings zu behandeln, die auf denselben Bucket-Hash zugreifen. In früheren Versionen von Java 6 hatte die Hashtabelle eine (fest verdrahtete) konstante Größe. Als -XX:StringTableSize für Java 6 wurde ein -XX:StringTableSize (-XX:StringTableSize) hinzugefügt. Bei einer Aktualisierung auf Java 7 wurde die Standardgröße des Pools von 1009 auf 60013 .

Die Quintessenz ist, dass, wenn Sie intern intensiv in Ihrem Code verwenden intern , es *ratsam ist* , eine Java-Version auszuwählen, in der die Hashtable-Größe eingestellt werden kann, und stellen Sie sicher, dass Sie die Größe entsprechend anpassen. Andernfalls kann sich die Leistung des intern verschlechtern, wenn der Pool größer wird.

Internierung als potenzieller Denial-of-Service-Vektor

Der Hashcode-Algorithmus für Strings ist allgemein bekannt. Wenn Sie interne Zeichenfolgen verwenden, die von böswilligen Benutzern oder Anwendungen bereitgestellt werden, kann dies als Teil eines DoS-Angriffs (Denial of Service) verwendet werden. Wenn der böswillige Agent veranlasst, dass alle von ihm bereitgestellten Zeichenfolgen denselben Hash-Code aufweisen, kann dies zu einer unausgeglichenen Hash-Tabelle und einer O(N) -Leistung für intern ... führen, wobei N die Anzahl der kollidierten Zeichenfolgen ist.

(Es gibt einfachere / effektivere Methoden, um einen DoS-Angriff auf einen Dienst zu starten. Dieser Vektor könnte jedoch verwendet werden, wenn das Ziel des DoS-Angriffs darin besteht, die Sicherheit zu brechen oder DoS-Abwehrlinien der ersten Wahl zu umgehen.)

Pitfall - Kleine Lese- / Schreibvorgänge für ungepufferte Streams sind ineffizient

Betrachten Sie den folgenden Code, um eine Datei in eine andere zu kopieren:

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

(Wir haben weggelassen normales Argument Überprüfung, Fehlerberichterstattung beraten und so

weiter , weil sie zu *Punkt* dieses Beispiels nicht relevant sind.)

Wenn Sie den obigen Code kompilieren und ihn zum Kopieren einer großen Datei verwenden, werden Sie feststellen, dass er sehr langsam ist. Tatsächlich ist es mindestens um einige Größenordnungen langsamer als die Standard-Dienstprogramme zum Kopieren von Dateien.

(*Fügen Sie hier die tatsächlichen Leistungsmessungen hinzu!*)

Der Hauptgrund dafür, dass das obige Beispiel langsam ist (im Fall der großen Datei), ist, dass Ein-Byte-Lesevorgänge und Ein-Byte-Schreibvorgänge für ungepufferte Byte-Streams ausgeführt werden. Die einfache Möglichkeit, die Leistung zu verbessern, besteht darin, die Streams mit gepufferten Streams zu umschließen. Zum Beispiel:

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

Diese kleinen Änderungen verbessern die Datenkopierrate *um mindestens* einige Größenordnungen, abhängig von verschiedenen plattformbezogenen Faktoren. Die gepufferten Stream-Wrapper bewirken, dass die Daten gelesen und in größere Blöcke geschrieben werden. Beide Instanzen haben Puffer, die als Byte-Arrays implementiert sind.

- Mit is , werden die Daten aus der Datei in den Puffer ein paar Kilobyte zu einem Zeitpunkt lesen. Wenn read() aufgerufen wird, gibt die Implementierung normalerweise ein Byte aus dem Puffer zurück. Es wird nur dann aus dem zugrunde liegenden Eingabestrom gelesen, wenn der Puffer geleert wurde.
- Das Verhalten für os ist analog. Aufrufe von os.write(int) schreiben einzelne Bytes in den Puffer. Daten werden nur in den Ausgabestrom geschrieben, wenn der Puffer voll ist oder wenn os geleert oder geschlossen wird.

Was ist mit zeichenbasierten Streams?

Wie Sie wissen sollten, bietet Java I / O verschiedene APIs zum Lesen und Schreiben von Binär- und Textdaten.

- InputStream und OutputStream sind die Basis-APIs für Stream-basierte binäre E / A
- Reader und Writer sind die Basis-APIs für Stream-basierte Text-E / A.

Für Text-E / A sind BufferedReader und BufferedWriter die Entsprechungen für BufferedInputStream und BufferedOutputStream .

Warum machen gepufferte Streams so viel Unterschied?

Der wahre Grund, dass gepufferte Streams die Leistung verbessern, hängt damit zusammen, wie eine Anwendung mit dem Betriebssystem kommuniziert:

- Die Java-Methode in einer Java-Anwendung oder native Prozeduraufrufe in den Laufzeitbibliotheken der JVM sind schnell. Sie nehmen in der Regel einige Maschinenanweisungen mit und haben nur minimale Auswirkungen auf die Leistung.

- Im Gegensatz dazu sind JVM-Laufzeitaufrufe an das Betriebssystem nicht schnell. Sie beinhalten etwas, das als "Syscall" bekannt ist. Das typische Muster für einen Syscall lautet wie folgt:

    1. Legen Sie die Syscall-Argumente in Registern ab.
    2. Führen Sie eine SYSENTER-Trap-Anweisung aus.
    3. Der Trap-Handler hat in den privilegierten Zustand gewechselt und die Zuordnungen des virtuellen Speichers geändert. Dann sendet es an den Code, um den spezifischen Syscall zu behandeln.
    4. Der syscall-Handler überprüft die Argumente und achtet darauf, dass ihm nicht mitgeteilt wird, dass er auf den Speicher zugreifen soll, den der Benutzerprozess nicht sehen sollte.
    5. Die Syscall-spezifische Arbeit wird ausgeführt. Im Fall einer read syscall kann das bedeuten:
        1. Überprüfen, ob Daten an der aktuellen Position des Dateideskriptors gelesen werden sollen
        2. den Dateisystem-Handler aufrufen, um die erforderlichen Daten von der Festplatte (oder wo auch immer sie gespeichert sind) in den Puffercache zu laden.
        3. Kopieren von Daten aus dem Puffercache an die von JVM bereitgestellte Adresse
        4. Anpassen der Position des punktweisen Dateideskriptors
    6. Rückkehr vom Syscall. Dies bedeutet, dass Sie die VM-Zuordnungen erneut ändern und den privilegierten Status ausschalten.

Wie Sie sich vorstellen können, kann ein einzelner Syscall Tausende von Maschinenanweisungen ausführen. Konservativ *mindestens* zwei Größenordnungen länger als ein normaler Methodenaufruf. (Wahrscheinlich drei oder mehr.)

Aus diesem Grund machen gepufferte Streams einen großen Unterschied, weil sie die Anzahl der Systemaufrufe drastisch reduzieren. Anstatt für jeden read() Aufruf ein Syscall auszuführen, liest der gepufferte Eingabestrom bei Bedarf eine große Datenmenge in einen Puffer. Die meisten read() Aufrufe im gepufferten Stream führen einige einfache Begrenzungen durch und geben ein byte , das zuvor gelesen wurde. Ähnliches gilt für den Ausgabestromfall und auch für den Zeichenstromfall.

(Einige Leute denken, dass die gepufferte E / A-Leistung aus dem Missverhältnis zwischen der Größe der Leseanforderung und der Größe eines Festplattenblocks, der Rotationslatenz der Festplatte und dergleichen resultiert.) In der Tat verwendet ein modernes Betriebssystem eine Reihe von Strategien, um sicherzustellen, dass die Die Anwendung muss *normalerweise* nicht auf die Festplatte warten. Dies ist keine echte Erklärung.)

Sind gepufferte Streams immer ein Gewinn?

Nicht immer. Gepufferte Streams sind definitiv ein Gewinn, wenn Ihre Anwendung viele "kleine" Lese- oder Schreibvorgänge ausführt. Wenn Ihre Anwendung jedoch nur große Lese- oder Schreibvorgänge in / von einem großen byte[] oder char[] ausführen muss, bieten gepufferte Streams keine echten Vorteile. Es kann sogar eine (winzige) Leistungsstrafe geben.

Ist dies der schnellste Weg, eine Datei in Java zu kopieren?

Nein, ist es nicht. Wenn Sie Java-Stream-basierte APIs verwenden, um eine Datei zu kopieren, entstehen Ihnen mindestens eine zusätzliche Kopie der Daten von Arbeitsspeicher. Es ist möglich , diese Option, wenn Ihre Nutzung der NIO zu vermeiden ByteBuffer und Channel - APIs. (*Fügen Sie hier einen Link zu einem separaten Beispiel hinzu.*)

Java-Fallstricke - Leistungsprobleme online lesen:
https://riptutorial.com/de/java/topic/5455/java-fallstricke---leistungsprobleme

Kapitel 73: Java-Fallstricke - Nulls und NullPointerException

Bemerkungen

Der Wert null ist der Standardwert für einen nicht initialisierten Wert eines Felds, dessen Typ ein Referenztyp ist.

NullPointerException (oder NPE) ist die Ausnahme, die ausgelöst wird, wenn Sie versuchen, eine unangemessene Operation für den null Objektverweis auszuführen. Solche Operationen umfassen:

- Aufruf einer Instanzmethode für ein null
- Zugriff auf ein Feld eines null ,
- Versuch, ein null Array-Objekt zu indizieren oder auf seine Länge zuzugreifen
- Verwenden einer null als Mutex in einem synchronized Block,
- Casting einer null Objektreferenz
- Unboxing einer null und
- eine null Objektreferenz werfen.

Die häufigsten Ursachen für NPEs:

- Vergessen, ein Feld mit einem Referenztyp zu initialisieren,
- Vergessen, Elemente eines Arrays eines Referenztyps zu initialisieren, oder
- nicht die Prüfung der Ergebnisse bestimmter API - Methoden , die als wiederkehrende *spezifiziert* sind null unter bestimmten Umständen.

Beispiele für häufig verwendete Methoden, die null sind:

- Die Methode get(key) in der Map API gibt eine null wenn Sie sie mit einem Schlüssel aufrufen, der keine Zuordnung hat.
- Die getResource(path) und getResourceAsStream(path) in den ClassLoader und Class APIs geben null wenn die Ressource nicht gefunden werden kann.
- Die get() -Methode in der Reference API gibt null wenn der Garbage Collector die Referenz gelöscht hat.
- Verschiedene getXxxx Methoden in den Java EE-Servlet-APIs geben null wenn Sie versuchen, einen nicht vorhandenen Anforderungsparameter, eine Sitzung oder ein Sitzungsattribut usw. abzurufen.

Es gibt Strategien, um unerwünschte NPEs zu vermeiden, z. B. das explizite Testen auf null oder die Verwendung von "Yoda Notation". Diese Strategien haben jedoch oft das unerwünschte Ergebnis, dass Probleme in Ihrem Code *verborgen* werden, die wirklich behoben werden müssen.

Examples

Fallstricke - Unnötige Verwendung von primitiven Wrappern kann zu NullPointerExceptions führen

Manchmal verwenden Programmierer mit neuem Java primitive Typen und Wrapper austauschbar. Dies kann zu Problemen führen. Betrachten Sie dieses Beispiel:

```
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
```

Unsere MyRecord Klasse [1] MyRecord auf der Standardinitialisierung, um die Werte in ihren Feldern zu initialisieren. Wenn wir also new eine Aufzeichnung, die a und b werden Felder auf Null gesetzt werden, und die c und d Felder gesetzt werden null .

Wenn wir versuchen, die standardmäßig initialisierten Felder zu verwenden, sehen wir, dass die int Felder die ganze Zeit funktionieren, aber die Integer Felder funktionieren in einigen Fällen und nicht in anderen. In dem Fall, dass ein null NullPointerException (mit d), tritt der Ausdruck rechts auf der rechten Seite auf, einen null Verweis NullPointerException , was dazu führt, dass die NullPointerException ausgelöst wird.

Es gibt mehrere Möglichkeiten, dies zu betrachten:

- Wenn die Felder c und d primitive Wrapper sein müssen, sollten wir uns entweder nicht auf die Standardinitialisierung verlassen oder auf null testen. Für ersteres gilt der richtige Ansatz, es *sei denn,* es gibt eine eindeutige Bedeutung für die Felder im null .
- Wenn es sich bei den Feldern nicht um primitive Wrapper handeln muss, ist es ein Fehler, sie als primitive Wrapper festzulegen. Zusätzlich zu diesem Problem haben die primitiven Wrapper relativ zu primitiven Typen zusätzlichen Aufwand.

Die Lektion hier ist, keine primitiven Wrapper-Typen zu verwenden, es sei denn, Sie müssen es wirklich tun.

---

1 - Diese Klasse ist kein Beispiel für eine gute Kodierungspraxis. Zum Beispiel hätte eine gut entworfene Klasse keine öffentlichen Felder. Dies ist jedoch nicht der Sinn dieses Beispiels.

Pitfall - Verwenden von null zur Darstellung eines leeren Arrays oder einer leeren Sammlung

Einige Programmierer denken, dass es eine gute Idee ist, Platz zu sparen, indem eine null , um ein leeres Array oder eine leere Sammlung darzustellen. Es ist zwar richtig, dass Sie wenig Speicherplatz sparen können, der Code wird jedoch auf der anderen Seite komplizierter und anfälliger. Vergleichen Sie diese beiden Versionen einer Methode zum Summieren eines Arrays:

In der ersten Version wird normalerweise die Methode codiert:

```
/**
 * 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;
}
```

Die zweite Version ist, wie Sie die Methode codieren müssen, wenn Sie normalerweise null , um ein leeres Array darzustellen.

```
/**
 * 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;
}
```

Wie Sie sehen, ist der Code etwas komplizierter. Dies ist direkt auf die Entscheidung zurückzuführen, null auf diese Weise zu verwenden.

Überlegen Sie nun, ob dieses Array, das möglicherweise eine null ist, an vielen Stellen verwendet wird. An jedem Ort, an dem Sie es verwenden, müssen Sie überlegen, ob Sie auf null testen müssen. Wenn Sie eine verpassen null Test, der es sein muss, riskieren Sie eine NullPointerException . Daher führt die Strategie der Verwendung von null auf diese Weise dazu, dass Ihre Anwendung fragiler wird. dh anfälliger für die Folgen von Programmierfehlern.

Die Lektion hier ist, leere Arrays und leere Listen zu verwenden, wenn Sie das meinen.

```
int[] values = new int[0];                      // always empty
List<Integer> list = new ArrayList();           // initially empty
List<Integer> list = Collections.emptyList();  // always empty
```

Der Platzaufwand ist gering, und es gibt andere Möglichkeiten, ihn zu minimieren, wenn sich dies lohnt.

Pitfall - unerwartete Nullen "gut machen"

Auf StackOverflow wird in den Antworten häufig Code wie folgt angezeigt:

```
public String joinStrings(String a, String b) {
    if (a == null) {
        a = "";
    }
    if (b == null) {
        b = "";
    }
    return a + ": " + b;
}
```

Dies wird häufig von einer Behauptung begleitet, die "bewährte NullPointerException " ist, um auf null zu testen, um NullPointerException zu vermeiden.

Ist es die beste Praxis? Kurz gesagt: Nein.

Es gibt einige Grundannahmen, die hinterfragt werden müssen, bevor wir sagen können, ob dies in unseren joinStrings :

Was bedeutet es, dass "a" oder "b" null ist?

Ein String Wert kann null oder mehr Zeichen sein, daher haben wir bereits die Möglichkeit, einen leeren String darzustellen. Bedeutet null etwas anderes als "" ? Wenn nein, ist es problematisch, eine leere Zeichenfolge auf zwei Arten darzustellen.

Kommt die Null aus einer nicht initialisierten Variablen?

Eine null kann aus einem nicht initialisierten Feld oder einem nicht initialisierten Array-Element stammen. Der Wert kann von Entwurf oder von Zufall nicht initialisiert werden. Wenn es ein Zufall war, dann ist dies ein Fehler.

Steht die Null für "Weiß nicht" oder "Fehlender Wert"?

Manchmal kann eine null eine echte Bedeutung haben. B., dass der tatsächliche Wert einer

Variablen unbekannt oder nicht verfügbar oder "optional" ist. In Java 8 bietet die Optional Klasse eine bessere Möglichkeit, dies auszudrücken.

Wenn dies ein Fehler (oder ein Konstruktionsfehler) ist, sollten wir "gut machen"?

Eine Interpretation des Codes besteht darin, dass wir eine unerwartete null durch eine leere Zeichenfolge ersetzen. Ist die richtige Strategie? Wäre es besser, die NullPointerException passieren zu lassen, die Ausnahme weiter oben im Stack NullPointerException und als Fehler zu protokollieren?

Das Problem beim "Wiedergutmachen" besteht darin, dass das Problem entweder verdeckt wird oder die Diagnose schwieriger wird.

Ist das effizient / gut für die Codequalität?

Wenn der "Make Good" -Ansatz konsequent verwendet wird, enthält Ihr Code viele "defensive" Nulltests. Dies macht es länger und schwieriger zu lesen. Außerdem können all diese Tests und "Wiedergutmachung" die Leistung Ihrer Anwendung beeinträchtigen.

in Summe

Wenn null ein sinnvoller Wert ist, ist der korrekte Ansatz der Test auf den null . Die Folge ist, dass, wenn ein null sinnvoll ist, dieser in den Javadocs von Methoden, die den null akzeptieren oder ihn zurückgeben, eindeutig dokumentiert werden sollte.

Andernfalls ist es eine bessere Idee, eine unerwartete null als Programmierfehler zu behandeln und die NullPointerException passieren zu lassen, damit der Entwickler NullPointerException , dass ein Problem im Code vorliegt.

Pitfall - Rückgabe von Null anstelle einer Ausnahme

Einige Java-Programmierer haben eine generelle Abneigung gegen das Auslösen oder Weiterleiten von Ausnahmen. Dies führt zu Code wie dem folgenden:

```
public Reader getReader(String pathname) {
    try {
        return new BufferedReader(FileReader(pathname));
    } catch (IOException ex) {
        System.out.println("Open failed: " + ex.getMessage());
        return null;
    }
```

}

Was ist das Problem damit?

Das Problem ist, dass der getReader eine null als speziellen Wert getReader um anzuzeigen, dass der Reader nicht geöffnet werden konnte. Nun muss der Rückgabewert überprüft werden , um zu sehen , ob es null , bevor sie verwendet wird. Wenn der Test ausgelassen wird, ist das Ergebnis eine NullPointerException .

Hier gibt es eigentlich drei Probleme:

1. Die IOException wurde zu früh aufgefangen.
2. Aufgrund des Aufbaus dieses Codes besteht die Gefahr, dass eine Ressource ausläuft.
3. Eine null wurde verwendet und dann zurückgegeben, da kein "echter" Reader verfügbar war.

Unter der Annahme, dass die Ausnahme so früh gefasst werden musste, gab es mehrere Alternativen für die Rückgabe von null :

1. Es wäre möglich, eine NullReader Klasse zu implementieren. B. eine, bei der sich die Operationen der API so verhalten, als befände sich der Leser bereits an der Position "Dateiende".

2. Mit Java 8 könnte getReader als Optional<Reader> .

Fallstricke - Nicht geprüft, ob ein E / A-Datenstrom beim Schließen noch nicht einmal initialisiert wurde

Um Speicherverluste zu vermeiden, sollten Sie nicht vergessen, einen Eingabestrom oder einen Ausgabestrom zu schließen, dessen Job erledigt ist. Dies geschieht normalerweise mit einer try catch finally Anweisung ohne den catch Teil:

```
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();
    }
}
```

Obwohl der obige Code unschuldig wirkt, weist er einen Fehler auf, der das Debuggen unmöglich machen kann. Wenn die Zeile , in der out initialisiert (out = new FileOutputStream(filename)) eine Ausnahme auslöst, dann out wird null , wenn out.close() ausgeführt wird, was zu einem heftigen NullPointerException !

Um dies zu verhindern, stellen Sie einfach sicher, dass der Stream nicht null bevor Sie versuchen, ihn zu schließen.

```
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();
    }
}
```

Ein noch besserer Ansatz ist es try -with-resources zu try , da der Stream automatisch mit einer Wahrscheinlichkeit von 0 geschlossen wird, um eine NPE zu werfen, ohne dass ein finally Block erforderlich ist.

```
void writeNullBytesToAFile(int count, String filename) throws IOException {
    try (FileOutputStream out = new FileOutputStream(filename)) {
        for(; count > 0; count--)
            out.write(0);
    }
}
```

Pitfall - Verwenden der "Yoda-Notation", um NullPointerException zu vermeiden

Eine Menge von Beispielcode, der in StackOverflow veröffentlicht wird, enthält folgende Ausschnitte:

```
if ("A".equals(someString)) {
    // do something
}
```

Dies "verhindert" oder "vermeidet" eine mögliche NullPointerException für den Fall, dass someString null . Darüber hinaus ist das fraglich

```
    "A".equals(someString)
```

ist besser als:

```
    someString != null && someString.equals("A")
```

(Es ist kürzer und kann unter Umständen effizienter sein. Wie wir jedoch weiter unten argumentieren, könnte die Prägnanz negativ sein.)

Die eigentliche Fallstricke verwendet jedoch den Yoda-Test , **um NullPointerExceptions** aus Gewohnheit **zu vermeiden** .

Wenn Sie "A".equals(someString) schreiben, "A".equals(someString) Sie tatsächlich den "Fall" gut, in dem someString zufällig null . Ein anderes Beispiel (Pitfall - "Making good" unerwartete Nullen) erklärt jedoch, dass "gute" null aus verschiedenen Gründen schädlich sein können.

Dies bedeutet, dass Yoda-Bedingungen keine "Best Practice" sind [1] . Wenn keine null erwartet wird, ist es besser, die NullPointerException passieren zu lassen, damit ein Unit-Test-Fehler (oder ein Fehlerbericht) auftreten kann. Auf diese Weise können Sie den Fehler finden und beheben, durch den die unerwartete / unerwünschte null angezeigt wurde.

Yoda-Bedingungen sollten nur in Fällen verwendet werden, in denen die null *erwartet wird,* da das von Ihnen getestete Objekt von einer API stammt , *die* als null . Und es könnte besser sein, eine der weniger hübschen Methoden zu verwenden, um den Test auszudrücken, da dies den null für jemanden hervorhebt, der Ihren Code überprüft.

---

1 - Laut Wikipedia : *"Beste Kodierungspraktiken sind eine Reihe informeller Regeln, die die Softwareentwicklungsgemeinschaft im Laufe der Zeit gelernt hat und die dazu beitragen kann, die Qualität der Software zu verbessern."* . Die Verwendung der Yoda-Notation erreicht dies nicht. In vielen Situationen wird der Code dadurch schlechter.

Java-Fallstricke - Nulls und NullPointerException online lesen:
https://riptutorial.com/de/java/topic/5680/java-fallstricke---nulls-und-nullpointerexception

Kapitel 74: Java-Fallstricke - Sprachsyntax

Einführung

Bei einem Missbrauch von Java-Programmiersprachen kann es vorkommen, dass ein Programm fehlerhafte Ergebnisse generiert, obwohl es korrekt kompiliert wurde. Das Hauptziel dieses Themas ist es, häufige Fallstricke mit ihren Ursachen aufzulisten und den richtigen Weg vorzuschlagen, um zu vermeiden, dass solche Probleme auftreten.

Bemerkungen

In diesem Thema werden bestimmte Aspekte der Java-Sprachsyntax behandelt, die entweder fehleranfällig sind oder auf bestimmte Weise nicht verwendet werden sollten.

Examples

Pitfall - Ignoriert die Sichtbarkeit der Methode

Selbst erfahrene Java-Entwickler neigen zu der Annahme, dass Java nur drei Schutzmodifikatoren hat. Die Sprache hat eigentlich vier! Das **Paket private** (auch bekannt als Standard) Sichtbarkeitsebene wird häufig vergessen.

Sie sollten darauf achten, welche Methoden Sie veröffentlichen. Die öffentlichen Methoden in einer Anwendung sind die sichtbare API der Anwendung. Dies sollte so klein und kompakt wie möglich sein, insbesondere wenn Sie eine wiederverwendbare Bibliothek schreiben (siehe auch das SOLID- Prinzip). Es ist wichtig, die Sichtbarkeit aller Methoden in ähnlicher Weise zu berücksichtigen und den geschützten oder geschützten privaten Zugriff nur bei Bedarf zu verwenden.

Wenn Sie Methoden deklarieren, die **privat** als öffentlich gelten sollen, legen Sie die internen Implementierungsdetails der Klasse offen.

Eine Folge davon ist, dass Sie nur die öffentlichen Methoden Ihrer Klasse als Unit-Test testen. Tatsächlich können Sie **nur** öffentliche Methoden testen. Es ist nicht ratsam, die Sichtbarkeit privater Methoden zu erhöhen, nur um Unit-Tests mit diesen Methoden durchführen zu können. Das Testen öffentlicher Methoden, die die Methoden mit einer restriktiveren Sichtbarkeit aufrufen, sollte ausreichen, um eine gesamte API zu testen. Sie sollten Ihre API **niemals** mit mehr öffentlichen Methoden erweitern, um nur Komponententests zu ermöglichen.

Pitfall - Fehlende "Pause" in einem "Switch" -Fall

Diese Java-Probleme können sehr peinlich sein und bleiben bis zum Start in der Produktion unentdeckt. Durchbruchsverhalten in switch-Anweisungen ist häufig nützlich. Wenn jedoch ein Schlüsselwort „break" fehlt, wenn ein solches Verhalten nicht erwünscht ist, kann dies katastrophale Folgen haben. Wenn Sie vergessen haben, im folgenden Codebeispiel eine "Pause" in "Fall 0" einzugeben, schreibt das Programm "Null" gefolgt von "Eins", da der Steuerfluss hier die gesamte "switch" -Anweisung durchläuft es erreicht eine "Pause". Zum Beispiel:

```
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");
        }
}
```

In den meisten Fällen besteht die einfachere Lösung darin, Schnittstellen zu verwenden und Code mit spezifischem Verhalten in separate Implementierungen zu verschieben (*Komposition über Vererbung*).

Wenn eine switch-Anweisung unvermeidbar ist, wird empfohlen, "erwartete" Fallthroughs zu dokumentieren, wenn sie auftreten. Auf diese Weise zeigen Sie anderen Entwicklern, dass Sie über die fehlende Unterbrechung Bescheid wissen und dass dies ein erwartetes Verhalten ist.

```
switch(caseIndex) {
    [...]
    case 2:
        System.out.println("Two");
        // fallthrough
    default:
        System.out.println("Default");
```

Pitfall - falsch platzierte Semikolons und fehlende Klammern

Dies ist ein Fehler, der bei Java-Anfängern echte Verwirrung stiftet, zumindest beim ersten Mal. Anstatt dies zu schreiben:

```
if (feeling == HAPPY)
    System.out.println("Smile");
else
    System.out.println("Frown");
```

Sie schreiben aus Versehen:

```
if (feeling == HAPPY);
    System.out.println("Smile");
else
    System.out.println("Frown");
```

und sind verwirrt, wenn der Java-Compiler ihnen mitteilt, dass das else falsch ist. Der Java-Compiler interpretiert das oben wie folgt:

```
if (feeling == HAPPY)
    /*empty statement*/ ;
System.out.println("Smile");   // This is unconditional
else                           // This is misplaced.  A statement cannot
                               // start with 'else'
System.out.println("Frown");
```

In anderen Fällen treten keine Kompilierungsfehler auf, der Code entspricht jedoch nicht dem, was der Programmierer beabsichtigt. Zum Beispiel:

```
for (int i = 0; i < 5; i++);
    System.out.println("Hello");
```

druckt nur einmal "Hallo". Wiederum bedeutet das falsche Semikolon, dass der Rumpf der for Schleife eine leere Anweisung ist. Das bedeutet, dass der folgende println Aufruf unbedingt ist.

Eine andere Variante:

```
for (int i = 0; i < 5; i++);
    System.out.println("The number is " + i);
```

Dies führt zu einem Fehler "Ich konnte kein Symbol finden" für i . Das Vorhandensein des falschen Semikolons bedeutet, dass der Aufruf von println versucht, i außerhalb seines Gültigkeitsbereichs zu verwenden.

In diesen Beispielen gibt es eine einfache Lösung: Löschen Sie einfach das falsche Semikolon. Es gibt jedoch einige tiefere Lehren aus diesen Beispielen:

1. Das Semikolon in Java ist kein "syntaktisches Rauschen". Das Vorhandensein oder Fehlen eines Semikolons kann die Bedeutung Ihres Programms ändern. Fügen Sie sie nicht einfach am Ende jeder Zeile hinzu.
2. Vertrauen Sie dem Einzug Ihres Codes nicht. In der Java-Sprache werden zusätzliche Leerzeichen am Anfang einer Zeile vom Compiler ignoriert.
3. Verwenden Sie einen automatischen Indenter. Alle IDEs und viele einfache Texteditoren wissen, wie Java-Code korrekt eingerückt wird.
4. Dies ist die wichtigste Lektion. Folgen Sie den neuesten Java-Stilrichtlinien und setzen Sie die Anweisungen "then" und "else" sowie die body-Anweisung einer Schleife in Klammern. Die offene Klammer ({) sollte sich nicht in einer neuen Zeile befinden.

Wenn der Programmierer die Stilregeln befolgt, würde das if Beispiel mit einem falsch platzierten Semikolon folgendermaßen aussehen:

```
if (feeling == HAPPY); {
    System.out.println("Smile");
} else {
    System.out.println("Frown");
}
```

Das sieht für ein erfahrenes Auge merkwürdig aus. Wenn Sie diesen Code automatisch eingerückt haben, würde er wahrscheinlich so aussehen:

```
if (feeling == HAPPY); {
                           System.out.println("Smile");
                       } else {
                           System.out.println("Frown");
                       }
```

das sollte sogar für einen Anfänger als falsch auffallen.

Fallstricke - Klammern weglassen: Probleme mit dem "baumelnden wenn" und "hängenden anderen"

Die neueste Version des Oracle Java-Style-Guide verlangt, dass die Anweisungen "then" und "else" in einer if Anweisung immer in "geschweifte Klammern" oder "geschweifte Klammern" eingeschlossen werden. Ähnliche Regeln gelten für die Körper verschiedener Schleifenanweisungen.

```
if (a) {           // <- open brace
    doSomething();
    doSomeMore();
}                  // <- close brace
```

Dies ist für die Java-Sprachsyntax eigentlich nicht erforderlich. Wenn der "dann" -Teil einer if Anweisung eine einzige Anweisung ist, ist es legal, die Klammern wegzulassen

```
if (a)
    doSomething();
```

oder auch

```
if (a) doSomething();
```

Es gibt jedoch Gefahren, wenn Java-Regeln ignoriert und die Klammern weggelassen werden. Insbesondere erhöhen Sie das Risiko, dass Code mit fehlerhafter Einrückung falsch gelesen wird.

Das "baumelnde if" Problem:

Betrachten Sie den Beispielcode von oben, ohne Klammern umgeschrieben.

```
if (a)
   doSomething();
   doSomeMore();
```

Dieser Code *scheint zu sagen,* dass die Aufrufe von doSomething und doSomeMore sowohl *dann als auch nur dann* auftreten, *wenn* a true . Tatsächlich ist der Code falsch eingerückt. Die Java-Sprachspezifikation, die der Aufruf von doSomeMore() nach der if Anweisung darstellt. Die richtige Einrückung lautet wie folgt:

```
if (a)
   doSomething();
doSomeMore();
```

Das Problem des "baumelnden anderen"

Ein zweites Problem tritt auf, wenn wir der Mischung else hinzufügen. Betrachten Sie das folgende Beispiel mit fehlenden geschweiften Klammern.

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
else
   doZ();
```

Der obige Code *scheint zu sagen,* dass doZ wird, wenn a false . In der Tat ist der Einzug erneut falsch. Die richtige Einrückung für den Code lautet:

```
if (a)
   if (b)
      doX();
   else if (c)
      doY();
   else
      doZ();
```

Wenn der Code gemäß den Regeln des Java-Stils geschrieben wurde, würde er tatsächlich so aussehen:

```
if (a) {
   if (b) {
      doX();
   } else if (c) {
      doY();
   } else {
      doZ();
   }
}
```

Um zu veranschaulichen, warum das besser ist, nehmen Sie an, Sie hätten den Code versehentlich falsch eingerückt. Sie könnten mit so etwas enden:

```
if (a) {                          if (a) {
   if (b) {                           if (b) {
      doX();                             doX();
   } else if (c) {                    } else if (c) {
      doY();                             doY();
} else {                          } else {
   doZ();                             doZ();
}                                     }
}                                 }
```

In beiden Fällen sieht der falsch eingerückte Code jedoch für einen erfahrenen Java-Programmierer falsch aus.

Pitfall - Überladen statt überschreiben

Betrachten Sie das folgende Beispiel:

```
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();
    }
}
```

Dieser Code verhält sich nicht wie erwartet. Das Problem ist , dass die equals und hashcode Methoden zur Person nicht über die Standardmethoden durch definierte außer Kraft setzen Object .

- Die Methode equals hat die falsche Signatur. Es sollte als equals(Object) nicht equals(String) deklariert werden.
- Die hashcode Methode hat den falschen Namen. Es sollte hashCode() (beachten Sie die Hauptstadt **C**).

Diese Fehler bedeuten, dass wir versehentliche Überladungen deklariert haben. Diese werden nicht verwendet, wenn Person in einem polymorphen Kontext verwendet wird.

Es gibt jedoch einen einfachen Weg, um damit umzugehen (ab Java 5). Verwenden Sie die Annotation @Override wenn Sie *beabsichtigen, dass* Ihre Methode eine Überschreibung ist:

Java SE 5

```
public final class Person {
    ...
```

```
    @Override
    public boolean equals(String other) {
        ....
    }

    @Override
    public hashcode() {
        ....
    }
}
```

Wenn wir eine hinzufügen @Override Anmerkung zu einer Methodendeklaration, prüft der Compiler , dass die Methode überschreibt , *nicht* (oder implementieren) eine Methode in einer übergeordneten Klasse oder Schnittstelle deklarierte. Im obigen Beispiel gibt der Compiler zwei Kompilierungsfehler aus, die ausreichen sollten, um auf den Fehler aufmerksam zu machen.

Pitfall - Oktal Literale

Betrachten Sie den folgenden Codeausschnitt:

```
// 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);
```

Ein Java-Anfänger könnte überrascht sein, dass das obige Programm die falsche Antwort ausgibt. Tatsächlich wird die Summe der Zahlen 1 bis 8 ausgegeben.

Der Grund ist, dass ein Integer-Literal, das mit der Ziffer Null ('0') beginnt, vom Java-Compiler als Oktal-Literal und nicht als Dezimal-Literal interpretiert wird. Somit ist 010 die Oktalzahl 10, die in Dezimalzahl 8 ist.

Pitfall - Deklarieren von Klassen mit denselben Namen wie Standardklassen

Manchmal machen Programmierer, die mit Java noch nicht vertraut sind, den Fehler, eine Klasse mit einem Namen zu definieren, der mit einer weit verbreiteten Klasse identisch ist. Zum Beispiel:

```
package com.example;

/**
 * My string utilities
 */
public class String {
    ....
}
```

Dann fragen sie sich, warum sie unerwartete Fehler bekommen. Zum Beispiel:

```
package com.example;

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

Wenn Sie die obigen Klassen kompilieren und dann versuchen, sie auszuführen, erhalten Sie eine Fehlermeldung:

```
$ 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
```

Jemand, der sich den Code für die Test Klasse ansieht, würde die Deklaration von main sehen, sich die Signatur ansehen und sich fragen, worüber sich der java Befehl beschwert. Tatsächlich sagt der java Befehl jedoch die Wahrheit.

Wenn wir eine Version von String im selben Paket wie Test deklarieren, hat diese Version Vorrang vor dem automatischen Import von java.lang.String . Die Signatur der Test.main Methode ist also tatsächlich

```
void main(com.example.String[] args)
```

anstatt

```
void main(java.lang.String[] args)
```

und der java Befehl erkennt *das nicht* als eine Trypoint-Methode.

Lektion: Definieren Sie keine Klassen, die denselben Namen wie vorhandene Klassen in java.lang oder andere häufig verwendete Klassen in der Java SE-Bibliothek haben. Wenn Sie das tun, öffnen Sie sich für alle Arten von dunklen Fehlern.

Pitfall - Verwenden Sie '==', um einen Boolean zu testen

Manchmal schreibt ein neuer Java-Programmierer Code wie folgt:

```
public void check(boolean ok) {
    if (ok == true) {           // Note 'ok == true'
        System.out.println("It is OK");
    }
}
```

Ein erfahrener Programmierer würde das als unbeholfen wahrnehmen und möchte es wie folgt umschreiben:

```
public void check(boolean ok) {
    if (ok) {
       System.out.println("It is OK");
    }
}
```

Es ist jedoch mehr falsch mit ok == true als einfache Ungeschicklichkeit. Betrachten Sie diese Variante:

```
public void check(boolean ok) {
    if (ok = true) {           // Oooops!
        System.out.println("It is OK");
    }
}
```

Hier hat der Programmierer == as = ... falsch geschrieben, und jetzt hat der Code einen subtilen Fehler. Der Ausdruck x = true weist bedingungslos true zu x und wertet dann zu true . Mit

anderen Worten, die check gibt nun "Es ist OK" aus, unabhängig von dem Parameter.

Die Lektion hier ist, sich der Gewohnheit zu == false , == false und == true . Sie sind nicht nur ausführlich, sondern machen sie auch fehleranfälliger.

Hinweis: Eine mögliche Alternative zu ok == true , die die Gefahr vermeidet, besteht in der Verwendung von Yoda-Bedingungen . dh setzen Sie das Literal auf die linke Seite des relationalen Operators, wie in true == ok . Dies funktioniert, aber die meisten Programmierer würden wahrscheinlich zustimmen, dass die Bedingungen von Yoda ungerade aussehen. Sicher ist ok (oder !ok) prägnanter und natürlicher.

Fallstricke - Wildcard-Importe können Ihren Code brüchig machen

Betrachten Sie das folgende Teilbeispiel:

```
import com.example.somelib.*;
import com.acme.otherlib.*;

public class Test {
    private Context x = new Context();   // from com.example.somelib
    ...
}
```

Angenommen, Sie haben den Code bei der ersten Entwicklung gegen Version 1.0 von somelib und Version 1.0 von otherlib . Zu einem späteren Zeitpunkt müssen Sie Ihre Abhängigkeiten auf eine otherlib Version aktualisieren, und Sie entscheiden sich für otherlib Version 2.0. Nehmen Sie außerdem an, dass eine der Änderungen, die sie zwischen 1.0 und 2.0 an otherlib , das Hinzufügen einer Context Klasse war.

Wenn Sie jetzt den Test kompilieren, wird ein Kompilierungsfehler angezeigt, der Sie darüber Context dass Context ein mehrdeutiger Import ist.

Wenn Sie mit der Codebase vertraut sind, ist dies wahrscheinlich nur eine geringfügige Unannehmlichkeit. Wenn nicht, dann haben Sie einige Arbeit zu tun, um dieses Problem hier und möglicherweise anderswo anzugehen.

Das Problem hier sind die Wildcard-Importe. Einerseits kann die Verwendung von Platzhaltern die Anzahl der Klassen um ein paar Zeilen verkürzen. Auf der anderen Seite:

- Aufwärtskompatible Änderungen an anderen Teilen Ihrer Codebase, an Java-Standardbibliotheken oder an Drittanbieter-Bibliotheken können zu Kompilierungsfehlern führen.
- Lesbarkeit leidet. Wenn Sie keine IDE verwenden, kann es schwierig sein, herauszufinden, welcher der Importe von Platzhaltern eine benannte Klasse zieht.

Die Lektion ist, dass es keine gute Idee ist, Wildcard-Importe in Code zu verwenden, der langlebig sein muss. Bestimmte (Nicht-Platzhalter) -Importe sind bei der Verwendung einer IDE nur schwer zu pflegen, und der Aufwand lohnt sich.

Pitfall: Verwenden von 'Assert' für die Validierung von Argumenten oder Benutzereingaben

Eine Frage , die gelegentlich auf Stackoverflow ist , ob es angemessen ist , zu verwenden assert Argumente durch den Benutzer auf ein Verfahren oder auch Eingänge vorgesehen geliefert zu validieren.

Die einfache Antwort ist, dass es nicht angemessen ist.

Bessere Alternativen sind:

- Eine IllegalArgumentException mit benutzerdefiniertem Code auslösen.
- Verwenden der in Google Guava-Bibliothek verfügbaren Preconditions Methoden.

- Verwenden der Validate Methoden, die in der Apache Commons Lang3-Bibliothek verfügbar sind.

Dies empfiehlt die Java-Sprachspezifikation (JLS 14.10 für Java 8) in dieser Angelegenheit:

> Normalerweise ist die Assertionsprüfung während der Programmentwicklung und beim Testen aktiviert und für die Bereitstellung deaktiviert, um die Leistung zu verbessern.
>
> Da Assertions möglicherweise deaktiviert sind, dürfen Programme nicht davon ausgehen, dass die in Assertions enthaltenen Ausdrücke ausgewertet werden. Daher sollten diese booleschen Ausdrücke im Allgemeinen frei von Nebenwirkungen sein. Die Auswertung eines solchen booleschen Ausdrucks sollte keinen Status beeinflussen, der nach Abschluss der Auswertung sichtbar ist. Es ist nicht illegal, dass ein boolescher Ausdruck, der in einer Assertion enthalten ist, Nebeneffekte hat, aber im Allgemeinen ist dies ungeeignet, da das Programmverhalten davon abhängen kann, ob Assertions aktiviert oder deaktiviert wurden.
>
> Vor diesem Hintergrund sollten Assertions nicht zur Argumentprüfung in öffentlichen Methoden verwendet werden. Die Überprüfung von Argumenten ist normalerweise Teil des Vertrages einer Methode. Dieser Vertrag muss bestätigt werden, unabhängig davon, ob Zusicherungen aktiviert oder deaktiviert sind.
>
> Ein sekundäres Problem bei der Verwendung von Assertions für die Argumentprüfung ist, dass fehlerhafte Argumente zu einer entsprechenden Laufzeitausnahme führen sollten (z. B. IllegalArgumentException , ArrayIndexOutOfBoundsException oder NullPointerException). Bei einem Assertionsfehler wird keine entsprechende Ausnahme ausgelöst. Es ist wiederum nicht illegal, Zusicherungen für die Argumentprüfung öffentlicher Methoden zu verwenden, ist aber generell unangemessen. Es ist beabsichtigt, dass AssertionError niemals abgefangen wird, aber es ist möglich, dies zu tun. Daher sollten die Regeln für try-Anweisungen Assertions behandeln, die in einem try-Block erscheinen, ähnlich wie die aktuelle Behandlung von throw-Anweisungen.

Fallstricke beim automatischen Auspacken von Nullobjekten in Grundkörper

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

        // example:
        Boolean ignore = null;
        if (ignore == false) {
            System.out.println("Do not ignore!");
        }
    }
}
```

Die Falle ist hier, dass null mit false verglichen wird. Da wir ein primitives boolean mit einem Boolean , versucht Java, das Boolean Object in ein primitives Äquivalent zu *entpacken,* das zum Vergleich bereit ist. Da dieser Wert jedoch null , wird eine NullPointerException ausgelöst.

Java kann NullPointerException primitiven Typen mit null , was zur Laufzeit eine NullPointerException verursacht. Betrachten Sie den primitiven Fall der Bedingung false == null ; Dies würde einen *Kompilierzeitfehler* erzeugen *, der nicht vergleichbar ist* incomparable types: int and <null> .

Java-Fallstricke - Sprachsyntax online lesen: https://riptutorial.com/de/java/topic/5382/java-fallstricke---sprachsyntax

Kapitel 75: Java-Fallstricke - Threads und Parallelität

Examples

Fallstricke: falsche Verwendung von wait () / notify ()

Die Methoden object.wait() , object.notify() und object.notifyAll() sollen auf ganz bestimmte Weise verwendet werden. (siehe http://stackoverflow.com/documentation/java/5409/wait-notify#t=20160811161648303307)

Das Problem "Lost Notification"

Ein häufiger Anfängerfehler ist der bedingungslose Aufruf von object.wait()

```
private final Object lock = new Object();

public void myConsumer() {
    synchronized (lock) {
        lock.wait();     // DON'T DO THIS!!
    }
    doSomething();
}
```

Dies ist falsch, weil es von einem anderen Thread abhängt, der lock.notify() oder lock.notifyAll() , aber nichts garantiert, dass der andere Thread diesen Aufruf nicht *vor* dem Consumer-Thread namens lock.wait() .

lock.notify() und lock.notifyAll() führen nichts aus, wenn ein anderer Thread nicht *bereits* auf die Benachrichtigung wartet. Der Thread, der myConsumer() in diesem Beispiel aufruft, myConsumer() für immer hängen, wenn es zu spät ist, um die Benachrichtigung myConsumer() .

Der "Illegal Monitor State" Fehler

Wenn Sie wait() oder notify() für ein Objekt aufrufen wait() ohne die Sperre zu halten, IllegalMonitorStateException die JVM die IllegalMonitorStateException .

```
public void myConsumer() {
    lock.wait();      // throws exception
    consume();
}

public void myProducer() {
    produce();
    lock.notify();    // throws exception
}
```

(Der Entwurf für wait() / notify() erfordert das Halten der Sperre, da dies zur Vermeidung systembedingter Race-Bedingungen erforderlich ist. Wenn wait() oder notify() ohne Sperren aufgerufen werden konnte, ist die Implementierung unmöglich der primäre Anwendungsfall für diese Grundelemente: Warten auf das Auftreten einer Bedingung.)

Warten / Benachrichtigen ist zu niedrig

Probleme mit wait() und notify() vermeiden Sie am *besten* , notify() sie nicht verwenden. Die meisten Synchronisationsprobleme können durch die Verwendung von Synchronisationsobjekten auf höherer Ebene (Warteschlangen, Barrieren, Semaphoren usw.) gelöst werden, die im Paket java.utils.concurrent verfügbar sind.

Pitfall - Erweiterung 'java.lang.Thread'

Der Javadoc für die Thread Klasse zeigt zwei Möglichkeiten zum Definieren und Verwenden eines Threads:

Verwenden einer benutzerdefinierten Thread-Klasse:

```
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();
```

Mit einem 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();
```

(Quelle: java.lang.Thread javadoc .)

Der Ansatz der benutzerdefinierten Thread-Klasse funktioniert, hat jedoch einige Probleme:

1. Es ist umständlich, PrimeThread in einem Kontext zu verwenden, der einen klassischen Threadpool, einen Executor oder das ForkJoin-Framework verwendet. (Es ist nicht unmöglich, weil PrimeThread indirekt implementiert Runnable , aber mit einer benutzerdefinierten Thread Klasse als ein Runnable sicherlich ungeschickt ist und nicht lebensfähig sein ... in Abhängigkeit von anderen Aspekten der Klasse.)

2. Es gibt mehr Möglichkeiten für Fehler in anderen Methoden. Wenn Sie beispielsweise ein PrimeThread.start() deklariert PrimeThread.start() ohne an Thread.start() zu Thread.start() , erhalten Sie am Ende einen "Thread", der im aktuellen Thread ausgeführt wurde.

Durch den Ansatz, die Thread-Logik in ein Runnable diese Probleme vermieden. Wenn Sie eine anonyme Klasse (Java 1.1 und höher) verwenden, um Runnable zu implementieren, Runnable das Ergebnis prägnanter und lesbarer als in den obigen Beispielen.

```
final long minPrime = ...
new Thread(new Runnable() {
    public void run() {
        // compute primes larger than minPrime
         . . .
    }
```

```
}.start();
```

Mit einem Lambda-Ausdruck (ab Java 8) würde das obige Beispiel noch eleganter werden:

```
final long minPrime = ...
new Thread(() -> {
   // compute primes larger than minPrime
    . . .
}).start();
```

Pitfall - Zu viele Threads machen eine Anwendung langsamer.

Viele Leute, die sich mit Multi-Threading noch nicht auskennen, denken, dass die Verwendung von Threads die Anwendung automatisch beschleunigt. In der Tat ist es viel komplizierter. Wir können jedoch mit Sicherheit sagen, dass für jeden Computer die Anzahl der Threads, die gleichzeitig ausgeführt werden können, begrenzt ist:

- Ein Computer hat eine feste Anzahl von *Kernen* (oder *Hyperthreads*).
- Ein Java - Thread muss auf einen Kern oder Hyperthread *geplant* , um zu laufen.
- Wenn mehr ausführbare Java-Threads als (verfügbare) Kerne / Hyperthreads verfügbar sind, müssen einige von ihnen warten.

Dies sagt uns , dass nur die Schaffung von mehr und mehr Java - Threads der Anwendung *nicht* schneller und schneller gehen machen. Es gibt aber auch andere Überlegungen:

- Jeder Thread benötigt einen Off-Heap-Speicherbereich für seinen Threadstapel. Die typische (Standard-) Thread-Stack-Größe beträgt 512 KB oder 1 MB. Wenn Sie über eine große Anzahl von Threads verfügen, kann die Speicherauslastung erheblich sein.

- Jeder aktive Thread verweist auf eine Anzahl von Objekten im Heap. Dies erhöht die Anzahl der *erreichbaren* Objekte, was sich auf die Speicherbereinigung und die physische Speicherauslastung auswirkt.

- Der Aufwand beim Wechseln zwischen Threads ist nicht trivial. In der Regel ist ein Wechsel in den Betriebssystemkernel erforderlich, um eine Thread-Scheduling-Entscheidung zu treffen.

- Die Overheads der Thread-Synchronisierung und der Signalisierung zwischen Threads (z. B. wait (), notify () / notifyAll) *können von* Bedeutung sein.

Abhängig von den Details Ihrer Anwendung bedeuten diese Faktoren im Allgemeinen, dass es einen "Sweet Spot" für die Anzahl der Threads gibt. Darüber hinaus führt das Hinzufügen weiterer Threads zu einer minimalen Leistungsverbesserung und kann die Leistung verschlechtern.

Wenn Ihre Anwendung für jede neue Aufgabe erstellt, kann eine unerwartete Erhöhung der Arbeitslast (z. B. eine hohe Anforderungsrate) zu katastrophalem Verhalten führen.

Ein besserer Weg, um damit umzugehen, ist die Verwendung eines gebundenen Thread-Pools, dessen Größe Sie (statisch oder dynamisch) steuern können. Wenn zu viel Arbeit besteht, muss die Anwendung die Anforderungen in die Warteschlange stellen. Wenn Sie einen ExecutorService , kümmert sich dieser um die Verwaltung des Thread-Pools und die Warteschlange für Aufgaben.

Pitfall - Thread-Erstellung ist relativ teuer

Betrachten Sie diese beiden Mikro-Benchmarks:

Der erste Benchmark erstellt, startet und verbindet Threads. Das Runnable des Runnable funktioniert nicht.

```
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
```

Bei einem typischen modernen PC mit Linux mit 64-Bit-Java 8 u101 zeigt dieser Benchmark eine durchschnittliche Zeit für das Erstellen, Starten und Verbinden eines Threads zwischen 33,6 und 33,9 Mikrosekunden.

Der zweite Benchmark entspricht dem ersten, verwendet jedoch einen ExecutorService zum Übergeben von Aufgaben und einen Future zum Rendezvous mit dem Ende der Aufgabe.

```
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
```

Wie Sie sehen, liegen die Durchschnittswerte zwischen 5,3 und 5,6 Mikrosekunden.

Während die tatsächlichen Zeiten von einer Vielzahl von Faktoren abhängen, ist der Unterschied zwischen diesen beiden Ergebnissen erheblich. Die Verwendung eines Thread-Pools zum Recyceln von Threads ist eindeutig schneller als das Erstellen neuer Threads.

Fallstricke: Gemeinsame Variablen erfordern eine korrekte Synchronisation

Betrachten Sie dieses Beispiel:

```
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.
    }
}
```

Das Programm beabsichtigt, einen Thread zu starten, ihn für 1000 Millisekunden laufen zu lassen und dann durch Setzen des stop Flags zu stop .

Funktioniert es wie beabsichtigt?

Vielleicht ja vielleicht nein.

Eine Anwendung wird nicht notwendigerweise beendet, wenn die main wird. Wenn ein anderer Thread erstellt wurde und dieser Thread nicht als Daemon-Thread markiert wurde, wird die Anwendung nach dem Ende des Haupt-Threads weiter ausgeführt. In diesem Beispiel bedeutet dies, dass die Anwendung so lange ausgeführt wird, bis der untergeordnete Thread endet. tt.stop sollte tt.stop , wenn tt.stop auf true .

Das ist aber eigentlich nicht richtig. In der Tat wird das Kind Thread zu stoppen , nachdem er *beobachtete* stop mit dem Wert true . Wird das passieren? Vielleicht ja vielleicht nein.

Die Java-Sprachspezifikation *garantiert,* dass in einem Thread erstellte Lese- und Schreibvorgänge für den Thread in der Reihenfolge der Anweisungen im Quellcode sichtbar sind. Im Allgemeinen ist dies jedoch NICHT garantiert, wenn ein Thread schreibt und ein anderer Thread

(anschließend) liest. Um eine garantierte Sichtbarkeit zu erhalten, muss eine Kette von *Ereignissen vor dem* Schreiben und einem nachfolgenden Lesen vorhanden sein. Im obigen Beispiel gibt es keine solche Kette für die Aktualisierung des stop Flags. Daher kann nicht garantiert werden, dass der untergeordnete Thread stop in true ändert.

(Anmerkung für Autoren: Es sollte ein separates Thema zum Java-Speichermodell geben, um auf die detaillierten technischen Details einzugehen.)

Wie lösen wir das Problem?

In diesem Fall gibt es zwei einfache Möglichkeiten, um sicherzustellen, dass das stop Update sichtbar ist:

1. Deklarieren Sie stop als volatile ; dh

   ```
   private volatile boolean stop = false;
   ```

   Bei einer volatile Variablen gibt die JLS an, dass eine *Vor-Ereignis-* Beziehung zwischen einem Schreiben eines Threads und einem späteren Lesen eines zweiten Threads besteht.

2. Verwenden Sie einen Mutex zum Synchronisieren wie folgt:

```
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.
        }
    }
}
```

Neben der Sicherstellung eines gegenseitigen Ausschlusses gibt die JLS an, dass es eine *zufällige* Beziehung zwischen dem Freigeben eines Mutex in einem Thread und dem Erhalten des gleichen Mutex in einem zweiten Thread gibt.

Aber ist die Zuordnung nicht atomar?

Ja, so ist es!

Dies bedeutet jedoch nicht, dass die Auswirkungen der Aktualisierung für alle Threads gleichzeitig sichtbar sind. Nur eine richtige Kette von *Geschehnissen vor den* Ereignissen garantiert das.

Warum haben sie das gemacht?

Programmierer, die zum ersten Mal Multithreading-Programmierung in Java durchführen, finden, dass das Speichermodell eine Herausforderung darstellt. Programme verhalten sich nicht intuitiv, weil die natürliche Erwartung ist, dass Schreibvorgänge einheitlich sichtbar sind. Warum also die Java-Designer das Speichermodell so gestalten.

Es geht eigentlich um einen Kompromiss zwischen Leistung und Benutzerfreundlichkeit (für den Programmierer).

Eine moderne Computerarchitektur besteht aus mehreren Prozessoren (Kernen) mit individuellen Registersätzen. Der Hauptspeicher ist entweder für alle Prozessoren oder für Prozessorgruppen zugänglich. Eine andere Eigenschaft moderner Computerhardware besteht darin, dass der Zugriff auf Register normalerweise um Größenordnungen schneller als der Zugriff auf den Hauptspeicher ist. Wenn die Anzahl der Kerne zunimmt, kann man leicht erkennen, dass das Lesen und Schreiben in den Hauptspeicher zu einem Hauptleistungsengpass eines Systems werden kann.

Diese Nichtübereinstimmung wird durch Implementieren einer oder mehrerer Ebenen des Zwischenspeichers zwischen den Prozessorkernen und dem Hauptspeicher behoben. Jeder Kern greift über seinen Cache auf Speicherzellen zu. Normalerweise findet ein Hauptspeicherlesevorgang nur statt, wenn ein Cache-Fehler vorliegt, und ein Hauptspeicherzugriff erfolgt nur, wenn eine Cachezeile geleert werden muss. Bei einer Anwendung, bei der die Arbeitsspeicherbereiche jedes Kerns in seinen Cache passen, ist die Kerngeschwindigkeit nicht mehr durch die Hauptspeichergeschwindigkeit / Bandbreite begrenzt.

Das gibt uns jedoch ein neues Problem, wenn mehrere Kerne gemeinsam genutzte Variablen lesen und schreiben. Die neueste Version einer Variablen kann sich im Cache eines Cores befinden. Wenn der betreffende Kern die Cache-Zeile nicht in den Hauptspeicher schreibt UND ANDERE Kerne ihre zwischengespeicherte Kopie älterer Versionen ungültig machen, kann es vorkommen, dass einige von ihnen veraltete Versionen der Variablen sehen. Wenn die Caches jedoch jedes Mal in den Speicher geschrieben werden, wird ein Cache-Schreibvorgang ausgeführt ("nur für den Fall", dass ein anderer Kern gelesen wurde), der die Hauptspeicherbandbreite unnötig verbrauchen würde.

Die Standardlösung, die auf der Ebene der Hardwarebefehlssätze verwendet wird, besteht darin, Anweisungen für die Cache-Ungültigmachung und einen Cache-Durchschreibvorgang bereitzustellen und es dem Compiler zu überlassen, zu entscheiden, wann er verwendet wird.

Rückkehr zu Java Das Speichermodell ist so konzipiert, dass die Java-Compiler keine Anweisungen zum Ungültigmachen des Cache und Durchschreibbefehle ausgeben müssen, wenn sie nicht wirklich benötigt werden. Es wird davon ausgegangen, dass der Programmierer einen geeigneten Synchronisationsmechanismus (z. B. primitive Mutexe, volatile , übergeordnete Parallelitätsklassen usw.) verwendet, um anzuzeigen, dass der Arbeitsspeicher sichtbar ist. Wenn keine *Ereignis-vor-* Beziehung vorliegt, können die Java-Compiler *davon ausgehen,* dass keine Cache-Operationen (oder ähnliches) erforderlich sind.

Dies hat erhebliche Leistungsvorteile für Multithread-Anwendungen, aber der Nachteil ist, dass das Schreiben korrekter Multithread-Anwendungen keine einfache Angelegenheit ist. Der Programmierer *muss* verstehen, was er tut.

Warum kann ich das nicht reproduzieren?

Es gibt eine Reihe von Gründen, warum solche Probleme schwer reproduzierbar sind:

1. Wie oben erläutert, ist die Folge des Problems, Probleme mit der Sichtbarkeit des Speichers nicht richtig zu behandeln, in der *Regel,* dass Ihre kompilierte Anwendung die Speicher-Caches nicht korrekt verarbeitet. Wie bereits erwähnt, werden Speichercaches jedoch ohnehin geleert.

2. Wenn Sie die Hardwareplattform ändern, können sich die Eigenschaften der Speichercaches ändern. Dies kann zu einem anderen Verhalten führen, wenn Ihre Anwendung nicht ordnungsgemäß synchronisiert wird.

3. Möglicherweise beobachten Sie die Auswirkungen einer *zufälligen* Synchronisation. Wenn Sie beispielsweise Traceprints hinzufügen, geschieht dies normalerweise im Hintergrund in den E / A-Streams mit einer Synchronisierung, die Cache-Flushen verursacht. Durch das Hinzufügen von Traceprints verhält sich die Anwendung *oft* anders.

4. Wenn Sie eine Anwendung unter einem Debugger ausführen, wird sie vom JIT-Compiler anders kompiliert. Haltepunkte und Einzelschritte verstärken dies. Diese Effekte ändern häufig das Verhalten einer Anwendung.

Diese Dinge machen Fehler, die auf eine unzureichende Synchronisierung zurückzuführen sind, besonders schwer zu lösen.

Java-Fallstricke - Threads und Parallelität online lesen:
https://riptutorial.com/de/java/topic/5567/java-fallstricke---threads-und-parallelitat

Kapitel 76: Java-Fallstricke - Verwendung von Ausnahmen

Einführung

Bei einem Missbrauch von Java-Programmiersprachen kann es vorkommen, dass ein Programm fehlerhafte Ergebnisse generiert, obwohl es korrekt kompiliert wurde. Das Hauptziel dieses Themas ist es, häufige **Fallstricke im** Zusammenhang mit der **Ausnahmebehandlung aufzulisten** und den richtigen Weg vorzuschlagen, um solche Fallstricke zu vermeiden.

Examples

Pitfall - Ignorieren oder Quetschen von Ausnahmen

In diesem Beispiel werden Ausnahmen absichtlich ignoriert oder "gestaucht". Oder genauer gesagt, es geht darum, eine Ausnahme so zu erfassen und zu behandeln, dass sie ignoriert wird. Bevor wir jedoch beschreiben, wie dies zu tun ist, sollten wir zunächst darauf hinweisen, dass das Quetschen von Ausnahmen im Allgemeinen nicht die richtige Vorgehensweise ist.

Ausnahmen werden normalerweise (durch etwas) ausgelöst, um andere Teile des Programms darüber zu informieren, dass ein bedeutendes (dh "außergewöhnliches") Ereignis aufgetreten ist. Im Allgemeinen (wenn auch nicht immer) bedeutet eine Ausnahme, dass etwas schiefgegangen ist. Wenn Sie Ihr Programm so programmieren, dass es die Ausnahme unterdrückt, besteht eine gute Chance, dass das Problem in einer anderen Form erneut auftritt. Um es noch schlimmer zu machen, wenn Sie die Exception quetschen, werfen Sie die Informationen im Exception-Objekt und der zugehörigen Stack-Ablaufverfolgung weg. Das macht es wahrscheinlich schwieriger, die ursprüngliche Ursache des Problems herauszufinden.

In der Praxis kommt es häufig zu Ausnahmebereitschaft, wenn Sie die automatische Korrekturfunktion einer IDE verwenden, um einen Kompilierungsfehler zu beheben, der durch eine nicht behandelte Ausnahme verursacht wurde. Beispielsweise könnte Code wie folgt angezeigt werden:

```
try {
    inputStream = new FileInputStream("someFile");
} catch (IOException e) {
    /* add exception handling code here */
}
```

Natürlich hat der Programmierer den Vorschlag der IDE akzeptiert, den Kompilierungsfehler zu beseitigen, der Vorschlag war jedoch unangemessen. (Wenn das Öffnen der Datei fehlgeschlagen ist, sollte das Programm höchstwahrscheinlich etwas dagegen unternehmen. Bei der obigen "Korrektur" kann das Programm später versagen; z. B. mit einer NullPointerException da inputStream jetzt null .)

Dies ist jedoch ein Beispiel für das absichtliche Zusammendrücken einer Ausnahme. (Nehmen Sie zur Argumentation an, dass wir festgestellt haben, dass ein Interrupt beim Anzeigen des Selfies harmlos ist.) Der Kommentar sagt dem Leser, dass wir die Ausnahme absichtlich zusammengedrückt haben und warum wir das getan haben.

```
try {
    selfie.show();
} catch (InterruptedException e) {
    // It doesn't matter if showing the selfie is interrupted.
}
```

Eine andere konventionelle Möglichkeit, hervorzuheben, dass wir eine Ausnahme *absichtlich* ausschließen, ohne zu sagen, warum, ist dies mit dem Namen der Ausnahmevariable anzugeben:

```
try {
    selfie.show();
```

```
} catch (InterruptedException ignored) {  }
```

Einige IDEs (wie IntelliJ IDEA) zeigen keine Warnung bezüglich des leeren catch-Blocks an, wenn der Variablenname auf ignored .

Pitfall - Abfangen von Throwable, Exception, Error oder RuntimeException

Ein gemeinsames Denkmuster für unerfahrenen Java - Programmierer ist , dass Ausnahmen „ein Problem" oder „Last" sind und der beste Weg , damit umzugehen ist , sie alle [1] so schnell wie möglich zu fangen. Dies führt zu Code wie folgt:

```
....
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (Exception ex) {
    System.out.println("Could not open file " + fileName);
}
```

Der obige Code hat einen erheblichen Fehler. Der catch tatsächlich mehr Ausnahmen, als der Programmierer erwartet. Nehmen wir an, dass der Wert der fileName ist null , aufgrund eines Fehlers an anderer Stelle in der Anwendung. Dies bewirkt, dass der FileInputStream Konstruktor eine NullPointerException . Der Handler wird dies abfangen und dem Benutzer mitteilen:

```
    Could not open file null
```

das ist nicht hilfreich und verwirrend. Schlimmer noch, nehmen Sie an, dass es der Code "Verarbeitung der Eingabe" war, der die unerwartete Ausnahmebedingung ausgelöst hat (aktiviert oder nicht angehakt!). Der Benutzer erhält nun die irreführende Meldung für ein Problem, das beim Öffnen der Datei nicht aufgetreten ist und möglicherweise nicht mit der E / A zusammenhängt.

Die Ursache des Problems ist, dass der Programmierer einen Handler für Exception codiert hat. Das ist fast immer ein Fehler:

- Durch das Abfangen von Exception werden alle markierten Ausnahmen und die meisten ungeprüften Ausnahmen ebenfalls erfasst.
- RuntimeException Abfangen von RuntimeException fängt die meisten ungeprüften Ausnahmen ab.
- Catching Error fängt ungeprüfte Ausnahmen auf, die interne JVM-Fehler anzeigen. Diese Fehler können im Allgemeinen nicht behoben werden und sollten nicht abgefangen werden.
- Catching Throwable fängt alle möglichen Ausnahmen auf.

Das Problem beim Erfassen einer zu breiten Anzahl von Ausnahmen besteht darin, dass der Handler normalerweise nicht alle angemessen behandeln kann. Im Falle der Exception und so weiter, ist es schwierig für den Programmierer vorherzusagen , was gefangen werden *könnten;* dh was zu erwarten ist.

Im Allgemeinen ist die richtige Lösung mit den Ausnahmen zu behandeln , die ausgelöst *werden.* Beispielsweise können Sie sie fangen und in situ handhaben:

```
try {
    InputStream is = new FileInputStream(fileName);
    // process the input
} catch (FileNotFoundException ex) {
    System.out.println("Could not open file " + fileName);
}
```

oder Sie können sie als von der einschließenden Methode thrown erklären.

Es gibt sehr wenige Situationen, in denen das Ausnehmen von Exception angemessen ist. Die

einzige, die häufig auftritt, ist etwa so:

```
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);
    }
}
```

Hier wollen wir uns wirklich mit allen Ausnahmen befassen, also ist das Fangen von Exception (oder sogar Throwable) richtig.

---

1 - Wird auch als Pokemon-Ausnahmebehandlung bezeichnet .

Pitfall - Throwable Throwable, Exception, Error oder RuntimeException

Die Exception Throwable , Exception , Error und RuntimeException sind zwar schlecht, aber sie zu werfen ist noch schlimmer.

Das grundlegende Problem ist, dass die Ausnahmebedingungen der obersten Ebene die Unterscheidung zwischen verschiedenen Fehlerbedingungen schwierig machen, wenn Ihre Anwendung mit Ausnahmen umgehen muss. Zum Beispiel

```
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
}
```

Das Problem ist, dass wir, weil wir eine Exception Instanz geworfen haben, gezwungen werden, sie zu fangen. Wie in einem anderen Beispiel beschrieben, ist das Abfangen von Exception jedoch schlecht. In dieser Situation wird es schwierig , zwischen dem „erwarteten" Fall einen diskriminieren Exception , die ausgelöst wird , wenn somethingBad ist true , und der unerwarteten Fall, dass wir tatsächlich eine ungeprüfte Ausnahme wie fangen NullPointerException .

Wenn sich die Ausnahme der obersten Ebene ausbreiten darf, treten andere Probleme auf:

- Wir müssen uns nun an die verschiedenen Gründe erinnern, die wir auf die oberste Ebene geworfen haben, und diese diskriminieren / behandeln.
- Im Falle von Exception und Throwable wir diese Exceptions auch der throws Klausel von Methoden hinzufügen, wenn die Exception propagiert werden soll. Dies ist problematisch, wie unten beschrieben.

Kurz gesagt, werfen Sie diese Ausnahmen nicht auf. Wirft eine spezifischere Ausnahme, die das aufgetretene "außergewöhnliche Ereignis" genauer beschreibt. Definieren und verwenden Sie ggf. eine benutzerdefinierte Ausnahmeklasse.

Das Deklarieren von Throwable oder Exception in "Throws" einer Methode ist problematisch.

Es ist verlockend , eine lange Liste von Ausnahmen geworfen in einem Verfahren zu ersetzen , der throws Klausel mit Exception oder sogar `Throwable. Das ist eine schlechte Idee:

1. Der Aufrufer wird gezwungen, die Exception zu behandeln (oder zu verbreiten).
2. Wir können uns nicht mehr darauf verlassen, dass der Compiler uns bestimmte geprüfte Ausnahmen mitteilt, die behandelt werden müssen.
3. Die richtige Handhabung von Exception ist schwierig. Es ist schwer zu wissen, welche Ausnahmen tatsächlich gefasst werden könnten, und wenn Sie nicht wissen, was gefangen werden könnte, ist es schwierig zu wissen, welche Erholungsstrategie geeignet ist.
4. Der Umgang mit Throwable ist noch schwieriger, da Sie jetzt auch mit potenziellen Ausfällen fertig werden müssen, bei denen niemals eine Throwable ist.

Dieser Hinweis bedeutet, dass bestimmte andere Muster vermieden werden sollten. Zum Beispiel:

```
try {
    doSomething();
} catch (Exception ex) {
    report(ex);
    throw ex;
}
```

Die oben genannten Versuche, alle Ausnahmen zu protokollieren, ohne sie endgültig zu behandeln. Leider wurde vor Java 7 der throw ex; Die Anweisung veranlasste den Compiler zu der Annahme, dass eine Exception ausgelöst werden könnte. Dies kann dazu führen, dass Sie die einschließende Methode als throws Exception deklarieren. Der Compiler weiß ab Java 7, dass der Satz von Ausnahmen, die dort (erneut ausgelöst) werden könnten, kleiner ist.

Pitfall - InterruptedException abfangen

Wie bereits in anderen Fallstricken dargelegt, alle Ausnahmen mit fangen

```
try {
    // Some code
} catch (Exception) {
    // Some error handling
}
```

Kommt mit vielen verschiedenen Problemen. Ein besonderes Problem ist jedoch, dass es zu Deadlocks kommen kann, da das Interrupt-System beim Schreiben von Multithread-Anwendungen unterbrochen wird.

Wenn Sie einen Thread starten, müssen Sie in der Regel aus verschiedenen Gründen abrupt stoppen können.

```
Thread t = new Thread(new Runnable() {
    public void run() {
         while (true) {
             //Do something indefinetely
         }
    }
}

t.start();

//Do something else

// The thread should be canceld if it is still active.
// A Better way to solve this is with a shared variable that is tested
// regularily by the thread for a clean exit, but for this example we try to
// forcibly interrupt this thread.
if (t.isAlive()) {
```

```
   t.interrupt();
   t.join();
}

//Continue with program
```

t.interrupt() eine InterruptedException in diesem Thread aus, um den Thread herunterzufahren. Was aber, wenn der Thread einige Ressourcen bereinigen muss, bevor er vollständig gestoppt wird? Dafür kann es die InterruptedException abfangen und einige Bereinigungen durchführen.

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

Wenn Sie jedoch einen Catch-All-Ausdruck in Ihrem Code haben, wird die InterruptedException ebenfalls davon abgefangen und die Unterbrechung wird nicht fortgesetzt. Was in diesem Fall zu einem Deadlock führen könnte, da der übergeordnete Thread unbestimmt darauf wartet, dass dieser mit 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();
        }
    }
}
```

Es ist also besser, Exceptions einzeln abzufangen, aber wenn Sie auf der Verwendung eines Catch-All bestehen, sollten Sie die InterruptedException mindestens vorher einzeln abfangen.

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

Fallstricke - Verwenden von Ausnahmen für die normale Flusskontrolle

Es gibt ein Mantra, das einige Java-Experten zu rezitieren pflegen:

> "Ausnahmen sollten nur in Ausnahmefällen verwendet werden."

(Zum Beispiel: http://programmers.stackexchange.com/questions/184654)

Das Wesentliche dabei ist, dass es (in Java) eine schlechte Idee ist, Ausnahmen und Ausnahmebehandlung zu verwenden, um die normale Flusssteuerung zu implementieren. Vergleichen Sie zum Beispiel diese beiden Arten des Umgangs mit einem Parameter, der null sein könnte.

```
public String truncateWordOrNull(String word, int maxLength) {
    if (word == null) {
        return "";
    } else {
        return word.substring(0, Math.min(word.length(), maxLength));
    }
}

public String truncateWordOrNull(String word, int maxLength) {
    try {
        return word.substring(0, Math.min(word.length(), maxLength));
    } catch (NullPointerException ex) {
        return "";
    }
}
```

In diesem Beispiel behandeln wir (durch das Design) den Fall, dass das word null als wäre es ein leeres Wort. Die beiden Versionen behandeln null entweder mit konventionellem *if ... else* und oder *versuchen Sie ... catch* . Wie sollen wir entscheiden, welche Version besser ist?

Das erste Kriterium ist die Lesbarkeit. Obwohl Lesbarkeit schwer objektiv zu quantifizieren ist, würden sich die meisten Programmierer darin einig sein, dass die grundlegende Bedeutung der ersten Version leichter zu erkennen ist. Um das zweite Formular wirklich verstehen zu können, müssen Sie NullPointerException , dass eine NullPointerException nicht von den Math.min oder String.substring Methoden String.substring werden kann.

Das zweite Kriterium ist die Effizienz. In Java-Versionen vor Java 8 ist die zweite Version erheblich (um Größenordnungen) langsamer als die erste Version. Insbesondere erfordert der Aufbau eines Ausnahmeobjekts das Erfassen und Aufzeichnen der Stapelrahmen, nur für den Fall, dass die Stapelverfolgung erforderlich ist.

Auf der anderen Seite gibt es viele Situationen, in denen die Verwendung von Ausnahmen lesbarer, effizienter und (manchmal) korrekter ist als die Verwendung von Bedingungscode für "außergewöhnliche" Ereignisse. In seltenen Fällen ist es notwendig, sie für "nicht

außergewöhnliche" Ereignisse zu verwenden. dh Ereignisse, die relativ häufig auftreten. Für Letzteres lohnt es sich, nach Möglichkeiten zu suchen, um den Aufwand für das Erstellen von Ausnahmeobjekten zu reduzieren.

Pitfall - Übermäßige oder unangemessene Stacktraces

Das Ärgerlichste, was Programmierer tun können, ist, Aufrufe von printStackTrace() in ihrem gesamten Code zu printStackTrace() .

Das Problem ist, dass printStackTrace() den Stacktrace in die Standardausgabe schreibt.

- Für eine Anwendung, die für Endbenutzer gedacht ist, die kein Java-Programmierer sind, ist ein Stacktrace bestenfalls nicht informativ und im schlimmsten Fall alarmierend.
- Bei einer serverseitigen Anwendung besteht die Möglichkeit, dass niemand die Standardausgabe betrachtet.

Besser ist es, printStackTrace direkt printStackTrace Wenn Sie es aufrufen, tun Sie dies so, dass die Stack-Ablaufverfolgung in eine Protokolldatei oder Fehlerdatei und nicht in die Konsole des Endbenutzers geschrieben wird.

Eine Möglichkeit, dies zu tun, besteht darin, ein Protokollierungsframework zu verwenden und das Exception-Objekt als Parameter des Protokollereignisses zu übergeben. Die Protokollierung der Ausnahme kann jedoch schädlich sein, wenn sie nicht ordnungsgemäß ausgeführt wird. Folgendes berücksichtigen:

```
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;
    }
}
```

Wenn die Ausnahme in method2 ausgelöst wird, werden in der method2 wahrscheinlich zwei Kopien desselben Stacktraces angezeigt, die demselben Fehler entsprechen.

Kurz gesagt, entweder die Ausnahme protokollieren oder erneut ausführen (möglicherweise mit einer anderen Ausnahme verpackt). Beides nicht tun

Pitfall - Direkt Unterklasse "werfen"

Throwable hat zwei direkte Unterklassen, Exception und Error . Es ist zwar möglich, eine neue Klasse zu erstellen, die Throwable direkt erweitert. Dies ist jedoch nicht zu Throwable , da viele Anwendungen nur Exception und Error Throwable .

Throwable auf den Punkt zu bringen, gibt es keinen praktischen Nutzen für die direkte Unterklasse Throwable , da die resultierende Klasse Throwable nur eine geprüfte Ausnahme ist. Subclassing Exception stattdessen zu demselben Verhalten, vermittelt jedoch Ihre Absicht klarer.

Java-Fallstricke - Verwendung von Ausnahmen online lesen:

https://riptutorial.com/de/java/topic/5381/java-fallstricke---verwendung-von-ausnahmen

Kapitel 77: Java-Gleitkommaoperationen

Einführung

Fließkommazahlen sind Zahlen mit gebrochenen Teilen (normalerweise mit Dezimalpunkt ausgedrückt). In Java gibt es zwei Grundtypen für Gleitkommazahlen, die float (4 Bytes) und double (8 Bytes) sind. Diese Dokumentationsseite enthält ausführliche Beispieloperationen, die für Gleitkommazahlen in Java ausgeführt werden können.

Examples

Vergleich von Gleitkommawerten

Beim Vergleich von Gleitkommawerten (float oder double) mit relationalen Operatoren sollten Sie vorsichtig sein: == != , < Und so weiter. Diese Operatoren liefern Ergebnisse entsprechend den binären Darstellungen der Gleitkommawerte. Zum Beispiel:

```
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"
    }
}
```

Die Berechnung oneThird hat zu einem kleinen Rundungsfehler geführt. Wenn Sie oneThird mit 3 multiplizieren, erhalten Sie ein Ergebnis, das sich leicht von 1.0 .

Dieses Problem ungenauer Repräsentationen wird deutlicher, wenn wir versuchen, double und float in Berechnungen zu mischen. Zum Beispiel:

```
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
    }
}
```

Die in Java für die float und double Typen verwendeten Gleitkommadarstellungen haben eine begrenzte Anzahl von Ziffern. Für den float Typ beträgt die Genauigkeit 23 Binärstellen oder etwa 8 Dezimalstellen. Für den double sind es 52 Bits oder etwa 15 Dezimalstellen. Darüber hinaus führen einige Rechenoperationen zu Rundungsfehlern. Wenn ein Programm Fließkommawerte vergleicht, ist es daher üblich, ein **akzeptables Delta** für den Vergleich zu definieren. Wenn die Differenz zwischen den beiden Zahlen kleiner als das Delta ist, werden sie als gleich betrachtet. Zum Beispiel

```
if (Math.abs(v1 - v2) < delta)
```

Delta-Vergleichsbeispiel:

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

Ergebnis:

```
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
```

Auch für den Vergleich von double und float Primitivtypen können statische compare des entsprechenden Boxtyps verwendet werden. Zum Beispiel:

```
double a = 1.0;
double b = 1.0001;

System.out.println(Double.compare(a, b));//-1
System.out.println(Double.compare(b, a));//1
```

Schließlich kann es schwierig sein zu bestimmen, welche Deltas für einen Vergleich am besten geeignet sind. Ein häufig verwendeter Ansatz ist die Auswahl von Delta-Werten, die unserer Intuition zufolge ungefähr richtig sind. Wenn Sie jedoch die Skalierung und (wahre) Genauigkeit der Eingabewerte sowie die durchgeführten Berechnungen kennen, kann es möglich sein, mathematisch fundierte Grenzen für die Genauigkeit der Ergebnisse und damit für die Deltas zu finden. (Es gibt einen formalen Zweig der Mathematik, bekannt als Numerische Analyse, der Computerwissenschaftlern beigebracht wurde, die diese Art von Analyse abdeckten.)

OverFlow und UnderFlow

Float- Datentyp

Der Float-Datentyp ist ein 32-Bit-IEEE 754-Gleitkomma mit einfacher Genauigkeit.

Float

Maximal möglicher Wert ist 3.4028235e+38 Wenn dieser Wert überschritten wird, wird Infinity erzeugt

```
float f = 3.4e38f;
float result = f*2;
System.out.println(result); //Infinity
```

Float **UnderFlow**

Der Mindestwert ist 1.4e-45f. Wenn dieser Wert unter 0 liegt, wird 0.0 erzeugt

```
    float f = 1e-45f;
    float result = f/1000;
    System.out.println(result);
```

doppelter Datentyp

Der doppelte Datentyp ist ein 64-bit IEEE 754-Gleitkomma mit doppelter Genauigkeit.

Double **Überlauf**

Maximal möglicher Wert ist 1.7976931348623157e+308 , Wenn er diesen Wert überschreitet, wird Infinity erzeugt

```
double d = 1e308;
double result=d*2;
System.out.println(result); //Infinity
```

Double **UnderFlow**

Der Mindestwert ist 4.9e-324. Wenn dieser Wert unterschritten wird, wird 0.0 erzeugt

```
    double d = 4.8e-323;
```

```
    double result = d/1000;
    System.out.println(result); //0.0
```

Formatieren der Gleitkommawerte

Gleitkommazahlen Zahlen können mit Hilfe von String.format mit 'f' String.format als Dezimalzahl formatiert werden

```
    //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"
```

Fließkommazahlen können mit DecimalFormat als Dezimalzahl formatiert werden

```
   // 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
```

Strikte Einhaltung der IEEE-Spezifikation

Standardmäßig halten Gleitkommaoperationen für float und double *nicht* strikt die Regeln der IEEE 754-Spezifikation ein. Ein Ausdruck darf implementierungsspezifische Erweiterungen für den Bereich dieser Werte verwenden. im Wesentlichen so dass sie *genauer* als erforderlich sein.

strictfp deaktiviert dieses Verhalten. Es wird auf eine Klasse, Schnittstelle oder eine Methode, und gilt für alles , was darin enthalten ist , wie beispielsweise Klassen, Schnittstellen, Methoden, Konstruktoren Variableninitialisierungen usw. Mit strictfp , die Zwischenwerte eines Gleitkommaausdruck sein *muss* innerhalb der Floatwertsatz oder der Doppelwertsatz. Dies führt dazu, dass die Ergebnisse solcher Ausdrücke genau die sind, die die IEEE 754-Spezifikation vorhersagt.

Alle konstanten Ausdrücke sind implizit streng, auch wenn sie sich nicht innerhalb eines strictfp Bereichs befinden.

strictfp hat daher den Effekt, dass bestimmte strictfp manchmal *weniger* genau sind. Außerdem können Gleitkommaoperationen *langsamer werden* (da die CPU jetzt mehr Arbeit leistet, um sicherzustellen, dass die native Genauigkeit zusätzlich das Ergebnis nicht beeinflusst). Es

führt jedoch auch dazu, dass die Ergebnisse auf allen Plattformen exakt gleich sind. Es ist daher nützlich für Dinge wie wissenschaftliche Programme, bei denen die Reproduzierbarkeit wichtiger ist als die Geschwindigkeit.

```
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;
        }
    }
}
```

Java-Gleitkommaoperationen online lesen: https://riptutorial.com/de/java/topic/6167/java-gleitkommaoperationen

Kapitel 78: Java-Leistungsoptimierung

Examples

Allgemeiner Ansatz

Das Internet enthält zahlreiche Tipps zur Leistungsverbesserung von Java-Programmen. Vielleicht ist der wichtigste Tipp das Bewusstsein. Das bedeutet:

- Mögliche Leistungsprobleme und Engpässe erkennen.
- Verwenden Sie Analyse- und Testwerkzeuge.
- Kennen Sie bewährte Praktiken und schlechte Praktiken.

Der erste Punkt sollte während der Entwurfsphase gemacht werden, wenn über ein neues System oder Modul gesprochen wird. Wenn Sie über Legacy-Code sprechen, kommen Analyse- und Test-Tools zum Tragen. Das grundlegendste Werkzeug für die Analyse Ihrer JVM-Leistung ist JVisualVM, das im JDK enthalten ist.

Der dritte Punkt ist vor allem über Erfahrung und umfangreiche Forschung und natürlich roh Tipps , die auf dieser Seite und andere, wie Show wird dies .

Anzahl der Saiten reduzieren

In Java ist es zu "einfach", viele nicht benötigte String-Instanzen zu erstellen. Dies und andere Gründe können dazu führen, dass Ihr Programm viele Strings hat, die der GC gerade bereinigt.

Einige Möglichkeiten, wie Sie String-Instanzen erstellen können:

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

Oder noch schlimmer, in einer Schleife oder Rekursion:

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

Das Problem ist, dass jedes + einen neuen String erstellt (normalerweise, da neue Compiler einige Fälle optimieren). Eine mögliche Optimierung kann mit StringBuilder oder StringBuffer :

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

Wenn Sie häufig lange Zeichenfolgen erstellen (z. B. SQLs), verwenden Sie eine Zeichenfolgenerstellungs-API.

Andere Dinge zu beachten:

- Reduzieren Sie die Verwendung von replace , substring usw.
- Vermeiden Sie String.toArray() , insbesondere in häufig aufgerufenem Code.
- Protokollausdrucke, die zum Filtern bestimmt sind (z. B. aufgrund des Protokollierungsniveaus), sollten nicht erstellt werden (Protokollierungsgrad sollte vorab geprüft werden).
- Verwenden Sie bei Bedarf Bibliotheken wie diese .
- StringBuilder ist besser, wenn die Variable nicht gemeinsam genutzt wird (über Threads hinweg).

Ein evidenzbasierter Ansatz für die Java-Leistungsoptimierung

Donald Knuth wird oft so zitiert:

> „Programmierer verschwenden enorme Mengen an Zeit darüber nachzudenken, oder sich Gedanken über die Geschwindigkeit von nicht kritischen Teile ihrer Programme, und diese Versuche der Effizienz haben tatsächlich eine starke negative Auswirkungen , wenn die Fehlersuche und Wartung berücksichtigt werden. *Wir sollten über kleine Effizienz vergessen, sagen In 97% der Fälle* : vorzeitige Optimierung ist die Wurzel allen Übels. Dennoch sollten wir unsere Chancen in den kritischen 3% nicht aufgeben. "

Quelle

In Anbetracht dieses weisen Ratschlags wird hier das empfohlene Verfahren zur Optimierung von Programmen empfohlen:

1. Entwerfen und codieren Sie zunächst Ihr Programm oder Ihre Bibliothek mit einem Fokus auf Einfachheit und Korrektheit. Um zu beginnen, geben Sie sich nicht viel Mühe für die Leistung.
2. Bringen Sie es in einen funktionierenden Zustand und entwickeln Sie (idealerweise) Komponententests für die wichtigsten Teile der Codebase.
3. Entwickeln Sie einen Performance-Benchmark auf Anwendungsebene. Der Benchmark sollte die leistungskritischen Aspekte Ihrer Anwendung abdecken und eine Reihe von Aufgaben ausführen, die typisch für die Verwendung der Anwendung in der Produktion sind.
4. Messen Sie die Leistung.
5. Vergleichen Sie die gemessene Leistung mit Ihren Kriterien, wie schnell die Anwendung sein muss. (Vermeiden Sie unrealistische, unerreichbare oder nicht quantifizierbare Kriterien wie "so schnell wie möglich".)
6. Wenn Sie die Kriterien erfüllt haben, STOPPEN. Deine Arbeit ist erledigt. (Jede weitere Anstrengung ist wahrscheinlich Zeitverschwendung.)
7. Profilieren Sie die Anwendung, während Ihr Leistungsbenchmark ausgeführt wird.
8. Überprüfen Sie die Ergebnisse der Profilerstellung und wählen Sie die größten (nicht optimierten) "Leistungs-Hotspots" aus. Dies sind Abschnitte des Codes, in denen die Anwendung die meiste Zeit zu verbringen scheint.
9. Analysieren Sie den Hotspot-Codeabschnitt, um herauszufinden, warum er einen Engpass darstellt, und überlegen Sie, wie er schneller werden kann.
10. Implementieren Sie dies als vorgeschlagene Codeänderung, testen und debuggen.
11. Führen Sie den Benchmark erneut aus, um zu sehen, ob die Codeänderung die Leistung verbessert hat:
    - Wenn ja, kehren Sie zu Schritt 4 zurück.
    - Wenn nein, brechen Sie die Änderung ab und kehren Sie zu Schritt 9 zurück. Wenn Sie keine Fortschritte erzielen, wählen Sie einen anderen Hotspot aus.

Schließlich werden Sie zu einem Punkt gelangen, an dem die Anwendung entweder schnell genug ist oder Sie alle wichtigen Hotspots berücksichtigt haben. An diesem Punkt müssen Sie diesen Ansatz stoppen. Wenn ein Codeabschnitt etwa 1% der Gesamtzeit beansprucht, führt eine Verbesserung um 50% dazu, dass die Anwendung insgesamt nur um 0,5% schneller wird.

Natürlich gibt es einen Punkt, bei dem die Optimierung von Hotspots eine Verschwendung von Aufwand bedeutet. Wenn Sie an diesen Punkt gelangen, müssen Sie radikaler vorgehen. Zum Beispiel:

- Sehen Sie sich die algorithmische Komplexität Ihrer Kernalgorithmen an.
- Wenn die Anwendung viel Zeit für die Garbage Collection aufbringt, suchen Sie nach Wegen, um die Objekterstellungsrate zu reduzieren.
- Wenn wichtige Teile der Anwendung CPU-intensiv und Single-Threaded sind, sollten Sie nach Möglichkeiten für Parallelität suchen.
- Wenn die Anwendung bereits mehrere Threads umfasst, suchen Sie nach Parallelitätsengpässen.

Verlassen Sie sich jedoch, wo immer möglich, auf Werkzeuge und Messungen und nicht auf Instinkt, um Ihre Optimierungsanstrengungen zu steuern.

Java-Leistungsoptimierung online lesen: https://riptutorial.com/de/java/topic/4160/java-leistungsoptimierung

Kapitel 79: Java-Plugin-Systemimplementierungen

Bemerkungen

Wenn Sie eine IDE und / oder ein Build-System verwenden, ist das Einrichten dieser Art von Projekt viel einfacher. Sie erstellen ein Hauptanwendungsmodul, dann ein API-Modul, dann ein Plugin-Modul und machen es vom API-Modul oder von beiden abhängig. Als Nächstes konfigurieren Sie, wo die Projektartefakte platziert werden sollen. In unserem Fall können die kompilierten Plug-In-Jars direkt in das Plug-In-Verzeichnis gesendet werden, sodass keine manuellen Bewegungen erforderlich sind.

Examples

URLClassLoader verwenden

Es gibt verschiedene Möglichkeiten, ein Plug-In-System für eine Java-Anwendung zu implementieren. Eine der einfachsten ist die Verwendung von *URLClassLoader* . Das folgende Beispiel enthält etwas JavaFX-Code.

Angenommen, wir haben ein Modul einer Hauptanwendung. Dieses Modul soll Plugins in Form von Jars aus dem Plugins-Ordner laden. Anfangscode:

```
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);
    }
}
```

Dann erstellen wir eine Schnittstelle, die ein Plugin darstellt.

```
package main;

public interface Plugin
{
    default void initialize()
    {
        System.out.println("Initialized "+this.getClass().getName());
    }
    default String name(){return getClass().getSimpleName();}
}
```

Wir möchten Klassen laden, die diese Schnittstelle implementieren. Daher müssen wir zunächst Dateien filtern, die die Erweiterung '.jar' haben:

```
File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
```

Wenn Dateien vorhanden sind, müssen Sammlungen von URLs und Klassennamen erstellt werden:

```
    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);
        }

    }
```

Fügen *wir* der MainApplication ein statisches HashSet *hinzu*, das geladene Plugins enthält:

```
static HashSet<Plugin> plugins=new HashSet<>();
```

Als Nächstes instantiieren wir einen *URLClassLoader* und iterieren über Klassennamen und instanziieren Klassen, die die *Plugin*- Schnittstelle implementieren:

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

Dann können wir zum Beispiel Plugins-Methoden aufrufen, um sie zu initialisieren:

```
if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded plugins:"));
    plugins.forEach(plugin -> {
        plugin.initialize();
        loadedPlugins.getChildren().add(new Label(plugin.name()));
});
```

Der endgültige Code von *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);
    }
}
```

Lassen Sie uns zwei Plugins erstellen. Natürlich sollte sich die Quelle des Plugins in einem separaten Modul befinden.

```
package plugins;

import main.Plugin;

public class FirstPlugin implements Plugin
{
    //this plugin has default behaviour
}
```

Zweites 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"));
    }
}
```

Diese Plugins müssen in Standard-Jars gepackt werden - dieser Vorgang hängt von Ihrer IDE oder anderen Tools ab.

Wenn Jars direkt in 'Plugins' *eingefügt* werden, *erkennt MainApplication* diese und instanziiert entsprechende Klassen.

Java-Plugin-Systemimplementierungen online lesen:
https://riptutorial.com/de/java/topic/7160/java-plugin-systemimplementierungen

Kapitel 80: Java-Sockets

Einführung

Sockets sind eine untergeordnete Netzwerkschnittstelle, mit deren Hilfe eine Verbindung zwischen zwei Programmen hergestellt werden kann, hauptsächlich Clients, die möglicherweise auf demselben Computer ausgeführt werden.

Socket-Programmierung ist eines der am häufigsten verwendeten Netzwerkkonzepte.

Bemerkungen

Es gibt zwei Arten von Internet Protocol Traffic –
1. TCP – Transmission Control Protocol (Protokoll für die Übertragungssteuerung) 2. UDP – User Datagram Protocol (Protokoll für Benutzerdaten)

TCP ist ein verbindungsorientiertes Protokoll.
UDP ist ein verbindungsloses Protokoll.

TCP eignet sich für Anwendungen, die eine hohe Zuverlässigkeit erfordern, und die Übertragungszeit ist relativ unkritisch.

UDP eignet sich für Anwendungen, die eine schnelle und effiziente Übertragung erfordern, wie z. B. Spiele. Die Stateless-Eigenschaft von UDP ist auch für Server nützlich, die kleine Anfragen von einer großen Anzahl von Clients beantworten.

In einfacheren Worten –
Verwenden Sie TCP, wenn Sie es sich nicht leisten können, Daten zu verlieren, und wenn die Zeit zum Senden und Empfangen von Daten keine Rolle spielt. Verwenden Sie UDP, wenn Sie es sich nicht leisten können, Zeit zu verlieren und wenn der Datenverlust keine Rolle spielt.

Es besteht die absolute Garantie, dass die übertragenen Daten erhalten bleiben und in der gleichen Reihenfolge ankommen, in der sie bei TCP gesendet wurden.
Es gibt jedoch keine Garantie dafür, dass die gesendeten Nachrichten oder Pakete in UDP überhaupt erreicht werden.

Examples

Ein einfacher TCP-Echo-Back-Server

Unser TCP-Echo-Back-Server wird ein separater Thread sein. Es ist ganz einfach als Anfang. Es wird nur das wiedergeben, was Sie senden, jedoch in Großschreibung.

```
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);
        }
    }

}
```

Jetzt Verbindungen annehmen. Lassen Sie uns die Run-Methode aktualisieren.

```
@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);
    }
}
```

Wenn Sie nun Telnet öffnen und versuchen, eine Verbindung herzustellen, wird eine Begrüßungsnachricht angezeigt.

Sie müssen eine Verbindung mit dem angegebenen Port und der IP-Adresse herstellen.

Sie sollten ein ähnliches Ergebnis sehen:

```
Welcome!

Connection to host lost.
```

Nun, die Verbindung ging verloren, weil wir sie abgebrochen haben. Manchmal müssen wir unseren eigenen TCP-Client programmieren. In diesem Fall benötigen wir einen Client, um die Eingabe vom Benutzer anzufordern und über das Netzwerk zu senden.

Wenn der Server zuerst Daten sendet, muss der Client die Daten zuerst lesen.

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

Aktualisieren Sie jetzt den Server

```
ostream.writeUTF("Welcome!");

String inString = istream.readUTF();    // Read what the user sent
String outString = inString.toUpperCase();  // Change it to caps
```

```
ostream.writeUTF(outString);

// Close the connection
istream.close();
```

Führen Sie nun den Server und den Client aus. Sie sollten eine ähnliche Ausgabe haben

```
Welcome!
>
```

Java-Sockets online lesen: https://riptutorial.com/de/java/topic/9923/java-sockets

Kapitel 81: Java-Speichermodell

Bemerkungen

Das Java-Speichermodell ist der Abschnitt des JLS, der die Bedingungen angibt, unter denen ein Thread garantiert die Auswirkungen von Speicherschreibvorgängen eines anderen Threads erkennt. Der relevante Abschnitt in den letzten Ausgaben ist "JLS 17.4 Memory Model" (in Java 8 , Java 7 , Java 6).

In Java 5 wurde das Java-Speichermodell grundlegend überarbeitet, was (unter anderem) die Art und Weise geändert hat, wie volatile funktioniert. Seitdem ist das Speichermodell im Wesentlichen unverändert.

Examples

Motivation für das Speichermodell

Betrachten Sie das folgende Beispiel:

```
public class Example {
    public int a, b, c, d;

    public void doIt() {
       a = b + 1;
       c = d + 1;
    }
}
```

Wenn diese Klasse eine Single-Thread-Anwendung ist, ist das beobachtbare Verhalten genau so, wie Sie es erwarten würden. Zum Beispiel:

```
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);
    }
}
```

wird ausgeben:

```
0, 0
1, 1
```

Soweit der "main" doIt() *kann* , werden die Anweisungen in der main() Methode und der doIt() Methode in der Reihenfolge ausgeführt, in der sie im Quellcode geschrieben werden. Dies ist eine klare Anforderung der Java Language Specification (JLS).

Betrachten Sie nun dieselbe Klasse, die in einer Multithread-Anwendung verwendet wird.

```
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);
        }
    }
}
```

Was wird das drucken?

Tatsächlich ist es laut JLS nicht möglich vorherzusagen, dass dies gedruckt wird:

- Sie werden wahrscheinlich ein paar Zeilen von 0, 0 , um damit zu beginnen.
- Dann sehen Sie wahrscheinlich Zeilen wie N, N oder N, N + 1 .
- Möglicherweise sehen Sie Zeilen wie N + 1, N
- Theoretisch sehen Sie vielleicht sogar, dass die 0, 0 Zeilen für immer bestehen [1] .

1 - In der Praxis kann das Vorhandensein der println Anweisungen zu einer println Synchronisation und zum Leeren des Speichercaches führen. Das wird wahrscheinlich einige der Effekte verdecken, die das oben genannte Verhalten verursachen würden.

Wie können wir das erklären?

Neuordnung der Aufträge

Eine mögliche Erklärung für unerwartete Ergebnisse ist, dass der JIT-Compiler die Reihenfolge der Zuweisungen in der Methode doIt() geändert hat. Das JLS erfordert, dass Anweisungen *aus der Sicht des aktuellen* Threads der Reihe *nach* ausgeführt werden. In diesem Fall kann nichts im Code der Methode doIt() die Auswirkung einer (hypothetischen) Neuordnung dieser beiden Anweisungen feststellen. Dies bedeutet, dass der JIT-Compiler dazu berechtigt ist.

Warum sollte es das tun?

Bei einer typischen modernen Hardware werden Maschinenbefehle unter Verwendung einer Befehlspipeline ausgeführt, die es ermöglicht, dass eine Folge von Anweisungen in verschiedenen Stufen abläuft. Einige Phasen der Befehlsausführung dauern länger als andere, und Speicheroperationen dauern in der Regel länger. Ein intelligenter Compiler kann den Befehlsdurchsatz der Pipeline optimieren, indem er die Anweisungen so anordnet, dass die Überlappung maximiert wird. Dies kann dazu führen, dass Teile von Anweisungen außerhalb der Reihenfolge ausgeführt werden. Das JLS erlaubt dies vorausgesetzt, dass das Ergebnis der Berechnung *aus Sicht des aktuellen Threads* nicht beeinflusst *wird* .

Auswirkungen von Speicher-Caches

Eine zweite mögliche Erklärung ist der Effekt des Memory-Caching. In einer klassischen Computerarchitektur verfügt jeder Prozessor über einen kleinen Satz von Registern und eine größere Menge an Speicher. Der Zugriff auf Register ist viel schneller als der Zugriff auf den Hauptspeicher. In modernen Architekturen gibt es Speichercaches, die langsamer als Register sind, aber schneller als Hauptspeicher.

Ein Compiler wird dies ausnutzen, indem er versucht, Kopien von Variablen in Registern oder in den Speichercaches zu halten. Wenn eine Variable *muss* nicht in dem Hauptspeicher geleert werden, oder *muss* nicht aus dem Speicher gelesen wird, gibt es erhebliche Leistungsvorteile in nicht tun. Wenn für das JLS keine Speicheroperationen für einen anderen Thread sichtbar sein müssen, fügt der Java-JIT-Compiler wahrscheinlich nicht die Anweisungen "Lesesperre" und "Schreibsperre" hinzu, die das Lesen und Schreiben von Hauptspeicher erzwingen. Wieder einmal sind die Leistungsvorteile dabei erheblich.

Richtige Synchronisation

Bisher haben wir gesehen, dass JLS es dem JIT-Compiler ermöglicht, Code zu generieren, der Single-Thread-Code schneller macht, indem Speicheroperationen neu angeordnet oder vermieden werden. Was passiert aber, wenn andere Threads den Status der (gemeinsam genutzten) Variablen im

Hauptspeicher beobachten können?

Die Antwort ist, dass die anderen Threads Variablenzustände beobachten können, die als unmöglich erscheinen ... basierend auf der Codereihenfolge der Java-Anweisungen. Die Lösung hierfür ist die Verwendung einer geeigneten Synchronisation. Die drei Hauptansätze sind:

- Verwenden primitiver Mutexe und der synchronized Konstrukte.
- Verwendung volatile Variablen
- Unterstützung für Parallelität auf höherer Ebene verwenden; zB Klassen in den java.util.concurrent Paketen.

Trotzdem ist es wichtig zu wissen, wo Synchronisierung erforderlich ist und auf welche Effekte Sie sich verlassen können. Hier kommt das Java Memory Model ins Spiel.

Das Speichermodell

Das Java-Speichermodell ist der Abschnitt des JLS, der die Bedingungen angibt, unter denen ein Thread garantiert die Auswirkungen von Speicherschreibvorgängen eines anderen Threads erkennt. Das Speichermodell ist mit einer gewissen *formalen Strenge* spezifiziert und erfordert (als Ergebnis) detailliertes und sorgfältiges Lesen, um es zu verstehen. Das Grundprinzip besteht jedoch darin, dass bestimmte Konstrukte eine "Vor-Vorher-Beziehung" zwischen dem Schreiben einer Variablen durch einen Thread und einem nachfolgenden Lesen derselben Variablen durch einen anderen Thread erzeugen. Wenn die „geschieht vor" Beziehung existiert, wird die JIT - Compiler *verpflichtet* Code zu generieren , die sicherstellen, dass der Lesevorgang den Wert durch den Schreib geschrieben sieht.

Damit ist es möglich, über die Speicherkohärenz in einem Java-Programm zu entscheiden und zu entscheiden, ob dies für *alle* Ausführungsplattformen vorhersehbar und konsistent ist.

Geschieht vor Beziehungen

(Nachfolgend finden Sie eine vereinfachte Version der Java-Sprachspezifikation. Für ein tieferes Verständnis müssen Sie die Spezifikation selbst lesen.)

Happens-before-Beziehungen sind der Teil des Speichermodells, mit dem wir die Sichtbarkeit des Speichers verstehen und begründen können. Wie die JLS sagt (JLS 17.4.5):

> "Zwei *Aktionen* können durch eine *zufällige* Beziehung geordnet werden. Wenn eine Aktion *vor einer* anderen *passiert* , ist die erste für die zweite sichtbar und vor ihr angeordnet."

Was bedeutet das?

Aktionen

Die Aktionen, auf die sich das obige Zitat bezieht, sind in JLS 17.4.2 angegeben . Es gibt 5 Arten von Aktionen, die durch die Spezifikation definiert werden:

- Lesen: Lesen einer nichtflüchtigen Variablen.
- Schreiben: Schreiben einer nichtflüchtigen Variablen.
- Synchronisationsaktionen:
  - Flüchtiges Lesen: Lesen einer flüchtigen Variablen.
  - Flüchtiges Schreiben: Schreiben einer flüchtigen Variablen.
  - Sperren. Einen Monitor sperren
  - Freischalten. Entsperren eines Monitors

- Die (synthetischen) ersten und letzten Aktionen eines Threads.
- Aktionen, die einen Thread starten oder feststellen, dass ein Thread beendet wurde.

- Externe Aktionen. Eine Aktion, deren Ergebnis von der Umgebung abhängt, in der sich das Programm befindet.
- Thread-Divergenzaktionen. Diese modellieren das Verhalten bestimmter Arten von Endlosschleifen.

Programmreihenfolge und Synchronisationsreihenfolge

Diese beiden Ordnungen (JLS 17.4.3 und JLS 17.4.4) bestimmen die Ausführung von Anweisungen in Java

Die Programmreihenfolge beschreibt die Reihenfolge der Anweisungsausführung innerhalb eines einzelnen Threads.

Die Synchronisationsreihenfolge beschreibt die Reihenfolge der Anweisungsausführung für zwei durch eine Synchronisation verbundene Anweisungen:

- Eine Entsperraktion auf dem Monitor wird *synchronisiert - mit* allen nachfolgenden Sperraktionen auf diesem Monitor.
- Ein Schreibvorgang in eine flüchtige Variable wird *mit* allen nachfolgenden Lesevorgängen derselben Variablen durch einen beliebigen Thread *synchronisiert* .
- Eine Aktion, die einen Thread startet (dh der Aufruf von Thread.start()), *synchronisiert sich mit* der ersten Aktion in dem Thread, den er startet (dh dem Aufruf der run() Methode des Threads).
- Die Standardinitialisierung der Felder wird *mit* der ersten Aktion in jedem Thread *synchronisiert* . (Eine Erklärung hierzu finden Sie in der JLS.)
- Die abschließende Aktion in einem Thread wird *mit* jeder Aktion in einem anderen Thread, die die Beendigung erkennt, *synchronisiert* . isTerminated() die Rückgabe eines join() Aufrufs oder ein isTerminated() Aufruf, der true zurückgibt.
- Wenn ein Thread einen anderen Thread unterbricht, wird der Interruptaufruf im ersten Thread *mit* dem Punkt *synchronisiert, an* dem ein anderer Thread feststellt, dass der Thread unterbrochen wurde.

Passiert vor der Bestellung

Diese Reihenfolge (JLS 17.4.5) bestimmt, ob ein Speicherschreiben garantiert für ein nachfolgendes Lesen des Speichers sichtbar ist.

Genauer gesagt, ein Lesen einer Variablen v garantiert, dass ein Schreiben in v genau dann beobachtet wird, wenn das write(v) *vor dem* read(v) *geschieht* UND es kein intervenierendes Schreiben in v . Wenn es dazwischenliegende Schreibvorgänge gibt, werden beim read(v) die Ergebnisse eher als bei den früheren angezeigt.

Die Regeln, die die *zufällige* Reihenfolge definieren, lauten wie folgt:

- **Happens-Before-Regel # 1** - Wenn x und y Aktionen des gleichen Threads sind und x vor y in der *Programmreihenfolge steht* , *passiert* x *vor* y.
- **Happens-Before-Regel Nr. 2** - Es gibt eine **Vor-** Ereignis-Kante vom Ende eines Konstruktors eines Objekts bis zum Beginn eines Finalizers für dieses Objekt.
- **Happens-Before-Regel Nr. 3** - Wenn eine Aktion x *mit* einer nachfolgenden Aktion y

synchronisiert wird , *passiert* x *vor* y.

- Passiert **vor Regel 4** - Wenn x *passiert - bevor* y und y *passiert - bevor* z dann x *passiert - bevor* z.

Darüber hinaus werden verschiedene Klassen in den Java-Standardbibliotheken als definierende *Vorgängerbeziehungen* definiert. Sie können dies so interpretieren, dass es *irgendwie* passiert, ohne dass Sie genau wissen müssen, wie die Garantie erfüllt wird.

Geschieht, bevor die Argumentation auf einige Beispiele angewendet wird

Wir werden einige Beispiele vorstellen, um zu demonstrieren, wie die Anwendung *angewendet wird, bevor* geprüft wird, ob Schreibvorgänge für nachfolgende Lesevorgänge sichtbar sind.

Single-Threaded-Code

Wie zu erwarten, sind Schreibvorgänge immer für nachfolgende Lesevorgänge in einem Singlethread-Programm sichtbar.

```
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)
    }
}
```

Durch Happens-Before-Regel Nr. 1:

1. Die write(a) *geschieht vor* der write(b) .
2. Die write(b) *geschieht vor* der read(a) .
3. Die Aktion read(a) *geschieht vor* der Aktion read(a) .

Durch Happens-Before-Regel Nr. 4:

4. write(a) *passiert - bevor* write(b) UND write(b) *passiert - bevor* read(a) IMPLIES write(a) *passiert - bevor* read(a) .
5. write(b) *geschieht vor dem* read(a) UND read(a) *geschieht vor dem* read(b) IMPLIES write(b) *geschieht vor dem* read(b) .

Zusammenfassen:

6. Die write(a) *-heat-before-* read(a) Beziehung read(a) bedeutet, dass die Anweisung a + b garantiert den korrekten Wert von a erkennt.
7. Die write(b) *-Ohre-* read(b) Beziehung read(b) bedeutet, dass die Anweisung a + b garantiert den korrekten Wert von b .

Verhalten von 'flüchtig' in einem Beispiel mit 2 Threads

Wir werden den folgenden Beispielcode verwenden, um einige Implikationen des Speichermodells für `volatil zu untersuchen.

```
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)
    }
}
```

Betrachten Sie zunächst die folgende Abfolge von Anweisungen, die 2 Threads betreffen:

1. Eine einzelne Instanz von VolatileExample wird erstellt. nennen sie es ve ,
2. ve.update(1, 2) wird in einem Thread aufgerufen und
3. ve.observe() wird in einem anderen Thread aufgerufen.

Durch Happens-Before-Regel Nr. 1:

1. Die write(a) *geschieht vor* der volatile-write(a) .
2. Die volatile-read(a) Aktion volatile-read(a) *geschieht vor* der read(b) Aktion read(b) .

Durch Happens-Before-Regel Nr. 2:

3. Die volatile-write(a) im ersten Thread *geschieht vor* der volatile-read(a) im zweiten Thread.

Durch Happens-Before-Regel Nr. 4:

4. Die write(b) -Aktion im ersten Thread *geschieht vor* der read(b) -Aktion im zweiten Thread.

Mit anderen Worten, für diese bestimmte Sequenz wird garantiert, dass der 2. Thread die Aktualisierung der nichtflüchtigen Variablen b sieht, die vom ersten Thread vorgenommen wird. Es sollte jedoch auch klar sein, dass, wenn die Zuweisungen in der update umgekehrt sind oder die observe() -Methode die Variable b vor a liest, die *Ereigniskette vorher* gebrochen würde. Die Kette wäre auch gebrochen, wenn volatile-read(a) im zweiten Thread nicht auf volatile-write(a) im ersten Thread folgt.

Wenn die Kette unterbrochen wird, kann nicht *garantiert werden,* dass observe() den korrekten Wert von b sieht.

Flüchtig mit drei Threads

Nehmen wir an, wir fügen dem vorherigen Beispiel einen dritten Thread hinzu:

1. Eine einzelne Instanz von VolatileExample wird erstellt. nennen sie es ve ,
2. update zwei Threads aufrufen:
    - ve.update(1, 2) wird in einem Thread aufgerufen.
    - ve.update(3, 4) wird im zweiten Thread aufgerufen.
3. ve.observe() wird anschließend in einem dritten Thread aufgerufen.

Um dies vollständig zu analysieren, müssen wir alle möglichen Verschachtelungen der Anweisungen in Thread eins und Thread zwei berücksichtigen. Stattdessen betrachten wir nur zwei davon.

Szenario 1 - annehmen , dass update(1, 2) voraus update(3,4) erhalten wir diese Reihenfolge:

```
write(b, 1), write-volatile(a, 2)     // first thread
write(b, 3), write-volatile(a, 4)     // second thread
read-volatile(a), read(b)             // third thread
```

In diesem Fall ist es leicht zu erkennen, dass es eine ununterbrochene *Vor-Ereignis*- Kette vom write(b, 3) zum read(b) . Darüber hinaus gibt es kein Schreiben an b . In diesem Szenario wird garantiert, dass b für den dritten Thread den Wert 3 .

Szenario 2 - Nehmen Sie an, dass sich update(1, 2) und update(3,4) überlappen und die ations wie folgt verschachtelt sind:

```
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
```

Während es nun eine *Vor-Ereignis*- Kette von write(b, 3) zu read(b) gibt, gibt es eine intervenierende write(b, 1) die vom anderen Thread ausgeführt wird. Dies bedeutet, dass wir nicht sicher sein können, welchen Wert read(b) sehen wird.

(Nebenbei: Dies zeigt, dass wir uns nicht auf volatile um die Sichtbarkeit nichtflüchtiger Variablen sicherzustellen, außer in sehr begrenzten Situationen.)

So vermeiden Sie, das Speichermodell verstehen zu müssen

Das Speichermodell ist schwer zu verstehen und nur schwer anzuwenden. Dies ist nützlich, wenn Sie über die Richtigkeit von Multithreadcode nachdenken müssen, diese Argumentation jedoch nicht für jede Multithread-Anwendung, die Sie schreiben, erforderlich ist.

Wenn Sie beim Schreiben von simultanem Code in Java die folgenden Prinzipien anwenden, können Sie die Notwendigkeit, *vor einer* Argumentation vorzugehen, *weitgehend* vermeiden.

- Verwenden Sie möglichst unveränderliche Datenstrukturen. Eine ordnungsgemäß implementierte unveränderliche Klasse ist Thread-sicher und führt nicht zu Thread-Sicherheitsproblemen, wenn Sie sie mit anderen Klassen verwenden.
- Verstehen und vermeiden Sie "unsichere Veröffentlichung".
- Verwenden Sie primitive Mutexe oder Lock , um den Zugriff auf den Status in veränderlichen Objekten zu synchronisieren, die threadsicher sein müssen [1] .
- Verwenden Sie Executor / ExecutorService oder das Fork-Join-Framework, anstatt zu versuchen, Management-Threads direkt zu erstellen.
- Verwenden Sie die `java.util.concurrent-Klassen, die erweiterte Sperren, Semaphore, Latches und Barrieren bereitstellen, anstatt direkt mit wait / notify / notifyAll zu arbeiten.
- Verwenden Sie die java.util.concurrent Versionen von Karten, Sets, Listen, Warteschlangen und Deques anstelle der externen Synchronisierung von nicht gleichzeitig ablaufenden Sammlungen.

Das allgemeine Prinzip besteht darin, zu versuchen, die eingebauten Parallelitätsbibliotheken von Java zu verwenden, anstatt die Parallelität "zu rollen". Sie können sich darauf verlassen, dass sie funktionieren, wenn Sie sie richtig verwenden.

---

1 - Nicht alle Objekte müssen threadsicher sein. Wenn zum Beispiel ein Objekt oder Objekte *fadenbeschränkt sind* (dh es ist nur für einen Thread zugänglich), ist die Thread-Sicherheit nicht relevant.

Java-Speichermodell online lesen: https://riptutorial.com/de/java/topic/6829/java-speichermodell

Kapitel 82: Java-Speicherverwaltung

Bemerkungen

In Java werden Objekte im Heapspeicher zugewiesen, und der Heapspeicher wird durch die automatische Garbage Collection wiederhergestellt. Ein Anwendungsprogramm kann ein Java-Objekt nicht explizit löschen.

Die Grundprinzipien der Java Garbage Collection werden im Garbage Collection- Beispiel beschrieben. Andere Beispiele beschreiben die Finalisierung, wie man den Garbage Collector von Hand auslöst, und das Problem von Speicherlecks.

Examples

Finalisierung

Ein Java-Objekt kann eine finalize Methode deklarieren. Diese Methode wird aufgerufen, kurz bevor Java den Speicher für das Objekt freigibt. Es wird normalerweise so aussehen:

```
public class MyClass {

    //Methods for the class

    @Override
    protected void finalize() throws Throwable {
        // Cleanup code
    }
}
```

Es gibt jedoch einige wichtige Einschränkungen beim Verhalten von Java-Finalisierungen.

- Java übernimmt keine Garantie dafür, wann eine finalize() -Methode aufgerufen wird.
- Java garantiert nicht einmal, dass eine finalize() -Methode einige Zeit während der Laufzeit der laufenden Anwendung aufgerufen wird.
- Die einzige Sache, die garantiert ist, ist, dass die Methode aufgerufen wird, bevor das Objekt gelöscht wird ... wenn das Objekt gelöscht wird.

Die oben genannten Einschränkungen bedeuten, dass es keine gute Idee ist, sich auf die finalize Methode zu verlassen, um Bereinigungs- (oder andere) Aktionen auszuführen, die rechtzeitig ausgeführt werden müssen. Ein zu starkes Vertrauen in die Finalisierung kann zu Speicherverlusten, Speicherverlusten und anderen Problemen führen.

Kurz gesagt, es gibt nur wenige Situationen, in denen die Fertigstellung tatsächlich eine gute Lösung darstellt.

Finalizer laufen nur einmal

Normalerweise wird ein Objekt gelöscht, nachdem es abgeschlossen wurde. Dies passiert jedoch nicht immer. Betrachten Sie das folgende Beispiel [1] :

```
public class CaptainJack {
    public static CaptainJack notDeadYet = null;

    protected void finalize() {
        // Resurrection!
        notDeadYet = this;
    }
}
```

Wenn eine Instanz von CaptainJack mehr erreichbar ist und der Garbage Collector versucht, diese

zurückzufordern, weist die Methode notDeadYet finalize() der notDeadYet Variablen einen Verweis auf die Instanz zu. Dadurch wird die Instanz erneut erreichbar, und der Garbage Collector löscht sie nicht.

Frage: Ist Captain Jack unsterblich?

Antwort: Nein

Der Haken ist, dass die JVM einen Finalizer für ein Objekt nur einmal im Leben ausführt. Wenn Sie notDeadYet Wert null notDeadYet sodass eine wiederhergestellte Instanz erneut nicht erreichbar ist, ruft der Garbage Collector nicht finalize() für das Objekt auf.

1 - Siehe https://en.wikipedia.org/wiki/Jack_Harkness .

GC manuell auslösen

Sie können den Garbage Collector manuell auslösen, indem Sie ihn aufrufen

```
System.gc();
```

Java garantiert jedoch nicht, dass der Garbage Collector ausgeführt wurde, wenn der Aufruf zurückkehrt. Diese Methode "schlägt" der JVM (Java Virtual Machine) einfach vor, dass sie den Garbage Collector ausführen soll, zwingt sie jedoch nicht dazu.

Es wird im Allgemeinen als schlechte Praxis betrachtet, zu versuchen, die Garbage Collection manuell auszulösen. Die JVM kann mit der Option -XX:+DisableExplicitGC , um Aufrufe von System.gc() zu deaktivieren. Das Auslösen der Garbage Collection durch Aufrufen von System.gc() kann die normalen Garbage Management / Object Promotion-Aktivitäten der von der JVM verwendeten spezifischen Garbage Collection- System.gc() .

Müllsammlung

Der C ++ - Ansatz - neu und löschen

In einer Sprache wie C ++ ist das Anwendungsprogramm dafür verantwortlich, den von dynamisch zugewiesenen Speicher verwendeten Speicher zu verwalten. Wenn ein Objekt mit dem new Operator im C ++ - Heap erstellt wird, muss der delete Operator entsprechend verwendet werden, um das Objekt zu entsorgen:

- Wenn das Programm das delete eines Objekts vergisst und nur das Objekt "vergisst", geht der zugehörige Speicher für die Anwendung verloren. Der Begriff für diese Situation ist ein *Speicherverlust* , und zu viel Speicherverlust, so dass eine Anwendung mehr und mehr Speicher benötigt und schließlich abstürzt.
- Wenn eine Anwendung andererseits versucht, dasselbe Objekt zweimal zu delete oder ein Objekt zu verwenden, nachdem es gelöscht wurde, kann die Anwendung aufgrund von Problemen mit der Speicherbeschädigung abstürzen

In einem komplizierten C ++ - Programm kann das Implementieren der Speicherverwaltung mit new und delete zeitaufwändig sein. In der Tat ist die Speicherverwaltung eine häufige Fehlerquelle.

Der Java-Ansatz - Speicherbereinigung

Java verfolgt einen anderen Ansatz. Anstelle eines expliziten delete stellt Java einen automatischen Mechanismus bereit, der als Garbage Collection bezeichnet wird, um den Speicher wiederzuerlangen, der von nicht mehr benötigten Objekten beansprucht wird. Das Java-Laufzeitsystem übernimmt die Verantwortung für das Auffinden der zu entsorgenden Objekte. Diese Aufgabe wird von einer Komponente ausgeführt, die als *Garbage Collector* oder kurz GC bezeichnet wird.

Während der Ausführung eines Java-Programms können Sie jederzeit die Menge aller vorhandenen Objekte in zwei unterschiedliche Teilmengen [1] aufteilen:

- Erreichbare Objekte werden von der JLS wie folgt definiert:

  > Ein erreichbares Objekt ist jedes Objekt, auf das bei jeder möglichen fortlaufenden Berechnung von einem beliebigen Live-Thread aus zugegriffen werden kann.

  In der Praxis bedeutet dies, dass es eine Kette von Verweisen gibt, die von einer lokalen Variablen im Gültigkeitsbereich oder einer static Variablen ausgehen, über die ein Code das Objekt erreichen kann.

- Nicht erreichbare Objekte sind Objekte, *die nicht* wie oben beschrieben erreichbar sind.

Alle Objekte, die nicht erreichbar sind, *können* für die Garbage Collection verwendet werden. Dies bedeutet nicht , dass sie Müll gesammelt *werden*. Eigentlich:

- Ein nicht erreichbares Objekt *wird nicht* sofort abgerufen, wenn *es nicht* erreichbar ist [1] .
- Ein unerreichbarer Gegenstand wird *möglicherweise niemals* als Müll gesammelt.

Die Java-Sprachspezifikation gibt einer JVM-Implementierung viel Spielraum, um zu entscheiden, wann nicht erreichbare Objekte erfasst werden sollen. In der Praxis wird auch die Erlaubnis erteilt, dass eine JVM-Implementierung bei der Erkennung nicht erreichbarer Objekte konservativ ist.

Die JLS garantiert nur, dass keine *erreichbaren* Objekte gesammelt werden.

Was passiert, wenn ein Objekt unerreichbar wird?

Zunächst einmal geschieht nichts gesagt , wenn ein Objekt nicht mehr erreichbar *ist*. Dies geschieht nur, wenn der Garbage Collector ausgeführt wird *und* erkennt, dass das Objekt nicht erreichbar ist. Es ist außerdem üblich, dass ein GC-Lauf nicht alle nicht erreichbaren Objekte erkennt.

Wenn der GC ein nicht erreichbares Objekt erkennt, können die folgenden Ereignisse auftreten.

1. Wenn Reference sind, die auf das Objekt verweisen, werden diese Referenzen gelöscht, bevor das Objekt gelöscht wird.
2. Wenn das Objekt *finalizable* ist, dann wird es fertig gestellt sein. Dies geschieht, bevor das Objekt gelöscht wird.
3. Das Objekt kann gelöscht werden und der Speicher, den es belegt, kann zurückgefordert werden.

Beachten Sie, dass es eine klare Reihenfolge gibt, in der die obigen Ereignisse auftreten *können*. Der Garbage Collector muss jedoch nicht die endgültige Löschung eines bestimmten Objekts in einem bestimmten Zeitraum durchführen.

Beispiele für erreichbare und nicht erreichbare Objekte

Betrachten Sie die folgenden Beispielklassen:

```
// 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
    }
}
```

Lassen Sie uns untersuchen, was passiert, wenn test() aufgerufen wird. Die Anweisungen T1, T2 und T3 erzeugen Node , und die Objekte sind alle über die Variablen n1 , n2 und n3 erreichbar. Anweisung T4 weist dem next Feld des ersten Feldes die Referenz auf das 2. Node zu. Node ist der 2. Node über zwei Pfade erreichbar:

```
n2 -> Node2
n1 -> Node1, Node1.next -> Node2
```

In Anweisung T5 weisen wir n2 null zu. Dadurch wird die erste der Erreichbarkeitsketten für Node2 , die zweite bleibt jedoch ungebrochen, sodass Node2 weiterhin erreichbar ist.

In Anweisung T6 weisen wir n3 null zu. Dies Node3 die einzige Erreichbarkeitskette für Node3 , wodurch Node3 nicht erreichbar ist. Node1 und Node2 sind jedoch beide weiterhin über die Variable n1 erreichbar.

Wenn die test() -Methode zurückkehrt, gehen ihre lokalen Variablen n1 , n2 und n3 schließlich aus dem Gültigkeitsbereich heraus und sind daher für nichts zugänglich. Dies bricht die verbleibenden Erreichbarkeitsketten für Node1 und Node2 , und alle der Node Objekte sind noch nicht erreichbar und *kommen* für die Garbage Collection.

---

1 - Dies ist eine Vereinfachung, die Finalisierungs- und Reference ignoriert. 2 - Hypothetisch könnte eine Java-Implementierung dies tun, aber der damit verbundene Performance-Aufwand macht dies unpraktisch.

Festlegen der Heap-, PermGen- und Stack-Größen

Wenn eine virtuelle Java-Maschine gestartet wird, muss sie wissen, wie groß der Heap ist, und die Standardgröße für Thread-Stacks. Diese können mithilfe von Befehlszeilenoptionen im java Befehl angegeben werden. Bei Java-Versionen vor Java 8 können Sie auch die Größe der PermGen-Region des Heap angeben.

Beachten Sie, dass PermGen in Java 8 entfernt wurde. Wenn Sie versuchen, die PermGen-Größe festzulegen, wird die Option ignoriert (mit einer Warnmeldung).

Wenn Sie die Heap- und Stack-Größen nicht explizit angeben, verwendet die JVM Standardwerte, die auf einer Version und plattformspezifisch berechnet werden. Dies kann dazu führen, dass Ihre Anwendung zu wenig oder zu viel Speicher verwendet. Dies ist normalerweise für Thread-Stacks in Ordnung, für ein Programm, das viel Speicher benötigt, kann dies jedoch problematisch sein.

Festlegen der Heap-, PermGen- und Standard-Stack-Größen:

Die folgenden JVM-Optionen legen die Heapgröße fest:

- -Xms<size> - legt die anfängliche Größe des -Xms<size>
- -Xmx<size> - -Xmx<size> die maximale Größe des -Xmx<size>
- -XX:PermSize<size> - Legt die anfängliche PermGen-Größe fest
- -XX:MaxPermSize<size> - Legt die maximale PermGen-Größe fest
- -Xss<size> - -Xss<size> die Standard- -Xss<size>

Der <size> -Parameter kann eine Anzahl von Bytes sein oder ein Suffix von k , m oder g . Letztere geben die Größe in Kilobyte, Megabyte und Gigabyte an.

Beispiele:

```
$ java -Xms512m -Xmx1024m JavaApp
$ java -XX:PermSize=64m -XX:MaxPermSize=128m JavaApp
$ java -Xss512k JavaApp
```

Ermitteln der Standardgrößen:

Mit der Option -XX:+printFlagsFinal können Sie die Werte aller Flags drucken, bevor Sie die JVM starten. Dies kann verwendet werden, um die Standardeinstellungen für die Einstellungen für Heap und Stapelgröße wie folgt zu drucken:

- Für Linux, Unix, Solaris und Mac OSX

  $ java -XX: + PrintFlagsFinal -version | grep -iE 'HeapSize | PermSize | ThreadStackSize'

- Für Windows:

  java -XX: + PrintFlagsFinal -version | findstr / i "HeapSize PermSize ThreadStackSize"

Die Ausgabe der obigen Befehle ähnelt der folgenden:

```
uintx InitialHeapSize                          := 20655360        {product}
uintx MaxHeapSize                              := 331350016       {product}
uintx PermSize                                  = 21757952        {pd product}
uintx MaxPermSize                               = 85983232        {pd product}
 intx ThreadStackSize                           = 1024            {pd product}
```

Die Größen werden in Bytes angegeben.

Speicherlecks in Java

Im Beispiel für die Garbage-Sammlung wurde impliziert, dass Java das Problem von Speicherverlusten löst. Das stimmt eigentlich nicht. Ein Java-Programm kann Speicher verlieren, obwohl die Ursachen der Lecks ziemlich unterschiedlich sind.

Erreichbare Objekte können auslaufen

Betrachten Sie die folgende naive Stack-Implementierung.

```
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;
    }
}
```

Wenn Sie ein Objekt push und es sofort wieder pop , wird immer noch ein Verweis auf das Objekt im stack Array angezeigt.

Die Logik der Stack-Implementierung bedeutet, dass diese Referenz nicht an einen Client der API zurückgegeben werden kann. Wenn ein Objekt geknackt wurde, können wir nachweisen, dass es *"nicht in einer möglichen fortlaufenden Berechnung von einem Live-Thread aus zugänglich ist"* . Das Problem ist, dass eine JVM der aktuellen Generation dies nicht beweisen kann. JVMs der aktuellen Generation berücksichtigen die Logik des Programms nicht bei der Bestimmung, ob Referenzen erreichbar sind. (Zunächst einmal ist es nicht praktisch.)

Abgesehen von der Frage, was *Erreichbarkeit* wirklich bedeutet, haben wir hier eindeutig eine Situation, in der die NaiveStack Implementierung an Objekten "hängt", die zurückgefordert werden müssen. Das ist ein Speicherleck.

In diesem Fall ist die Lösung einfach:

```
    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;
    }
```

Caches können Speicherlecks sein

Eine übliche Strategie zur Verbesserung der Serviceleistung ist das Zwischenspeichern von Ergebnissen. Die Idee ist, dass Sie häufige Anforderungen und ihre Ergebnisse in einer Datenstruktur im Arbeitsspeicher speichern, die als Cache bezeichnet wird. Jedes Mal, wenn eine Anforderung erfolgt, suchen Sie die Anforderung im Cache nach. Wenn die Suche erfolgreich ist, geben Sie die entsprechenden gespeicherten Ergebnisse zurück.

Diese Strategie kann sehr effektiv sein, wenn sie richtig umgesetzt wird. Bei falscher Implementierung kann ein Cache jedoch ein Speicherverlust sein. Betrachten Sie das folgende Beispiel:

```
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;
    }
}
```

Das Problem bei diesem Code besteht darin, dass der Aufruf von doRequest zwar einen neuen Eintrag zum Cache hinzufügen kann, jedoch nicht entfernt werden kann. Wenn der Dienst ständig andere Aufgaben erhält, verbraucht der Cache schließlich den gesamten verfügbaren Speicherplatz. Dies ist eine Form von Speicherverlust.

Ein Lösungsansatz besteht darin, einen Cache mit einer maximalen Größe zu verwenden und alte

Einträge zu verwerfen, wenn der Cache den Maximalwert überschreitet. (Das Löschen des am wenigsten verwendeten Eintrags ist eine gute Strategie.) Ein anderer Ansatz besteht darin, den Cache mit WeakHashMap sodass die JVM Cache-Einträge WeakHashMap kann, wenn der WeakHashMap zu voll wird.

Java-Speicherverwaltung online lesen: https://riptutorial.com/de/java/topic/2804/java-speicherverwaltung

Kapitel 83: JAXB

Einführung

JAXB oder Java Architecture for XML Binding (JAXB) ist ein Software-Framework, mit dem Java-Entwickler Java-Klassen XML-Repräsentationen zuordnen können. Auf dieser Seite werden die Leser in JAXB anhand von ausführlichen Beispielen zu ihren Funktionen eingeführt, die hauptsächlich für das Marshalling und das Marshaling von Java-Objekten im XML-Format und umgekehrt dienen.

Syntax

- `JAXB.marshall (object, fileObjOfXML);`
- `Objekt obj = JAXB.unmarshall (fileObjOfXML, Klassenname);`

Parameter

Parameter	Einzelheiten
fileObjOfXML	`File` einer XML-Datei
Klassenname	Name einer Klasse mit der Erweiterung `.class`

Bemerkungen

Mit dem im JDK verfügbaren XJC-Tool kann Java-Code für eine in einem XML-Schema (.xsd Datei) beschriebene XML-Struktur automatisch generiert werden. Siehe XJC-Thema .

Examples

Schreiben einer XML-Datei (Marshalling eines Objekts)

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

Mit der Annotation XMLRootElement können wir eine Klasse als Wurzelelement einer XML-Datei markieren.

```
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() wird verwendet, um den Inhalt des Objekts in eine XML-Datei zu schreiben. Hier werden das user und ein neues File Objekt als Argumente an den marshal() .

Bei erfolgreicher Ausführung wird eine XML-Datei mit dem Namen UserDetails.xml im Klassenpfad mit dem folgenden Inhalt erstellt.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

Lesen einer XML-Datei (unmarshalling)

So lesen Sie eine XML-Datei mit dem Namen UserDetails.xml mit dem folgenden Inhalt

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

Wir benötigen eine POJO-Klasse mit dem Namen User.java (User.java unten)

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

Hier haben wir die Variablen und den Klassennamen gemäß den XML-Knoten erstellt. Um sie zuzuordnen, verwenden wir die Annotation XmlRootElement für die Klasse.

```
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!");
        }
    }
}
```

Hier wird die unmarshal() -Methode verwendet, um die XML-Datei zu analysieren. Der XML-Dateiname und der Klassentyp werden als zwei Argumente verwendet. Dann können wir die Getter-Methoden des Objekts verwenden, um die Daten zu drucken.

Verwenden von XmlAdapter zum Generieren des gewünschten XML-Formats

Wenn das gewünschte XML-Format vom Java-Objektmodell abweicht, kann eine XmlAdapter-Implementierung verwendet werden, um das Modellobjekt in ein Objekt im XML-Format und umgekehrt zu transformieren. In diesem Beispiel wird veranschaulicht, wie ein Feldwert in ein Attribut eines Elements mit dem Feldnamen eingefügt wird.

```
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);
    }
}
```

XML-Mapping-Konfiguration für automatisches Feld / Eigenschaft (@XmlAccessorType)

Die Anmerkung `@XmlAccessorType` legt fest, ob Felder / Eigenschaften automatisch in XML serialisiert werden. Beachten Sie, dass die Feld- und Methodenanmerkungen `@XmlElement` , `@XmlAttribute` oder `@XmlTransient` den Standardeinstellungen `@XmlTransient` .

```
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
```

Ausgabe

```
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>
```

Manuelle Konfiguration von Feld- / Eigenschafts-XML-Mappings

Die Anmerkungen `@XmlElement` , `@XmlAttribute` oder `@XmlTransient` und andere im Paket `javax.xml.bind.annotation` ermöglichen dem Programmierer, anzugeben, welche und wie markierte Felder oder Eigenschaften serialisiert werden sollen.

```
@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!");
        }
    }
}
```

Angeben einer XmlAdapter-Instanz, um vorhandene Daten (erneut) zu verwenden

Manchmal sollten bestimmte Dateninstanzen verwendet werden. Eine Wiederherstellung ist nicht erwünscht und das Referenzieren static Daten hätte einen Codegeruch.

Es ist möglich, eine XmlAdapter Instanz Unmarshaller , die der Unmarshaller soll. Unmarshaller kann der Benutzer XmlAdapter ohne Zero- XmlAdapter -Konstruktor verwenden und / oder Daten an den Adapter übergeben.

Beispiel

Benutzerklasse

Die folgende Klasse enthält einen Namen und ein Benutzerbild.

```
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);
    }

}
```

Adapter

Um zu vermeiden, dass dasselbe Bild zweimal im Speicher erstellt wird (und die Daten erneut heruntergeladen werden), speichert der Adapter die Bilder in einer Karte.

Java SE 7

Ersetzen getImage für gültigen Java 7-Code die getImage Methode durch

```
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();
    }

}
```

Beispiel-XMLs

Die folgenden 2 xmls sind für *Jon Skeet* und sein Gegenstück zu Earth 2, die beide genau gleich aussehen und daher den gleichen Avatar verwenden.

```
<?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"
```

Verwendung des Adapters

```
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());
```

Binden eines XML-Namespaces an eine serialisierbare Java-Klasse.

Dies ist ein Beispiel für eine package-info.java Datei, die einen XML-Namespace an eine serialisierbare Java-Klasse bindet. Diese sollte in demselben Paket wie die Java-Klassen abgelegt werden, die mit dem Namespace serialisiert werden sollen.

```
/**
 * 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;
```

Zeichenkette mit XmlAdapter trimmen.

```
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();
    }
}
```

Fügen Sie in package-info.java folgende Deklaration hinzu.

```
@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;
```

JAXB online lesen: https://riptutorial.com/de/java/topic/147/jaxb

Kapitel 84: JAX-WS

Examples

Basisauthentifizierung

Ein JAX-WS-Aufruf mit Standardauthentifizierung ist etwas unklar.

Hier ist ein Beispiel, bei dem Service die Serviceklassendarstellung und Port der Serviceport ist, auf den Sie zugreifen möchten.

```
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();
```

JAX-WS online lesen: https://riptutorial.com/de/java/topic/4105/jax-ws

Kapitel 85: JMX

Einführung

Die JMX-Technologie bietet die Tools zum Erstellen verteilter, webbasierter, modularer und dynamischer Lösungen zum Verwalten und Überwachen von Geräten, Anwendungen und serviceorientierten Netzwerken. Dieser Standard eignet sich prinzipiell für die Anpassung von Altsystemen, die Implementierung neuer Verwaltungs- und Überwachungslösungen und die Einbindung in zukünftige Systeme.

Examples

Einfaches Beispiel mit Platform MBean Server

Nehmen wir an, wir haben einen Server, der neue Benutzer registriert und sie mit einer Nachricht begrüßt. Wir möchten diesen Server überwachen und einige seiner Parameter ändern.

Erstens brauchen wir eine Schnittstelle zu unseren Überwachungs- und Kontrollmethoden

```
public interface UserCounterMBean {
    long getSleepTime();

    void setSleepTime(long sleepTime);

    int getUserCount();

    void setUserCount(int userCount);

    String getGreetingString();

    void setGreetingString(String greetingString);

    void stop();
}
```

Und einige einfache Implementierungen, die uns zeigen lassen, wie es funktioniert und wie wir es beeinflussen

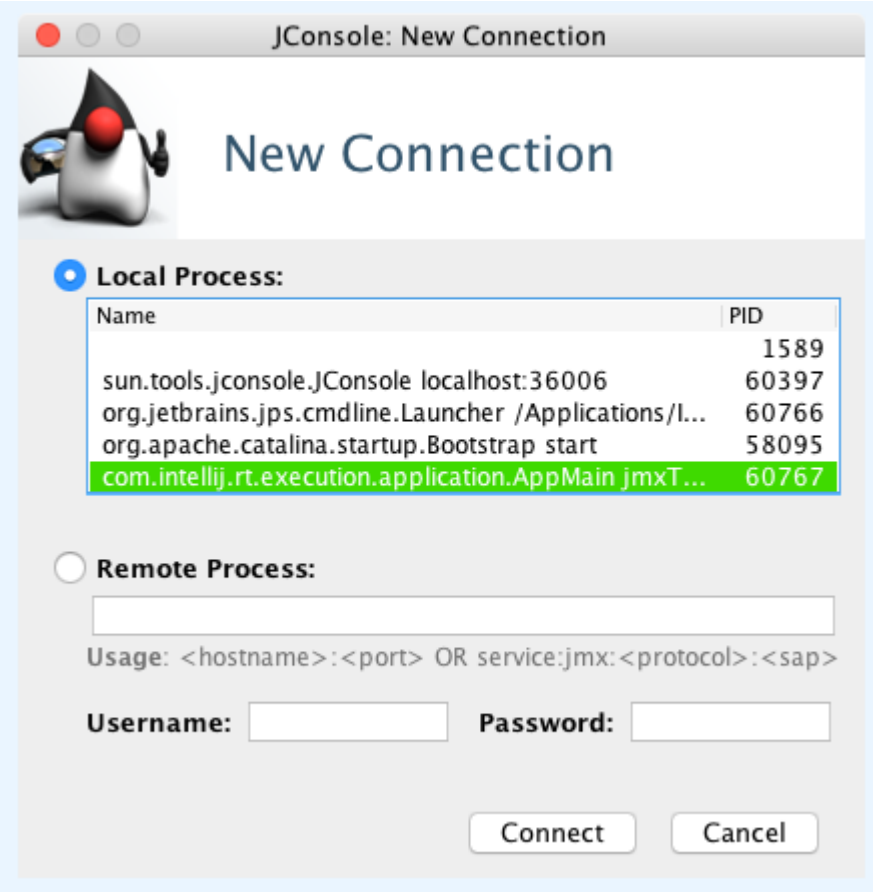
```
public class UserCounter implements UserCounterMBean, Runnable {
    private AtomicLong sleepTime = new AtomicLong(10000);
    private AtomicInteger userCount = new AtomicInteger(0);
    private AtomicReference<String> greetingString = new AtomicReference<>("welcome");
    private AtomicBoolean interrupted = 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) {
            }
        }
    }
}
```

Für ein einfaches Beispiel mit lokaler oder Remote-Verwaltung müssen wir unser MBean registrieren:

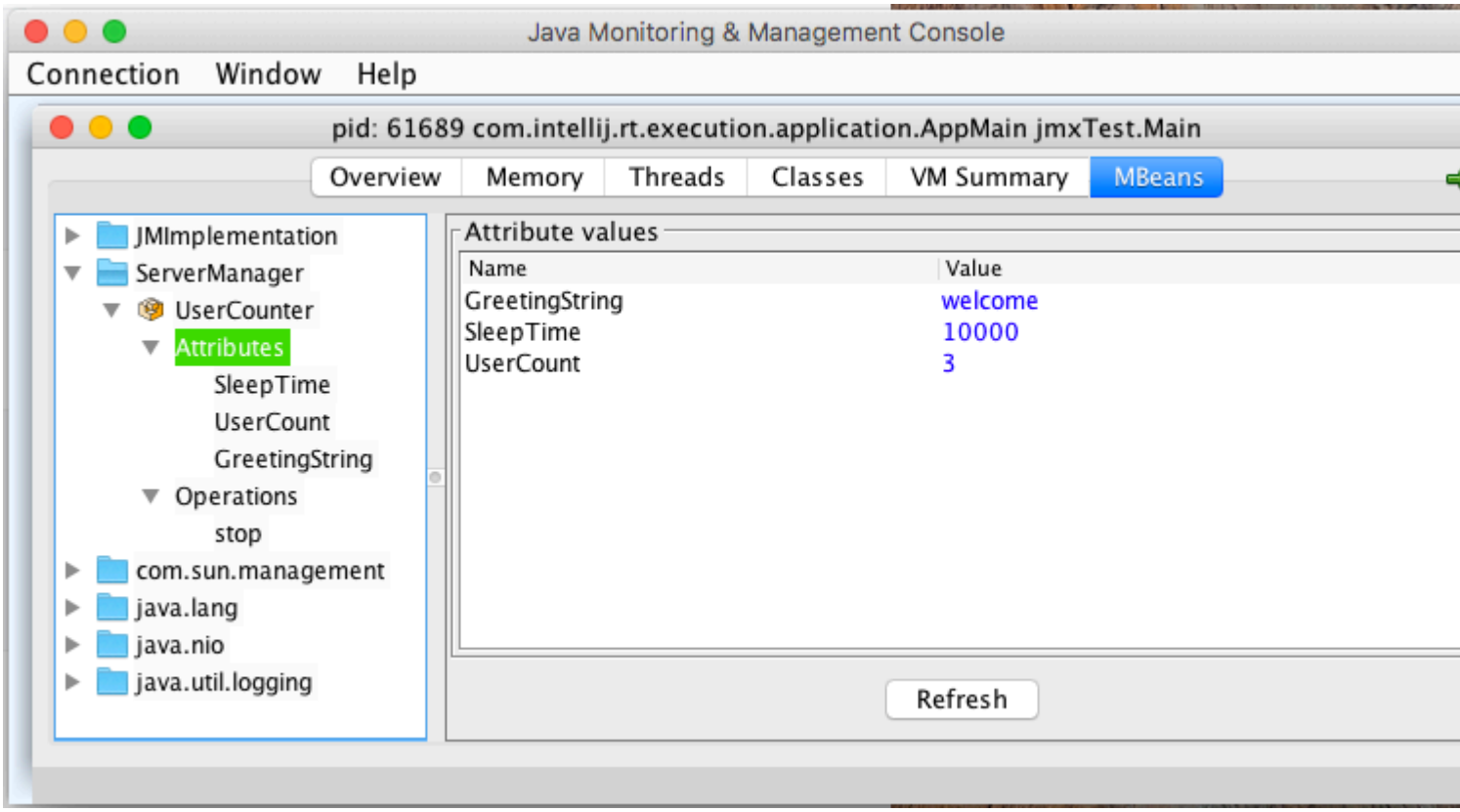
```
import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;

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();
    }
}
```

Danach können wir unsere Anwendung ausführen und über jConsole eine Verbindung herstellen, die sich in Ihrem $JAVA_HOME/bin . Zuerst müssen wir mit unserer Anwendung unseren lokalen Java-

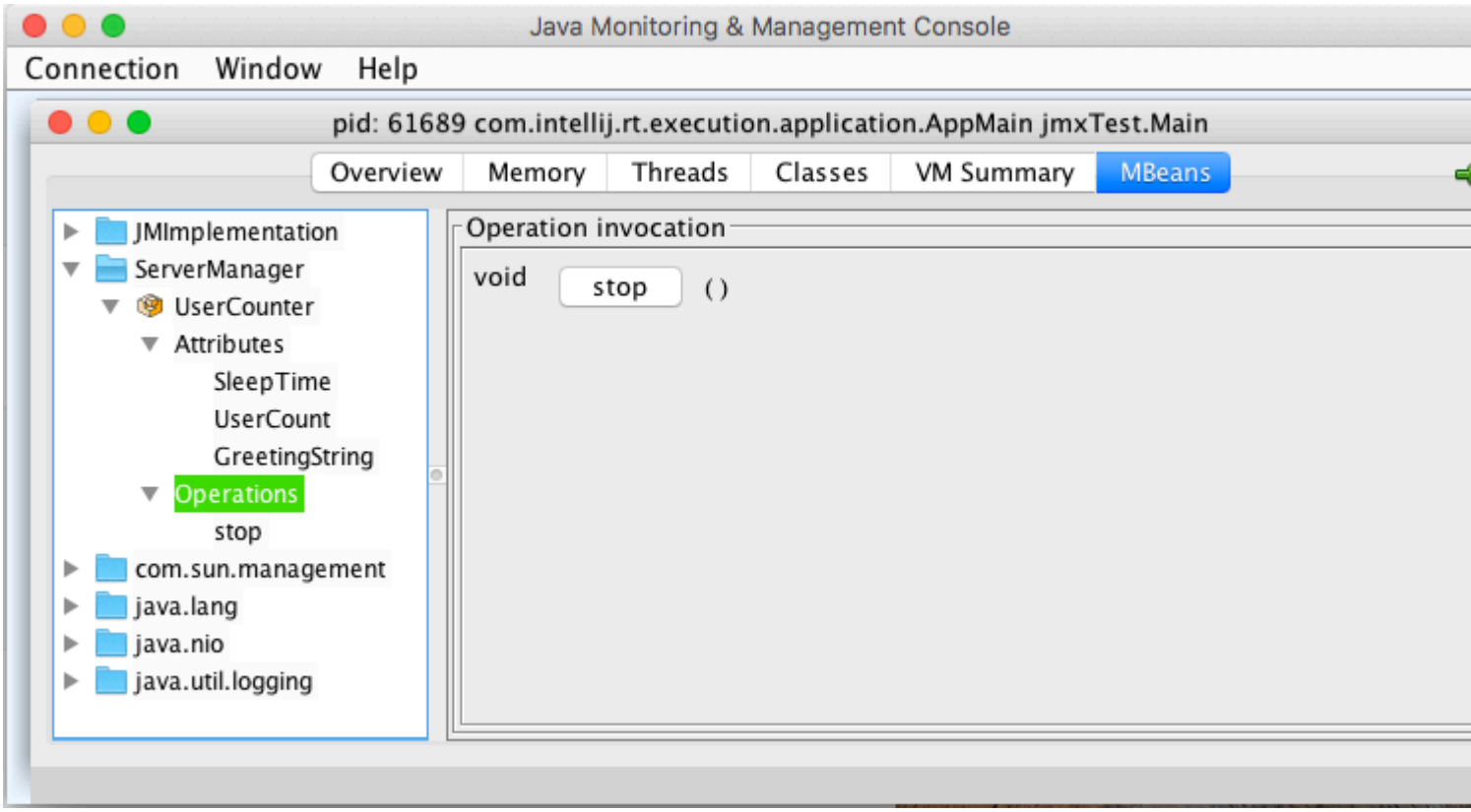


Prozess finden

Wechseln Sie dann zur Registerkarte MBeans und suchen Sie die MBean, die wir in unserer Main-Klasse als ObjectName (im obigen Beispiel ist es ServerManager). Im Abschnitt " Attributes " können wir Attribute erkennen. Wenn Sie nur die Get-Methode angegeben haben, ist das Attribut lesbar, aber nicht schreibbar. Wenn Sie sowohl get- als auch set-Methoden angegeben haben, wäre das Attribut lesbar und schreibbar.



Bestimmte Methoden können im Operations Abschnitt aufgerufen werden.



Wenn Sie die Remote-Verwaltung verwenden möchten, benötigen Sie zusätzliche JVM-Parameter wie:

```
-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
```

Diese Parameter finden Sie in Kapitel 2 der JMX-Handbücher . Danach können Sie über jConsole über jConsole remote mit jconsole host:port oder mit Angabe von host:port oder service:jmx:rmi:///jndi/rmi://hostName:portNum/jmxrmi in der jConsole-GUI verbinden.

Nützliche Links:

- JMX-Anleitungen
- JMX Best Practices

JMX online lesen: https://riptutorial.com/de/java/topic/9278/jmx

Kapitel 86: JNDI

Examples

RMI über JNDI

Dieses Beispiel zeigt, wie JNDI in RMI funktioniert. Es hat zwei Rollen:

- Um dem Server eine Bind / Unind / Re-Bind-API für die RMI-Registry bereitzustellen
- Um dem Client eine Lookup / List-API für die RMI-Registry bereitzustellen.

Die RMI-Registry ist Teil von RMI, nicht von JNDI.

Um dies zu java.rmi.registry.CreateRegistry() , verwenden wir java.rmi.registry.CreateRegistry() , um die RMI-Registry zu erstellen.

1. Server.java (der JNDI-Server)

```
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 (der JNDI-Client)

```
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 (RMI-Serverklasse)

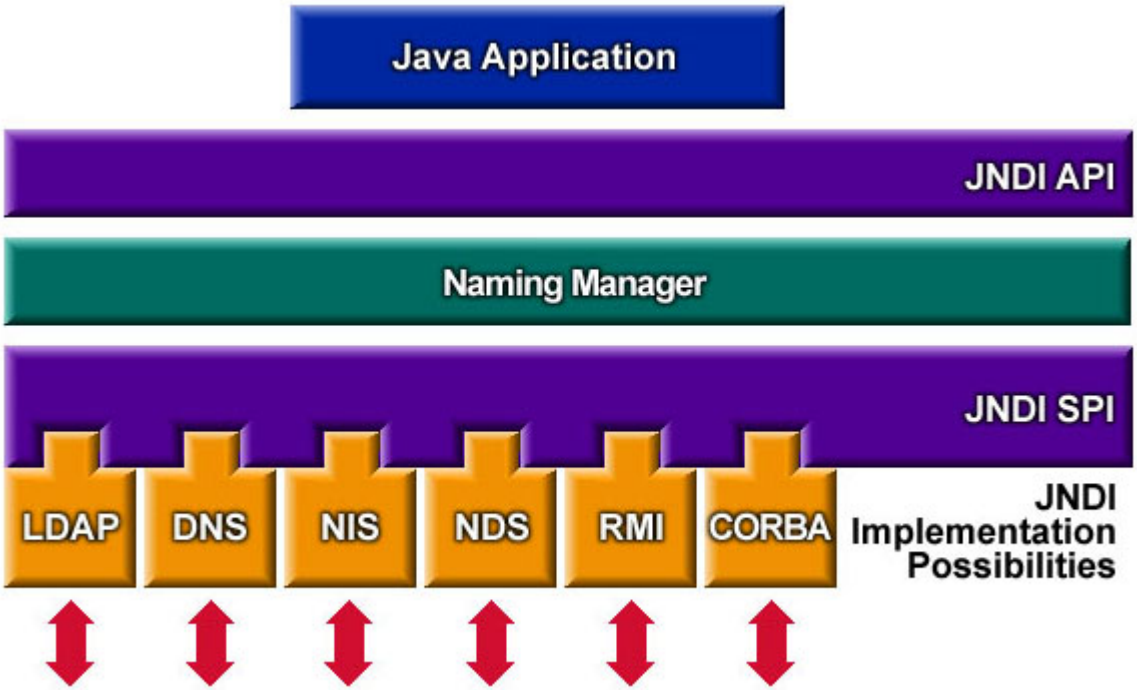
```
package com.neohope.jndi.test;

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;
    }
}
```

So führen Sie den eaxmple aus:

1. Erstellen und starten Sie den Server
2. Erstellen und starten Sie den Client

Vorstellen



JNDI (Java Naming and Directory Interface) ist eine Java-API für einen Verzeichnisdienst, mit der Java-Software-Clients Daten und Objekte über einen Namen ermitteln und suchen können. Es ist so konzipiert, dass es unabhängig von bestimmten Namens- oder Verzeichnisdienstimplementierungen ist.

Die JNDI-Architektur besteht aus einer **API** (Application Programming Interface) und einer **SPI** (Service Provider Interface). Java-Anwendungen verwenden diese API, um auf verschiedene Namens- und Verzeichnisdienste zuzugreifen. Das SPI ermöglicht das transparente Einfügen einer Vielzahl von Namens- und Verzeichnisdiensten, sodass die Java-Anwendung mit Hilfe der API der JNDI-Technologie auf ihre Dienste zugreifen kann.

Wie Sie im obigen Bild sehen können, unterstützt JNDI LDAP, DNS, NIS, NDS, RMI und CORBA. Sie können es natürlich erweitern.

Wie es funktioniert

In diesem Beispiel verwendet Java RMI die JNDI-API, um Objekte in einem Netzwerk nachzuschlagen. Wenn Sie ein Objekt suchen möchten, benötigen Sie mindestens zwei Informationen:

- Wo finde ich das Objekt?

Die RMI-Registry verwaltet die Namensbindungen und gibt an, wo das Objekt zu finden ist.

- Der Name des Objekts

Was ist ein Objektname? Normalerweise handelt es sich um eine Zeichenfolge. Es kann sich auch um ein Objekt handeln, das die Namensschnittstelle implementiert.

Schritt für Schritt

1. Zunächst benötigen Sie eine Registry, die die Namensbindung verwaltet. In diesem Beispiel verwenden wir java.rmi.registry.LocateRegistry .

```
//This will start a registry on localhost, port 1234
registry = LocateRegistry.createRegistry(1234);
```

2. Sowohl Client als auch Server benötigen einen Kontext. Server verwenden den Kontext, um den Namen und das Objekt zu binden. Der Client verwendet den Kontext, um den Namen abzurufen und das Objekt abzurufen.

```
//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. Der Server bindet den Namen und das Objekt

```
//The jndi name is "/neohope/jndi/test01"
bindJNDI("/neohope/jndi/test01", msg);
```

4. Der Client sucht das Objekt unter dem Namen "/ neohope / jndi / test01".

```
//look up the object by name "java:com/neohope/jndi/test01"
NeoMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
```

5. Nun kann der Client das Objekt verwenden

6. Wenn der Server beendet wird, müssen Sie aufräumen.

```
ctx.unbind("/neohope/jndi/test01");
ctx.close();
```

JNDI online lesen: https://riptutorial.com/de/java/topic/5720/jndi

Kapitel 87: JShell

Einführung

JShell ist ein interaktives REPL für Java, das in JDK 9 hinzugefügt wurde. Entwickler können Ausdrücke sofort testen, Klassen testen und mit der Java-Sprache experimentieren. Ein früher Zugriff für jdk 9 ist erhältlich unter: http://jdk.java.net/9/

Syntax

- $ jshell - Starten Sie die JShell REPL
- jshell> / <command> - Führt einen angegebenen JShell-Befehl aus
- jshell> / exit - Beenden Sie JShell
- jshell> / help - Zeigt eine Liste der JShell-Befehle an
- jshell> <java_expression> - Den angegebenen Java-Ausdruck auswerten (optionales Semikolon)
- jshell> / vars OR / Methods OR / types - Zeigt eine Liste von Variablen, Methoden bzw. Klassen an.
- jshell> / open <file> - Liest eine Datei als Eingabe in die Shell
- jshell> / edit <Bezeichner> - Bearbeiten Sie ein Snippet im Set-Editor
- jshell> / set editor <Befehl> - Legt den Befehl fest, der zum Bearbeiten von Ausschnitten mit / edit verwendet wird
- jshell> / drop <Bezeichner> - Ein Snippet löschen
- jshell> / reset - Setzt die JVM zurück und löscht alle Snippets

Bemerkungen

JShell benötigt das Java 9 JDK, das aktuell (März 2017) als frühzeitiger Zugriff auf Snapshots von jdk9.java.net heruntergeladen werden kann . Wenn Sie beim Ausführen des jshell einen Fehler erhalten, der mit der Fehlermeldung Unable to locate an executable nicht JAVA_HOME wird JAVA_HOME wird, müssen Sie sicherstellen, dass JAVA_HOME richtig eingestellt ist.

Standardimporte

Die folgenden Pakete werden beim Start von JShell automatisch importiert:

```
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

JShell eingeben und beenden

JShell wird gestartet

Stellen Sie vor dem Start von JShell sicher, dass die Umgebungsvariable JAVA_HOME auf eine JDK 9-Installation verweist. Führen Sie den folgenden Befehl aus, um JShell zu starten:

```
$ jshell
```

Wenn alles gut geht, sollte eine jshell> Eingabeaufforderung angezeigt werden.

Beenden von JShell

Führen Sie den folgenden Befehl an der JShell-Eingabeaufforderung aus, um JShell zu beenden:

```
jshell> /exit
```

Ausdrücke

In JShell können Sie Java-Ausdrücke mit oder ohne Semikolons auswerten. Diese können von grundlegenden Ausdrücken und Anweisungen bis zu komplexeren Ausdrücken reichen:

```
jshell> 4+2
jshell> System.out.printf("I am %d years old.\n", 421)
```

Loops und Bedingungen sind auch in Ordnung:

```
jshell> for (int i = 0; i<3; i++) {
   ...> System.out.println(i);
   ...> }
```

Beachten Sie, dass **Ausdrücke in Blöcken Semikolons enthalten müssen!**

Variablen

Sie können lokale Variablen in JShell deklarieren:

```
jshell> String s = "hi"
jshell> int i = s.length
```

Beachten Sie, dass Variablen mit unterschiedlichen Typen neu deklariert werden können. Dies gilt vollkommen für JShell:

```
jshell> String var = "hi"
jshell> int var = 3
```

Geben Sie /vars an der JShell-Eingabeaufforderung ein, um eine Liste der Variablen anzuzeigen.

Methoden und Klassen

Sie können Methoden und Klassen in JShell definieren:

```
jshell> void speak() {
   ...> System.out.println("hello");
   ...> }

jshell> class MyClass {
   ...> void doNothing() {}
   ...> }
```

Es sind keine Zugriffsmodifizierer erforderlich. Wie bei anderen Blöcken sind Semikolons innerhalb von Methodenkörpern erforderlich. Beachten Sie, dass es wie bei Variablen möglich ist, Methoden und Klassen neu zu definieren. Um eine Liste der Methoden oder Klassen anzuzeigen, geben Sie /methods method oder /types an der JShell-Eingabeaufforderung ein.

Schnipsel bearbeiten

Die grundlegende Codeeinheit, die von JShell verwendet wird, ist das **Snippet** oder der **Quelleneintrag** . Jedes Mal, wenn Sie eine lokale Variable deklarieren oder eine lokale Methode oder Klasse definieren, erstellen Sie ein Snippet, dessen Name der Bezeichner der Variablen / Methode / Klasse ist. Sie können ein von Ihnen erstelltes Snippet jederzeit mit dem Befehl /edit . Zum Beispiel, sagen wir mal ich die Klasse erstellt haben Foo mit einem einzigen, Verfahren, bar :

```
jshell> class Foo {
   ...> void bar() {
   ...> }
   ...> }
```

Nun möchte ich den Körper meiner Methode ausfüllen. Anstatt die gesamte Klasse neu zu schreiben, kann ich sie bearbeiten:

```
jshell> /edit Foo
```

Standardmäßig wird ein Swing-Editor mit den grundlegendsten Funktionen angezeigt. Sie können jedoch den von JShell verwendeten Editor ändern:

```
jshell> /set editor emacs
jshell> /set editor vi
jshell> /set editor nano
jshell> /set editor -default
```

Wenn **die neue Version des Snippets Syntaxfehler enthält, wird sie möglicherweise nicht gespeichert**. Ebenso wird ein Snippet nur erstellt, wenn die ursprüngliche Deklaration / Definition syntaktisch korrekt ist. Folgendes funktioniert nicht:

```
jshell> String st = String 3
//error omitted
jshell> /edit st
|  No such snippet: st
```

Snippets können jedoch trotz bestimmter Fehler bei der Kompilierung, z. B. nicht übereinstimmenden Typen, kompiliert und somit bearbeitet werden. Dies funktioniert folgendermaßen:

```
jshell> int i = "hello"
//error omitted
jshell> /edit i
```

Schließlich können Ausschnitte mit dem Befehl /drop gelöscht werden:

```
jshell> int i = 13
jshell> /drop i
jshell> System.out.println(i)
|  Error:
|  cannot find symbol
|    symbol:   variable i
|  System.out.println(i)
|
```

Um alle Snippets zu löschen und dabei den Status der JVM zurückzusetzen, verwenden Sie \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
|  ^
```

JShell online lesen: https://riptutorial.com/de/java/topic/9511/jshell

Kapitel 88: JSON in Java

Einführung

JSON (JavaScript Object Notation) ist ein leichtes, textbasiertes, sprachunabhängiges Datenaustauschformat, das von Menschen und Maschinen leicht gelesen und geschrieben werden kann. JSON kann zwei strukturierte Typen darstellen: Objekte und Arrays. JSON wird häufig in Ajax-Anwendungen, -Konfigurationen, Datenbanken und RESTful-Webdiensten verwendet. Die Java-API für die JSON-Verarbeitung bietet portable APIs zum Analysieren, Generieren, Transformieren und Abfragen von JSON.

Bemerkungen

Dieses Beispiel konzentriert sich auf das Analysieren und Erstellen von JSON in Java mithilfe verschiedener Bibliotheken wie der **Google Gson-** Bibliothek, Jackson Object Mapper und anderen.

Beispiele für andere Bibliotheken finden Sie hier: So analysieren Sie JSON in Java

Examples

Daten als JSON kodieren

Wenn Sie ein JSONObject erstellen und Daten darin JSONObject , betrachten Sie das folgende Beispiel:

```
// 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"]
   }
*/
```

JSON-Daten dekodieren

Wenn Sie Daten von einem JSONObject , beachten Sie das folgende Beispiel:

```
String json =
"{\"foo\":\"bar\",\"temperature\":21.5,\"year\":2016,\"message\":{\"Hello\":\"world\"},\"months\":[\"Ja


// Decode the JSON-encoded string
JSONObject object = new JSONObject(json);

// Retrieve some values
String foo = object.getString("foo");
double temperature = object.getDouble("temperature");
int year = object.getInt("year");

// Retrieve another object
JSONObject secondary = object.getJSONObject("message");
String world = secondary.getString("Hello");

// Retrieve an array
JSONArray someMonths = object.getJSONArray("months");
// Get some values from the array
int nMonths = someMonths.length();
String february = someMonths.getString(1);
```

optXXX vs getXXX-Methoden

JSONObject und JSONArray verfügen über einige Methoden, die sehr nützlich sind, wenn es um die Möglichkeit geht, dass ein Wert, den Sie erhalten JSONArray , nicht existiert oder von einem anderen Typ ist.

```
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"
```

Die gleichen Regeln gelten für die getXXX / optXXX Methoden von JSONArray .

Objekt für JSON (Gson Library)

Nehmen wir an, Sie haben eine Klasse namens Person mit nur 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"}
```

Natürlich muss sich das Gson- Glas auf dem Klassenpfad befinden.

JSON in Objekt (Gson-Bibliothek)

Nehmen wir an, Sie haben eine Klasse namens Person mit nur 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
```

Sie müssen eine Gson-Bibliothek in Ihrem Klassenpfad haben.

Einzelne Elemente aus JSON extrahieren

```
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
```

Jackson Object Mapper verwenden

Pojo-Modell

```
public class Model {
    private String firstName;
    private String lastName;
    private int age;
    /* Getters and setters not shown for brevity */
}
```

Beispiel: String to Object

```
Model outputObject = objectMapper.readValue(
     "{\"firstName\":\"John\",\"lastName\":\"Doe\",\"age\":23}",
     Model.class);
System.out.println(outputObject.getFirstName());
//result: John
```

Beispiel: Object to String

```
String jsonString = objectMapper.writeValueAsString(inputObject));
//result: {"firstName":"John","lastName":"Doe","age":23}
```

Einzelheiten

Importanweisung erforderlich:

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

Maven-Abhängigkeit: Jackson-Datenbind

`ObjectMapper` Instanz

```
//creating one
ObjectMapper objectMapper = new ObjectMapper();
```

- ObjectMapper ist THREAD
- empfohlen: eine gemeinsam genutzte statische Instanz haben

Deserialisierung:

```
<T> T readValue(String content, Class<T> valueType)
```

- valueType muss angegeben werden - die Rückgabe wird von diesem Typ sein
- Wirft
    - IOException - bei einem E / A-Problem auf niedriger Ebene
    - JsonParseException - Wenn die zugrunde liegende Eingabe ungültigen Inhalt enthält
    - JsonMappingException - wenn die Eingabe-JSON-Struktur nicht mit der Objektstruktur übereinstimmt

Verwendungsbeispiel (jsonString ist die Eingabezeichenfolge):

```
Model fromJson = objectMapper.readValue(jsonString, Model.class);
```

Methode zur Serialisierung:

String writeValueAsString (Objektwert)

- Wirft
    - JsonProcessingException im Fehlerfall
    - Anmerkung: Vor Version 2.1 enthielt die Throws-Klausel IOException. 2.1 entfernt.

JSON-Iteration

Durchlaufen Sie die JSONObject Eigenschaften

```
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
}
```

Durchlaufen Sie die JSONArray Werte

```
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 - Verkettungsmethoden

Sie können die Methodenverkettung verwenden, während Sie mit JSONObject und JSONArray .

JSONObject-Beispiel

```
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

Wenn Sie eine Immobilie mit einem hinzufügen müssen null Wert, sollten Sie die vordefinierten statische Endverwendung JSONObject.NULL und nicht die Standard - Java null Referenz.

JSONObject.NULL ist ein Sentinel-Wert, mit dem eine Eigenschaft explizit mit einem leeren Wert definiert wird.

```
JSONObject obj = new JSONObject();
obj.put("some", JSONObject.NULL);   //Creates: {"some":null}
System.out.println(obj.get("some"));//prints: null
```

Hinweis

```
JSONObject.NULL.equals(null); //returns true
```

Welches ist eine **klare Verletzung** des Java.equals() Vertrags:

> Für jeden Referenzwert x, der nicht Null ist, sollte x.equals (null) den Wert false zurückgeben

JsonArray zu Java-Liste (Gson-Bibliothek)

Hier ist ein einfacher JsonArray, den Sie in eine Java ArrayList konvertieren möchten:

```
{
    "list": [
                "Test_String_1",
                "Test_String_2"
            ]
}
```

JsonArray nun die JsonArray -Liste an die folgende Methode, die eine entsprechende Java-ArrayList zurückgibt:

```
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;
}
```

Sie sollten Ihrer POM.xml Datei die folgende Maven-Abhängigkeit 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>
```

Oder Sie sollten den jar com.google.code.gson:gson:jar:<version> in Ihrem Klassenpfad haben.

Deserialisieren Sie die JSON-Sammlung in eine Sammlung von Objekten mit Jackson

Angenommen, Sie haben eine Pojo-Klasse Person

```
public class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Und Sie möchten es in ein JSON-Array oder eine Karte von Personenobjekten analysieren. Aufgrund der Typenlöschung können Sie zur Laufzeit keine Klassen aus List<Person> und Map<String, Person> direkt *erstellen (und daher zur Deserialisierung von JSON verwenden)* . Um diese Einschränkung zu überwinden, bietet jackson zwei Ansätze an - TypeFactory und TypeReference .

TypeFactory

Der hier verfolgte Ansatz besteht darin, eine Factory (und ihre statische Dienstprogrammfunktion) zu verwenden, um Ihren Typ für Sie zu erstellen. Die dafür benötigten Parameter sind die Sammlung, die Sie verwenden möchten (Liste, Satz usw.), und die Klasse, die Sie in dieser Sammlung speichern möchten.

TypReferenz

Der Typ-Referenz-Ansatz scheint einfacher zu sein, da er Ihnen weniger Schreibarbeit erspart und sauberer aussieht. TypeReference akzeptiert einen Typparameter, bei dem Sie den gewünschten Typ

List<Person> . Sie instanziieren dieses TypeReference-Objekt einfach und verwenden es als Typcontainer.

Sehen wir uns nun an, wie Sie Ihr JSON tatsächlich in ein Java-Objekt deserialisieren. Wenn Ihr JSON als Array formatiert ist, können Sie es als Liste deserialisieren. Wenn eine komplexere verschachtelte Struktur vorhanden ist, müssen Sie eine Deserialisierung für eine Map durchführen. Wir werden uns Beispiele für beide ansehen.

Deserialisierung des JSON-Arrays

```
String jsonString = "[{\"name\": \"Alice\"}, {\"name\": \"Bob\"}]"
```

TypeFactory-Ansatz

```
CollectionType listType =
    factory.constructCollectionType(List.class, Person.class);
List<Preson> list = mapper.readValue(jsonString, listType);
```

TypeReference-Ansatz

```
TypeReference<Person> listType = new TypeReference<List<Person>>() {};
List<Person> list = mapper.readValue(jsonString, listType);
```

Deserialisierung der JSON-Karte

```
String jsonString = "{\"0\": {\"name\": \"Alice\"}, \"1\": {\"name\": \"Bob\"}}"
```

TypeFactory-Ansatz

```
CollectionType mapType =
    factory.constructMapLikeType(Map.class, String.class, Person.class);
List<Person> list = mapper.readValue(jsonString, mapType);
```

TypeReference-Ansatz

```
TypeReference<Person> mapType = new TypeReference<Map<String, Person>>() {};
Map<String, Person> list = mapper.readValue(jsonString, mapType);
```

Einzelheiten

Import-Anweisung verwendet:

```
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.type.CollectionType;
```

Verwendete Instanzen:

```
ObjectMapper mapper = new ObjectMapper();
TypeFactory factory = mapper.getTypeFactory();
```

Hinweis

Während der TypeReference Ansatz möglicherweise besser aussieht, hat er mehrere Nachteile:

1. TypeReference sollte mithilfe einer anonymen Klasse instanziiert werden
2. Sie sollten generische Erklärung geben

Andernfalls kann das generische Argument verloren gehen, was zu einem Deserialisierungsfehler führen kann.

JSON in Java online lesen: https://riptutorial.com/de/java/topic/840/json-in-java

Kapitel 89: Just in Time (JIT) -Compiler

Bemerkungen

Geschichte

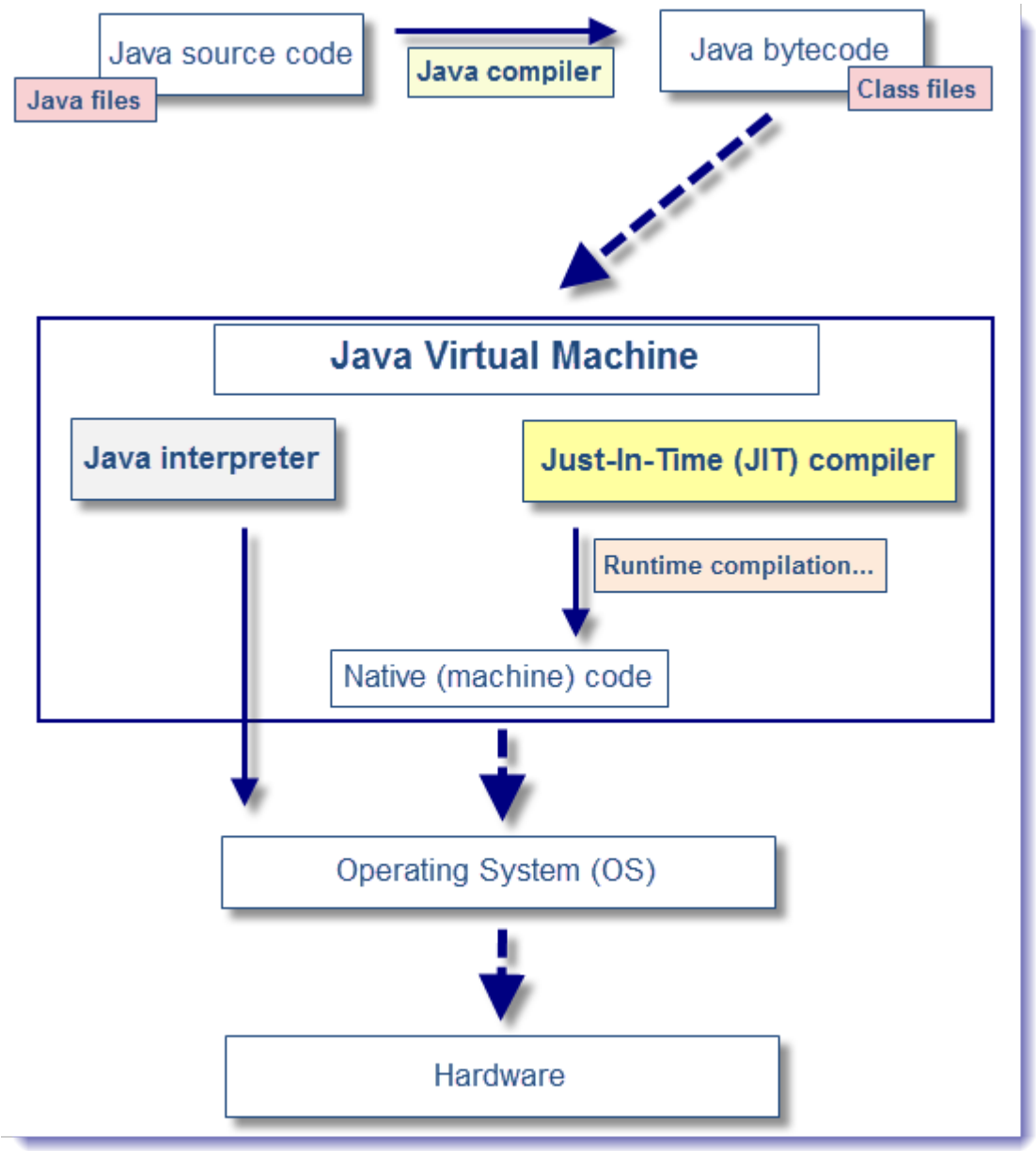
Der Symantec JIT-Compiler war ab 1.1.5 in Sun Java verfügbar, hatte jedoch Probleme.

Der Hotspot-JIT-Compiler wurde in Sun Java 1.2.2 als Plugin hinzugefügt. In Java 1.3 wurde JIT standardmäßig aktiviert.

(Quelle: Wann hat Java einen JIT-Compiler bekommen?

Examples

Überblick



Der Just-In-Time-Compiler (JIT) ist eine Komponente der Java-Laufzeitumgebung, die die Leistung von Java-Anwendungen zur Laufzeit verbessert.

- Java-Programme bestehen aus Klassen, die plattformneutrale Bytecodes enthalten, die von

einer JVM auf vielen verschiedenen Computerarchitekturen interpretiert werden können.
- Zur Laufzeit lädt die JVM die Klassendateien, bestimmt die Semantik jedes einzelnen Bytecodes und führt die entsprechende Berechnung aus.

  Die zusätzliche Prozessor- und Speicherverwendung während der Interpretation bedeutet, dass eine Java-Anwendung langsamer arbeitet als eine native Anwendung.

  Der JIT-Compiler verbessert die Leistung von Java-Programmen, indem er zur Laufzeit Bytecodes in nativen Maschinencode kompiliert.

Der JIT-Compiler ist standardmäßig aktiviert und wird beim Aufruf einer Java-Methode aktiviert. Der JIT-Compiler kompiliert die Bytecodes dieser Methode in nativen Maschinencode und kompiliert ihn "just in time" zur Ausführung.

Wenn eine Methode kompiliert wurde, ruft die JVM den kompilierten Code dieser Methode direkt auf, anstatt sie zu interpretieren. Wenn die Kompilierung keine Prozessorzeit und Speicherauslastung erfordert, kann theoretisch das Kompilieren jeder Methode die Geschwindigkeit des Java-Programms ermöglichen, sich der einer nativen Anwendung anzunähern.

Bei der JIT-Kompilierung sind Prozessorzeit und Speicherbedarf erforderlich. Beim ersten Start der JVM werden Tausende von Methoden aufgerufen. Das Kompilieren all dieser Methoden kann die Startzeit erheblich beeinflussen, auch wenn das Programm letztendlich eine sehr gute Spitzenleistung erreicht.

- In der Praxis werden Methoden beim ersten Aufruf nicht kompiliert. Für jede Methode unterhält die JVM einen call count der bei jedem call count der Methode inkrementiert wird.
- Die JVM interpretiert eine Methode so lange, bis ihre Aufrufzählung einen Schwellenwert für die JIT-Kompilierung überschreitet.
- Daher werden häufig verwendete Methoden bald nach dem Start der JVM kompiliert. Weniger verwendete Methoden werden viel später oder gar nicht kompiliert.
- Der Schwellenwert für die JIT-Kompilierung hilft der JVM, schnell zu starten und dennoch die Leistung zu verbessern.
- Die Schwelle wurde sorgfältig ausgewählt, um ein optimales Gleichgewicht zwischen Anlaufzeiten und langfristiger Leistung zu erzielen.
- Nachdem eine Methode kompiliert wurde, wird ihre Aufrufzählung auf null zurückgesetzt, und nachfolgende Aufrufe der Methode erhöhen weiterhin ihre Zählung.
- Wenn der Aufrufzähler einer Methode einen Schwellenwert für die JIT-Neukompilierung erreicht, kompiliert der JIT-Compiler dies ein zweites Mal und wendet eine größere Auswahl an Optimierungen an als bei der vorherigen Kompilierung.
- Dieser Vorgang wird wiederholt, bis der maximale Optimierungsgrad erreicht ist.

  Die aktivsten Methoden eines Java-Programms werden immer äußerst aggressiv optimiert, um die Leistungsvorteile des JIT-Compilers zu maximieren.

Der JIT-Compiler kann auch operational data at run time messen und diese Daten verwenden, um die Qualität weiterer Rekompilierungen zu verbessern.

  Der JIT-Compiler kann deaktiviert werden. In diesem Fall wird das gesamte Java-Programm interpretiert. Das Deaktivieren des JIT-Compilers wird nicht empfohlen, es sei denn, JIT-Kompilierungsprobleme werden diagnostiziert oder umgangen.

Just in Time (JIT) -Compiler online lesen: https://riptutorial.com/de/java/topic/5152/just-in-time--jit---compiler

Kapitel 90: JVM-Flags

Bemerkungen

Es wird dringend empfohlen, nur diese Optionen zu verwenden:

- Wenn Sie ein gründliches Verständnis Ihres Systems haben.
- Beachten Sie, dass sich diese Optionen bei unsachgemäßer Verwendung negativ auf die Stabilität oder Leistung Ihres Systems auswirken können.

Informationen aus der offiziellen Java-Dokumentation .

Examples

-XXaggressiv

-XXaggressive ist eine Sammlung von Konfigurationen, durch die die JVM mit hoher Geschwindigkeit arbeitet und so schnell wie möglich einen stabilen Zustand erreicht. Um dieses Ziel zu erreichen, verwendet die JVM beim Start mehr interne Ressourcen. Es bedarf jedoch einer weniger anpassungsfähigen Optimierung, sobald das Ziel erreicht ist. Es wird empfohlen, diese Option für langlebige, speicherintensive Anwendungen zu verwenden, die alleine funktionieren.

Verwendungszweck:

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

<param>	Beschreibung
opt	Planen Sie adaptive Optimierungen früher und aktivieren Sie neue Optimierungen, die in zukünftigen Releases als Standard gelten sollen.
memory	Konfiguriert das Speichersystem für speicherintensive Workloads und legt fest, dass große Speicherressourcen zur Verfügung stehen, um einen hohen Durchsatz zu gewährleisten. JRockit JVM verwendet, falls verfügbar, auch große Seiten.

-XXallocClearChunks

Mit dieser Option können Sie einen TLA für Referenzen und Werte zum Zeitpunkt der TLA-Zuweisung löschen und den nächsten Block vorabholen. Wenn eine Ganzzahl, eine Referenz oder etwas anderes deklariert wird, hat sie einen Standardwert von 0 oder Null (je nach Typ). Zu gegebener Zeit müssen Sie diese Referenzen und Werte löschen, um den Speicher auf dem Heap freizugeben, damit Java es verwenden oder wiederverwenden kann. Sie können dies entweder tun, wenn das Objekt zugewiesen wird, oder mithilfe dieser Option, wenn Sie einen neuen TLA anfordern.

Verwendungszweck:

```
-XXallocClearChunks
```

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

Das oben genannte ist eine boolesche Option und wird generell bei IA64-Systemen empfohlen. Letztendlich hängt die Verwendung von der Anwendung ab. Wenn Sie die Größe der gelöschten Chunks festlegen möchten, kombinieren Sie diese Option mit -XXallocClearChunkSize . Wenn Sie dieses Flag verwenden, aber keinen booleschen Wert angeben, ist der Standardwert true .

-XXallocClearChunkSize

Bei Verwendung mit -XXallocClearChunkSize legt diese Option die Größe der zu löschenden Chunks fest. Wenn dieses Flag verwendet wird, aber kein Wert angegeben wird, sind 512 Bytes voreingestellt.

Verwendungszweck:

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

-XXcallProfiling

Diese Option ermöglicht die Verwendung von Anrufprofilen für Codeoptimierungen. Das Profiling zeichnet nützliche Laufzeitstatistiken auf, die für die Anwendung spezifisch sind, und kann in vielen Fällen die Leistung erhöhen, da JVM diese Statistiken dann verwenden kann.

> Hinweis: Diese Option wird von der JRockit JVM R27.3.0 und späteren Version unterstützt. In zukünftigen Versionen kann es zur Standardeinstellung werden.

Verwendungszweck:

```
java -XXcallProfiling myApp
```

Diese Option ist standardmäßig deaktiviert. Sie müssen es aktivieren, um es verwenden zu können.

-XXdisableFatSpin

Diese Option deaktiviert den Fat Lock-Spin-Code in Java, sodass Threads, die den Versuch blockieren, eine Fat Lock zu erwerben, direkt in den Ruhezustand versetzt werden.

Objekte in Java werden zu einer Sperre, sobald ein Thread einen synchronisierten Block für dieses Objekt eingibt. Alle Schlösser werden gehalten (das heißt, sie bleiben gesperrt), bis sie vom Verriegelungsfaden freigegeben werden. Wenn die Sperre nicht sehr schnell aufgehoben wird, kann sie zu einer "fetten Sperre" aufgeblasen werden. "Spinning" tritt auf, wenn ein Thread, der eine bestimmte Sperre wünscht, diese Sperre fortlaufend überprüft, um festzustellen, ob sie noch belegt ist und in einer Spinnerei läuft enge Schleife, wie es die Prüfung macht. Das Drehen gegen einen Fettriegel ist im Allgemeinen vorteilhaft, obwohl dies in einigen Fällen teuer sein kann und die Leistung beeinträchtigen kann. -XXdisableFatSpin können Sie das Drehen gegen einen Fat Lock abschalten und den potenziellen Performance-Schlag beseitigen.

Verwendungszweck:

```
-XXdisableFatSpin
```

-XXdisableGCHeuristics

Diese Option deaktiviert die Änderungen der Speicherbereinigungsstrategie. Verdichtungsheuristiken und Größengrößenheuristiken sind von dieser Option nicht betroffen. Standardmäßig sind die Garbage Collection-Heuristiken aktiviert.

Verwendungszweck:

```
-XXdisableFatSpin
```

-XXdumpSize

Mit dieser Option wird eine Speicherabbilddatei generiert, in der Sie die relative Größe dieser Datei angeben können (dh klein, mittel oder groß).

Verwendungszweck:

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

<Größe>	Beschreibung
none	Erzeugt keine Dump-Datei.
small	Unter Windows wird eine kleine Speicherabbilddatei erstellt (unter Linux wird ein vollständiger Core-Speicherabzug erstellt). Ein kleiner Speicherauszug enthält nur die Threadstapel einschließlich ihrer Spuren und sehr wenig anderes. Dies war der Standard in der JRockit JVM 8.1 mit Service Packs 1 und 2 sowie 7.0 mit Service Pack 3 und höher.
normal	Bewirkt, dass auf allen Plattformen ein normaler Speicherauszug generiert wird. Diese Dump-Datei enthält den gesamten Speicher außer dem Java-Heap. Dies ist der Standardwert für die JRockit JVM 1.4.2 und höher.
large	Enthält alles, was sich im Speicher befindet, einschließlich des Java-Heapspeichers. Diese Option macht -XXdumpSize gleichbedeutend mit -XXdumpFullState .

-XXexitOnOutOfMemory

Diese Option bewirkt, dass JRockit JVM beim ersten Auftreten eines Speichermangels beendet wird. Es kann verwendet werden, wenn Sie lieber eine Instanz von JRockit JVM neu starten möchten, als wenn Sie aus Speicherfehler herausarbeiten. Geben Sie diesen Befehl beim Start ein, um die JRockit-JVM beim ersten Auftreten eines Speichermangels zu beenden.

Verwendungszweck:

```
-XXexitOnOutOfMemory
```

JVM-Flags online lesen: https://riptutorial.com/de/java/topic/2500/jvm-flags

Kapitel 91: JVM-Tool-Schnittstelle

Bemerkungen

JVM TM Tool-Schnittstelle

Version 1.2

http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

Examples

Objekte, die vom Objekt aus erreichbar sind, durchlaufen (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);
            }
        }
    }
}
```

Holen Sie sich eine JVMTI-Umgebung

Innerhalb der Agent_OnLoad-Methode:

```
jvmtiEnv* jvmti;
/* Get JVMTI environment */
vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
```

Beispiel für die Initialisierung in der Agent_OnLoad-Methode

```
/* 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;
}
```

JVM-Tool-Schnittstelle online lesen: https://riptutorial.com/de/java/topic/3316/jvm-tool-schnittstelle

Kapitel 92: Kalender und seine Unterklassen

Bemerkungen

Mit Java 8 wurden Calendar und seine Unterklassen durch das Paket java.time und seine Unterpakete ersetzt. Sie sollten bevorzugt werden, es sei denn, eine ältere API erfordert einen Kalender.

Examples

Kalenderobjekte erstellen

Calendar können mit getInstance() oder mit dem Konstruktor GregorianCalendar .

Es ist wichtig zu bemerken , dass Monate im Calendar sind Null basiert, was bedeutet , dass Januar durch ein dargestellt wird int Wert 0. Um einen besseren Code zu liefern, immer verwenden Calendar Konstanten, wie Calendar.JANUARY um Missverständnisse zu vermeiden.

```
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);
```

Anmerkung : Verwenden **Sie** immer die Monatskonstanten: Die numerische Darstellung ist irreführend , z. B. Calendar.JANUARY hat den Wert 0

Kalenderfelder vergrößern / verkleinern

add() und roll() können Calendar vergrößert / verkleinert werden.

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

Die add() Methode wirkt sich auf alle Felder aus und verhält sich effektiv, wenn tatsächliche Daten aus dem Kalender hinzugefügt oder subtrahiert werden

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

Durch die oben genannte Operation werden sechs Monate aus dem Kalender entfernt und wir kehren zum 30. September 2015 zurück.

Um ein bestimmtes Feld zu ändern, ohne die anderen Felder zu beeinflussen, verwenden Sie roll() .

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

Die obige Operation entfernt sechs Monate aus dem aktuellen *Monat* , sodass der Monat als September bezeichnet wird. Es wurden keine anderen Felder angepasst. Das Jahr hat sich bei dieser Operation nicht geändert.

Suche nach AM / PM

Mit der Kalender-Klasse können Sie AM oder PM leicht finden.

```
  Calendar cal = Calendar.getInstance();
  cal.setTime(new Date());
  if (cal.get(Calendar.AM_PM) == Calendar.PM)
```

```
      System.out.println("It is PM");
```

Kalender abziehen

Verwenden Sie die getTimeInMillis() Methode, um einen Unterschied zwischen zwei Calendar :

```
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)
```

Kalender und seine Unterklassen online lesen:
https://riptutorial.com/de/java/topic/165/kalender-und-seine-unterklassen

Kapitel 93: Karten

Einführung

Die java.util.Map-Schnittstelle stellt eine Zuordnung zwischen Schlüsseln und ihren Werten dar. Eine Map darf keine doppelten Schlüssel enthalten. und jeder Schlüssel kann höchstens einem Wert zugeordnet werden.

Da Map eine Schnittstelle ist, müssen Sie eine konkrete Implementierung dieser Schnittstelle instanziieren, um sie verwenden zu können. Es gibt mehrere Map Implementierungen, und meist werden java.util.HashMap und java.util.TreeMap

Bemerkungen

Eine *Karte* ist ein Objekt, das *Schlüssel* mit einem zugehörigen *Wert* für jeden Schlüssel speichert. Ein Schlüssel und sein Wert werden manchmal als *Schlüssel / Wert-Paar* oder als *Eintrag bezeichnet* . Karten bieten normalerweise diese Funktionen:

- Die Daten werden in Schlüssel / Wert-Paaren in der Karte gespeichert.
- Die Karte kann nur einen Eintrag für einen bestimmten Schlüssel enthalten. Wenn eine Map einen Eintrag mit einem bestimmten Schlüssel enthält und Sie versuchen, einen zweiten Eintrag mit demselben Schlüssel zu speichern, ersetzt der zweite Eintrag den ersten. Mit anderen Worten, dies ändert den mit der Taste verknüpften Wert.
- Maps bieten schnelle Vorgänge zum Testen, ob ein Schlüssel in der Map vorhanden ist, um den mit einem Schlüssel verknüpften Wert abzurufen und ein Schlüssel / Wert-Paar zu entfernen.

Die am häufigsten verwendete Kartenimplementierung ist HashMap . Es funktioniert gut mit Schlüsseln, die Zeichenketten oder Zahlen sind.

Einfache Karten wie HashMap sind nicht geordnet. Das Durchlaufen von Schlüssel / Wert-Paaren kann einzelne Einträge in beliebiger Reihenfolge zurückgeben. Wenn Sie Mapeinträge kontrolliert durchlaufen müssen, sollten Sie Folgendes beachten:

- Sortierte Karten wie TreeMap durchlaufen die Schlüssel in ihrer natürlichen Reihenfolge (oder in einer Reihenfolge, die Sie durch Angabe eines Comparators angeben können). Es würde beispielsweise erwartet, dass eine sortierte Karte, die Zahlen als Schlüssel verwendet, ihre Einträge in numerischer Reihenfolge durchläuft.

- LinkedHashMap ermöglicht das Durchlaufen von Einträgen in der Reihenfolge, in der sie in die Map eingefügt wurden, oder in der Reihenfolge der zuletzt verwendeten.

Examples

Fügen Sie ein Element hinzu

1. Zusatz

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
System.out.println(map.get(1));
```

Ausgabe: First element.

2. Überschreiben

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "First element.");
map.put(1, "New element.");
System.out.println(map.get(1));
```

Ausgabe: New element.

HashMap wird als Beispiel verwendet. Andere Implementierungen, die die Map Schnittstelle implementieren, können ebenfalls verwendet werden.

Fügen Sie mehrere Elemente hinzu

Wir können V put(K key,V value) :

> Ordnet den angegebenen Wert in dieser Map dem angegebenen Schlüssel zu (optionale Operation). Wenn die Zuordnung zuvor eine Zuordnung für den Schlüssel enthielt, wird der alte Wert durch den angegebenen Wert ersetzt.

```
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());
```

Ausgabe:

3

Um viele Elemente hinzuzufügen, können Sie eine innere Klasse wie folgt verwenden:

```
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");
}};
```

Beachten Sie, dass das Erstellen einer anonymen inneren Klasse nicht immer effizient ist und zu Speicherverlusten führen kann. Verwenden Sie daher, wenn möglich, einen Initialisierungsblock:

```
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");
}
```

Das obige Beispiel macht die Karte statisch. Es kann auch in einem nicht statischen Kontext verwendet werden, indem alle static Vorkommnisse entfernt werden.

Darüber hinaus unterstützen die meisten Implementierungen putAll , mit dem alle Einträge in

einer Map wie putAll werden können:

```
another.putAll(one);
```

Verwenden von Standardmethoden von Map aus Java 8

Beispiele für die Verwendung von in Java 8 in der Map-Schnittstelle eingeführten Standardmethoden

1. **GetOrDefault verwenden**

Gibt den dem Schlüssel zugeordneten Wert zurück. Wenn der Schlüssel nicht vorhanden ist, wird der Standardwert zurückgegeben

```
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. **ForEach verwenden**

Ermöglicht die Ausführung der in der "Aktion" für jeden Karteneintrag angegebenen Operation

```
 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. **ReplaceAll verwenden**

Wird nur durch einen neuen Wert ersetzt, wenn der Schlüssel vorhanden ist

```
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. **PutIfAbsent verwenden**

Das Schlüssel-Wert-Paar wird der Karte hinzugefügt, wenn der Schlüssel nicht vorhanden ist oder auf Null gestellt wird

```
   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. Mit **entfernen**

   Entfernt den Schlüssel nur, wenn er mit dem angegebenen Wert verknüpft ist

```
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. Verwenden Sie **ersetzen**

   Wenn der Schlüssel vorhanden ist, wird der Wert durch einen neuen Wert ersetzt. Wenn der Schlüssel nicht vorhanden ist, tut dies nichts.

```
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. **ComputeIfAbsent verwenden**

Diese Methode fügt einen Eintrag in der Map hinzu. Der Schlüssel wird in der Funktion angegeben und der Wert ist das Ergebnis der Anwendung der Mapping-Funktion

```
    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. **ComputeIfPresent verwenden**

Diese Methode fügt einen Eintrag hinzu oder ändert einen vorhandenen Eintrag in der Map. Macht nichts, wenn ein Eintrag mit diesem Schlüssel nicht vorhanden ist

```
    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. **Berechnen verwenden**

Diese Methode ersetzt den Wert eines Schlüssels durch den neu berechneten Wert

```
    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. **Merge verwenden**

Fügt das Schlüssel-Wert-Paar zur Karte hinzu, wenn der Schlüssel nicht vorhanden ist oder der Wert für den Schlüssel Null ist. Ersetzt den Wert durch den neu berechneten Wert. Wenn der Schlüssel vorhanden ist, wird der Schlüssel aus der Karte entfernt, wenn der berechnete Wert

Null ist

```
    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}
```

Karte löschen

```
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
```

Durchlaufen den Inhalt einer Map

Karten bieten Methoden, mit denen Sie auf die Schlüssel, Werte oder Schlüsselwertpaare der Karte als Sammlungen zugreifen können. Sie können diese Sammlungen durchlaufen. Zum Beispiel die folgende Karte:

```
Map<String, Integer> repMap = new HashMap<>();
repMap.put("Jon Skeet", 927_654);
repMap.put("BalusC", 708_826);
repMap.put("Darin Dimitrov", 715_567);
```

Iteration durch Kartenschlüssel:

```
for (String key : repMap.keySet()) {
    System.out.println(key);
}
```

Drucke:

```
Darin Dimitrov
Jon Skeet
BalusC
```

keySet() stellt die Schlüssel der Karte als Set bereit. Set wird verwendet, da die Schlüssel keine doppelten Werte enthalten können. Durch das Durchlaufen des Satzes erhält man jede Taste. HashMaps werden nicht bestellt, daher können die Schlüssel in diesem Beispiel in beliebiger Reihenfolge zurückgegeben werden.

Durchlaufen von Kartenwerten:

```
for (Integer value : repMap.values()) {
    System.out.println(value);
}
```

Drucke:

```
715567
927654
708826
```

values() gibt die Werte der Map als Collection . Das Durchlaufen der Sammlung führt zu jedem Wert. Die Werte können auch in beliebiger Reihenfolge zurückgegeben werden.

Durchlaufen von Schlüsseln und Werten zusammen

```
for (Map.Entry<String, Integer> entry : repMap.entrySet()) {
    System.out.printf("%s = %d\n", entry.getKey(), entry.getValue());
}
```

Drucke:

```
Darin Dimitrov = 715567
Jon Skeet = 927654
BalusC = 708826
```

entrySet() gibt eine Auflistung von Map.Entry Objekten zurück. Map.Entry ermöglicht den Zugriff auf den Schlüssel und den Wert für jeden Eintrag.

Karten zusammenführen, kombinieren und komponieren

Verwenden Sie putAll , um jedes Mitglied einer Karte in eine andere zu setzen. Schlüssel, die bereits in der Karte vorhanden sind, werden mit den entsprechenden Werten überschrieben.

```
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)
```

Dies ergibt die folgende Zuordnung in numbers :

```
"One" -> 1
"Two" -> 2
"Three" -> 4 //old value 3 was overwritten by new value 4
```

Wenn Sie Werte kombinieren möchten, anstatt sie zu überschreiben, können Sie Map.merge , das in Java 8 hinzugefügt wurde. In diesem Map.merge mithilfe einer vom Benutzer bereitgestellten BiFunction Werte für doppelte Schlüssel zusammengeführt. merge wirkt sich auf einzelne Schlüssel und Werte aus, daher müssen Sie eine Schleife oder Map.forEach . Hier verketten wir Strings für doppelte Schlüssel:

```
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));
```

Wenn Sie die Einschränkung erzwingen möchten, gibt es keine doppelten Schlüssel. Sie können eine Zusammenführungsfunktion verwenden, die einen AssertionError :

```
mapA.forEach((k, v) ->
    mapB.merge(k, v, (v1, v2) ->
        {throw new AssertionError("duplicate values for key: "+k);}));
```

Karte <X, Y> und Karte <Y, Z> erstellen, um Karte <X, Z> zu erhalten

Wenn Sie zwei Zuordnungen erstellen möchten, können Sie dies wie folgt tun

```
    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)));
```

Dies ergibt die folgende Abbildung

```
    "key1" -> 1.0
    "key2" -> 2.0
    "key3" -> 3.0
```

Überprüfen Sie, ob der Schlüssel vorhanden ist

```
Map<String, String> num = new HashMap<>();
num.put("one", "first");

if (num.containsKey("one")) {
    System.out.println(num.get("one")); // => first
}
```

Karten können Nullwerte enthalten

Bei Karten muss man vorsichtig sein, um "das Enthalten eines Schlüssels" nicht mit "einen Wert" zu verwechseln. Zum Beispiel können HashMap s null enthalten. Dies bedeutet, dass das folgende Verhalten völlig normal ist:

```
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
}
```

Formal gibt es keine Garantie, dass map.contains(key) <=> map.get(key)!=null

Karteneinträge effizient iterieren

Dieser Abschnitt enthält Code und Benchmarks für zehn eindeutige Beispielimplementierungen, die die Einträge einer Map<Integer, Integer> und die Summe der Integer Werte generieren. Alle Beispiele weisen eine algorithmische Komplexität von $\Theta(n)$. Die Benchmarks sind jedoch immer noch nützlich, um Einblick zu gewinnen, welche Implementierungen in einer "realen" Umgebung effizienter sind.

1. Implementierung mit **Iterator** mit **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. Implementierung mit **for** mit **Map.Entry**

```
for (Map.Entry<Integer, Integer> pair : map.entrySet()) {
    sum += pair.getKey() + pair.getValue();
}
```

3. Implementierung mit **Map.forEach** (Java 8+)

```
map.forEach((k, v) -> sum[0] += k + v);
```

4. Implementierung mit **Map.keySet** mit **for**

```
for (Integer key : map.keySet()) {
    sum += key + map.get(key);
}
```

5. Implementierung mit **Map.keySet** mit **Iterator**

```
Iterator<Integer> it = map.keySet().iterator();
while (it.hasNext()) {
    Integer key = it.next();
    sum += key + map.get(key);
}
```

6. Implementierung mit **for** mit **Iterator** und **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. Implementierung mit **Stream.forEach** (Java 8+)

```
map.entrySet().stream().forEach(e -> sum += e.getKey() + e.getValue());
```

8. Implementierung mit **Stream.forEach** mit **Stream.parallel** (Java 8+)

```
map.entrySet()
   .stream()
   .parallel()
```

```
    .forEach(e -> sum += e.getKey() + e.getValue());
```

9. Implementierung mit **IterableMap** aus Apache Collections

```
MapIterator<Integer, Integer> mit = iterableMap.mapIterator();
while (mit.hasNext()) {
    sum += mit.next() + it.getValue();
}
```

10. Implementierung mit **MutableMap** aus Eclipse Collections

```
mutableMap.forEachKeyValue((key, value) -> {
    sum += key + value;
});
```

Leistungstests (*Code auf Github verfügbar*)
Testumgebung: Windows 8.1 64-Bit, Intel i7-4790 3,60 GHz, 16 GB

1. Durchschnittliche Leistung von 10 Versuchen (100 Elemente) Best: 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. Durchschnittliche Leistung von 10 Versuchen (10000 Elemente) Best: 37.606 ± 0,790 µs / Op

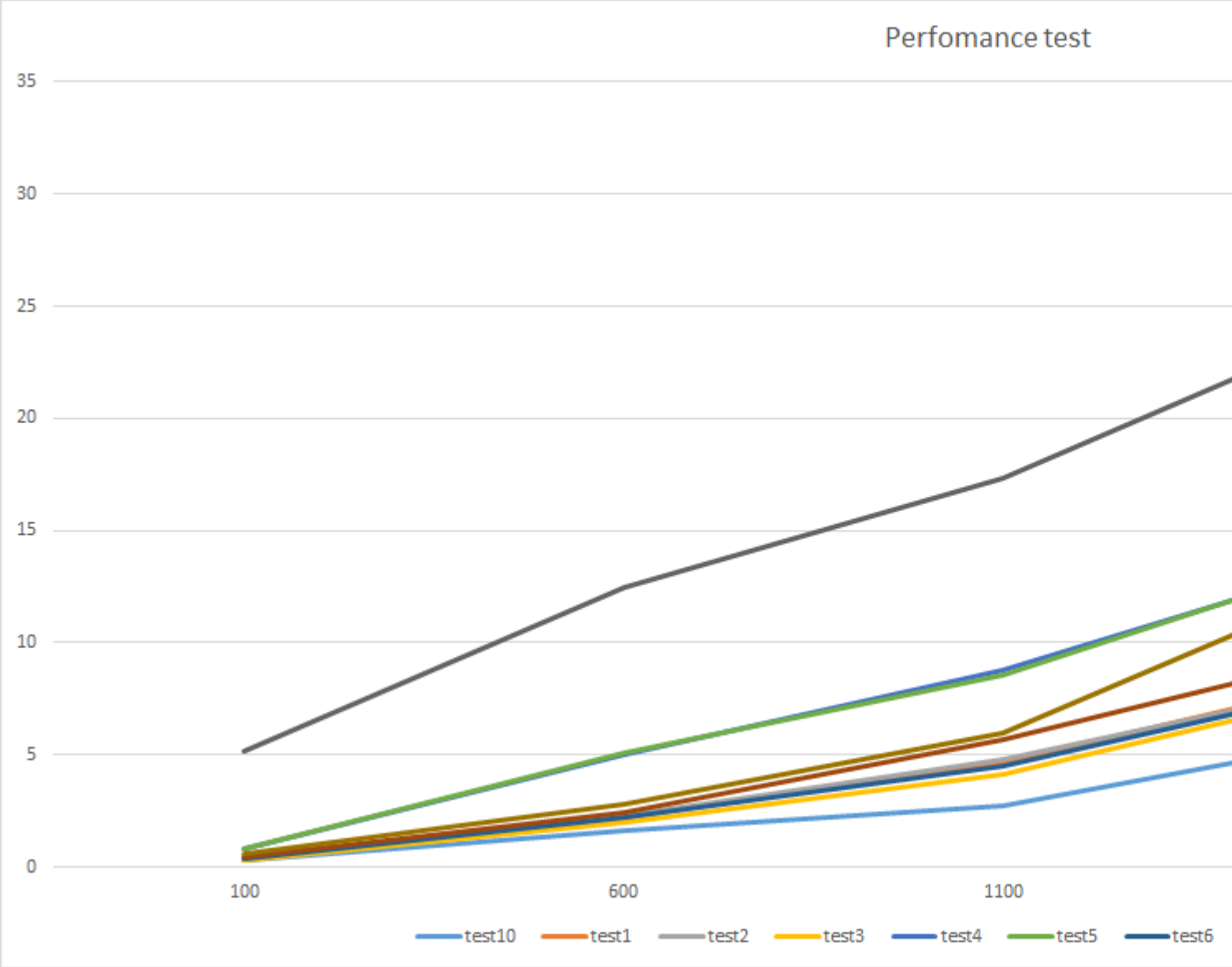
```
Benchmark                              Score     Error  Units
test10_UsingEclipseMutableMap          37606  ±    790  ns/op
test3_UsingForEachAndJava8             50368  ±    887  ns/op
test6_UsingForAndIterator              50332  ±    507  ns/op
test2_UsingForEachAndMapEntry          51406  ±   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. Durchschnittliche Leistung von 10 Versuchen (100000 Elemente) Best: 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. Ein Vergleich der Leistungsabweichungen in Bezug auf die Kartengröße




```
      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
```

Verwenden Sie ein benutzerdefiniertes Objekt als Schlüssel

Bevor Sie Ihr eigenes Objekt als Schlüssel verwenden, müssen Sie die Methode hashCode () und

equals () Ihres Objekts überschreiben.

Im einfachen Fall hätten Sie so etwas wie:

```
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 entscheidet, zu welchem Hash-Bucket der Schlüssel gehört, und equals entscheidet, welches Objekt in diesem Hash-Bucket liegt.

Ohne diese Methode wird die Referenz Ihres Objekts für den obigen Vergleich verwendet. Dies funktioniert nicht, wenn Sie nicht jedes Mal dieselbe Objektreferenz verwenden.

Verwendung von HashMap

HashMap ist eine Implementierung der Map-Schnittstelle, die eine Datenstruktur zum Speichern von Daten in Schlüssel-Wert-Paaren bereitstellt.

1. HashMap deklarieren

```
Map<KeyType, ValueType> myMap = new HashMap<KeyType, ValueType>();
```

KeyType und ValueType müssen gültige Typen in Java sein, wie – String, Integer, Float oder eine benutzerdefinierte Klasse wie Employee, Student usw.

Zum Beispiel: Map<String,Integer> myMap = new HashMap<String,Integer>();

2. Werte in HashMap setzen.

Um einen Wert in die HashMap einzufügen, müssen Sie die put Methode für das HashMap-Objekt aufrufen put indem Sie die Parameter Key und Value als Parameter übergeben.

```
myMap.put("key1", 1);
myMap.put("key2", 2);
```

Wenn Sie die put-Methode mit dem bereits in der Map vorhandenen Key aufrufen, überschreibt die Methode ihren Wert und gibt den alten Wert zurück.

3. Werte von HashMap abrufen.

Um den Wert aus einer HashMap zu erhalten, müssen Sie die get Methode aufrufen, indem Sie den Key als Parameter übergeben.

```
myMap.get("key1");    //return 1 (class Integer)
```

Wenn Sie einen Schlüssel übergeben, der nicht in der HashMap vorhanden ist, gibt diese Methode null

4. Prüfen Sie, ob sich der Schlüssel in der Karte befindet oder nicht.

```
myMap.containsKey(varKey);
```

5. Prüfen Sie, ob sich der Wert in der Karte befindet oder nicht.

```
myMap.containsValue(varValue);
```

Die obigen Methoden geben einen boolean Wert true oder false zurück, wenn key, value in der Map vorhanden ist oder nicht.

Karten erstellen und initialisieren

Einführung

Maps speichert Schlüssel / Wert-Paare, wobei jedem Schlüssel ein Wert zugeordnet ist. Bei einem bestimmten Schlüssel kann die Karte den zugehörigen Wert sehr schnell nachschlagen.

Maps , auch assoziiertes Array genannt, ist ein Objekt, in dem die Daten in Form von Schlüsseln und Werten gespeichert werden. In Java werden Karten mit der Map-Schnittstelle dargestellt, die keine Erweiterung der Erfassungsschnittstelle ist.

- Weg 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);
```

- Weg 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);
```

- Weg 3: -

```
 Map<String, Object> map = new HashMap<String, Object>(){{
     put("name", "A");
     put("address", "Malviya-Nagar");
     put("city", "Jaipur");
 }};
 System.out.println(map);
```

- Weg 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);
```

- Weg 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);
```

- Weg 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");
    }
```

- Möglichkeit 7: - Erstellen einer unveränderlichen Karte mit einem einzigen Schlüsselwert.

```
    //Immutable single key-value map
    Map<String, String> singletonMap = Collections.singletonMap("key", "value");
```

Bitte beachten Sie, dass **eine solche Karte nicht geändert werden kann** .

Jeder Versuch, die Karte zu ändern, löst die UnsupportedOperationException aus.

```
    //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
```

Karten online lesen: https://riptutorial.com/de/java/topic/105/karten

Kapitel 94: Klasse - Java Reflection

Einführung

Die Klasse java.lang.Class stellt viele Methoden bereit, mit denen Metadaten abgerufen, das Laufzeitverhalten einer Klasse untersucht und geändert werden können.

Die Pakete java.lang und java.lang.reflect stellen Klassen für die Java-Reflektion bereit.

Wo wird es verwendet?

Die Reflection-API wird hauptsächlich verwendet in:

IDE (Integrated Development Environment) zB Eclipse, MyEclipse, NetBeans usw. Debugger-Testtools usw.

Examples

getClass () - Methode der Object-Klasse

```
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);
    }
}
```

Klasse - Java Reflection online lesen: https://riptutorial.com/de/java/topic/10151/klasse---java-reflection

Kapitel 95: Klassen und Objekte

Einführung

Objekte haben Zustände und Verhalten. Beispiel: Ein Hund hat Zustände – Farbe, Name, Rasse sowie Verhalten – Schwanzwedeln, Bellen, Fressen. Ein Objekt ist eine Instanz einer Klasse.

Klasse – Eine Klasse kann als Vorlage / Plan definiert werden, die das Verhalten / den Zustand beschreibt, den das Objekt seines Typs unterstützt.

Syntax

- class Beispiel {} // Klassenschlüssel, Name, Hauptteil

Examples

Einfachste mögliche Klasse

```
class TrivialClass {}
```

Eine Klasse besteht mindestens aus dem Schlüsselwort class , einem Namen und einem Körper, der leer sein kann.

Sie instanziieren eine Klasse mit dem new Operator.

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

Objektmitglied vs statisches Mitglied

Mit dieser Klasse:

```
class ObjectMemberVsStaticMember {

    static int staticCounter = 0;
    int memberCounter = 0;

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

das folgende Code-Snippet:

```
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);
```

produziert diese Ausgabe:

```
o1 static counter 3
o1 member counter 1

o2 static counter 3
o2 member counter 2

ObjectMemberVsStaticMember.staticCounter = 3
```

Anmerkung: Sie sollten static Member nicht für Objekte, sondern für Klassen aufrufen. Während es für die JVM keinen Unterschied macht, werden menschliche Leser das zu schätzen wissen.

static Member sind Teil der Klasse und existieren nur einmal pro Klasse. Für Instanzen gibt es nicht static Member. Für jede Instanz gibt es eine unabhängige Kopie. Dies bedeutet auch, dass Sie Zugriff auf ein Objekt dieser Klasse benötigen, um auf seine Mitglieder zugreifen zu können.

Überladungsmethoden

Manchmal muss die gleiche Funktionalität für verschiedene Arten von Eingaben geschrieben werden. Zu diesem Zeitpunkt kann derselbe Methodenname mit einem anderen Parametersatz verwendet werden. Jeder unterschiedliche Parametersatz wird als Methodensignatur bezeichnet. Wie aus dem Beispiel hervorgeht, kann eine einzelne Methode mehrere Signaturen haben.

```
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"
    }
}
```

Der Vorteil ist, dass die gleiche Funktionalität mit zwei unterschiedlichen Eingangszahlen aufgerufen wird. Beim Aufrufen der Methode gemäß der übergebenen Eingabe (in diesem Fall entweder ein Stringwert oder zwei Stringwerte) wird die entsprechende Methode ausgeführt.

Methoden können überlastet sein:

1. Basierend auf der **Anzahl der übergebenen Parameter** .

   Beispiel: method(String s) und method(String s1, String s2) .

2. Basierend auf der **Reihenfolge der Parameter** .

   Beispiel: method(int i, float f) und method(float f, int i)) .

Hinweis: *Methoden können nicht durch* eine *Änderung nur den Rückgabetyp (überlastet werden int method() - String method() RuntimeException int method() gilt das gleiche wie String method() - String method() und einen werfen RuntimeException wenn versucht* wird *). Wenn Sie den Rückgabetyp ändern, müssen Sie auch die Parameter ändern, um überladen zu werden.*

Grundlegende Objektkonstruktion und Verwendung

Objekte kommen in eine eigene Klasse, ein einfaches Beispiel wäre ein Auto (detaillierte Erklärungen unten):

```
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;
    }

}
```

Objekte sind **Instanzen** ihrer Klasse. Die Art und Weise, wie Sie **ein Objekt erstellen** würden , wäre, die Car-Klasse auf **zwei Arten** in Ihrer Hauptklasse aufzurufen (Hauptmethode in Java oder onCreate in Android).

Option 1

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

Bei Option 1 teilen Sie dem Programm beim Erstellen des Objekts im Wesentlichen alles über das Auto mit. Wenn Sie eine Eigenschaft des Autos repaintCar , müssen Sie eine der Methoden aufrufen, z. B. die Methode repaintCar . Beispiel:

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

Anmerkung: Stellen Sie sicher, dass Sie den korrekten Datentyp an die Methode übergeben. Im obigen Beispiel können Sie auch eine Variable an die Methode repaintCar , **solange der Datentyp korrekt ist** .

Dies war ein Beispiel für das Ändern der Eigenschaften eines Objekts. Für das Empfangen von Eigenschaften eines Objekts müsste eine Methode aus der Klasse Car verwendet werden, die einen Rückgabewert enthält (dh eine Methode, die nicht void). Beispiel:

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

Option 1 ist die **beste** Option, wenn Sie zum Zeitpunkt der Erstellung über **alle Daten des Objekts** verfügen.

Option 2

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

Option 2 erzielt den gleichen Effekt, erfordert jedoch mehr Arbeit, um ein Objekt korrekt zu erstellen. Ich möchte mich an diesen Konstruktor in der Autoklasse erinnern:

```
public void Car(){
```

```
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }
```

Beachten Sie, dass Sie keine Parameter tatsächlich an das Objekt übergeben müssen, um es zu erstellen. Dies ist sehr nützlich, wenn Sie nicht alle Aspekte des Objekts haben, aber die Teile verwenden müssen, die Sie haben. Dadurch werden generische Daten in jede der Instanzvariablen des Objekts festgelegt, sodass, wenn Sie Daten anfordern, die nicht vorhanden sind, keine Fehler ausgegeben werden.

Hinweis: Vergessen Sie nicht, dass Sie später die Teile des Objekts festlegen müssen, mit denen Sie es nicht initialisiert haben. Zum Beispiel,

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

Dies ist ein häufiger Fehler bei Objekten, die nicht mit all ihren Daten initialisiert werden. Fehler wurden vermieden, da es einen Konstruktor gibt, der die Erstellung eines leeren Car-Objekts mit **Stand-In-Variablen ermöglicht** (`public Car(){}`), aber kein Teil des myCar wurde tatsächlich angepasst. **Korrektes Beispiel für das Erstellen eines Car-Objekts:**

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

Rufen Sie als Erinnerung die Eigenschaften eines Objekts ab, indem Sie eine Methode in Ihrer Hauptklasse aufrufen. Beispiel:

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

Konstrukteure

Konstruktoren sind spezielle Methoden, die nach der Klasse und ohne Rückgabetyp benannt werden, und werden zum Erstellen von Objekten verwendet. Konstruktoren können wie Methoden Eingabeparameter annehmen. Konstruktoren werden zum Initialisieren von Objekten verwendet. Abstrakte Klassen können auch Konstruktoren haben.

```
public class Hello{
    // constructor
    public Hello(String wordToPrint){
        printHello(wordToPrint);
    }
    public void printHello(String word){
        System.out.println(word);
    }
}
// instantiates the object during creating and prints out the content
// of wordToPrint
```

Es ist wichtig zu verstehen, dass Konstruktoren sich in mehrfacher Hinsicht von Methoden unterscheiden:

1. Konstruktoren können nur die Modifizierer public , private und protected und können nicht als abstract , final , static oder synchronized deklariert werden.

2. Konstruktoren haben keinen Rückgabetyp.

3. Konstruktoren MÜSSEN genauso benannt werden wie der Klassenname. Im Hello Beispiel Hello der Konstruktorname des Hello Objekts mit dem Klassennamen überein.

4. Das Schlüsselwort this hat eine zusätzliche Verwendung in Konstruktoren. this.method(...) ruft eine Methode in der aktuellen Instanz auf, während this(...) auf einen anderen Konstruktor in der aktuellen Klasse mit unterschiedlichen Signaturen verweist.

Konstruktoren können auch durch Vererbung mit dem Schlüsselwort super aufgerufen werden.

```
public class SuperManClass{

    public SuperManClass(){
        // some implementation
    }

    // ... methods
}


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

Siehe Java-Sprachspezifikation Nr. 8.8 und Nr. 15.9

Statische Endfelder werden mit einem statischen Initialisierer initialisiert

Um ein static final Endfeld zu initialisieren, für das mehr als ein einzelner Ausdruck erforderlich ist, kann der Wert mit einem static Initialisierer zugewiesen werden. Das folgende Beispiel initialisiert eine nicht veränderbare Menge von String s:

```
public class MyClass {

    public static final Set<String> WORDS;

    static {
        Set<String> set = new HashSet<>();
        set.add("Hello");
        set.add("World");
        set.add("foo");
        set.add("bar");
        set.add("42");
        WORDS = Collections.unmodifiableSet(set);
    }
}
```

Erklären, was Methodenüberladen und Überschreiben ist.

Überschreiben und Überladen von Methoden sind zwei von Java unterstützte Formen von Polymorphismus.

Methodenüberladung

Überladen von Methoden (auch als statischer Polymorphismus bezeichnet) ist eine Möglichkeit, zwei (oder mehr) Methoden (Funktionen) mit demselben Namen in einer einzigen Klasse zu haben. Ja, so einfach ist das.

```
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;
    }
}
```

Auf diese Weise kann der Benutzer dieselbe Methode für den Bereich aufrufen, abhängig von der Art der Form, die er hat.

Aber die eigentliche Frage ist nun, wie der Java-Compiler unterscheidet, welcher Methodenkörper ausgeführt werden soll.

Nun, Java hat deutlich gemacht, dass die **Methodennamen** (area() in unserem Fall) **zwar gleich sein können, die Argumente jedoch anders sind.**

> Überladene Methoden müssen eine unterschiedliche Argumentliste haben (Anzahl und Typen).

Allerdings können wir keine andere Methode hinzufügen, um die Fläche eines Quadrats wie folgt zu berechnen: public Double area(Long side) da in diesem Fall die Flächenmethode des Kreises in Konflikt gerät und **Mehrdeutigkeiten** für den Java-Compiler auftreten.

Gott sei Dank, gibt es einige Entspannungen beim Schreiben überladener Methoden wie

> Kann unterschiedliche Rückgabetypen haben.
>
> Kann unterschiedliche Zugriffsmodifizierer haben.
>
> Kann verschiedene Ausnahmen werfen.

Warum heißt das statischer Polymorphismus?

Das liegt daran, dass die überladene Methode zur Kompilierzeit festgelegt wird, basierend auf der tatsächlichen Anzahl der Argumente und den Kompilierzeittypen der Argumente.

> Einer der häufigsten Gründe für das Überladen von Methoden ist die Einfachheit des bereitgestellten Codes. Erinnern Sie sich beispielsweise an String.valueOf() das fast alle Arten von Argumenten String.valueOf() ? Was hinter der Szene steht, ist wahrscheinlich so:

```
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)
```

Überschreiben der Methode

Nun, das Überschreiben von Methoden (Ja, Sie denken, es wird auch als dynamischer Polymorphismus bezeichnet) ist ein etwas interessanteres und komplexeres Thema.

Beim Überschreiben der Methode überschreiben wir den von der übergeordneten Klasse bereitgestellten Methodenkörper. Ich habs? Nein? Gehen wir ein Beispiel durch.

```
public abstract class Shape{

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

Wir haben also eine Klasse namens Shape und eine Methode namens area, die wahrscheinlich den Bereich der Form zurückgibt.

Nehmen wir an, wir haben jetzt zwei Klassen mit dem Namen Circle und 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;
    }
}
```

Ähnlich Rechteckklasse:

```
 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;
    }
}
```

Jetzt haben beide Kind-Klassen den von der Eltern-Klasse (Shape) bereitgestellten Methodenkörper aktualisiert. Nun ist die Frage, wie das Ergebnis zu sehen ist. Nun, machen wir es auf die alte psvm Weise.

```
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());
```

```
    }
}
```

Beeindruckend! ist es nicht toll? Zwei Objekte desselben Typs, die dieselben Methoden aufrufen und unterschiedliche Werte zurückgeben. Mein Freund, das ist die Kraft des dynamischen Polymorphismus.

Hier ist ein Diagramm zum besseren Vergleich der Unterschiede zwischen diesen beiden: -

Methodenüberladung	**Überschreiben der Methode**
Methodenüberladung wird verwendet, um die Lesbarkeit des Programms zu erhöhen.	Das Überschreiben von Methoden wird verwendet, um die spezifische Implementierung der Methode bereitzustellen, die bereits von ihrer Oberklasse bereitgestellt wird.
Das Überladen der Methode wird innerhalb der Klasse durchgeführt.	Das Überschreiben von Methoden tritt in zwei Klassen auf, die eine IS-A-Beziehung (Vererbung) haben.
Bei Überladung der Methode müssen die Parameter unterschiedlich sein.	Im Falle eines Überschreibens der Methode muss der Parameter derselbe sein.
Methodenüberladung ist das Beispiel für die Kompilierzeit-Polymorphie.	Das Überschreiben von Methoden ist das Beispiel für Laufzeitpolymorphismus.
In Java kann das Überladen von Methoden nicht durchgeführt werden, indem nur der Rückgabetyp der Methode geändert wird. Der Rückgabetyp kann in der Methodenüberladung gleich oder verschieden sein. Sie müssen jedoch den Parameter ändern.	Der Rückgabetyp muss beim Überschreiben der Methode gleich oder kovariant sein.

Klassen und Objekte online lesen: https://riptutorial.com/de/java/topic/114/klassen-und-objekte

Kapitel 96: Klassenlader

Bemerkungen

Ein Classloader ist eine Klasse, deren Hauptzweck darin besteht, die Position und das Laden der von einer Anwendung verwendeten Klassen zu vermitteln. Ein Klassenlader kann auch *Ressourcen* finden und laden.

Die Standard-Classloader-Klassen können Klassen und Ressourcen aus Verzeichnissen im Dateisystem und aus JAR- und ZIP-Dateien laden. Sie können JAR- und ZIP-Dateien auch von einem Remote-Server herunterladen und cachen.

Klassenladeprogramme werden normalerweise verkettet, sodass die JVM versucht, Klassen aus den Standardklassenbibliotheken zu bevorzugen, die von der Anwendung bereitgestellten Quellen bevorzugt werden. Benutzerdefinierte Klassenladeprogramme ermöglichen es dem Programmierer, dies zu ändern. Sie können auch Dinge wie das Entschlüsseln von Bytecode-Dateien und die Bytecode-Änderung durchführen.

Examples

Instantiieren und Verwenden eines Klassenladers

Dieses grundlegende Beispiel zeigt, wie eine Anwendung einen Classloader instanziieren und verwenden kann, um eine Klasse dynamisch zu laden.

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

Der in diesem Beispiel erstellte Klassenlader hat den Standardklassenlader als übergeordnetes Element und versucht zuerst, eine Klasse im übergeordneten Klassenlader zu finden, bevor er in "extra.jar" sucht. Wenn die angeforderte Klasse bereits geladen wurde, gibt der findClass Aufruf den Verweis auf die zuvor geladene Klasse zurück.

Der findClass Aufruf kann auf verschiedene Weise fehlschlagen. Die häufigsten sind:

- Wenn die benannte Klasse nicht gefunden werden kann, wird der Aufruf mit ClassNotFoundException .
- Wenn die benannte Klasse von einer anderen Klasse abhängt, die nicht gefunden werden kann, gibt der Aufruf NoClassDefFoundError .

Implementieren eines benutzerdefinierten classLoader

Jeder benutzerdefinierte Loader muss die Klasse java.lang.ClassLoader direkt oder indirekt erweitern. Die wichtigsten *Erweiterungspunkte* sind die folgenden Methoden:

- findClass(String) - Überladen Sie diese Methode, wenn Ihr Classloader dem Standard-Delegierungsmodell zum Laden von Klassen folgt.
- loadClass(String, boolean) - Überladen Sie diese Methode, um ein alternatives Delegierungsmodell zu implementieren.
- findResource und findResources - Überladen Sie diese Methoden, um das Laden von Ressourcen anzupassen.

Die defineClass Methoden, die für das tatsächliche Laden der Klasse aus einem Byte-Array verantwortlich sind, sind final , um ein Überladen zu verhindern. Jedes benutzerdefinierte Verhalten muss vor dem Aufrufen von defineClass .

Hier ist eine einfache Methode, die eine bestimmte Klasse aus einem Byte-Array lädt:

```
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);
        }
    }
}
```

Da wir die findClass Methode nur überschrieben haben, findClass dieses benutzerdefinierte Klassenladeprogramm beim loadClass wie folgt.

1. Die loadClass Methode des loadClass ruft findLoadedClass auf, um findLoadedClass , ob eine Klasse mit diesem Namen bereits von diesem Klassenladeprogramm geladen wurde. Wenn dies erfolgreich ist, wird das resultierende Class Objekt an den Anforderer zurückgegeben.
2. Die loadClass Methode wird dann an den übergeordneten Classloader delegiert, indem ihr loadClass Aufruf aufgerufen wird. Wenn die Eltern mit der Bitte umgehen kann, wird es eine Rückkehr Class Objekt , das wird dann an den Anforderer zurückgegeben.
3. Wenn das übergeordnete Klassenladeprogramm die Klasse nicht laden kann, ruft findClass unsere überschreibbare findClass Methode auf und übergibt den Namen der zu ladenden Klasse.
4. Wenn der angeforderte Name mit this.classname , rufen wir defineClass auf, um die tatsächliche Klasse aus dem this.classfile defineClass zu laden. Das resultierende Class Objekt wird dann zurückgegeben.
5. Wenn der Name nicht übereinstimmt, wird ClassNotFoundException .

Laden einer externen .class-Datei

Um eine Klasse zu laden, müssen wir sie zuerst definieren. Die Klasse wird vom ClassLoader definiert. Es gibt nur ein Problem: Oracle hat den Code des ClassLoader nicht mit dieser verfügbaren Funktion geschrieben. Um die Klasse zu definieren, müssen wir auf eine Methode namens defineClass() die eine private Methode des ClassLoader .

Um darauf zuzugreifen, erstellen wir eine neue Klasse, ByteClassLoader , und erweitern sie auf ClassLoader . Nun , da wir unsere Klasse erweitert ClassLoader , können wir den Zugriff auf ClassLoader ,s private Methoden. Um defineClass() verfügbar zu machen, erstellen wir eine neue Methode, die als Spiegel für die private defineClass() -Methode fungiert. Die private Methode nennen wir den Klassennamen, müssen name , die Klasse Bytes, classBytes , das erste Byte des Offset, der sein wird , 0 , weil classBytes 'Daten beginnt bei classBytes[0] , und das letzte Byte des Offset, der sein classBytes.lenght weil es die Größe der Daten darstellt, die der letzte Versatz ist.

```
public class ByteClassLoader extends ClassLoader {

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

}
```

Jetzt haben wir eine öffentliche defineClass() -Methode. Es kann aufgerufen werden, indem der Name der Klasse und die Klassenbytes als Argumente übergeben werden.

Nehmen wir an, wir haben die Klasse MyClass im Paket stackoverflow ...

Um die Methode aufzurufen, benötigen wir die Klassenbytes, sodass wir ein Path Objekt erstellen, das den Path unserer Klasse darstellt. Paths.get() verwenden Sie die Paths.get() Methode und übergeben den Pfad der binären Klasse als Argument. Nun können wir die Klassenbytes mit Files.readAllBytes(path) . Also erstellen wir eine ByteClassLoader Instanz und verwenden die von uns erstellte Methode defineClass() . Wir haben bereits die Klassenbytes, aber zum Aufruf unserer Methode benötigen wir auch den Klassennamen, der durch den Paketnamen (Punkt) und den kanonischen Klassennamen angegeben wird, in diesem Fall stackoverflow.MyClass .

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

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

Anmerkung : Die defineClass() -Methode gibt ein Class<?> Objekt zurück. Sie können es speichern, wenn Sie möchten.

Um die Klasse zu laden, rufen wir einfach loadClass() und übergeben den Klassennamen. Diese Methode kann eine ClassNotFoundException daher müssen wir einen try-Cath-Block verwenden

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

Klassenlader online lesen: https://riptutorial.com/de/java/topic/5443/klassenlader

Kapitel 97: Konsolen-E / A

Examples

Benutzereingaben von der Konsole lesen

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());
}
```

Die folgenden Importe werden für diesen Code benötigt:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

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 + "!");
```

Der folgende Import wird für dieses Beispiel benötigt:

```
import java.util.Scanner;
```

Um mehr als eine Zeile zu lesen, rufen Sie scanner.nextLine() wiederholt auf:

```
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 + "!");
```

Es gibt zwei Methoden, um Strings , next() und nextLine() . next() gibt den Text bis zum ersten Leerzeichen zurück (auch als "Token" bezeichnet), und nextLine() gibt den gesamten vom Benutzer eingegebenen Text zurück, bis er die Eingabetaste drückt.

Scanner bietet auch Dienstprogrammmethoden zum Lesen anderer Datentypen als String . Diese schließen ein:

```
scanner.nextByte();
scanner.nextShort();
scanner.nextInt();
scanner.nextLong();
scanner.nextFloat();
scanner.nextDouble();
scanner.nextBigInteger();
scanner.nextBigDecimal();
```

Das Präfixieren einer dieser Methoden mit has (wie in hasNextLine() , hasNextInt()) gibt true zurück true wenn der Stream weitere hasNextInt() . Hinweis: Diese Methoden führen zum Absturz des Programms, wenn die Eingabe nicht vom angeforderten Typ ist (z. B. "a" für nextInt()). Sie können einen try {} catch() {} , um dies zu verhindern (siehe: Ausnahmen).

```
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");
}
```

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();
```

Vorteile :

- Lesemethoden werden synchronisiert
- Formatstringsyntax kann verwendet werden

Hinweis : Dies funktioniert nur, wenn das Programm von einer echten Befehlszeile aus ausgeführt wird, ohne die Standardeingabe- und -ausgabeströme umzuleiten. Es funktioniert nicht, wenn das Programm innerhalb bestimmter IDEs ausgeführt wird, z. B. Eclipse. In den anderen Beispielen finden Sie Code, der in IDEs und mit Stream-Umleitung funktioniert.

Grundlegendes Befehlszeilenverhalten implementieren

Für grundlegende Prototypen oder grundlegendes Befehlszeilenverhalten ist die folgende Schleife hilfreich.

```
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");
    }
}
```

Zeichenketten in der Konsole ausrichten

Die Methode PrintWriter.format (aufgerufen über System.out.format) kann verwendet werden, um ausgerichtete Zeichenfolgen in der Konsole zu drucken. Die Methode erhält einen String mit den Formatinformationen und einer Reihe von zu formatierenden Objekten:

```
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();
}
```

Ausgabe:

```
1   1          1
1234 1234    1234
1234567 1234567 123456
123456789 12345678 123456
```

Die Verwendung von Formatzeichenfolgen mit fester Größe ermöglicht das Drucken der Zeichenfolgen in Tabellenform mit Spalten mit fester Größe:

```
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();
}
```

Ausgabe:

```
1   1             1
123 1234       1234
123 1234567  123456
123 12345678 123456
```

Beispiele für Formatstrings

- %s : Nur eine Zeichenfolge ohne Formatierung
- %5s : Formatieren Sie die Zeichenfolge mit **mindestens** 5 Zeichen. Wenn die Zeichenfolge kürzer ist, wird sie auf 5 Zeichen **aufgefüllt** und nach **rechts** ausgerichtet
- %-5s : Formatieren Sie die Zeichenfolge mit **mindestens** 5 Zeichen. Wenn die Zeichenfolge kürzer ist, wird sie auf 5 Zeichen **aufgefüllt** und **links** ausgerichtet
- %5.10s : Formatieren Sie die Zeichenfolge mit **mindestens** 5 und **maximal** 10 Zeichen. Ist die Zeichenfolge kürzer als 5, wird sie auf 5 Zeichen **aufgefüllt** und nach **rechts** ausgerichtet. Wenn die Zeichenfolge länger als 10 ist, wird sie auf 10 Zeichen **gekürzt** und **rechtsbündig** ausgerichtet
- %-5.5s : Formatieren Sie die Zeichenfolge mit einer **festen** Größe von 5 Zeichen (Minimum und Maximum sind gleich). Ist die Zeichenfolge kürzer als 5, wird sie auf 5 Zeichen **aufgefüllt** und **linksbündig** ausgerichtet. Wenn der String länger als 5 ist, wird er auf 5 Zeichen **gekürzt** und **linksbündig** ausgerichtet

Konsolen-E / A online lesen: https://riptutorial.com/de/java/topic/126/konsolen-e---a

Kapitel 98: Konstrukteure

Einführung

Konstruktoren in Java sind zwar nicht erforderlich, aber vom Compiler erkannte Methoden, um bestimmte Werte für die Klasse zu instanziieren, die für die Rolle des Objekts wesentlich sein können. In diesem Thema wird die ordnungsgemäße Verwendung von Java-Klassenkonstruktoren veranschaulicht.

Bemerkungen

In der Java-Sprachspezifikation wird ausführlich über die genaue Natur der Konstruktorsemantik gesprochen. Sie können in JLS §8.8 gefunden werden

Examples

Standardkonstruktor

Der "Standard" für Konstruktoren ist, dass sie keine Argumente haben. Wenn Sie **keinen** Konstruktor angeben, generiert der Compiler einen Standardkonstruktor für Sie.
Dies bedeutet, dass die folgenden zwei Ausschnitte semantisch äquivalent sind:

```
public class TestClass {
    private String test;
}
```

```
public class TestClass {
    private String test;
    public TestClass() {

    }
}
```

Die Sichtbarkeit des Standardkonstruktors entspricht der Sichtbarkeit der Klasse. Daher hat eine Klasse, die als paket-privat definiert wurde, einen paket-privaten Standardkonstruktor

Wenn Sie jedoch keinen Standardkonstruktor verwenden, generiert der Compiler keinen Standardkonstruktor für Sie. Also sind diese nicht gleichwertig:

```
public class TestClass {
    private String test;
    public TestClass(String arg) {
    }
}
```

```
public class TestClass {
    private String test;
    public TestClass() {
    }
    public TestClass(String arg) {
    }
}
```

Beachten Sie, dass der generierte Konstruktor keine nicht standardisierte Initialisierung durchführt. Das bedeutet, dass alle Felder Ihrer Klasse ihren Standardwert haben, sofern sie keinen Initialisierer haben.

```
public class TestClass {

    private String testData;

    public TestClass() {
        testData = "Test"
    }
}
```

Konstruktoren werden so genannt:

```
TestClass testClass = new TestClass();
```

Konstruktor mit Argumenten

Konstruktoren können mit beliebigen Argumenten erstellt werden.

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
        this.testData = testData;
    }
}
```

So genannt:

```
TestClass testClass = new TestClass("Test Data");
```

Eine Klasse kann mehrere Konstruktoren mit unterschiedlichen Signaturen haben. Um Konstruktoraufrufe zu verketten (rufen Sie beim Instanziieren einen anderen Konstruktor derselben Klasse an), verwenden Sie this() .

```
public class TestClass {

    private String testData;

    public TestClass(String testData) {
        this.testData = testData;
    }

    public TestClass() {
        this("Test"); // testData defaults to "Test"
    }
}
```

So genannt:

```
TestClass testClass1 = new TestClass("Test Data");
TestClass testClass2 = new TestClass();
```

Rufen Sie den übergeordneten Konstruktor auf

Angenommen, Sie haben eine Elternklasse und eine Kindklasse. Um eine Child-Instanz zu erstellen, muss immer ein Parent-Konstruktor beim ersten Start des Child-Konstruktors ausgeführt werden. Wir können den gewünschten super(...) Konstruktor auswählen, indem wir super(...) mit den entsprechenden Argumenten explizit als unsere erste Child-Konstruktoranweisung aufrufen. Dadurch

sparen Sie Zeit, indem Sie den Konstruktor der übergeordneten Klassen wiederverwenden, anstatt den gleichen Code im Konstruktor der untergeordneten Klassen neu zu schreiben.

Ohne super(...) **Methode:**

(implizit wird die no-args-Version super() als unsichtbar bezeichnet.)

```
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;
    }
}
```

Mit super() **Methode:**

```
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
    }
}
```

Anmerkung: Aufrufe an einen anderen Konstruktor (Verkettung) oder den Super-Konstruktor **MÜSSEN** die erste Anweisung innerhalb des Konstruktors sein.

Wenn Sie den super(...) explizit aufrufen, muss ein übereinstimmender übergeordneter Konstruktor vorhanden sein (das ist unkompliziert, nicht wahr?).

Wenn Sie keinen super(...) explizit aufrufen, muss Ihre übergeordnete Klasse über einen no-args-Konstruktor verfügen. Dieser kann entweder explizit geschrieben oder vom Compiler als Standardwert erstellt werden, wenn die übergeordnete Klasse keine Angaben macht irgendein Konstruktor.

```
class Parent{
    public Parent(String tName, int tAge) {}
}
```

```
class Child extends Parent{
    public Child(){}
}
```

Die Klasse Parent hat keinen Standardkonstruktor, daher kann der Compiler nicht super im Child-Konstruktor hinzufügen. Dieser Code wird nicht kompiliert. Sie müssen die Konstruktoren ändern, um auf beide Seiten zu passen, oder Ihren eigenen super Aufruf schreiben, wie folgt:

```
class Child extends Parent{
    public Child(){
          super("",0);
    }
}
```

Konstrukteure online lesen: https://riptutorial.com/de/java/topic/682/konstrukteure

Kapitel 99: Konvertieren in und aus Strings

Examples

Konvertieren anderer Datentypen in String

- Sie können den Wert anderer primitiver Datentypen als String valueOf indem Sie eine der valueOf Methoden der String-Klasse verwenden.

  Zum Beispiel:

  ```
  int i = 42;
  String string = String.valueOf(i);
  //string now equals "42”.
  ```

  Diese Methode ist auch für andere Datentypen wie float , double , boolean und sogar Object überladen.

- Sie können auch jedes andere Objekt (jede Instanz einer beliebigen Klasse) als String erhalten, indem Sie .toString aufrufen. Damit dies eine nützliche Ausgabe ergibt, muss die Klasse toString() überschreiben. Die meisten Standard-Java-Bibliotheksklassen wie Date und andere.

  Zum Beispiel:

  ```
  Foo foo = new Foo(); //Any class.
  String stringifiedFoo = foo.toString().
  ```

  Hier enthält stringifiedFoo eine Darstellung von foo als String.

Sie können auch einen beliebigen Zahlentyp in eine kurze Notation wie folgt konvertieren.

```
int i = 10;
String str = i + "";
```

Oder einfach auf einfache Weise

```
String str = 10 + "";
```

Umwandlung in / von Bytes

Um eine Zeichenfolge in ein Byte-Array zu kodieren, können Sie einfach die Methode String#getBytes() mit einem der Standardzeichensätze verwenden, die in jeder Java-Laufzeitumgebung verfügbar sind:

```
byte[] bytes = "test".getBytes(StandardCharsets.UTF_8);
```

und zu entschlüsseln:

```
String testString = new String(bytes, StandardCharsets.UTF_8);
```

Sie können den Aufruf weiter vereinfachen, indem Sie einen statischen Import verwenden:

```
import static java.nio.charset.StandardCharsets.UTF_8;
...
byte[] bytes = "test".getBytes(UTF_8);
```

Bei weniger gebräuchlichen Zeichensätzen können Sie den Zeichensatz mit einer Zeichenfolge angeben:

```
byte[] bytes = "test".getBytes("UTF-8");
```

und umgekehrt:

```
String testString = new String (bytes, "UTF-8");
```

Dies bedeutet jedoch, dass Sie die geprüfte UnsupportedCharsetException .

Der folgende Aufruf verwendet den Standardzeichensatz. Der Standardzeichensatz ist plattformspezifisch und unterscheidet sich im Allgemeinen zwischen Windows-, Mac- und Linux-Plattformen.

```
byte[] bytes = "test".getBytes();
```

und umgekehrt:

```
String testString = new String(bytes);
```

Beachten Sie, dass ungültige Zeichen und Bytes durch diese Methoden ersetzt oder übersprungen werden können. Für mehr Kontrolle - zum Beispiel zur Überprüfung der Eingabe - sollten Sie die Klassen CharsetEncoder und CharsetDecoder verwenden.

Base64-Kodierung / Dekodierung

Gelegentlich müssen Sie binäre Daten als *Base64-* kodierte Zeichenfolge codieren.

Dazu können wir die DatatypeConverter Klasse aus dem Paket javax.xml.bind :

```
import javax.xml.bind.DatatypeConverter;
import java.util.Arrays;

// arbitrary binary data specified as a byte array
byte[] binaryData = "some arbitrary data".getBytes("UTF-8");

// convert the binary data to the base64-encoded string
String encodedData = DatatypeConverter.printBase64Binary(binaryData);
// encodedData is now "c29tZSBhcmJpdHJhcnkgZGF0YQ=="

// convert the base64-encoded string back to a byte array
byte[] decodedData = DatatypeConverter.parseBase64Binary(encodedData);

// assert that the original data and the decoded data are equal
assert Arrays.equals(binaryData, decodedData);
```

Apache Commons-Codec

Alternativ können wir Base64 von Apache Commons-Codec verwenden .

```
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)));
```

Wenn Sie dieses Programm bei laufendem Betrieb prüfen, werden Sie someBinaryData dass someBinaryData für c29tZUJpbmFyeURhdGE= kodiert, ein sehr verwaltbares *UTF-8*- String-Objekt.

Java SE 8

Details dazu finden Sie unter 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);
```

Referenz

Analysieren von Zeichenfolgen mit einem numerischen Wert

Zeichenfolge für einen primitiven numerischen Typ oder einen numerischen Wrapper-Typ:

Jede numerische Wrapper-Klasse stellt eine parseXxx Methode parseXxx , die einen String in den entsprechenden parseXxx konvertiert. Der folgende Code konvertiert einen String mit der Integer.parseInt Methode in ein int :

```
String string = "59";
int primitive = Integer.parseInteger(string);
```

Um einen String in eine Instanz einer numerischen Wrapper-Klasse zu konvertieren, können Sie entweder eine Überladung der valueOf Methode der Wrapper-Klassen verwenden:

```
String string = "59";
Integer wrapper = Integer.valueOf(string);
```

oder verlassen Sie sich auf das automatische Boxen (Java 5 und höher):

```
String string = "59";
Integer wrapper = Integer.parseInteger(string);  // 'int' result is autoboxed
```

Das obige Muster funktioniert für byte , short , int , long , float und double und die entsprechenden Wrapper-Klassen (Byte , Short , Integer , Long , Float und Double).

Zeichenfolge in Ganzzahl mit 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
```

Ausnahmen

Die nicht geprüfte NumberFormatException- Ausnahme wird ausgelöst, wenn eine numerische valueOf(String) oder parseXxx(...) -Methode für eine Zeichenfolge aufgerufen wird, die keine akzeptable numerische Darstellung ist oder einen Wert außerhalb des Bereichs darstellt.

Einen String aus einem InputStream holen

Ein String kann mit dem Byte-Array-Konstruktor aus einem InputStream gelesen werden.

```
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);
}
```

Dies verwendet den Standard-Zeichensatz des Systems, obwohl ein alternativer Zeichensatz angegeben werden kann:

```
return new String(bytes, Charset.forName("UTF-8"));
```

String in andere Datentypen konvertieren.

Sie können eine **numerische** Zeichenfolge folgendermaßen in verschiedene numerische Java-Typen konvertieren:

String nach int:

```
String number = "12";
int num = Integer.parseInt(number);
```

Zeichenfolge zu schweben:

```
String number = "12.0";
float num = Float.parseFloat(number);
```

Zeichenfolge zu verdoppeln:

```
String double = "1.47";
double num = Double.parseDouble(double);
```

Zeichenfolge nach Boolean:

```
String falseString = "False";
boolean falseBool = Boolean.parseBoolean(falseString);   // falseBool = false

String trueString = "True";
boolean trueBool = Boolean.parseBoolean(trueString);     // trueBool = true
```

Zeichenfolge zu lang:

```
String number = "47";
long num = Long.parseLong(number);
```

String zu BigInteger:

```
String bigNumber = "21";
BigInteger reallyBig = new BigInteger(bigNumber);
```

Zeichenfolge zu BigDecimal:

```
String bigFraction = "17.21455";
BigDecimal reallyBig = new BigDecimal(bigFraction);
```

Konvertierungsausnahmen:

Die obigen numerischen Konvertierungen NumberFormatException alle eine (ungeprüfte) NumberFormatException wenn Sie versuchen, eine Zeichenfolge zu analysieren, die keine geeignet formatierte Zahl ist oder für den NumberFormatException außerhalb des NumberFormatException Bereichs liegt. Das Thema Ausnahmen behandelt den Umgang mit solchen Ausnahmen.

Wenn Sie testen tryParse... dass Sie eine Zeichenfolge analysieren können, können Sie eine tryParse... Methode wie tryParse... implementieren:

```
boolean tryParseInt (String value) {
    try {
        String somechar = Integer.parseInt(value);
        return true;
     } catch (NumberFormatException e) {
        return false;
     }
}
```

Das Aufrufen dieser tryParse... -Methode unmittelbar vor dem Parsen ist jedoch (vermutlich) schlechte Praxis. Es wäre besser, einfach die Methode parse... aufzurufen und mit der Ausnahme fertig zu werden.

Konvertieren in und aus Strings online lesen:
https://riptutorial.com/de/java/topic/6678/konvertieren-in-und-aus-strings

Kapitel 100: Lambda-Ausdrücke

Einführung

Lambda-Ausdrücke bieten eine klare und präzise Möglichkeit, eine Schnittstelle mit einer Methode mithilfe eines Ausdrucks zu implementieren. Mit ihnen können Sie die Menge an Code reduzieren, die Sie erstellen und verwalten müssen. Obwohl sie anonymen Klassen ähnlich sind, haben sie selbst keine Typinformationen. Typ Inferenz muss passieren.

Methodenreferenzen implementieren funktionale Schnittstellen mit vorhandenen Methoden anstelle von Ausdrücken. Sie gehören auch zur Lambda-Familie.

Syntax

- () -> {return expression; } // Zero-Arity mit Funktionsrumpf, um einen Wert zurückzugeben.
- () -> Ausdruck // Abkürzung für die obige Deklaration; Es gibt kein Semikolon für Ausdrücke.
- () -> {function-body} // Nebeneffekt im Lambda-Ausdruck zum Ausführen von Operationen.
- Parametername -> Ausdruck // Lambda-Ausdruck mit einer Arität. In Lambda-Ausdrücken mit nur einem Argument kann die Klammer entfernt werden.
- (Typ parameterName, type secondParameterName, ...) -> Ausdruck // Lambda, der einen Ausdruck mit links aufgelisteten Parametern auswertet
- (parameterName, secondParameterName, ...) -> Ausdruck // Abkürzung, die die Parametertypen für die Parameternamen entfernt. Kann nur in Kontexten verwendet werden, die vom Compiler abgeleitet werden können, wenn die angegebene Parameterlistengröße mit einer (und nur einer) Größe der erwarteten funktionalen Schnittstellen übereinstimmt.

Examples

Verwenden von Lambda-Ausdrücken zum Sortieren einer Sammlung

Listen sortieren

Vor Java 8 war es erforderlich, die java.util.Comparator Schnittstelle mit einer anonymen (oder benannten) Klasse zu implementieren, wenn eine Liste [1] sortiert wird:

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());
        }
    }
);
```

Beginnend mit Java 8 kann die anonyme Klasse durch einen Lambda-Ausdruck ersetzt werden. Beachten Sie, dass die Typen für die Parameter p1 und p2 können, da der Compiler sie automatisch ableitet:

```
Collections.sort(
    people,
    (p1, p2) -> p1.getFirstName().compareTo(p2.getFirstName())
);
```

Das Beispiel kann vereinfacht werden, indem Comparator.comparing und Methodenreferenzen verwendet werden, die mit dem Symbol :: (Doppelpunkt) ausgedrückt werden.

```
Collections.sort(
    people,
    Comparator.comparing(Person::getFirstName)
);
```

Ein statischer Import erlaubt es uns, dies genauer zu formulieren, aber es ist fraglich, ob dies die Lesbarkeit insgesamt verbessert:

```
import static java.util.Collections.sort;
import static java.util.Comparator.comparing;
//...
sort(people, comparing(Person::getFirstName));
```

Auf diese Weise aufgebaute Komparatoren können auch miteinander verkettet werden. Wenn Sie zum Beispiel Personen nach ihrem Vornamen vergleichen und Personen mit demselben Vornamen verwenden, kann die thenComparing Methode auch mit dem thenComparing verglichen werden:

```
sort(people, comparing(Person::getFirstName).thenComparing(Person::getLastName));
```

---

1 - Beachten Sie, dass Collections.sort (...) nur für Sammlungen funktioniert, die Untertypen von `List` . Die `Set` und `Collection` APIs implizieren keine Anordnung der Elemente.

Karten sortieren

Sie können die Einträge einer HashMap auf ähnliche Weise nach Wert sortieren. (Beachten Sie, dass eine LinkedHashMap als Ziel verwendet werden muss. Die Schlüssel in einer normalen HashMap sind ungeordnet.)

```
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));
```

Einführung in Java-Lambdas

Funktionale Schnittstellen

Lambdas können nur an einer funktionalen Schnittstelle arbeiten, bei der es sich um eine Schnittstelle mit nur einer abstrakten Methode handelt. Funktionsschnittstellen können eine beliebige Anzahl von default oder static Methoden haben. (Aus diesem Grund werden sie manchmal als Single Abstract Method Interfaces oder SAM Interfaces bezeichnet).

```
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();
}
```

Bei der Deklaration einer funktionalen Schnittstelle kann die Annotation `@FunctionalInterface` hinzugefügt werden. Dies hat keine besonderen Auswirkungen, es wird jedoch ein Compiler-Fehler generiert, wenn diese Anmerkung auf eine nicht funktionale Schnittstelle angewendet wird, die als Erinnerung daran erinnert, dass die Schnittstelle nicht geändert werden sollte.

```
@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
}
```

Umgekehrt ist dies **keine** funktionale Schnittstelle, da es mehr als **eine abstrakte** Methode gibt:

```
interface BadFoo {
    void bar();
    void quux(); // <-- Second method prevents lambda: which one should
                 // be considered as lambda?
}
```

Dies ist **auch keine** funktionale Schnittstelle, da es keine Methoden gibt:

```
interface BlankFoo2 { }
```

Beachten Sie folgendes. Angenommen, Sie haben

`interface Parent { public int parentMethod(); }`

und

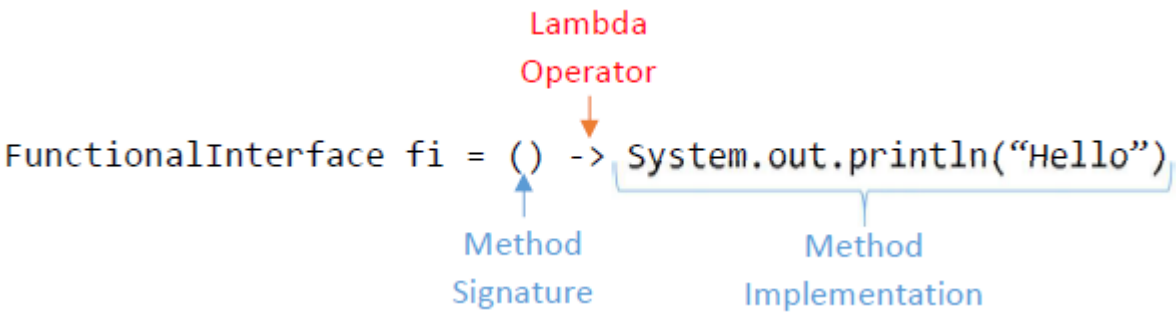
`interface Child extends Parent { public int ChildMethod(); }`

Dann **kann** Child **keine** funktionale Schnittstelle sein, da es zwei angegebene Methoden hat.

Java 8 bietet auch eine Reihe von generischen Templates mit funktionalen Schnittstellen im Paket `java.util.function` . Die eingebaute Schnittstelle Predicate<T> umgibt eine einzelne Methode, die einen Wert vom Typ T boolean und einen boolean Wert zurückgibt.

Lambda-Ausdrücke

Die Grundstruktur eines Lambda-Ausdrucks lautet:



fi wird dann eine Singleton-Instanz einer Klasse enthalten, ähnlich einer anonymen Klasse, die FunctionalInterface implementiert und die Definition der einen Methode lautet { System.out.println("Hello"); } . Mit anderen Worten, das obige ist größtenteils äquivalent zu:

```
FunctionalInterface fi = new FunctionalInterface() {
    @Override
    public void theOneMethod() {
        System.out.println("Hello");
    }
};
```

Das Lambda ist der anonymen Klasse nur "weitgehend gleichwertig", da in einem Lambda die Bedeutung von Ausdrücken wie this , super oder toString() auf die Klasse verweist, in der die Zuweisung stattfindet, nicht auf das neu erstellte Objekt.

Sie können den Namen der Methode nicht angeben, wenn Sie ein Lambda verwenden. Dies sollte jedoch nicht erforderlich sein, da eine funktionale Schnittstelle nur eine abstrakte Methode haben muss, also überschreibt Java diese.

In Fällen, in denen der Typ des Lambda nicht sicher ist (z. B. überladene Methoden), können Sie dem Lambda einen Abguss hinzufügen, um dem Compiler mitzuteilen, welcher Typ es sein sollte:

```
Object fooHolder = (Foo1) () -> System.out.println("Hello");
System.out.println(fooHolder instanceof Foo1); // returns true
```

Wenn für die einzelne Methode der funktionalen Schnittstelle Parameter erforderlich sind, sollten die lokalen formalen Namen zwischen den Klammern des Lambda stehen. Es ist nicht erforderlich, den Typ des Parameters zu deklarieren oder zurückzugeben, da diese von der Schnittstelle abgerufen werden (obwohl es kein Fehler ist, die Parametertypen zu deklarieren, wenn Sie möchten). Somit sind diese beiden Beispiele gleichwertig:

```
Foo2 longFoo = new Foo2() {
    @Override
    public int bar(boolean baz) {
        return baz ? 1 : 0;
    }
};
Foo2 shortFoo = (x) -> { return x ? 1 : 0; };
```

Die Klammern um das Argument können weggelassen werden, wenn die Funktion nur ein Argument hat:

```
Foo2 np = x -> { return x ? 1 : 0; }; // okay
Foo3 np2 = x, y -> x.toString() + y // not okay
```

Implizite Rückgaben

Wenn der in einem Lambda platzierte Code eher ein Java- *Ausdruck* als eine *Anweisung ist* , wird er als Methode behandelt, die den Wert des Ausdrucks zurückgibt. Somit sind die folgenden zwei

äquivalent:

```
IntUnaryOperator addOneShort = (x) -> (x + 1);
IntUnaryOperator addOneLong = (x) -> { return (x + 1); };
```

Zugriff auf lokale Variablen (Werteverschlüsse)

Da Lambdas eine syntaktische Abkürzung für anonyme Klassen sind, gelten dieselben Regeln für den Zugriff auf lokale Variablen im umschließenden Bereich. Die Variablen müssen als final behandelt und nicht innerhalb des Lambda geändert werden.

```
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
}
```

Wenn eine sich ändernde Variable auf diese Weise umbrochen werden soll, sollte ein reguläres Objekt verwendet werden, das eine Kopie der Variablen enthält. Lesen Sie mehr in Java Closures mit Lambda-Ausdrücken.

Lambdas akzeptieren

Da ein Lambda eine Implementierung einer Schnittstelle ist, muss nichts Besonderes getan werden, damit eine Methode ein Lambda akzeptiert: Jede Funktion, die eine funktionale Schnittstelle übernimmt, kann auch ein Lambda akzeptieren.

```
public void passMeALambda(Foo1 f) {
    f.bar();
}
passMeALambda(() -> System.out.println("Lambda called"));
```

Der Typ eines Lambda-Ausdrucks

Ein Lambda-Ausdruck an sich hat keinen bestimmten Typ. Es ist zwar richtig, dass die Typen und die Anzahl der Parameter zusammen mit dem Typ eines Rückgabewerts einige Typinformationen übermitteln können. Diese Informationen beschränken jedoch nur, welchen Typen sie zugewiesen werden können. Das Lambda erhält einen Typ, wenn es auf eine der folgenden Arten einem funktionalen Schnittstellentyp zugewiesen wird:

- Direkte Zuordnung zu einem funktionalen Typ, zB myPredicate = s -> s.isEmpty()
- stream.filter(s -> s.isEmpty()) als Parameter mit einem funktionalen Typ, z. B. stream.filter(s -> s.isEmpty())
- Rückgabe von einer Funktion, die einen Funktionstyp zurückgibt, z. B. return s -> s.isEmpty()
- Casting in einen funktionalen Typ, z. B. (Predicate<String>) s -> s.isEmpty()

Bis zu einer solchen Zuordnung zu einem funktionalen Typ hat das Lambda keinen bestimmten Typ. Betrachten Sie zur Veranschaulichung den Lambda-Ausdruck o -> o.isEmpty() . Derselbe Lambda-Ausdruck kann vielen verschiedenen Funktionstypen zugewiesen werden:

```
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();
```

Jetzt, da sie zugewiesen sind, sind die Beispiele völlig unterschiedlich, auch wenn die Lambda-Ausdrücke gleich aussehen und sie nicht einander zugeordnet werden können.

Methodenreferenzen

Über Methodenreferenzen können vordefinierte statische oder Instanzmethoden, die einer kompatiblen funktionalen Schnittstelle entsprechen, als Argumente anstelle eines anonymen Lambda-Ausdrucks übergeben werden.

Angenommen, wir haben ein Modell:

```
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();
```

Instanzmethodenreferenz (auf eine beliebige Instanz)

```
people.stream().map(Person::getName)
```

Das Äquivalent Lambda:

```
people.stream().map(person -> person.getName())
```

In diesem Beispiel wird eine Methodenreferenz auf die Instanzmethode getName() vom Typ Person übergeben. Da bekannt ist, dass es vom Collection-Typ ist, wird die Methode der Instanz (später bekannt) aufgerufen.

Instanzmethodenreferenz (auf eine bestimmte Instanz)

```
people.forEach(System.out::println);
```

Da System.out eine Instanz von PrintStream , wird eine Methodenreferenz auf diese bestimmte Instanz als Argument übergeben.

Das Äquivalent Lambda:

```
people.forEach(person -> System.out.println(person));
```

Statische Methodenreferenz

Auch für die Transformation von Streams können Verweise auf statische Methoden angewendet werden:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
numbers.stream().map(String::valueOf)
```

In diesem Beispiel wird ein Verweis auf die statische valueOf() -Methode für den String Typ übergeben. Daher wird das valueOf() in der Auflistung als Argument an valueOf() .

Das Äquivalent Lambda:

```
 numbers.stream().map(num -> String.valueOf(num))
```

Verweis auf einen Konstruktor

```
List<String> strings = Arrays.asList("1", "2", "3");
strings.stream().map(Integer::new)
```

Lesen Sie Sammeln von Elementen eines Streams in einer Sammlung, um zu erfahren, wie Elemente zur Sammlung gesammelt werden.

Der einzelne String-Argumentkonstruktor vom Typ Integer wird hier verwendet, um eine Ganzzahl zu erstellen, die die als Argument angegebene Zeichenfolge enthält. In diesem Fall wird der Stream auf Integer-Werte abgebildet, solange der String eine Zahl darstellt. Das Äquivalent Lambda:

```
strings.stream().map(s -> new Integer(s));
```

Spickzettel

Methodenreferenzformat	Code	Äquivalentes Lambda
Statische Methode	TypeName::method	(args) -> TypeName.method(args)
Nicht statische Methode (in Instanz *)	instance::method	(args) -> instance.method(args)
Nicht statische Methode (keine Instanz)	TypeName::method	(instance, args) -> instance.method(args)
Konstrukteur **	TypeName::new	(args) -> new TypeName(args)
Array-Konstruktor	TypeName[]::new	(int size) -> new TypeName[size]

---

* instance kann ein beliebiger Ausdruck sein, der eine Referenz auf eine Instanz auswertet, z. B. die getInstance()::method , this::method

** Wenn TypeName eine nicht statische innere Klasse ist, ist die Konstruktorreferenz nur innerhalb des Bereichs einer äußeren Klasseninstanz gültig

Implementierung mehrerer Schnittstellen

Manchmal möchten Sie vielleicht einen Lambda-Ausdruck, der mehr als eine Schnittstelle implementiert. Dies ist vor allem bei Markerschnittstellen (z. B. java.io.Serializable) hilfreich, da sie keine abstrakten Methoden hinzufügen.

Sie möchten beispielsweise ein TreeSet mit einem benutzerdefinierten Comparator erstellen, es serialisieren und über das Netzwerk senden. Der triviale Ansatz:

```
TreeSet<Long> ts = new TreeSet<>((x, y) -> Long.compare(y, x));
```

funktioniert nicht, da das Lambda für den Komparator kein Serializable implementiert. Sie können dies beheben, indem Sie Schnitttypen verwenden und explizit angeben, dass dieses Lambda serialisierbar sein muss:

```
TreeSet<Long> ts = new TreeSet<>(
    (Comparator<Long> & Serializable) (x, y) -> Long.compare(y, x));
```

Wenn Sie häufig Schnitttypen verwenden (wenn Sie beispielsweise ein Framework wie Apache Spark verwenden, bei dem fast alles serialisierbar sein muss), können Sie leere Schnittstellen erstellen und diese stattdessen in Ihrem Code verwenden:

```
public interface SerializableComparator extends Comparator<Long>, Serializable {}

public class CustomTreeSet {
  public CustomTreeSet(SerializableComparator comparator) {}
}
```

Auf diese Weise wird garantiert, dass der übergebene Komparator serialisierbar ist.

Lambdas und Execute-around-Pattern

Es gibt mehrere gute Beispiele für die Verwendung von Lambdas als FunctionalInterface in einfachen Szenarien. Ein recht häufiger Anwendungsfall, der durch Lambdas verbessert werden kann, ist das sogenannte Execute-Around-Muster. In diesem Muster verfügen Sie über einen Satz von Standard-Setup- / Teardown-Code, der für mehrere Szenarien benötigt wird, die Anwendungsfall-spezifischen Code betreffen. Ein paar häufige Beispiele hierfür sind file io, database io, try / catch-Blöcke.

```
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();
    }
}
```

Wenn Sie diese Methode dann mit einem Lambda aufrufen, könnte es so aussehen:

```
public static void updateMyDAO(MyVO vo) throws DatabaseException {
    doProcessing((Connection conn) -> MyDAO.update(conn, ObjectMapper.map(vo)));
}
```

Dies ist nicht auf E / A-Vorgänge beschränkt. Sie kann auf jedes Szenario angewendet werden, in dem ähnliche Auf- / Abbauaufgaben mit geringfügigen Abweichungen anwendbar sind. Der Hauptvorteil dieses Musters ist die Wiederverwendung von Code und das Durchsetzen von DRY (Don't Repeat Yourself).

Verwenden des Lambda-Ausdrucks mit Ihrer eigenen funktionalen Schnittstelle

Lambdas sollen Inline-Implementierungscode für Schnittstellen mit einer einzigen Methode bereitstellen und die Möglichkeit geben, sie wie normale Variablen weiterzugeben. Wir nennen sie funktionale Schnittstelle.

Das Schreiben einer ausführbaren Klasse in anonyme Klasse und das Starten eines Threads sieht beispielsweise folgendermaßen aus:

```
//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();
```

Nun, im Einklang mit dem obigen Beispiel, können Sie sagen, Sie haben eine benutzerdefinierte Schnittstelle:

```
interface TwoArgInterface {
    int operate(int a, int b);
}
```

Wie verwenden Sie Lambda, um diese Schnittstelle in Ihrem Code zu implementieren? Dasselbe wie das oben gezeigte ausführbare Beispiel. Siehe das Treiberprogramm unten:

```
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" kehrt nur vom Lambda zurück, nicht von der äußeren Methode

Die return kehrt nur vom Lambda zurück, nicht von der äußeren Methode.

Beachten Sie, dass diese von Scala und Kotlin *anders!*

```
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!
  });
}
```

Dies kann zu unerwartetem Verhalten führen , wenn versucht eigene Sprachkonstrukte zu schreiben, wie es in gebautet Konstrukte wie for Schleifen return verhält sich anders:

```
void demo2() {
  for (int i = 0; i < 3; i++) {
    System.out.println(i);
    return; // Return from 'demo2' entirely
  }
}
```

In Scala und Kotlin würden demo und demo2 nur 0 drucken. Dies ist jedoch *nicht* konsistenter. Der Java-Ansatz ist konsistent mit dem Refactoring und der Verwendung von Klassen - die return im Code oben und der Code unten verhält sich gleich:

```
void demo3() {
  threeTimes(new MyIntConsumer());
}

class MyIntConsumer implements IntConsumer {
  public void accept(int i) {
    System.out.println(i);
    return;
  }
}
```

Daher ist die Java return mehr im Einklang mit Klassenmethoden und Refactoring, aber weniger mit dem for und while builtins, bleiben diese besondere.

Aus diesem Grund sind die folgenden zwei in Java gleichwertig:

```
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);
```

Darüber hinaus ist die Verwendung von Try-with-Ressourcen in Java sicher:

```
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.
  });
}
```

wird before accept() after close() gedruckt. In der Scala- und Kotlin-Semantik werden die Try-with-Ressourcen nicht geschlossen, sondern nur before accept() gedruckt.

Java-Closures mit Lambda-Ausdrücken.

Ein Lambda-Abschluss wird erstellt, wenn ein Lambda-Ausdruck auf die Variablen eines umschließenden Bereichs (global oder lokal) verweist. Die Regeln dafür sind die gleichen wie für Inline-Methoden und anonyme Klassen.

Lokale Variablen aus einem einschließenden Bereich, die in einem Lambda verwendet werden, müssen final . Mit Java 8 (der frühesten Version, die Lambdas unterstützt) müssen sie im äußeren Kontext nicht *als* final *deklariert werden* , müssen jedoch so behandelt werden. Zum Beispiel:

```
int n = 0; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
```

Dies ist zulässig, solange der Wert der Variablen n nicht geändert wird. Wenn Sie versuchen, die Variable innerhalb oder außerhalb des Lambda zu ändern, wird der folgende Kompilierungsfehler angezeigt:

> msgstr "Lokale Variablen, auf die von einem Lambda - Ausdruck verwiesen wird, müssen *final* oder *effektiv final sein.* "

Zum Beispiel:

```
int n = 0;
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
n++; // Will generate an error.
```

Wenn innerhalb eines Lambda eine sich ändernde Variable verwendet werden muss, wird normalerweise eine final Kopie der Variablen deklariert und die Kopie verwendet. Zum Beispiel

```
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
```

Natürlich sieht der Körper des Lambda die Änderungen an der ursprünglichen Variablen nicht.

Beachten Sie, dass Java keine echten Schließungen unterstützt. Ein Java-Lambda kann nicht so erstellt werden, dass Änderungen in der Umgebung sichtbar werden, in der es instanziiert wurde. Wenn Sie einen Abschluss implementieren möchten, der die Umgebung beobachtet oder Änderungen daran vornimmt, sollten Sie ihn mit einer regulären Klasse simulieren. Zum Beispiel:

```
// Does not compile ...
public IntUnaryOperator createAccumulator() {
    int value = 0;
    IntUnaryOperator accumulate = (x) -> { value += x; return value; };
    return accumulate;
}
```

Das obige Beispiel wird aus den zuvor diskutierten Gründen nicht kompiliert. Wir können den Kompilierungsfehler wie folgt umgehen:

```
// Compiles, but is incorrect ...
public class AccumulatorGenerator {
    private int value = 0;

    public IntUnaryOperator createAccumulator() {
        IntUnaryOperator accumulate = (x) -> { value += x; return value; };
        return accumulate;
    }
}
```

Das Problem ist, dass dadurch der Designvertrag für die IntUnaryOperator Schnittstelle IntUnaryOperator , der besagt, dass Instanzen funktionsfähig und zustandslos sein sollten. Wenn ein solcher Abschluss an integrierte Funktionen übergeben wird, die Funktionsobjekte akzeptieren, kann dies zu Abstürzen oder fehlerhaftem Verhalten führen. Verschlüsse, die den veränderbaren Zustand einschließen, sollten als reguläre Klassen implementiert werden. Zum Beispiel.

```
// Correct ...
public class Accumulator {
   private int value = 0;

   public int accumulate(int x) {
      value += x;
      return value;
   }
}
```

Lambda - Hörer Beispiel

Anonymer Klassenlistener

Vor Java 8 ist es sehr üblich, dass eine anonyme Klasse verwendet wird, um das Click-Ereignis einer JButton-Klasse zu behandeln, wie im folgenden Code dargestellt. Dieses Beispiel zeigt, wie Sie einen anonymen Listener im Rahmen von btn.addActionListener .

```
JButton btn = new JButton("My Button");
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button was pressed");
    }
});
```

Lambda-Hörer

Da die ActionListener Schnittstelle nur eine Methode actionPerformed() , handelt es sich um eine funktionale Schnittstelle, actionPerformed() es gibt einen Platz, an dem Lambda-Ausdrücke verwendet werden können, um den Boilerplate-Code zu ersetzen. Das obige Beispiel kann wie folgt mit Lambda-Ausdrücken umgeschrieben werden:

```
JButton btn = new JButton("My Button");
btn.addActionListener(e -> {
    System.out.println("Button was pressed");
});
```

Traditioneller Stil im Lambda-Stil

Traditioneller Weg

```
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));
    }
}
```

Lambda-Stil

1. Entfernen Sie den Klassennamen und den funktionalen Schnittstellenkörper.

```
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. Optionale Typdeklaration

```
MathOperation isEven = (num) -> {
    return num%2 == 0;
};
```

3. Optionale Klammer um Parameter, wenn es sich um einen einzelnen Parameter handelt

```
MathOperation isEven = num -> {
    return num%2 == 0;
};
```

4. Optionale geschweifte Klammern, wenn der Funktionskörper nur eine Zeile enthält
5. Optionales return-Schlüsselwort, wenn der Funktionskörper nur eine Zeile enthält

```
MathOperation isEven = num -> num%2 == 0;
```

Lambdas und Speicherauslastung

Da Java-Lambdas Schließungen sind, können sie die Werte von Variablen im umgebenden lexikalischen Bereich "erfassen". Obwohl nicht alle Lambdas alles erfassen - einfache lambdas wie `s -> s.length()` erfaßt nichts und sind *staatenlos* genannt - Erfassung lambda erfordert ein temporäres Objekt die erfassten Variablen zu halten. In diesem Code-Snippet ist das Lambda `() -> j` ein Fang-Lambda und kann dazu führen, dass ein Objekt zugewiesen wird, wenn es ausgewertet wird:

```
public static void main(String[] args) throws Exception {
    for (int i = 0; i < 1000000000; i++) {
        int j = i;
        doSomethingWithLambda(() -> j);
    }
}
```

Obwohl es möglicherweise nicht sofort offensichtlich ist, da das new Schlüsselwort an keiner Stelle im Snippet angezeigt wird, kann dieser Code 1.000.000.000 separate Objekte erstellen, um die Instanzen des () -> j Lambda-Ausdrucks darzustellen. Es sollte jedoch auch beachtet werden, dass zukünftige Versionen von Java [1] dies möglicherweise so optimieren können, dass die Lambda-Instanzen zur *Laufzeit erneut* verwendet oder auf andere Weise dargestellt wurden.

---

1 - Zum Beispiel führt Java 9 eine optionale "Link" -Phase für die Java-Buildsequenz ein, die die Möglichkeit für globale Optimierungen wie diese bietet.

Verwenden von Lambda-Ausdrücken und Prädikaten, um bestimmte Werte aus einer Liste zu erhalten

Ab Java 8 können Sie Lambda-Ausdrücke und Prädikate verwenden.

Beispiel: Verwenden Sie einen Lambda-Ausdruck und ein Prädikat, um einen bestimmten Wert aus einer Liste zu erhalten. In diesem Beispiel wird jede Person mit der Tatsache ausgedruckt, ob sie 18 Jahre oder älter ist oder nicht.

Personenklasse:

```
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; }
}
```

Die integrierte Schnittstelle Predicate aus den Paketen java.util.function.Predicate ist eine funktionale Schnittstelle mit einer boolean test(T t) .

Verwendungsbeispiel:

```
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.");
            }
        }
    }
}
```

Der print(personList, p -> p.getAge() >= 18); Die Methode nimmt einen Lambda-Ausdruck (da das Prädikat ein Parameter ist), in dem Sie den benötigten Ausdruck definieren können. Die Testmethode des checker.test(person) überprüft, ob dieser Ausdruck korrekt ist oder nicht: checker.test(person) .

Sie können dies leicht in etwas anderes ändern, beispielsweise zum print(personList, p -> p.getName().startsWith("J")); . Dadurch wird geprüft, ob der Name der Person mit einem "J" beginnt.

Lambda-Ausdrücke online lesen: https://riptutorial.com/de/java/topic/91/lambda-ausdrucke

Kapitel 101: Laufzeitbefehle

Examples

Abschalthaken hinzufügen

Manchmal benötigen Sie einen Code, der ausgeführt wird, wenn das Programm beendet wird, z. Sie können einen Thread ausführen, wenn das Programm mit der addShutdownHook Methode beendet wird:

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    ImportantStuff.someImportantIOStream.close();
}));
```

Laufzeitbefehle online lesen: https://riptutorial.com/de/java/topic/7304/laufzeitbefehle

Kapitel 102: Leser und Schriftsteller

Einführung

Leser und Verfasser und ihre jeweiligen Unterklassen bieten einfache E / A für text- / zeichenbasierte Daten.

Examples

BufferedReader

Einführung

Die BufferedReader Klasse ist ein Wrapper für andere Reader Klassen, der zwei Hauptzwecken dient:

1. Ein BufferedReader bietet Pufferung für den eingebundenen Reader . Dadurch kann eine Anwendung Zeichen nacheinander ohne übermäßigen E / A-Aufwand lesen.
2. Ein BufferedReader bietet Funktionen zum Lesen von Text für Zeile.

Grundlagen zur Verwendung eines BufferedReader

Das normale Muster für die Verwendung eines BufferedReader ist wie folgt. Zuerst erhalten Sie den Reader , von dem Sie die Zeichen lesen möchten. Als Nächstes instanziieren Sie einen BufferedReader , der den Reader BufferedReader . Dann lesen Sie Zeichendaten. Schließlich schließen Sie den BufferedReader der den umwickelten `Reader schließt. Zum Beispiel:

```
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);
}
```

Sie können dieses Muster auf jeden Reader anwenden

Anmerkungen:

1. Wir haben Java 7 (oder später) *Try-With-Ressourcen verwendet*, um sicherzustellen, dass der zugrunde liegende Reader immer geschlossen ist. Dadurch wird ein potenzielles Ressourcenleck vermieden. In früheren Java-Versionen würden Sie den BufferedReader explizit in einem finally Block schließen.
2. Der Code im try Block ist praktisch identisch mit dem, was wir verwenden würden, wenn wir direkt aus dem FileReader lesen. Tatsächlich funktioniert ein BufferedReader genauso wie der Reader , den er wrappen würde. Der Unterschied ist, dass *diese* Version wesentlich effizienter ist.

Die Puffergröße des BufferedReader

Die BufferedReader.readLine () -Methode

Beispiel: Lesen aller Zeilen einer Datei in eine Liste

Dazu wird jede Zeile in einer Datei abgerufen und in eine List<String> . Die Liste wird dann zurückgegeben:

```
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 bietet eine einfachere Möglichkeit, dies mit der lines() -Methode zu tun:

```
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();
}
```

StringWriter-Beispiel

Die Java-StringWriter-Klasse ist ein Zeichenstrom, der die Ausgabe von Zeichenfolgenpuffer sammelt und zur Erstellung einer Zeichenfolge verwendet werden kann.

Die StringWriter-Klasse erweitert die Writer-Klasse.

In der StringWriter-Klasse werden keine Systemressourcen wie Netzwerk-Sockets und -Dateien verwendet. Das Schließen des StringWriter ist daher nicht erforderlich.

```
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();
    }
}
```

Das obige Beispiel hilft uns, ein einfaches Beispiel für StringWriter kennenzulernen, der BufferedReader verwendet, um Dateidaten aus dem Stream zu lesen.

Leser und Schriftsteller online lesen: https://riptutorial.com/de/java/topic/10618/leser-und-schriftsteller

Kapitel 103: LinkedHashMap

Einführung

Die LinkedHashMap-Klasse ist eine Hash-Tabelle und eine Implementierung der Linked-Liste der Map-Schnittstelle mit einer vorhersagbaren Iterationsreihenfolge. Es erbt die HashMap-Klasse und implementiert die Map-Schnittstelle.

Die wichtigsten Punkte zur Java-Klasse LinkedHashMap sind: Eine LinkedHashMap enthält Werte, die auf dem Schlüssel basieren. Es enthält nur eindeutige Elemente. Es kann einen Nullschlüssel und mehrere Nullwerte haben. Es ist dasselbe, wie HashMap stattdessen die Einfügereihenfolge beibehält.

Examples

Java LinkedHashMap-Klasse

Schlüsselpunkte: -

- Ist eine Hash-Tabelle und eine Linked List-Implementierung der Map-Schnittstelle mit vorhersagbarer Iterationsreihenfolge.
- erbt die HashMap-Klasse und implementiert die Map-Schnittstelle.
- enthält Werte, die auf dem Schlüssel basieren.
- nur eindeutige Elemente.
- kann einen Nullschlüssel und mehrere Nullwerte haben.
- Dasselbe wie bei HashMap bleibt die Reihenfolge beim Einfügen erhalten.

Methoden: -

- nicht löschen ().
- boolean containsKey (Objektschlüssel).
- Object get (Objektschlüssel).
- protected Boolean removeEldestEntry (Map.Entry Ältester)

Beispiel: -

```
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"));
```

```
    }
```

LinkedHashMap online lesen: https://riptutorial.com/de/java/topic/10750/linkedhashmap

Kapitel 104: Liste vs SET

Einführung

Welche Unterschiede bestehen zwischen der List- und Set-Sammlung auf der obersten Ebene und Wie wählen Sie aus, wann List in Java verwendet werden soll und wann Set in Java verwendet werden soll?

Examples

List vs Set

```
import java.util.ArrayList;
```

import java.util.HashSet; import java.util.List; import java.util.Set;

public class SetAndListExample {public static void main (String [] args) {System.out.println ("Listenbeispiel ....."); Liste Liste = neue 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);
    }
}
```

}

Ausgangslistenbeispiel ..... 1 2 3 4 1 Set-Beispiel 3 2 10 5 4

Liste vs SET online lesen: https://riptutorial.com/de/java/topic/10125/liste-vs-set

Kapitel 105: Listen

Einführung

Eine *Liste* ist eine *geordnete* Sammlung von Werten. In Java sind Listen Teil des Java Collections Framework . Listen implementieren die Schnittstelle java.util.List , die java.util.Collection .

Syntax

- ls.add (E-Element); // Fügt ein Element hinzu
- ls.remove (E-Element); // Entfernt ein Element
- for (E Element: ls) {} // Iteriert über jedes Element
- ls.toArray (neuer String [ls.length]); // Konvertiert eine Liste von Strings in ein Array von Strings
- ls.get (int index); // Gibt das Element am angegebenen Index zurück.
- ls.set (int index, E-Element); // Ersetzt das Element an einer angegebenen Position.
- ls.isEmpty (); // Gibt true zurück, wenn das Array keine Elemente enthält. Andernfalls wird false zurückgegeben.
- ls.indexOf (Objekt o); // Liefert den Index der ersten Position des angegebenen Elements o oder, falls nicht vorhanden, -1.
- ls.lastIndexOf (Objekt o); // Gibt den Index der letzten Position des angegebenen Elements zurück, o oder, falls nicht vorhanden, -1.
- ls.size (); // Gibt die Anzahl der Elemente in der Liste zurück.

Bemerkungen

Eine Liste ist ein Objekt, in dem eine geordnete Wertesammlung gespeichert wird. "Bestellt" bedeutet, dass die Werte in einer bestimmten Reihenfolge gespeichert werden - ein Element steht an erster Stelle, einer an zweiter Stelle usw. Die einzelnen Werte werden üblicherweise als "Elemente" bezeichnet. Java-Listen bieten normalerweise diese Funktionen:

- Listen können null oder mehr Elemente enthalten.
- Listen können doppelte Werte enthalten. Mit anderen Worten, ein Element kann mehrmals in eine Liste eingefügt werden.
- Listen speichern ihre Elemente in einer bestimmten Reihenfolge, das heißt, ein Element steht an erster Stelle, eines kommt an und so weiter.
- Jedes Element hat einen *Index* , der seine Position innerhalb der Liste angibt. Das erste Element hat den Index 0, das nächste Element den Index 1 und so weiter.
- Listen erlauben das Einfügen von Elementen am Anfang, am Ende oder an einem beliebigen Index innerhalb der Liste.
- Beim Testen, ob eine Liste einen bestimmten Wert enthält, müssen Sie in der Regel jedes Element in der Liste überprüfen. Dies bedeutet, dass die Zeit zum Durchführen dieser Prüfung O (n) ist , proportional zur Größe der Liste.

Durch Hinzufügen eines Werts zu einer Liste an einem anderen Punkt als dem Ende werden alle folgenden Elemente nach "unten" oder "nach rechts" verschoben. Mit anderen Worten, durch Hinzufügen eines Elements am Index *n wird* das Element, das sich zuvor am Index *n befand,* zum Index *n + 1* usw. verschoben. Zum Beispiel:

```
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

Eine generische Liste sortieren

Die Collections Klasse bietet zwei statische Standardmethoden zum Sortieren einer Liste:

- sort(List<T> list) gilt für Listen, bei denen T extends Comparable<? super T> und
- sort(List<T> list, Comparator<? super T> c) anwendbar für Listen eines beliebigen Typs.

Die erstere Anwendung erfordert die Änderung der Klasse der zu sortierenden Listenelemente, was nicht immer möglich ist. Es kann auch unerwünscht sein, da es zwar die Standardsortierung vorsieht, andere Sortierreihenfolgen jedoch unter verschiedenen Umständen erforderlich sein können oder dass das Sortieren nur eine einmalige Aufgabe ist.

Angenommen, wir haben die Aufgabe, Objekte zu sortieren, die Instanzen der folgenden Klasse sind:

```
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);
    }
}
```

Um Collections.sort(List<User> list) , müssen Sie die User ändern, um die Comparable Schnittstelle zu implementieren. Zum Beispiel

```
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);
    }
}
```

(Nebenbei: Viele Standard-Java-Klassen wie String , Long und Integer implementieren die Comparable Schnittstelle. Dadurch werden Listen dieser Elemente standardmäßig sortierbar und die Implementierung von compare oder compareTo in anderen Klassen vereinfacht.)

Mit der obigen Änderung können Sie eine Liste von User auf einfache Weise anhand der *natürlichen Reihenfolge* der Klassen sortieren. (In diesem Fall haben wir das definiert, um anhand von id Werten zu ordnen). Zum Beispiel:

```
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]
```

Nehmen wir jedoch an, wir wollten User nach name und nicht nach id sortieren. Nehmen Sie alternativ an, dass wir die Klasse nicht so ändern konnten, dass sie Comparable implementiert.

Hier ist die sort mit dem Comparator Argument hilfreich:

```
Collections.sort(users, new Comparator<User>() {
    @Override
    /* Order two 'User' objects based on their names. */
    public int compare(User left, User right) {
        return left.username.compareTo(right.username);
    }
});
System.out.print(users);
// [A:33, B:25, C:28]
```

Java SE 8

In Java 8 können Sie ein *Lambda* anstelle einer anonymen Klasse verwenden. Letzteres reduziert sich auf einen Einliner:

```
Collections.sort(users, (l, r) -> l.username.compareTo(r.username));
```

Ferner fügt es Java 8 eine Standard – sort auf der List Schnittstelle, die noch mehr erleichtert das Sortieren.

```
users.sort((l, r) -> l.username.compareTo(r.username))
```

Liste erstellen

Geben Sie Ihrer Liste einen Typ

Um eine Liste zu erstellen, benötigen Sie einen Typ (beliebige Klasse, z. B. String). Dies ist der Typ Ihrer List . Die List speichert nur Objekte des angegebenen Typs. Zum Beispiel:

```
List<String> strings;
```

Kann "string1" speichern, "hello world!" "goodbye" usw., aber es kann keine 9.2 speichern:

```
List<Double> doubles;
```

Kann 9.2 speichern, aber nicht "hello world!" .

Initialisieren Sie Ihre Liste

Wenn Sie versuchen, etwas zu den obigen Listen hinzuzufügen, erhalten Sie eine NullPointerException, da strings und doubles beide gleich **Null sind** !

Es gibt zwei Möglichkeiten, eine Liste zu initialisieren:

Option 1: Verwenden Sie eine Klasse, die List implementiert

List ist eine Schnittstelle, was bedeutet, dass es keinen Konstruktor gibt, sondern Methoden, die eine Klasse überschreiben muss. ArrayList ist die am häufigsten verwendete List , obwohl LinkedList auch häufig ist. Also initialisieren wir unsere Liste so:

```
List<String> strings = new ArrayList<String>();
```

oder

```
List<String> strings = new LinkedList<String>();
```

Java SE 7

Ab Java SE 7 können Sie einen *Diamantoperator verwenden* :

```
List<String> strings = new ArrayList<>();
```

oder

```
List<String> strings = new LinkedList<>();
```

Option 2: Verwenden Sie die Collections-Klasse

Die Collections Klasse bietet zwei nützliche Methoden zum Erstellen von Listen ohne List Variable:

- emptyList() : gibt eine leere Liste zurück.
- singletonList(T) : erstellt eine Liste vom Typ T und fügt das angegebene Element hinzu.

Und eine Methode, die eine vorhandene List , um Daten einzugeben:

- addAll(L, T...) : fügt der als ersten Parameter übergebenen Liste alle angegebenen Elemente hinzu.

Beispiele:

```
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);
```

Positionszugriffsvorgänge

Die Listen-API verfügt über acht Methoden für Positionszugriffsvorgänge:

- 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)

Wenn wir also eine Liste haben:

```
List<String> strings = new ArrayList<String>();
```

Und wir wollten die Zeichenfolgen "Hallo Welt!" und "Auf Wiedersehen Welt!" dazu würden wir es

so machen:

```
strings.add("Hello world!");
strings.add("Goodbye world!");
```

Und unsere Liste würde die zwei Elemente enthalten. Nun wollen wir sagen, wir wollten "Programm starten!" an der **Spitze** der Liste. Wir würden das so machen:

```
strings.add(0, "Program starting!");
```

HINWEIS: Das erste Element ist 0.

Nun, wenn wir die "Auf Wiedersehen Welt" entfernen wollten! Linie könnten wir es so machen:

```
strings.remove("Goodbye world!");
```

Und wenn wir die erste Zeile entfernen wollten (was in diesem Fall "Programmstart!" Wäre), könnten wir es so machen:

```
strings.remove(0);
```

Hinweis:

1. Durch das Hinzufügen und Entfernen von Listenelementen wird die Liste geändert. Dies kann zu einer ConcurrentModificationException führen, wenn die Liste gleichzeitig durchlaufen wird.
2. Das Hinzufügen und Entfernen von Elementen kann O(1) oder O(N) abhängig von der Listenklasse, der verwendeten Methode und davon, ob Sie ein Element am Anfang, am Ende oder in der Mitte der Liste hinzufügen oder entfernen.

Um ein Element der Liste an einer bestimmten Position abzurufen, können Sie den E get(int index); Methode der List API. Zum Beispiel:

```
strings.get(0);
```

gibt das erste Element der Liste zurück.

Sie können jedes Element an einer bestimmten Position ersetzen, indem Sie das set(int index, E element); . Zum Beispiel:

```
strings.set(0,"This is a replacement");
```

Dies setzt den String "Dies ist ein Ersatz" als erstes Element der Liste.

Hinweis: Die set-Methode überschreibt das Element an Position 0. Die neue Zeichenfolge wird nicht an Position 0 eingefügt und der alte an Position 1 verschoben.

Der int indexOf(Object o); gibt die Position des ersten Vorkommens des als Argument übergebenen Objekts zurück. Wenn das Objekt in der Liste nicht vorkommt, wird der Wert -1 zurückgegeben. In Fortsetzung des vorherigen Beispiels, wenn Sie anrufen:

```
strings.indexOf("This is a replacement")
```

Es wird erwartet, dass die 0 zurückgegeben wird, wenn wir den String "Dies ist ein Ersatz" an Position 0 unserer Liste setzen. Wenn es mehr als ein Vorkommen in der Liste gibt, wenn int indexOf(Object o); aufgerufen wird, dann wird wie erwähnt der Index des ersten Vorkommens zurückgegeben. Durch Aufruf des int lastIndexOf(Object o) Sie den Index des letzten Vorkommens in der Liste abrufen. Wenn wir also ein weiteres "Dies ist ein Ersatz" hinzufügen:

```
strings.add("This is a replacement");
strings.lastIndexOf("This is a replacement");
```

Diesmal wird die 1 zurückgegeben und nicht die 0;

Elemente in einer Liste durchlaufen

Nehmen wir als Beispiel an, wir haben eine Liste des Typs String, die vier Elemente enthält: "Hallo", "Wie", "Wie", "Sie", "Sie?"

Die beste Möglichkeit, jedes Element zu durchlaufen, ist die Verwendung einer for-each-Schleife:

```
public void printEachElement(List<String> list){
    for(String s : list){
        System.out.println(s);
    }
}
```

Was würde drucken:

```
hello,
how
are
you?
```

Um sie alle in derselben Zeile zu drucken, können Sie einen StringBuilder verwenden:

```
public void printAsLine(List<String> list){
    StringBuilder builder = new StringBuilder();
    for(String s : list){
        builder.append(s);
    }
    System.out.println(builder.toString());
}
```

Wird drucken:

```
hello, how are you?
```

Alternativ können Sie die Elementindizierung (wie unter Zugriff auf Element unter Index von ArrayList auf Element beschrieben) verwenden, um eine Liste zu durchlaufen. Warnung: Dieser Ansatz ist für verknüpfte Listen ineffizient.

Entfernen von Elementen aus der Liste B, die in der Liste A vorhanden sind

Nehmen wir an, Sie haben 2 Listen A und B, und Sie möchten alle Elemente, die Sie in **A** haben, von **B** entfernen. Die Methode in diesem Fall ist

```
 List.removeAll(Collection c);
```

#Beispiel:

```
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);
    }
```

Dies wird gedruckt

```
A: [1, 3, 4, 7, 5, 2]

B: [13, 32, 533, 3, 4, 2]

B gelöscht: [13, 32, 533]
```

Suche nach gemeinsamen Elementen zwischen 2 Listen

Angenommen, Sie haben zwei Listen: A und B, und Sie müssen die Elemente finden, die in beiden Listen vorhanden sind.

Sie können dies tun, indem Sie einfach die Methode List.retainAll() .

Beispiel:

```
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);

}
```

Konvertieren Sie eine Liste ganzer Zahlen in eine Liste von Zeichenfolgen

```
List<Integer> nums = Arrays.asList(1, 2, 3);
List<String> strings = nums.stream()
    .map(Object::toString)
    .collect(Collectors.toList());
```

Das ist:

1. Erstellen Sie einen Stream aus der Liste
2. Ordnen Sie jedes Element mithilfe von Object::toString
3. Sammeln Sie die String Werte mithilfe von Collectors.toList() in einer List

Element aus einer ArrayList erstellen, hinzufügen und entfernen

ArrayList ist eine der integrierten Datenstrukturen in Java. Es ist ein dynamisches Array (bei dem die Größe der Datenstruktur nicht zuerst deklariert werden musste) zum Speichern von Elementen (Objekten).

Es erweitert die AbstractList Klasse und implementiert die List Schnittstelle. Eine ArrayList kann doppelte Elemente enthalten, in denen die Einfügungsreihenfolge beibehalten wird. Es sollte beachtet werden, dass die Klasse ArrayList nicht synchronisiert ist. Beim Umgang mit Parallelität mit ArrayList ist daher Vorsicht geboten. ArrayList ermöglicht den Direktzugriff, da das Array auf der Indexbasis arbeitet. Die Bearbeitung in ArrayList ist langsam, da das Verschieben häufig auftritt, wenn ein Element aus der Array-Liste entfernt wird.

Eine ArrayList kann wie folgt erstellt werden:

```
List<T> myArrayList = new ArrayList<>();
```

Dabei ist T (Generics) der Typ, der in ArrayList gespeichert wird.

Der Typ der ArrayList kann ein beliebiges Objekt sein. Der Typ kann kein primitiver Typ sein (verwenden Sie stattdessen die *Wrapper-Klassen*).

Um der ArrayList ein Element hinzuzufügen, verwenden Sie die add() Methode:

```
myArrayList.add(element);
```

Oder um einen Artikel zu einem bestimmten Index hinzuzufügen:

```
myArrayList.add(index, element); //index of the element should be an int (starting from 0)
```

Um ein Element aus der ArrayList zu entfernen, verwenden Sie die remove() Methode:

```
myArrayList.remove(element);
```

Oder um ein Element aus einem bestimmten Index zu entfernen:

```
myArrayList.remove(index); //index of the element should be an int (starting from 0)
```

Direkter Austausch eines Listenelements

In diesem Beispiel wird ein List Element ersetzt, wobei sichergestellt wird, dass sich das Ersatzelement an derselben Position wie das Element befindet, das ersetzt wird.

Dies kann mit diesen Methoden geschehen:

- set (int index, T typ)
- int indexOf (T-Typ)

Betrachten Sie eine ArrayList mit den Elementen "Programm wird gestartet!", "Hallo Welt!" und "Auf Wiedersehen Welt!"

```
List<String> strings = new ArrayList<String>();
strings.add("Program starting!");
strings.add("Hello world!");
strings.add("Goodbye world!");
```

Wenn wir den Index des Elements kennen, das wir ersetzen möchten, können Sie set wie folgt verwenden:

```
strings.set(1, "Hi world");
```

Wenn wir den Index nicht kennen, können wir ihn zuerst suchen. Zum Beispiel:

```
int pos = strings.indexOf("Goodbye world!");
```

```
if (pos >= 0) {
    strings.set(pos, "Goodbye cruel world!");
}
```

Anmerkungen:

1. Die set Operation verursacht keine ConcurrentModificationException .
2. Die set Operation ist schnell (O(1)) für ArrayList aber langsam (O(N)) für eine LinkedList .
3. Eine indexOf Suche in einer ArrayList oder LinkedList ist langsam (O(N)).

Eine Liste unveränderlich machen

Die Collections-Klasse bietet eine Möglichkeit, eine Liste unveränderbar zu machen:

```
List<String> ls = new ArrayList<String>();
List<String> unmodifiableList = Collections.unmodifiableList(ls);
```

Wenn Sie eine unveränderbare Liste mit einem Element wünschen, können Sie Folgendes verwenden:

```
List<String> unmodifiableList = Collections.singletonList("Only string in the list");
```

Objekte in der Liste verschieben

Mit der Collections-Klasse können Sie Objekte in der Liste mit verschiedenen Methoden verschieben (ls ist die Liste):

Liste umkehren:

```
Collections.reverse(ls);
```

Rotierende Positionen von Elementen in einer Liste

Die Rotationsmethode erfordert ein ganzzahliges Argument. Dies ist, wie viele Punkte Sie entlang der Linie verschieben müssen. Ein Beispiel dafür ist unten:

```
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();
```

Dies wird gedruckt "Hallo, wie geht es dir?"

Elemente in einer Liste mischen

Mit der gleichen Liste oben können wir die Elemente in einer Liste mischen:

```
Collections.shuffle(ls);
```

Wir können ihm auch ein java.util.Random-Objekt geben, mit dem Objekte zufällig in Spots platziert werden:

```
Random random = new Random(12);
```

```
Collections.shuffle(ls, random);
```

Klassen zur Implementierung von List - Vor- und Nachteile

Die List Schnittstelle wird von verschiedenen Klassen implementiert. Jeder von ihnen hat seinen eigenen Weg, um ihn mit unterschiedlichen Strategien umzusetzen und verschiedene Vor- und Nachteile zu bieten.

Klassen, die List implementieren

Dies sind alle public Klassen in Java SE 8, die die Schnittstelle java.util.List implementieren:

1. **Abstrakte Klassen:**
   - AbstractList
   - AbstractSequentialList
2. **Konkrete Klassen:**
   - Anordnungsliste
   - Attributliste
   - CopyOnWriteArrayList
   - LinkedList
   - Rollenliste
   - RoleUnresolvedList
   - Stapel
   - Vektor

Vor- und Nachteile jeder Implementierung im Hinblick auf die Komplexität der Zeit

Anordnungsliste

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

ArrayList ist eine anpassbare Array-Implementierung der List-Schnittstelle. Wenn Sie die Liste in einem Array *speichern,* bietet *ArrayList* (zusätzlich zu den Methoden, die die *List*-Schnittstelle implementieren) Methoden zum Ändern der Größe des Arrays.

Initialisieren Sie ArrayList of Integer mit der Größe 100

```
List<Integer> myList = new ArrayList<Integer>(100); // Constructs an empty list with the
specified initial capacity.
```

- PROS:

Die Operationen size, isEmpty, ***get*** , ***set*** , iterator und listIterator werden in konstanter Zeit ausgeführt. Das Erhalten und Einstellen jedes Elements der Liste hat also die gleichen *Zeitkosten* :

```
int e1 = myList.get(0);  //   \
int e2 = myList.get(10); //    | => All the same constant cost => O(1)
myList.set(2,10);        //   /
```

- CONS:

Die Implementierung mit einem Array (statische Struktur) für das Hinzufügen von Elementen über die Größe des Arrays hinaus ist mit hohen Kosten verbunden, da für das gesamte Array eine neue

Zuordnung vorgenommen werden muss. Jedoch aus der Dokumentation :

> Die Hinzufügungsoperation wird in einer amortisierten konstanten Zeit ausgeführt, das heißt, das Hinzufügen von n Elementen erfordert O (n) Zeit

Das Entfernen eines Elements erfordert O (n) Zeit.

Attributliste

Auf kommen

CopyOnWriteArrayList

Auf kommen

LinkedList

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

LinkedList wird durch eine doppelt verknüpfte Liste implementiert, eine verknüpfte Datenstruktur, die aus einem Satz sequentiell verknüpfter Datensätze besteht, die als Knoten bezeichnet werden.

Initialisieren Sie LinkedList of Integer

```
List<Integer> myList = new LinkedList<Integer>(); // Constructs an empty list.
```

- PROS:

Das Hinzufügen oder Entfernen eines Elements vor oder nach dem Ende der Liste hat eine konstante Zeit.

```
myList.add(10);  // \
myList.add(0,2); //  | => constant time => O(1)
myList.remove(); // /
```

- CONS: Aus der Dokumentation :

> Operationen, die in die Liste aufgenommen werden, durchlaufen die Liste vom Anfang oder vom Ende, je nachdem, welcher Punkt näher am angegebenen Index liegt.

Operationen wie:

```
myList.get(10);    // \
myList.add(11,25); //  | => worst case done in O(n/2)
myList.set(15,35); // /
```

Rollenliste

Auf kommen

RoleUnresolvedList

Auf kommen

Stapel

Auf kommen

Vektor

Auf kommen

Listen online lesen: https://riptutorial.com/de/java/topic/2989/listen

Kapitel 106: Literale

Einführung

Ein Java-Literal ist ein syntaktisches Element (dh etwas, das Sie im *Quellcode* eines Java-Programms finden), das einen Wert darstellt. Beispiele sind 1 , 0.333F , false , 'X' und "Hello world\n" .

Examples

Hexadezimal-, Oktal- und Binärliterale

Eine hexadecimal ist ein Wert in Basis-16. Es gibt 16 Ziffern, 0-9 und die Buchstaben AF (Fall spielt keine Rolle). AF für 10-16 .

Eine octal ist ein Wert in Basis-8 und verwendet die Ziffern 0-7 .

Eine binary Zahl ist ein Wert in Basis-2 und verwendet die Ziffern 0 und 1 .

Alle diese Zahlen ergeben den gleichen Wert 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
```

Beachten Sie, dass die binäre Literal-Syntax in Java 7 eingeführt wurde.

Das Oktal-Literal kann leicht eine Falle für semantische Fehler sein. Wenn Sie eine führende '0' für Ihre Dezimal-Literale definieren, erhalten Sie den falschen Wert:

```
int a = 0100;        // Instead of 100, a == 64
```

Verwendung von Unterstreichungen zur Verbesserung der Lesbarkeit

Seit Java 7 ist es möglich, einen oder mehrere Unterstriche (_) zum Trennen von Zifferngruppen in einem primitiven Zahlenliteral zu verwenden, um deren Lesbarkeit zu verbessern.

Zum Beispiel sind diese beiden Erklärungen gleichwertig:

Java SE 7

```
int i1 = 123456;
int i2 = 123_456;
System.out.println(i1 == i2); // true
```

Dies kann wie folgt auf alle primitiven Zahlenliterale angewendet werden:

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;
```

Dies funktioniert auch mit Präfixen für binäre, oktale und hexadezimale Basen:

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;
```

Es gibt ein paar Regeln für Unterstriche, **die** ihre Platzierung an folgenden Stellen **verbieten** :

- Am Anfang oder Ende einer Nummer (zB _123 oder 123_ sind *nicht* gültig)
- Angrenzend an einen Dezimalpunkt in einem Fließkommaliteral (zB 1._23 oder 1_.23 sind *nicht* gültig)
- Vor einem Suffix von F oder L (zB 1.23_F oder 9999999_L sind *nicht* gültig)
- In Positionen, an denen eine Zeichenfolge erwartet wird (zB 0_xFFFF ist *nicht* gültig)

Escape-Sequenzen in Literalen

String- und Zeichenliterale bieten einen Escape-Mechanismus, der Express-Zeichencodes zulässt, die ansonsten im Literal nicht zulässig wären. Eine Escape-Sequenz besteht aus einem Backslash-Zeichen (\ ), gefolgt von einem oder mehreren anderen Zeichen. Die gleichen Sequenzen gelten für beide Zeichen- und Zeichenkettenliterale.

Der komplette Satz von Escape-Sequenzen lautet wie folgt:

Fluchtabfolge	Bedeutung
`\\`	Kennzeichnet einen umgekehrten Schrägstrich (`\`)
`\'`	Kennzeichnet ein Anführungszeichen (`'`)
`\"`	Kennzeichnet ein Anführungszeichen (`"`)
`\n`	Kennzeichnet ein Zeilenvorschubzeichen (`LF`)
`\r`	Kennzeichnet ein Wagenrücklaufzeichen (`CR`)
`\t`	Kennzeichnet ein horizontales Tabulatorzeichen (`HT`)
`\f`	Kennzeichnet ein Formularvorschubzeichen (`FF`)
`\b`	Kennzeichnet ein Backspace-Zeichen (`BS`)
`\<octal>`	Bezeichnet einen Zeichencode im Bereich von 0 bis 255.

Das <octal> im obigen Beispiel besteht aus einer, zwei oder drei Oktalstellen ('0' bis '7'), die eine Zahl zwischen 0 und 255 (dezimal) darstellen.

Beachten Sie, dass ein umgekehrter Schrägstrich gefolgt von einem anderen Zeichen eine ungültige Escape-Sequenz ist. Ungültige Escape-Sequenzen werden von der JLS als Kompilierungsfehler behandelt.

Referenz:

- JLS 3.10.6. Escape-Sequenzen für Zeichen- und Zeichenkettenliterale

Unicode entkommt

Zusätzlich zu den oben beschriebenen String- und Character-Escape-Sequenzen verfügt Java über einen allgemeineren Unicode-Escape-Mechanismus, wie in JLS 3.3 definiert . Unicode-Flucht Ein Unicode-Escape hat die folgende Syntax:

```
'\' 'u' <hex-digit> <hex-digit> <hex-digit> <hex-digit>
```

wobei <hex-digit> eine von '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' , '8' , '9' , 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'A' , 'B' , 'C' , 'D' , 'E' , 'F' .

Ein Unicode-Escape- *Code* wird vom Java-Compiler einem Zeichen zugeordnet (streng genommen eine 16-Bit-Unicode- *Codeeinheit*) und kann überall im Quellcode verwendet werden, wo das zugeordnete Zeichen gültig ist. Es wird häufig in Zeichen- und Zeichenkettenliteralen verwendet, wenn Sie ein Nicht-ASCII-Zeichen in einem Literal darstellen müssen.

Flucht in Regex

TBD

Dezimal-Integer-Literale

Integer-Literale bieten Werte, die verwendet werden können, wenn Sie eine byte , short , int , long oder char Instanz benötigen. (Dieses Beispiel konzentriert sich auf die einfachen Dezimalformen. In anderen Beispielen wird erläutert, wie Literale in Oktal-, Hexadezimal- und Binärform angegeben werden, und wie Unterstriche verwendet werden, um die Lesbarkeit zu verbessern.)

Gewöhnliche ganzzahlige Literale

Die einfachste und gebräuchlichste Form des Ganzzahl-Literal ist ein Dezimal-Ganzzahl-Literal. Zum Beispiel:

```
0       // The decimal number zero       (type 'int')
1       // The decimal number one        (type 'int')
42      // The decimal number forty two  (type 'int')
```

Sie müssen vorsichtig mit führenden Nullen sein. Eine führende Null bewirkt, dass ein Integer-Literal als *Oktal und* nicht als Dezimalzahl interpretiert wird.

```
077    // This literal actually means 7 x 8 + 7 ... or 63 decimal!
```

Ganzzahlige Literale sind nicht signiert. Wenn Sie etwas wie -10 oder +10 , sind dies eigentlich *Ausdrücke* , die die unären - und unären + -Operatoren verwenden.

Der Bereich der ganzzahligen Literale dieser Form hat einen intrinsischen Typ von int und muss im Bereich von 0 bis 2^{31} oder 2 147 483 648 liegen.

Beachten Sie, dass 2^{31} um 1 größer als Integer.MAX_VALUE . Literale von 0 bis zu 2147483647 kann überall eingesetzt werden, aber es ist ein Übersetzungsfehler verwenden 2147483648 ohne vorherigen einstellige - Operator. (Mit anderen Worten, es ist für die Angabe des Wertes von Integer.MIN_VALUE .)

```
int max = 2147483647;     // OK
int min = -2147483648;    // OK
int tooBig = 2147483648;  // ERROR
```

Lange ganzzahlige Literale

Literale vom Typ long werden durch Hinzufügen eines L Suffixes ausgedrückt. Zum Beispiel:

```
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
```

Beachten Sie, dass der Unterschied zwischen int und long Literalen an anderen Stellen signifikant ist. Zum Beispiel

```
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.
```

Referenz: JLS 3.10.1 - Integer Literals

Boolesche Literale

Boolesche Literale sind die einfachsten der Literale in der Programmiersprache Java. Die zwei möglichen boolean Werte werden durch die Literale true und false . Hierbei wird zwischen Groß- und Kleinschreibung unterschieden. Zum Beispiel:

```
boolean flag = true;    // using the 'true' literal
flag = false;           // using the 'false' literal
```

String-Literale

String-Literale bieten die bequemste Möglichkeit, String-Werte im Java-Quellcode darzustellen. Ein String-Literal besteht aus:

- Ein doppeltes Anführungszeichen (").
- Null oder mehrere andere Zeichen, die weder ein Anführungszeichen noch ein Zeilenumbruchzeichen sind. (Ein umgekehrter Schrägstrich (\ ) ändert die Bedeutung nachfolgender Zeichen; siehe Escape-Sequenzen in Literalen .)
- Ein schließendes Anführungszeichen.

Zum Beispiel:

```
"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
```

Beachten Sie, dass sich ein einzelnes String-Literal möglicherweise nicht über mehrere Quellcodezeilen erstreckt. Es ist ein Kompilierungsfehler, wenn ein Zeilenumbruch (oder das Ende der Quelldatei) vor dem schließenden Anführungszeichen eines Literal auftritt. Zum Beispiel:

```
"Jello world    // Compilation error (at the end of the line!)
```

Lange Saiten

Wenn Sie eine Zeichenfolge benötigen, die zu lang ist, um in eine Zeile zu passen, können Sie sie normalerweise in mehrere Literale aufteilen und den Verkettungsoperator (+) verwenden, um die Teile zu verbinden. Zum Beispiel

```
String typingPractice = "The quick brown fox " +
                        "jumped over " +
                        "the lazy dog"
```

Ein Ausdruck wie der oben genannte, der aus Zeichenkettenlitern und + erfüllt die Anforderungen, um ein konstanter Ausdruck zu sein . Das bedeutet, dass der Ausdruck vom Compiler ausgewertet und zur Laufzeit durch ein einzelnes String Objekt dargestellt wird.

Internierung von String-Literalen

Wenn Klassendatei Stringliterale enthält , durch die JVM geladen wird, werden die entsprechenden String sind Objekte , die von dem Laufzeitsystem *interniert*. Dies bedeutet, dass ein Zeichenfolgenliteral, das in mehreren Klassen verwendet wird, nicht mehr Speicherplatz beansprucht, als wenn es in einer Klasse verwendet würde.

Weitere Informationen zum Interning und zum String-Pool finden Sie im Beispiel zum String-Pool und zum Heap-Speicher im Thema Strings.

Das Null-Literal

Das Null-Literal (als null) repräsentiert den einzigen Wert des Null-Typs. Hier sind einige Beispiele

```
    MyClass object = null;
    MyClass[] objects = new MyClass[]{new MyClass(), null, new MyClass()};

    myMethod(null);

    if (objects != null) {
         // Do something
    }
```

Der Nulltyp ist eher ungewöhnlich. Es hat keinen Namen und kann daher nicht in Java-Quellcode ausgedrückt werden. (Und es hat auch keine Laufzeitdarstellung.)

Der einzige Zweck des Null - Typs ist der Typ zu sein , null . Es ist mit allen Referenztypen zuweisungskompatibel und kann in einen beliebigen Referenztyp umgewandelt werden. (Im letzteren Fall erfordert die Besetzung keine Überprüfung des Laufzeittyps.)

Schließlich hat null die Eigenschaft, dass die null instanceof <SomeReferenceType> des Typs als false ausgewertet wird.

Fließkomma-Literale

Fließkomma-Literale liefern Werte, die verwendet werden können, wenn Sie eine float oder double benötigen. Es gibt drei Arten von Gleitkomma-Literalen.

- Einfache Dezimalformen
- Skalierte Dezimalformen
- Hexadezimalformen

(Die JLS-Syntaxregeln kombinieren die beiden Dezimalformen in einer einzigen Form. Wir behandeln sie zur Vereinfachung der Erklärung getrennt.)

Es gibt verschiedene Literaltypen für float und double Literale, die mit Suffixen ausgedrückt werden. Die verschiedenen Formen verwenden Buchstaben, um verschiedene Dinge auszudrücken. Diese Buchstaben unterscheiden nicht zwischen Groß- und Kleinschreibung.

Einfache Dezimalformen

Die einfachste Form des Gleitpunktliteral besteht aus einer oder mehreren Dezimalstellen und einem Dezimalpunkt (.) Und einem optionalen Suffix (f , F , d oder D). Mit dem optionalen Suffix können Sie angeben, dass das Literal ein float (f oder F) oder ein double (d oder D) ist. Der Standardwert (wenn kein Suffix angegeben ist) ist double .

Zum Beispiel

```
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)
```

Dezimalstellen, gefolgt von einem Suffix, sind auch ein Fließkommaliteral.

```
1F      // means the same thing as 1.0F
```

Die Bedeutung eines Dezimalliteral ist die IEEE-Gleitkommazahl, die der mathematischen Unendlichkeitsgenauigkeit "Real" am *nächsten kommt* , die mit der dezimalen Gleitkommaform bezeichnet wird. Dieser Wert wird konzeptuelle Darstellung umgewandelt binäre Gleitkomma IEEE *Rund zum nächsten* verwendet wird . (Die genaue Semantik der Dezimalkonvertierung wird in den Javadocs für Double.valueOf(String) und Float.valueOf(String) , wobei zu berücksichtigen ist, dass Unterschiede in der Zahlensyntax bestehen.)

Skalierte Dezimalformen

Skalierte Dezimalformen bestehen aus einfachen Dezimalzahlen mit einem durch ein E oder e eingeführten Exponententeil, gefolgt von einer vorzeichenbehafteten Ganzzahl. Der Exponent-Teil ist eine kurze Hand zum Multiplizieren der Dezimalform mit einer Zehnerpotenz, wie in den folgenden Beispielen gezeigt. Es gibt auch ein optionales Suffix zur Unterscheidung von float und double . Hier sind einige Beispiele:

```
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)
```

Die Größe eines Literal wird durch die Darstellung (float oder double) begrenzt. Es ist ein Kompilierungsfehler, wenn der Skalierungsfaktor einen zu großen oder zu kleinen Wert ergibt.

Hexadezimalformen

Beginnend mit Java 6 ist es möglich, Fließkommaliter im Hexadezimalformat auszudrücken. Die Hexadezimalform hat eine analoge Syntax zu den einfachen und skalierten Dezimalformen mit den folgenden Unterschieden:

1. Jedes hexadezimale Fließkommaliteral beginnt mit einer Null (0) und dann einem x oder X
2. Die Ziffern der Zahl (aber *nicht* der Exponententeil!) Enthalten auch die hexadezimalen Ziffern a bis f und ihre Äquivalente in Großbuchstaben.
3. Der Exponent ist *obligatorisch* und wird durch den Buchstaben p (oder P) anstelle von e oder E . Der Exponent stellt einen Skalierungsfaktor dar, der eine Potenz von 2 anstelle einer Potenz von 10 ist.

Hier sind einige Beispiele:

```
0x0.0p0f    // this is zero expressed in hexadecimal form (`float`)
0xff.0p19   // this is 255.0 x 2^19 (`double`)
```

Hinweis: Da hexadezimale Fließkommaformulare für die meisten Java-Programmierer unbekannt sind, ist es ratsam, sie sparsam zu verwenden.

Unterstriche

Ab Java 7 sind in allen drei Formen des Fließkomma-Literales Unterstriche innerhalb der Ziffernfolgen zulässig. Dies gilt auch für die "Exponenten" -Teile. Siehe Unterstriche verwenden, um die Lesbarkeit zu verbessern .

Sonderfälle

Es ist ein Kompilierungsfehler, wenn ein Gleitkommaliteral eine Zahl angibt, die zu groß oder zu klein ist, um sie in der ausgewählten Darstellung darzustellen. dh wenn die Nummer zu + INF oder -INF überlaufen würde oder zu 0,0 unterlaufen würde. Es ist jedoch zulässig, dass ein Literal eine denormalisierte Zahl ungleich Null darstellt.

Die Fließkomma-Literal-Syntax bietet keine Literaldarstellungen für IEEE 754-Sonderwerte wie die INF- und NaN-Werte. Wenn Sie sie im Quellcode ausdrücken müssen, verwenden Sie die Konstanten java.lang.Float und java.lang.Double ; zB Float.NaN , Float.NEGATIVE_INFINITY und Float.POSITIVE_INFINITY .

Zeichenliterale

Zeichenliterale bieten die bequemste Möglichkeit, char im Java-Quellcode auszudrücken. Ein Zeichenliteral besteht aus:

- Ein anfängliches Anführungszeichen (').
- Eine Darstellung eines Charakters. Diese Darstellung kann kein einfaches Anführungszeichen oder Zeilenumbruchzeichen sein. Es kann sich jedoch um eine Escape-Sequenz handeln, die durch einen umgekehrten Schrägstrich (\ ) eingeführt wird. Siehe Escape-Sequenzen in Literalen .
- Ein abschließendes einfaches Anführungszeichen (').

Zum Beispiel:

```
char a = 'a';
char doubleQuote = '"';
char singleQuote = '\'';
```

Ein Zeilenumbruch in einem Zeichenliteral ist ein Kompilierungsfehler:

```
char newline = '
// Compilation error in previous line
char newLine = '\n';  // Correct
```

Literale online lesen: https://riptutorial.com/de/java/topic/8250/literale

Kapitel 107: log4j / log4j2

Einführung

Apache Log4j ist ein Java-basiertes Protokollierungsdienstprogramm. Es ist eines von mehreren Java-Protokollierungsframeworks. In diesem Thema wird gezeigt, wie Log4j in Java eingerichtet und konfiguriert wird, und enthält detaillierte Beispiele zu allen möglichen Aspekten der Verwendung.

Syntax

- Logger.debug ("zu protokollierender Text"); // Debugging-Informationen protokollieren
- Logger.info ("zu protokollierender Text"); // Allgemeine Informationen protokollieren
- Logger.error ("zu protokollierender Text"); // Fehlerinformationen protokollieren
- Logger.warn ("zu protokollierender Text"); // Warnungen protokollieren
- Logger.trace ("zu protokollierender Text"); // Trace-Informationen protokollieren
- Logger.fatal ("zu protokollierender Text"); // Protokollierung schwerwiegender Fehler
- Log4j2-Verwendung mit Parameterprotokollierung:
- Logger.debug ("Debug-Parameter {} {} {}", param1, param2, param3); // Protokollieren des Debuggens mit Parametern
- Logger.info ("Info-Parameter {} {} {}", param1, param2, param3); // Protokollieren von Informationen mit Parametern
- Logger.error ("Fehlerparameter {} {} {}", param1, param2, param3); // Protokollierungsfehler mit Parametern
- Logger.warn ("Parameter warnen {} {} {}", Parameter1, Parameter2, Parameter3); // Warnungen mit Parametern protokollieren
- Logger.trace ("Trace-Parameter {} {} {}", param1, param2, param3); // Ablaufverfolgung mit Parametern protokollieren
- Logger.fatal ("fatale Parameter {} {} {}", param1, param2, param3); // Protokollierung mit Parametern fatal
- Logger.error ("Ausnahme beim Abfangen:", ex); // Ausnahme mit Protokollierung und Stacktrace protokollieren (wird automatisch angehängt)

Bemerkungen

End of Life für Log4j 1 erreicht

> Am 5. August 2015 gab das Logging Services Project Management Committee bekannt, dass Log4j 1.x das Ende seiner Lebensdauer erreicht hat. Den vollständigen Text der Ankündigung finden Sie im Apache-Blog. **Benutzern von Log4j 1 wird empfohlen, auf Apache Log4j 2 zu aktualisieren .**

Von: http://logging.apache.org/log4j/1.2/

Examples

Wie erhalte ich Log4j?

Aktuelle Version (log4j2)

Maven benutzen:
Fügen Sie Ihrer POM.xml Datei die folgende Abhängigkeit 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>
```

Ivy verwenden:

```
<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>
```

Verwenden von 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'
}
```

Log4j 1.x bekommen

Hinweis: Log4j 1.x hat das End-of-Life (EOL) erreicht (siehe Anmerkungen).

Maven benutzen:

Deklarieren Sie diese Abhängigkeit in der POM.xml Datei:

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

Ivy verwenden:

```
<dependency org="log4j" name="log4j" rev="1.2.17"/>
```

Usign Gradle:

```
compile group: 'log4j', name: 'log4j', version: '1.2.17'
```

Buildr verwenden:

```
'log4j:log4j:jar:1.2.17'
```

Manuelles Hinzufügen in der Pfaderstellung:

Laden Sie es vom Log4j- Website-Projekt herunter

So verwenden Sie Log4j in Java-Code

Zuerst müssen Sie ein final static logger erstellen:

```
final static Logger logger = Logger.getLogger(classname.class);
```

Rufen Sie dann Protokollierungsmethoden auf:

```
//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);
```

Eigenschaftendatei einrichten

Log4j gibt Ihnen die Möglichkeit, Daten gleichzeitig in Konsole und Datei einzuloggen. Erstellen Sie eine log4j.properties Datei und log4j.properties diese Basiskonfiguration ein:

```
# 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
```

Wenn Sie Maven verwenden, geben Sie diese Eigenschaftendatei in den Pfad ein:

```
/ProjectFolder/src/java/resources
```

Grundlegende Konfigurationsdatei für 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>
```

Dies ist eine grundlegende Konfiguration von log4j2.xml mit einem Konsolen-Appender und einem Root-Logger. Das Musterlayout gibt an, welches Muster für die Protokollierung der Anweisungen verwendet werden soll.
Um das Laden von log4j2.xml zu debuggen, können Sie das Attribut status = <WARN | DEBUG | ERROR | FATAL | TRACE | INFO> im Konfigurations-Tag Ihrer log4j2.xml.
Sie können auch ein Überwachungsintervall hinzufügen, damit die Konfiguration nach dem angegebenen Intervall erneut geladen wird. Das Überwachungsintervall kann wie folgt zum Konfigurationstag hinzugefügt werden: monitorInterval = 30 . Das bedeutet, dass die Konfiguration alle 30 Sekunden geladen wird.

Migration von log4j 1.x nach 2.x

Wenn Sie aus Ihrem vorhandenen log4j 1.x-System zu log4j 2.x migrieren möchten, entfernen Sie alle vorhandenen log4j 1.x-Abhängigkeiten und fügen Sie die folgende Abhängigkeit hinzu:

Log4j 1.x API Bridge

Maven Build

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-1.2-api</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

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 Wenn in Ihrem Projekt Apache Commons Logging verwendet wird, das log4j 1.x verwendet und Sie es auf log4j 2.x migrieren möchten, fügen Sie die folgenden Abhängigkeiten hinzu:

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'
}
```

Hinweis: Entfernen Sie keine vorhandenen Abhängigkeiten der Apache-Commons-Protokollierung

Referenz: https://logging.apache.org/log4j/2.x/maven-artifacts.html

Eigenschaften-Datei zur Anmeldung an der DB

Für dieses Beispiel benötigen Sie einen JDBC-Treiber, der mit dem System kompatibel ist, auf dem die Datenbank ausgeführt wird. Eine Open Source-Ressource , mit der Sie eine Verbindung zu DB2-Datenbanken auf einem IBM System i herstellen können, finden Sie hier: JT400

Obwohl dieses Beispiel DB2-spezifisch ist, funktioniert es für fast jedes andere System, wenn Sie den Treiber austauschen und die JDBC-URL anpassen.

```
# 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
```

Logoutput nach Stufe filtern (log4j 1.x)

Sie können einen Filter verwenden , um nur Nachrichten „tiefer" als zB log ERROR Ebene. **Der Filter wird jedoch nicht von PropertyConfigurator unterstützt. Daher müssen Sie zur Verwendung der XML-Konfiguration wechseln** . Siehe log4j-Wiki zu Filtern .

Beispiel "spezifische Ebene"

```
<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>
```

Oder "Pegelbereich"

```
<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>
```

log4j / log4j2 online lesen: https://riptutorial.com/de/java/topic/2472/log4j---log4j2

Kapitel 108: Lokale innere Klasse

Einführung

Eine Klasse, z. B. innerhalb einer Methode erstellt, wird in Java als lokale innere Klasse bezeichnet. Wenn Sie die Methoden der lokalen inneren Klasse aufrufen möchten, müssen Sie diese Klasse in der Methode instanziieren.

Examples

Lokale innere Klasse

```
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();
 }
}
```

Lokale innere Klasse online lesen: https://riptutorial.com/de/java/topic/10160/lokale-innere-klasse

Kapitel 109: Lokalisierung und Internationalisierung

Bemerkungen

Java verfügt über einen leistungsstarken und flexiblen Mechanismus zum Lokalisieren Ihrer Anwendungen. Es ist jedoch auch leicht zu missbrauchen und mit einem Programm zu beenden, das das Gebietsschema des Benutzers außer Acht lässt oder das Verhalten des Programms davon abhängig macht.

Ihre Benutzer werden erwarten, dass Daten in den von ihnen gewohnten Formaten lokalisiert werden. Wenn Sie versuchen, dies manuell zu unterstützen, ist dies eine dumme Angelegenheit. Hier ist nur ein kleines Beispiel für die verschiedenen Arten, wie Benutzer erwarten, Inhalte zu sehen, von denen Sie annehmen, dass sie "immer" auf eine bestimmte Weise angezeigt werden:

	Termine	Zahlen	Landeswährung	Fremdwährung	Entfernungen
Brasilien					
China					
Ägypten					
Mexiko	20/3/16	1.234,56	1.000,50 $	1.000,50 USD	
Vereinigtes Königreich	20/3/16	1,234,56	1.000,50 £		100 km
Vereinigte Staaten von Amerika	3/20/16	1,234,56	1.000,50 $	1.000,50 MXN	60 mi

Allgemeine Ressourcen

- Wikipedia: Internationalisierung und Lokalisierung

Java-Ressourcen

- Java-Tutorial: Internationalisierung
- Oracle: Internationalisierung: Lokalisierung in der Java-Plattform
- JavaDoc: Locale

Examples

Automatisch formatierte Daten mit "locale"

SimpleDateFormatter ist prima, aber wie der Name schon sagt, skaliert es nicht gut.

Wenn Sie in Ihrer gesamten Anwendung "MM/dd/yyyy" Ihre internationalen Benutzer nicht zufrieden sein.

Lassen Sie Java die Arbeit für Sie erledigen

Verwenden Sie die static Methoden in DateFormat , um die richtige Formatierung für Ihren Benutzer abzurufen. Rufen Sie für eine Desktopanwendung (bei der Sie sich auf das Standardgebietsschema verlassen) einfach an:

```
String localizedDate = DateFormat.getDateInstance(style).format(date);
```

Dabei ist style eine der Formatierungskonstanten (FULL , LONG , MEDIUM , SHORT usw.), die in DateFormat angegeben DateFormat .

Für eine serverseitige Anwendung, in der der Benutzer als Teil der Anforderung das Gebietsschema angibt, sollten Sie es stattdessen explizit an getDateInstance() :

```
String localizedDate =
    DateFormat.getDateInstance(style, request.getLocale()).format(date);
```

Stringvergleich

Vergleichen Sie zwei Strings, die den Fall ignorieren:

```
"School".equalsIgnoreCase("school"); // true
```

Nicht verwenden

```
text1.toLowerCase().equals(text2.toLowerCase());
```

Sprachen haben unterschiedliche Regeln für die Konvertierung von Groß- und Kleinschreibung. Ein "Ich" würde auf Englisch in "Ich" umgewandelt. Aber auf Türkisch wird ein Ich zu einem I. Wenn Sie verwenden müssen toLowerCase() verwenden , um die Überlastung , die einen erwartet Locale : String.toLowerCase(Locale) .

Beim Vergleich zweier Strings werden geringfügige Unterschiede ignoriert:

```
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.PRIMARY);
collator.equals("Gärten", "gaerten"); // returns true
```

Sortieren Sie Strings unter Berücksichtigung der Reihenfolge der natürlichen Sprache und ignorieren Sie die Groß- / Kleinschreibung (verwenden Sie den Kollatierungsschlüssel, um:

```
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"}
```

Gebietsschema

Die Klasse java.util.Locale wird verwendet, um eine "geografische, politische oder kulturelle" Region zu repräsentieren, um einen bestimmten Text, eine bestimmte Anzahl, ein bestimmtes Datum oder eine bestimmte Operation zu lokalisieren. Ein Gebietsschema-Objekt kann somit ein Land, eine Region, eine Sprache und auch eine Variante einer Sprache enthalten, beispielsweise einen Dialekt, der in einer bestimmten Region eines Landes oder in einem anderen Land als dem Land, aus dem die Sprache stammt, gesprochen wird.

Die Locale-Instanz wird an Komponenten übergeben, die ihre Aktionen lokalisieren müssen, unabhängig davon, ob sie die Eingabe oder Ausgabe konvertieren oder nur für interne Vorgänge benötigen. Die Locale-Klasse kann selbst keine Internationalisierung oder Lokalisierung durchführen

Sprache

Die Sprache muss ein 2- oder 3-stelliger Sprachcode nach ISO 639 oder ein registrierter Sprachuntertitel mit bis zu 8 Zeichen sein. Wenn eine Sprache sowohl aus zwei als auch aus drei

Zeichen besteht, verwenden Sie den aus zwei Zeichen bestehenden Code. Eine vollständige Liste der Sprachcodes finden Sie in der IANA Language Subtag Registry.

Bei Sprachcodes wird die Groß- und Kleinschreibung nicht beachtet, die Locale-Klasse verwendet jedoch immer Versionen der Sprachcodes in Kleinbuchstaben

Ein Gebietsschema erstellen

Das Erstellen einer java.util.Locale Instanz kann auf vier verschiedene Arten erfolgen:

```
Locale constants
Locale constructors
Locale.Builder class
Locale.forLanguageTag factory method
```

Java ResourceBundle

Sie erstellen eine ResourceBundle-Instanz wie folgt:

```
Locale locale = new Locale("en", "US");
ResourceBundle labels = ResourceBundle.getBundle("i18n.properties");
System.out.println(labels.getString("message"));
```

i18n.properties ich habe eine Eigenschaftendatei i18n.properties :

```
message=This is locale
```

Ausgabe:

```
This is locale
```

Gebietsschema einstellen

Wenn Sie den Status mit anderen Sprachen reproduzieren möchten, können Sie die **setDefault()** - Methode verwenden. Seine Verwendung:

```
 setDefault(Locale.JAPANESE); //Set Japanese
```

Lokalisierung und Internationalisierung online lesen:
https://riptutorial.com/de/java/topic/4086/lokalisierung-und-internationalisierung

Kapitel 110: Methoden der Sammlungsfabrik

Einführung

Die Einführung von Java 9 bringt viele neue Funktionen in die Collections-API von Java, von denen eine die Methoden der Sammlungsfabrik darstellt. Diese Methoden ermöglichen die einfache Initialisierung **unveränderlicher** Sammlungen, unabhängig davon, ob sie leer oder nicht leer sind.

Beachten Sie, dass diese Factory-Methoden nur für die folgenden Schnittstellen verfügbar sind: `List<E>` , `Set<E>` und `Map<K, V>`

Syntax

- `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)`

Parameter

Methode mit Parameter	Beschreibung
`List.of(E e)`	Ein generischer Typ, der eine Klasse oder Schnittstelle sein kann.
`Set.of(E e)`	Ein generischer Typ, der eine Klasse oder Schnittstelle sein kann.
`Map.of(K k, V v)`	Ein Schlüsselwertpaar generischer Typen, die jeweils eine Klasse oder Schnittstelle sein können.
`Map.of(Map.Entry<? extends K, ? extends V> entry)`	Eine `Map.Entry` Instanz, bei der der Schlüssel `K` oder eines seiner `Map.Entry` und der Wert `V` oder eines seiner `Map.Entry` sein kann.

Examples

Liste Factory Method Beispiele

- `List<Integer> immutableEmptyList = List.of();`
  - Initialisiert eine leere, unveränderliche `List<Integer>` .
- `List<Integer> immutableList = List.of(1, 2, 3, 4, 5);`
  - Initialisiert eine unveränderliche `List<Integer>` mit fünf Anfangselementen.

- `List<Integer> mutableList = new ArrayList<>(immutableList);`
    - Initialisiert eine veränderliche `List<Integer>` aus einer unveränderlichen `List<Integer>` .

einstellen Factory Method Beispiele

- `Set<Integer> immutableEmptySet = Set.of();`
    - Initialisiert einen leeren, unveränderlichen `Set<Integer>` .
- `Set<Integer> immutableSet = Set.of(1, 2, 3, 4, 5);`
    - Initialisiert ein unveränderliches `Set<Integer>` mit fünf Anfangselementen.
- `Set<Integer> mutableSet = new HashSet<>(immutableSet);`
    - Initialisiert einen veränderbaren `Set<Integer>` aus einem unveränderlichen `Set<Integer>` .

Karte Factory Method Beispiele

- `Map<Integer, Integer> immutableEmptyMap = Map.of();`
    - Initialisiert eine leere, unveränderliche `Map<Integer, Integer>` .
- `Map<Integer, Integer> immutableMap = Map.of(1, 2, 3, 4);`
    - Initialisiert eine unveränderliche `Map<Integer, Integer>` mit zwei anfänglichen Schlüsselwerteinträgen.
- `Map<Integer, Integer> immutableMap = Map.ofEntries(Map.entry(1, 2), Map.entry(3, 4));`
    - Initialisiert eine unveränderliche `Map<Integer, Integer>` mit zwei anfänglichen Schlüsselwerteinträgen.
- `Map<Integer, Integer> mutableMap = new HashMap<>(immutableMap);`
    - Initialisiert eine veränderliche `Map<Integer, Integer>` aus einer unveränderlichen `Map<Integer, Integer>` .

Methoden der Sammlungsfabrik online lesen: https://riptutorial.com/de/java/topic/9783/methoden-der-sammlungsfabrik

Kapitel 111: Module

Syntax

- erfordert java.xml;
- erfordert öffentliche java.xml; # macht das Modul zur Verwendung an abhängige Personen
- exportiert com.example.foo; # abhängige Personen können öffentliche Typen in diesem Paket verwenden
- exportiert com.example.foo.impl nach com.example.bar; # Nutzung auf ein Modul beschränken

Bemerkungen

Die Verwendung von Modulen wird empfohlen, ist jedoch nicht erforderlich. Dadurch kann der vorhandene Code in Java 9 weiterarbeiten. Er ermöglicht auch einen schrittweisen Übergang zum modularen Code.

Nicht-modularer Code wird beim Kompilieren in ein nicht *benanntes Modul* eingefügt. Dies ist ein spezielles Modul, das Typen aus allen anderen Modulen verwenden kann, jedoch **nur aus Paketen, die eine exports** .

Alle Pakete im *unbenannten Modul* werden automatisch exportiert.

Schlüsselwörter, z. B. module usw., sind in der Moduldeklaration eingeschränkt, können aber an anderer Stelle weiterhin als Bezeichner verwendet werden.

Examples

Grundmodul definieren

Module werden in einer Datei namens module-info.java , die als module-info.java wird. Es sollte sich im Quellcode-Stammverzeichnis befinden:

```
|-- module-info.java
|-- com
    |-- example
        |-- foo
            |-- Foo.java
        |-- bar
            |-- Bar.java
```

Hier ist eine einfache Modulbeschreibung:

```
module com.example {
    requires java.httpclient;
    exports com.example.foo;
}
```

Der Modulname sollte eindeutig sein. Es wird empfohlen, dass Sie dieselbe Reverse-DNS-Benennungsnotation verwenden , die von Paketen verwendet wird, um dies sicherzustellen.

Das Modul java.base , das die grundlegenden Klassen von Java enthält, ist implizit für jedes Modul sichtbar und muss nicht eingeschlossen werden.

Die requires Erklärung ermöglicht es uns , weitere Module zu verwenden, im Beispiel das Modul java.httpclient importiert wird.

Ein Modul kann auch angeben, welche Pakete exports und macht es somit für andere Module sichtbar.

Das Paket com.example.foo in der deklarierten exports Klausel zu anderen Modulen sichtbar sein. Etwaige Unterpakete von com.example.foo nicht exportiert werden, sie ihre eigenen benötigen export Erklärungen.

Umgekehrt com.example.bar , die nicht in aufgeführt wird exports Klauseln werden nicht an andere Module sichtbar.

Module online lesen: https://riptutorial.com/de/java/topic/5286/module

Kapitel 112: Nashorn-JavaScript-Engine

Einführung

Nashorn ist eine JavaScript-Engine, die in Java von Oracle entwickelt wurde und mit Java 8 veröffentlicht wurde. Nashorn ermöglicht das Einbetten von Javascript in Java-Anwendungen über den JSR-223, die Entwicklung eigenständiger Javascript-Anwendungen, eine bessere Laufzeitleistung und eine bessere Einhaltung der ECMA normalisierte Javascript-Spezifikation.

Syntax

- ScriptEngineManager // Stellt einen Erkennungs- und Installationsmechanismus für ScriptEngine-Klassen bereit. verwendet eine SPI (Service Provider Interface)
- ScriptEngineManager.ScriptEngineManager () // Empfohlener Konstruktor
- ScriptEngine // Stellt die Schnittstelle zur Skriptsprache bereit
- ScriptEngine ScriptEngineManager.getEngineByName (String shortName) // Factory-Methode für die angegebene Implementierung
- Object ScriptEngine.eval (String-Skript) // Führt das angegebene Skript aus
- Object ScriptEngine.eval (Reader) // Lädt ein Skript aus der angegebenen Quelle und führt es aus
- ScriptContext ScriptEngine.getContext () // Gibt den Standardanbieter für Bindungen, Leser und Schreiber zurück
- void ScriptContext.setWriter (Writer Writer) // Legt das Ziel fest, an das die Skriptausgabe gesendet werden soll

Bemerkungen

Nashorn ist eine JavaScript-Engine, die in Java geschrieben und in Java 8 enthalten ist. Alles, was Sie brauchen, ist im Paket javax.script enthalten.

Beachten Sie, dass der ScriptEngineManager eine generische API bietet, mit der Sie Skriptmodule für verschiedene Skriptsprachen (dh nicht nur Nashorn, nicht nur JavaScript) erhalten.

Examples

Legen Sie globale Variablen fest

```
// 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
```

Hallo 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
```

Führen Sie die JavaScript-Datei aus

```
// 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!');
```

Skriptausgabe abfangen

```
// 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!'
```

Berechnen Sie arithmetische Zeichenfolgen

```
// 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.
```

Verwendung von Java-Objekten in JavaScript in Nashorn

Es ist möglich, Java-Objekte zur Verarbeitung in Java-Code an die Nashorn-Engine zu übergeben. Gleichzeitig gibt es einige JavaScript- (und Nashorn-) spezifische Konstruktionen, und es ist nicht immer klar, wie sie mit Java-Objekten arbeiten.

Nachfolgend finden Sie eine Tabelle, die das Verhalten nativer Java-Objekte in JavaScript-Konstruktionen beschreibt.

Getestete Konstruktionen:

1. Ausdruck in if-Klausel. Im JS-Ausdruck in if-Klauseln muss im Gegensatz zu Java kein boolescher Wert sein. Es wird als False für sogenannte Falschwerte (Null, undefiniert, 0, leere Zeichenfolgen usw.) ausgewertet.
2. Für jede Anweisung hat Nashorn eine spezielle Art von Schleife - für jede -, die verschiedene JS- und Java-Objekte durchlaufen kann.
3. Objektgröße abrufen In JS-Objekten haben Eigenschaften eine Länge, die die Größe eines Arrays oder einer Zeichenfolge zurückgibt.

Ergebnisse:

Art	Ob	für jeden	.Länge
Java null	falsch	Keine Iterationen	**Ausnahme**
Java leere Zeichenfolge	falsch	Keine Iterationen	0
Java-String	wahr	Durchläuft Zeichenfolgen	Länge der Zeichenfolge

Art	Ob	für jeden	.Länge
Java Integer / Long	Wert! = 0	Keine Iterationen	nicht definiert
Java ArrayList	wahr	Iteriert über Elemente	Länge der Liste
Java HashMap	wahr	Iteriert über Werte	Null
Java HashSet	wahr	Iteriert über Elemente	nicht definiert

Empfehlungen:

- Verwenden Sie if (some_string) , um zu überprüfen, ob eine Zeichenfolge nicht null und nicht leer ist
- for each kann sicher verwendet werden, um über eine Sammlung zu iterieren, und es werden keine Ausnahmen ausgelöst, wenn die Sammlung nicht iterierbar, null oder nicht definiert ist
- Bevor die Länge eines Objekts abgerufen werden kann, muss es auf null oder undefined geprüft werden (dasselbe gilt für jeden Versuch, eine Methode aufzurufen oder eine Eigenschaft eines Java-Objekts abzurufen)

Implementierung einer Schnittstelle aus einem Skript

```
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
    }
}
```

Globale Variablen setzen und abrufen

```
// 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
```

Nashorn-JavaScript-Engine online lesen: https://riptutorial.com/de/java/topic/166/nashorn-javascript-engine

Kapitel 113: Native Java-Schnittstelle

Parameter

Parameter	Einzelheiten
JNIEnv	Zeiger auf die JNI-Umgebung
Jobobjekt	Das Objekt, das die nicht `static native` Methode aufgerufen hat
jclass	Die Klasse, die die `static native` Methode aufgerufen hat

Bemerkungen

Zum Einrichten von JNI sind sowohl ein Java-Compiler als auch ein systemeigener Compiler erforderlich. Je nach IDE und Betriebssystem sind einige Einstellungen erforderlich. Eine Anleitung für Eclipse finden Sie hier . Eine vollständige Anleitung finden Sie hier .

Dies sind die Schritte zum Einrichten der Java-C ++ - Verknüpfung unter Windows:

- Kompilieren Sie die Java-Quelldateien (.java) mit javac in Klassen (.class).
- Erstellen Sie Header-Dateien (.h) aus den Java-Klassen, die native Methoden enthalten, mit javah . Diese Dateien "weisen" den nativen Code an, für welche Methoden er verantwortlich ist.
- Fügen Sie die Header - Dateien (#include) in der C ++ Quelldateien (.cpp) die Umsetzung native Methoden.
- Kompilieren Sie die C ++ - Quelldateien und erstellen Sie eine Bibliothek (.dll). Diese Bibliothek enthält die native Code-Implementierung.
- Geben Sie den Bibliothekspfad (-Djava.library.path) an und laden Sie ihn in die Java-Quelldatei (System.loadLibrary(...)).

Callbacks (Aufruf von Java-Methoden aus nativem Code) erfordert die Angabe eines Methodendeskriptors. Wenn der Deskriptor falsch ist, tritt ein Laufzeitfehler auf. Aus diesem Grund ist es hilfreich, die Deskriptoren für uns erstellen zu javap -s . Dies kann mit javap -s .

Examples

C ++ - Methoden von Java aus aufrufen

Statische und Member-Methoden in Java können als *native* markiert werden , um anzuzeigen, dass ihre Implementierung in einer gemeinsam genutzten Bibliotheksdatei zu finden ist. Bei der Ausführung einer systemeigenen Methode sucht die JVM nach einer entsprechenden Funktion in geladenen Bibliotheken (siehe Laden von systemeigenen Bibliotheken), verwendet ein einfaches Namensveränderungsschema, führt die Argumentkonvertierung und das Stack-Setup durch und übergibt die Kontrolle an den nativen Code.

Java-Code

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

C ++ - Code

Header-Dateien, die native Funktionsdeklarationen enthalten, sollten mit dem javah Tool für javah generiert werden. Führen Sie den folgenden Befehl im Build-Verzeichnis aus:

```
javah -o com_example_jni_JNIJava.hpp com.example.jni.JNIJava
```

... erzeugt die folgende Header-Datei (aus *Gründen der Kürze entfernte Kommentare*):

```
// 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
```

Hier ist eine Beispielimplementierung:

```
// 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;
}
```

Ausgabe

Wenn Sie die obige Beispielklasse ausführen, erhalten Sie die folgende Ausgabe:

```
C ++ 'printString' von Java aufgerufen
Ergebnis aus C ++ "Durchschnitt": 3.4
```

Java-Methoden von C ++ aus aufrufen (Callback)

Das Aufrufen einer Java-Methode aus nativem Code erfolgt in zwei Schritten:

1. einen Methodenzeiger mit der GetMethodID JNI-Funktion erhalten, wobei der Methodenname und der Deskriptor verwendet werden;
2. Rufen Sie eine der hier aufgeführten Call*Method Funktionen auf.

Java-Code

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

C ++ - Code

```
// 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);
}
```

Ausgabe

```
Erhielt Float aus C ++: 5.221
Bekam Int von C ++: 17
```

Den Deskriptor bekommen

Deskriptoren (oder *interne* **Typunterschriften**) werden mit dem **javap-** Programm der kompilierten .class Datei abgerufen. Hier ist die Ausgabe von 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
}
```

Laden nativer Bibliotheken

Das übliche Idiom zum Laden von gemeinsam genutzten Bibliotheksdateien in Java lautet wie folgt:

```
public class ClassWithNativeMethods {
    static {
```

```
        System.loadLibrary("Example");
    }

    public native void someNativeMethod(String arg);
    ...
```

Aufrufe von System.loadLibrary sind fast immer statisch, sodass sie beim Laden der Klasse auftreten. System.loadLibrary wird sichergestellt, dass keine native Methode ausgeführt werden kann, bevor die gemeinsam genutzte Bibliothek geladen wurde. Folgendes ist jedoch möglich:

```
public class ClassWithNativeMethods {
    // Call this before using any native method
    public static void prepareNativeMethods() {
        System.loadLibrary("Example");
    }

    ...
```

Dies ermöglicht, das Laden der gemeinsam genutzten Bibliothek zu verschieben, bis dies erforderlich ist, erfordert jedoch besondere Vorsicht, um java.lang.UnsatisfiedLinkError s zu vermeiden.

Zieldatei-Lookup

java.library.path gemeinsam genutzten Bibliotheksdateien wird in den Pfaden gesucht, die in der java.library.path definiert sind. java.library.path können mit dem Argument -Djava.library.path= JVM zur Laufzeit überschrieben werden:

```
java -Djava.library.path=path/to/lib/:path/to/other/lib MainClassWithNativeMethods
```

Achten Sie auf Systempfadtrennzeichen: zum Beispiel für Windows ; anstelle von :

Beachten Sie, dass System.loadLibrary Bibliotheksdateinamen System.loadLibrary auflöst: Der obige Code-Snippet erwartet unter Linux eine Datei namens libExample.so und Example.dll Windows Example.dll .

Eine Alternative zu System.loadLibrary ist System.load(String) , die den vollständigen Pfad zu einer gemeinsam genutzten Bibliotheksdatei verwendet und die Suche nach java.library.path umgeht:

```
public class ClassWithNativeMethods {
    static {
        System.load("/path/to/lib/libExample.so");
    }

    ...
```

Native Java-Schnittstelle online lesen: https://riptutorial.com/de/java/topic/168/native-java-schnittstelle

Kapitel 114: Neue Datei-E / A

Syntax

- Paths.get (String first, String ... mehr) // Erstellt eine Path-Instanz anhand ihrer String-Elemente
- Paths.get (URI uri) // Erstellt eine Pfadinstanz über einen URI

Examples

Pfade erstellen

Die Path Klasse wird verwendet, um einen Pfad im Dateisystem zu programmieren (und kann daher auf Dateien sowie Verzeichnisse verweisen, auch auf nicht vorhandene).

Ein Pfad kann mit der Hilfsklasse Paths abgerufen werden:

```
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");
```

Informationen über einen Pfad abrufen

Informationen zu einem Pfad können mit den Methoden eines Path Objekts Path werden:

- toString() gibt die String-Darstellung des Pfads zurück

  ```
  Path p1 = Paths.get("/var/www"); // p1.toString() returns "/var/www"
  ```

- getFileName() gibt den Dateinamen (oder genauer das letzte Element des Pfads) zurück

  ```
  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() gibt die Anzahl der Elemente zurück, die den Pfad bilden

  ```
  Path p1 = Paths.get("/var/www"); // p1.getNameCount() returns 2
  ```

- getName(int index) gibt das Element am angegebenen Index zurück

  ```
  Path p1 = Paths.get("/var/www"); // p1.getName(0) returns "var", p1.getName(1) returns
  "www"
  ```

- getParent() gibt den Pfad des übergeordneten Verzeichnisses zurück

  ```
  Path p1 = Paths.get("/var/www"); // p1.getParent().toString() returns "/var"
  ```

- getRoot() gibt die Wurzel des Pfads zurück

  ```
  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:\\"
```

Pfade manipulieren

Zwei Wege verbinden

Pfade können mit der Methode resolve() werden. Der übergebene Pfad muss ein Teilpfad sein. Dies ist ein Pfad, der das Stammelement nicht enthält.

```
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"
```

Pfad normalisieren

Pfade können die Elemente enthalten . (was auf das aktuelle Verzeichnis verweist) und .. (das auf das übergeordnete Verzeichnis zeigt).

Bei Verwendung in einem Pfad . kann jederzeit entfernt werden, ohne das Ziel des Pfades zu ändern, und .. kann zusammen mit dem vorhergehenden Element entfernt werden.

Bei der Paths-API erfolgt dies mit der Methode .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"
```

Informationen über das Dateisystem abrufen

Um mit dem Dateisystem zu interagieren, verwenden Sie die Methoden der Klasse Files .

Existenz prüfen

Um das Vorhandensein der Datei oder des Verzeichnisses zu überprüfen, auf das ein Pfad verweist, verwenden Sie die folgenden Methoden:

```
Files.exists(Path path)
```

und

```
Files.notExists(Path path)
```

!Files.exists(path) muss nicht notwendigerweise mit Files.notExists(path) , da es drei mögliche Szenarien gibt:

- Der Datei oder das Verzeichnis Existenz überprüft wird (exists kehrt true und notExists gibt false in diesem Fall)
- Die Nichtexistenz einer Datei oder eines Verzeichnisses wird überprüft (Vorhanden exists false und notExists gibt true)

- Weder die Existenz noch die Nichtexistenz einer Datei oder ein Verzeichnis kann (aufgrund von Zugangsbeschränkungen zum Beispiel) überprüft werden: Sowohl exists und nonExists false zurück.

Prüfen, ob ein Pfad auf eine Datei oder ein Verzeichnis verweist

Dies erfolgt mithilfe von Files.isDirectory(Path path) und 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
```

Eigenschaften erhalten

Dies kann mit den folgenden Methoden erfolgen:

```
Files.isReadable(Path path)
Files.isWritable(Path path)
Files.isExecutable(Path path)

Files.isHidden(Path path)
Files.isSymbolicLink(Path path)
```

MIME-Typ abrufen

```
Files.probeContentType(Path path)
```

Dadurch wird versucht, den MIME-Typ einer Datei abzurufen. Es gibt einen MIME-Typ String wie folgt zurück:

- text/plain für Textdateien
- text/html für HTML-Seiten
- application/pdf für PDF-Dateien
- image/png für PNG-Dateien

Dateien lesen

Dateien können byte- und zeilenweise mit der Files Klasse gelesen werden.

```
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));
byte[] content = Files.readAllBytes(p2);
List<String> linesOfContent = Files.readAllLines(p2);
```

Files.readAllLines() nimmt optional einen Zeichensatz als Parameter (Standard ist StandardCharsets.UTF_8):

```
List<String> linesOfContent = Files.readAllLines(p2, StandardCharsets.ISO_8859_1);
```

Dateien schreiben

Dateien können mit der Files Klasse biss- und zeilenweise geschrieben werden

```
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)
```

Bestehende Dateien werden überschrieben, nicht vorhandene Dateien werden erstellt.

Neue Datei-E / A online lesen: https://riptutorial.com/de/java/topic/5519/neue-datei-e---a

Kapitel 115: Nichtzugriffsmodifizierer

Einführung

Nicht-Zugriffsmodifizierer ***ändern die Zugänglichkeit von Variablen*** und Methoden nicht, bieten jedoch ***spezielle Eigenschaften*** .

Examples

Finale

final in Java kann sich auf Variablen, Methoden und Klassen beziehen. Es gibt drei einfache Regeln:

- Die letzte Variable kann nicht erneut zugewiesen werden
- Die endgültige Methode kann nicht überschrieben werden
- letzte Klasse kann nicht verlängert werden

Verwendungen

Gute Programmierpraxis

Einige Entwickler halten es für sinnvoll, eine Variable als final zu markieren, wenn Sie können. Wenn Sie eine Variable haben, die nicht geändert werden soll, markieren Sie sie als final.

Eine wichtige Verwendung des final Schlüsselworts if für Methodenparameter. Wenn Sie betonen möchten, dass eine Methode ihre Eingabeparameter nicht ändert, markieren Sie die Eigenschaften als endgültig.

```
public int sumup(final List<Integer> ints);
```

Dies betont, dass die sumup Methode die sumup nicht ändern ints .

Innere Klasse Zugang

Wenn Ihre anonyme innere Klasse auf eine Variable zugreifen möchte, sollte die Variable als final markiert werden

```
  public IPrintName printName(){
    String name;
    return new IPrintName(){
        @Override
        public void printName(){
            System.out.println(name);
        }
    };
}
```

Diese Klasse lässt sich nicht kompilieren, als name , nicht endgültig.

Java SE 8

Endgültige Variablen sind eine Ausnahme. Hierbei handelt es sich um lokale Variablen, die nur einmal geschrieben werden und daher als final deklariert werden könnten. Auf endgültige Variablen kann auch von anonymen Klassen zugegriffen werden.

final static Variable

Auch wenn der folgende Code völlig legal ist, wenn die final Variable foo nicht static , wird sie im static Fall nicht kompiliert:

```
class TestFinal {
    private final static List foo;

    public Test() {
        foo = new ArrayList();
    }
}
```

Der Grund ist, lassen Sie uns noch einmal wiederholen, die *letzte Variable kann nicht erneut zugewiesen werden* . Da foo statisch ist, wird es von allen Instanzen der Klasse TestFinal . Wenn eine neue Instanz einer Klasse TestFinal erstellt wird, wird deren Konstruktor aufgerufen, und foo wird neu zugewiesen, was der Compiler nicht zulässt. Eine korrekte Methode zum Initialisieren der Variablen foo in diesem Fall entweder:

```
class TestFinal {
    private static final List foo = new ArrayList();
    //..
}
```

oder mit einem statischen Initialisierer:

```
class TestFinal {
    private static final List foo;
    static {
        foo = new ArrayList();
    }
    //..
}
```

final Methoden sind nützlich, wenn die Basisklasse einige wichtige Funktionen implementiert, die von der abgeleiteten Klasse nicht geändert werden sollen. Sie sind auch schneller als nicht endgültige Methoden, da kein Konzept für virtuelle Tabellen vorhanden ist.

Alle Wrapper-Klassen in Java sind final, wie Integer , Long usw. Die Ersteller dieser Klassen wollten nicht, dass jeder Integer beispielsweise in seine eigene Klasse erweitern und das grundlegende Verhalten der Integer-Klasse ändern kann. Eine der Voraussetzungen, um eine Klasse unveränderlich zu machen, besteht darin, dass Unterklassen Methoden nicht überschreiben dürfen. Am einfachsten ist es, die Klasse als final zu deklarieren.

flüchtig

Der volatile Modifikator wird bei der Multithread-Programmierung verwendet. Wenn Sie ein Feld als volatile deklarieren, ist dies ein Signal an Threads, dass sie den neuesten Wert lesen müssen, nicht einen lokal zwischengespeicherten Wert. Darüber hinaus sind volatile Lese- und Schreibvorgänge garantiert atomar (der Zugriff auf einen volatile long oder double Speicher ist nicht atomar), wodurch bestimmte Lese- / Schreibfehler zwischen mehreren Threads vermieden werden.

```
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;
    }
}
```

statisch

Das static Schlüsselwort wird für eine Klasse, eine Methode oder ein Feld verwendet, damit sie unabhängig von jeder Instanz der Klasse funktionieren.

- Statische Felder gelten für alle Instanzen einer Klasse. Sie benötigen keine Instanz, um auf sie zuzugreifen.
- Statische Methoden können ohne eine Instanz der Klasse ausgeführt werden, in der sie sich befinden. Sie können jedoch nur auf statische Felder dieser Klasse zugreifen.
- Statische Klassen können innerhalb anderer Klassen deklariert werden. Sie benötigen keine Instanz der Klasse, in der sie sich befinden, um instanziiert zu werden.

```
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
```

abstrakt

Abstraktion ist ein Prozess, bei dem die Implementierungsdetails ausgeblendet werden und dem Benutzer nur die Funktionalität angezeigt wird. Eine abstrakte Klasse kann niemals instanziiert werden. Wenn eine Klasse als abstrakt deklariert wird, besteht der einzige Zweck darin, die Klasse zu erweitern.

```
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");
    }
}
```

synchronisiert

Synchronized Modifier wird verwendet, um den Zugriff einer bestimmten Methode oder eines Blocks durch mehrere Threads zu steuern. Nur ein Thread kann eine Methode oder einen Block betreten, die als synchronisiert deklariert sind. Das synchronisierte Schlüsselwort funktioniert für die intrinsische Sperre eines Objekts. Im Falle einer synchronisierten Methode sperren die aktuellen

Objekte und die statische Methode verwendet ein Klassenobjekt. Jeder Thread, der versucht, einen synchronisierten Block auszuführen, muss zuerst die Objektsperre erwerben.

```
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();
    }
}
```

vorübergehend

Eine als transient deklarierte Variable wird während der Objektserialisierung nicht serialisiert.

```
public transient int limit = 55;   // will not persist
public int b; // will persist
```

strictfp

Java SE 1.2

Der striktfp-Modifizierer wird für Gleitkommaberechnungen verwendet. Mit diesem Modifikator wird die Fließkommavariable über mehrere Plattformen hinweg konsistenter und es wird sichergestellt, dass alle Fließkommaberechnungen gemäß den IEEE 754-Standards ausgeführt werden, um Berechnungsfehler (Rundungsfehler), Überläufe und Unterläufe in der 32-Bit- und 64-Bit-Architektur zu vermeiden. Dies kann nicht auf abstrakte Methoden, Variablen oder Konstruktoren angewendet werden.

```
// strictfp keyword can be applied on methods, classes and interfaces.

strictfp class A{}

strictfp interface M{}

class A{
    strictfp void m(){}
}
```

Nichtzugriffsmodifizierer online lesen:
https://riptutorial.com/de/java/topic/4401/nichtzugriffsmodifizierer

Kapitel 116: NIO - Vernetzung

Bemerkungen

SelectionKey definiert die verschiedenen auswählbaren Operationen und Informationen zwischen Selector und Channel . Insbesondere kann der Anhang verwendet werden, um verbindungsbezogene Informationen zu speichern.

Der Umgang mit OP_READ ist ziemlich unkompliziert. Beim Umgang mit OP_WRITE jedoch Vorsicht OP_WRITE : In der OP_WRITE können Daten in Sockets geschrieben werden, sodass das Ereignis weiterhin OP_WRITE wird. OP_WRITE Sie sicher, dass OP_WRITE nur OP_WRITE registrieren, bevor Sie Daten schreiben möchten (siehe Antwort).

Auch OP_CONNECT soll abgebrochen werden , sobald der Kanal verbunden ist (weil, na ja, es verbunden *ist*. Sehen Sie dies und dass Antworten auf SO). Daraus ergibt sich die OP_CONNECT Entfernung nach finishConnect() erfolgreich war.

Examples

Mit Selector auf Ereignisse warten (Beispiel mit OP_CONNECT)

NIO erschien in Java 1.4 und führte das Konzept der "Channels" ein, die schneller als normale E / A sein sollen. Aus SelectableChannel des Netzwerks ist der SelectableChannel der interessanteste, da er die Überwachung verschiedener Status des Kanals ermöglicht. Es funktioniert auf ähnliche Weise wie der Aufruf von C select() : Wir werden geweckt, wenn bestimmte Ereignistypen auftreten:

- Verbindung empfangen (OP_ACCEPT)
- Verbindung hergestellt (OP_CONNECT)
- Daten im Lese-FIFO verfügbar (OP_READ)
- Daten können zum Schreiben des OP_WRITE (OP_WRITE) OP_WRITE

Es ermöglicht die Trennung zwischen Buchse *Erkennung* I / O (etwas gelesen / geschrieben / ...) und der *Durchführung* des E / A (Lesen / Schreiben / ...). Insbesondere kann die gesamte E / A-Erkennung in einem einzigen Thread für mehrere Sockets (Clients) durchgeführt werden, während E / A-Vorgänge in einem Thread-Pool oder an einem anderen Ort ausgeführt werden können. Dadurch kann eine Anwendung problemlos auf die Anzahl der verbundenen Clients skaliert werden.

Das folgende Beispiel zeigt die Grundlagen:

1. Erstellen Sie eine Selector
2. Erstellen Sie einen SocketChannel
3. Registrieren Sie den SocketChannel im Selector
4. Schleife mit dem Selector , um Ereignisse zu erkennen

```
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");
```

Würde folgende Ausgabe ergeben:

```
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
```

NIO - Vernetzung online lesen: https://riptutorial.com/de/java/topic/5513/nio---vernetzung

Kapitel 117: Objektklassenmethoden und Konstruktor

Einführung

Diese Dokumentationsseite ist für Details mit Beispiel über Java - Klasse zeigt , Konstrukteure und über Objektklassenmethoden , die von dem übergeordneten Klasse automatisch vererbt werden Object einer neu erstellten Klasse.

Syntax

- öffentliche endgültige native Klasse <?> getClass ()
- public final native void notify ()
- public final native void notifyAll ()
- public final native void wait (lange Zeitüberschreitung) löst InterruptedException aus
- public final void wait () löst InterruptedException aus
- public final void wait (lange Zeitüberschreitung, int nanos) löst InterruptedException aus
- public native int hashCode ()
- public boolean equals (Object obj)
- public String toString ()
- protected native Object clone () löst CloneNotSupportedException aus
- protected void finalize () wirft Throwable

Examples

toString () -Methode

Die toString() -Methode wird verwendet, um eine String Darstellung eines Objekts mithilfe des Objektinhalts zu erstellen. Diese Methode sollte beim Schreiben Ihrer Klasse überschrieben werden. toString() wird implizit aufgerufen, wenn ein Objekt wie in "hello " + anObject zu einer Zeichenfolge "hello " + anObject .

Folgendes berücksichtigen:

```
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"
    }
}
```

Hier wird toString() aus der Object Klasse in der User Klasse überschrieben, um beim Drucken aussagekräftige Daten zum Objekt bereitzustellen.

Bei Verwendung von println() wird die toString() Methode des Objekts implizit aufgerufen. Deshalb machen diese Aussagen dasselbe:

```
System.out.println(user); // toString() is implicitly called on `user`
System.out.println(user.toString());
```

Wenn toString() in der oben genannten User Klasse nicht überschrieben wird, kann System.out.println(user) User@659e0bfd oder einen ähnlichen String mit fast keinen nützlichen Informationen außer dem Klassennamen zurückgeben. Dies wird sein , weil der Anruf wird die Verwendung toString() Implementierung der Basis Java Object - Klasse , die nichts über die weiße User der Klasse oder Geschäftsregeln. Wenn Sie diese Funktionalität in Ihrer Klasse ändern möchten, überschreiben Sie einfach die Methode.

Methode equals ()

TL; DR

== testet auf Referenzgleichheit (ob es sich um ***dasselbe Objekt handelt***)

.equals() prüft auf .equals() ob sie ***logisch "gleich" sind***)

---

equals() ist eine Methode, mit der zwei Objekte auf Gleichheit verglichen werden. Die Standardimplementierung der equals() -Methode in der Object Klasse gibt true zurück true wenn beide Referenzen auf dieselbe Instanz zeigen. Es verhält sich daher wie beim Vergleich durch == .

```
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
    }
}
```

Obwohl foo1 und foo2 mit denselben Feldern erstellt werden, zeigen sie auf zwei verschiedene Objekte im Speicher. Die standardmäßige equals() -Implementierung wird daher zu false ausgewertet.

Um den Inhalt eines Objekts auf Gleichheit zu vergleichen, muss equals() überschrieben werden.

```
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
    }
}
```

Hier entscheidet die überschriebene equals() -Methode, dass die Objekte gleich sind, wenn ihre Felder gleich sind.

Beachten Sie, dass die hashCode() -Methode ebenfalls überschrieben wurde. Der Vertrag für diese Methode besagt, dass, wenn zwei Objekte gleich sind, auch ihre Hash-Werte gleich sein müssen. Deshalb muss man fast immer hashCode() und equals() gemeinsam überschreiben.

Achten Sie besonders auf den Argumenttyp der Methode equals . Es ist Object obj , nicht Foo obj . Wenn Sie letzteres in Ihre Methode einfügen, ist dies keine Überschreibung der equals Methode.

Beim Schreiben Ihrer eigenen Klasse müssen Sie eine ähnliche Logik schreiben, wenn Sie equals() und hashCode() überschreiben. Die meisten IDEs können dies automatisch für Sie generieren.

Ein Beispiel für eine equals() -Implementierung finden Sie in der String Klasse, die Teil der Core-Java-API ist. Anstatt Zeiger zu vergleichen, vergleicht die String Klasse den Inhalt des String .

Java SE 7

In Java 1.7 wurde die Klasse java.util.Objects eingeführt, die eine bequeme Methode, equals , zur Verfügung stellt, die zwei potenziell null vergleicht, sodass Implementierungen der equals Methode vereinfacht werden können.

```
@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);
}
```

Klassenvergleich

Da die equals-Methode für jedes Objekt ausgeführt werden kann, besteht eine der ersten Aufgaben, die die Methode (nach Überprüfung auf null) häufig macht, darin, zu prüfen, ob die Klasse des zu vergleichenden Objekts mit der aktuellen Klasse übereinstimmt.

```
@Override
public boolean equals(Object obj) {
    //...check for null
    if (getClass() != obj.getClass()) {
        return false;
    }
    //...compare fields
}
```

Dies erfolgt normalerweise wie oben durch Vergleich der Klassenobjekte. Dies kann jedoch in einigen speziellen Fällen fehlschlagen, die möglicherweise nicht offensichtlich sind. Beispielsweise generieren einige Frameworks dynamische Proxys von Klassen, und diese dynamischen Proxies sind tatsächlich eine andere Klasse. Hier ist ein Beispiel mit 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
}
```

Ein Mechanismus, um diese Einschränkung zu umgehen, ist das Vergleichen von Klassen mithilfe von instanceof

```
@Override
public final boolean equals(Object obj) {
    if (!(obj instanceof Foo)) {
        return false;
    }
    //...compare fields
}
```

Es gibt jedoch einige Fallstricke, die bei der Verwendung von instanceof vermieden werden müssen. Da Foo möglicherweise andere Unterklassen haben könnte und diese Unterklassen möglicherweise equals() überschreiben, könnten Sie in einen Fall geraten, in dem ein Foo einer FooSubclass , die FooSubclass jedoch nicht Foo .

```
Foo foo = new Foo(7);
FooSubclass fooSubclass = new FooSubclass(7, false);
foo.equals(fooSubclass) //true
fooSubclass.equals(foo) //false
```

Dies verstößt gegen die Eigenschaften von Symmetrie und Transitivität und ist daher eine ungültige Implementierung der equals() -Methode. Daher ist es bei der Verwendung von instanceof eine gute Praxis, die equals() -Methode als final (wie im obigen Beispiel). Dadurch wird sichergestellt, dass keine Unterklasse equals() überschreibt und gegen wichtige Annahmen verstößt.

hashCode () Methode

Wenn eine Java-Klasse die Methode equals überschreibt, sollte sie auch die Methode hashCode überschreiben. Wie im Vertrag der Methode definiert:

- Wenn die hashCode Methode während des Ausführens einer Java-Anwendung mehr als einmal für dasselbe Objekt aufgerufen wird, muss sie die gleiche ganze Zahl zurückgeben, vorausgesetzt, es werden keine Informationen geändert, die in Gleichheitsvergleichen für das Objekt verwendet werden. Diese Ganzzahl muss von einer Ausführung einer Anwendung zu einer anderen Ausführung derselben Anwendung nicht konsistent bleiben.
- Wenn zwei Objekte gemäß der equals(Object) -Methode gleich sind, muss das Aufrufen der hashCode Methode für jedes der beiden Objekte dasselbe Integer-Ergebnis erzeugen.
- Wenn zwei Objekte gemäß der equals(Object) -Methode ungleich sind, ist es nicht erforderlich, dass beim Aufruf der hashCode Methode für jedes der beiden Objekte unterschiedliche Integer-Ergebnisse erzeugt werden müssen. Dem Programmierer sollte jedoch bewusst sein, dass die Erzeugung eindeutiger ganzzahliger Ergebnisse für ungleiche Objekte die Leistung von Hashtabellen verbessern kann.

Hash-Codes werden in Hash-Implementierungen wie HashMap , HashTable und HashSet . Das Ergebnis der hashCode Funktion bestimmt den Bucket, in den ein Objekt hashCode wird. Diese Hash-Implementierungen sind effizienter, wenn die bereitgestellte Hash- hashCode Implementierung gut ist. Eine wichtige Eigenschaft einer guten hashCode Implementierung besteht darin, dass die Verteilung der hashCode Werte einheitlich ist. Mit anderen Worten, es besteht eine geringe Wahrscheinlichkeit, dass zahlreiche Instanzen im selben Bucket gespeichert werden.

Ein Algorithmus zum Berechnen eines Hash-Code-Werts kann dem folgenden ähnlich sein:

```
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;
    }
}
```

Verwenden von Arrays.hashCode () als Abkürzung

Java SE 1.2

In Java 1.2 und höher kann anstelle eines Algorithmus zur Berechnung eines Hash-Codes ein Algorithmus mit java.util.Arrays#hashCode generiert werden, indem ein Object- oder Primitives-Array mit den Feldwerten java.util.Arrays#hashCode wird:

```
@Override
public int hashCode() {
    return Arrays.hashCode(new Object[] {field1, field2, field3});
}
```

Java SE 7

Java 1.7 hat die Klasse java.util.Objects eingeführt, die eine bequeme Methode, hash(Object... objects) , bereitstellt, die einen Hash-Code basierend auf den Werten der ihm bereitgestellten Objekte berechnet. Diese Methode funktioniert genauso wie java.util.Arrays#hashCode .

```
@Override
public int hashCode() {
    return Objects.hash(field1, field2, field3);
}
```

Hinweis: Dieser Ansatz ist ineffizient und erzeugt bei jedem Aufruf Ihrer benutzerdefinierten hashCode() Methode hashCode() :

- Ein temporäres Object[] wird erstellt. (In der Objects.hash() -Version wird das Array durch den "varargs" -Mechanismus erstellt.)
- Wenn es sich bei den Feldern um primitive Typen handelt, müssen sie mit einem Box versehen werden, wodurch möglicherweise mehr temporäre Objekte erstellt werden.
- Das Array muss gefüllt sein.
- Das Array muss von der Methode Arrays.hashCode oder Objects.hash werden.
- Die Aufrufe von Object.hashCode() , die Arrays.hashCode oder Objects.hash (wahrscheinlich) machen muss, können nicht eingebettet werden.

Internes Caching von Hash-Codes

Da die Berechnung des Hash-Codes eines Objekts teuer sein kann, kann es attraktiv sein, den Hash-Code-Wert in dem Objekt zu speichern, wenn er zum ersten Mal berechnet wird. Zum Beispiel

```
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;
    }
}
```

Bei diesem Ansatz werden die Kosten für die (wiederholte) Berechnung des Hash-Codes gegen den Overhead eines zusätzlichen Felds abgewickelt, um den Hash-Code zwischenzuspeichern. Ob sich dies als Leistungsoptimierung auszahlt, hängt davon ab, wie oft ein bestimmtes Objekt gehasht (nachgeschlagen) wird, und von anderen Faktoren.

Sie werden auch feststellen, dass der Cache unwirksam ist, wenn der echte Hashcode eines ImmutableArray gleich Null ist (eine Chance in 2^{32}).

Schließlich ist es schwieriger, diesen Ansatz korrekt zu implementieren, wenn das zu hashende Objekt veränderbar ist. Es gibt jedoch größere Bedenken, wenn sich Hash-Codes ändern. siehe den Vertrag oben.

wait () und notify () Methoden

wait() und notify() arbeiten in Tandem - wenn ein Thread Anrufe wait() auf ein Objekt, wird dieser Thread blockiert , bis ein anderer Thread ruft notify() oder notifyAll() auf demselben Objekt.

(Siehe auch: 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!");
    }
}
```

Einige Beispielausgabe:

```
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!
```

getClass () -Methode

Mit der getClass() -Methode kann der Laufzeitklassentyp eines Objekts ermittelt werden. Siehe das Beispiel unten:

```
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"
}
```

Die getClass() -Methode gibt den spezifischsten Klassentyp zurück. Wenn getClass() für einen anotherSpecificUser , ist der Rückgabewert die class SpecificUser da dies die Vererbungsstruktur niedriger als User .

Es ist bemerkenswert, dass die getClass Methode zwar wie folgt deklariert ist:

```
public final native Class<?> getClass();
```

Der tatsächliche statische Typ, der von einem Aufruf an getClass ist Class<? extends T> wobei T der statische Typ des Objekts ist, für das getClass aufgerufen wird.

dh Folgendes wird kompiliert:

```
Class<? extends String> cls = "".getClass();
```

klon () methode

Die clone() -Methode wird verwendet, um eine Kopie eines Objekts zu erstellen und zurückzugeben. Diese Methode sollte vermieden werden, da sie problematisch ist und ein Kopierkonstruktor oder eine andere Methode zum Kopieren zu Gunsten von clone() .

Damit die Methode verwendet werden kann, müssen alle Klassen, die die Methode aufrufen, die Schnittstelle Cloneable implementieren.

Die Cloneable Schnittstelle selbst ist nur ein Tag - Interface verwendet , um das Verhalten der ändern native clone() Methode , die prüft , ob die anrufende Objekte Klasse implementiert Cloneable . Wenn der Aufrufer diese Schnittstelle nicht implementiert, wird eine CloneNotSupportedException ausgelöst.

Die Object Klasse selbst implementiert diese Schnittstelle nicht, so dass eine

CloneNotSupportedException wird, wenn das aufrufende Objekt die Klasse Object .

Damit ein Klon korrekt ist, muss er unabhängig von dem Objekt sein, aus dem geklont wird. Daher muss das Objekt möglicherweise geändert werden, bevor es zurückgegeben wird. Dies bedeutet, dass im Wesentlichen eine "tiefe Kopie" erstellt wird, indem auch eines der *veränderlichen* Objekte kopiert wird, aus denen die interne Struktur des geklonten Objekts besteht. Wenn dies nicht korrekt implementiert ist, ist das geklonte Objekt nicht unabhängig und hat dieselben Verweise auf die veränderlichen Objekte wie das Objekt, aus dem es geklont wurde. Dies würde zu einem inkonsistenten Verhalten führen, da Änderungen an denen in der einen die andere beeinflussen würden.

```
class Foo implements Cloneable {
    int w;
    String x;
    float[] y;
    Date z;

    public Foo clone() {
        try {
            Foo result = new Foo();
            // copy primitives by value
            result.w = this.w;
            // immutable objects like String can be copied by reference
            result.x = this.x;

            // The fields y and z refer to a mutable objects; clone them recursively.
            if (this.y != null) {
              result.y = this.y.clone();
            }
            if (this.z != null) {
              result.z = this.z.clone();
            }

            // Done, return the new object
            return result;

        } catch (CloneNotSupportedException e) {
            // in case any of the cloned mutable fields do not implement Cloneable
            throw new AssertionError(e);
        }
    }
}
```

finalize () -Methode

Dies ist eine *geschützte* und *nicht statische* Methode der Object Klasse. Diese Methode wird verwendet, um einige abschließende Vorgänge auszuführen oder ein Objekt zu bereinigen, bevor es aus dem Speicher entfernt wird.

> Laut dem Dokument wird diese Methode vom Garbage Collector für ein Objekt aufgerufen, wenn die Garbage Collection feststellt, dass keine weiteren Verweise auf das Objekt vorhanden sind.

Es gibt jedoch keine Garantie dafür, dass die finalize() -Methode aufgerufen wird, wenn das Objekt noch erreichbar ist oder kein Garbage Collectors ausgeführt wird, wenn das Objekt in Frage kommt. Deshalb ist es besser, sich **nicht** auf diese Methode zu verlassen.

In Java-Kernbibliotheken konnten einige Verwendungsbeispiele gefunden werden, zum Beispiel in FileInputStream.java :

```
protected void finalize() throws IOException {
```

```
    if ((fd != null) &&  (fd != FileDescriptor.in)) {
        /* if fd is shared, the references in FileDescriptor
         * will ensure that finalizer is only called when
         * safe to do so. All references using the fd have
         * become unreachable. We can call close()
         */
        close();
    }
}
```

In diesem Fall ist dies die letzte Chance, die Ressource zu schließen, wenn diese Ressource zuvor noch nicht geschlossen wurde.

Im Allgemeinen wird die Verwendung der Methode finalize() in Anwendungen jeglicher Art als schlechte Praxis betrachtet und sollte vermieden werden.

Finalizer sind *nicht* zum Freigeben von Ressourcen (z. B. zum Schließen von Dateien) gedacht. Der Garbage-Collector wird aufgerufen, wenn (wenn!) Das System über wenig Speicherplatz verfügt. Sie können sich nicht darauf verlassen, dass es aufgerufen wird, wenn das System nicht genügend Dateizugriffsnummern hat oder aus anderen Gründen.

Der vorgesehene Anwendungsfall für Finalizer ist, dass ein Objekt, das gerade zurückgefordert wird, ein anderes Objekt über den bevorstehenden Untergang informiert. Zu diesem Zweck gibt es jetzt einen besseren Mechanismus - die Klasse java.lang.ref.WeakReference<T> . Wenn Sie der Meinung sind, dass Sie eine finalize() Methode schreiben müssen, sollten Sie prüfen, ob Sie dasselbe Problem stattdessen mit WeakReference lösen können. Wenn sich das Problem dadurch nicht lösen lässt, müssen Sie möglicherweise Ihr Design auf einer tieferen Ebene überdenken.

Zur weiteren Lektüre hier ein Artikel über die Methode finalize() aus dem Buch "Effective Java" von Joshua Bloch.

Objektkonstruktor

Alle Konstruktoren in Java müssen den Object aufrufen. Dies geschieht mit dem Aufruf super() . Dies muss die erste Zeile eines Konstruktors sein. Der Grund dafür ist, dass das Objekt tatsächlich auf dem Heap erstellt werden kann, bevor eine weitere Initialisierung durchgeführt wird.

Wenn Sie den Aufruf von super() in einem Konstruktor nicht angeben, wird der Compiler ihn für Sie einfügen.

Alle drei Beispiele sind also funktional identisch

mit explizitem Aufruf an super() Konstruktor

```
public class MyClass {

    public MyClass() {
        super();
    }
}
```

mit implizitem Aufruf an super() Konstruktor

```
public class MyClass {

    public MyClass() {
        // empty
    }
}
```

mit implizitem Konstruktor

```
public class MyClass {

}
```

Was ist mit Konstruktor-Verkettung?

Es ist möglich, andere Konstruktoren als erste Anweisung eines Konstruktors aufzurufen. Da sowohl der explizite Aufruf eines Superkonstruktors als auch der Aufruf eines anderen Konstruktors beide erste Anweisungen sein müssen, schließen sie sich gegenseitig aus.

```
public class MyClass {

    public MyClass(int size) {

        doSomethingWith(size);

    }

    public MyClass(Collection<?> initialValues) {

        this(initialValues.size());
        addInitialValues(initialValues);
    }
}
```

Beim Aufruf von new MyClass(Arrays.asList("a", "b", "c")) wird der zweite Konstruktor mit dem List-Argument aufgerufen, das wiederum an den ersten Konstruktor delegiert wird (der implizit an super() delegiert. super()) und rufen dann addInitialValues(int size) mit der zweiten Größe der Liste auf. Dies wird verwendet, um die Code-Duplizierung zu reduzieren, wenn mehrere Konstruktoren dieselbe Arbeit ausführen müssen.

Wie rufe ich einen bestimmten Konstruktor an?

Im obigen Beispiel kann man entweder new MyClass("argument") oder new MyClass("argument", 0) aufrufen. Mit anderen Worten, ähnlich wie beim Überladen von Methoden rufen Sie einfach den Konstruktor mit den Parametern auf, die für den ausgewählten Konstruktor erforderlich sind.

Was passiert im Objektklassenkonstruktor?

In einer Unterklasse mit einem leeren Standardkonstruktor (außer dem Aufruf von super()) passiert nichts weiter.

Der leere Standardkonstruktor kann explizit definiert werden. Andernfalls wird er vom Compiler eingefügt, sofern noch keine anderen Konstruktoren definiert sind.

Wie wird dann ein Objekt aus dem Konstruktor in Object erstellt?

Die eigentliche Erstellung von Objekten liegt in der JVM. Jeder Konstruktor in Java erscheint als spezielle Methode mit dem Namen <init> die für die Initialisierung verantwortlich ist. Diese <init> -Methode wird vom Compiler bereitgestellt. Da <init> in Java kein gültiger Bezeichner ist, kann sie nicht direkt in der Sprache verwendet werden.

Wie ruft die JVM diese <init> -Methode auf?

Die JVM invokespecial Methode <init> mit der Anweisung invokespecial und kann nur für nicht initialisierte Klasseninstanzen aufgerufen werden.

Weitere Informationen finden Sie in der JVM-Spezifikation und der Java-Sprachspezifikation:

- Spezielle Methoden (JVM) - JVMS - 2.9
- Konstruktoren - JLS - 8.8

Objektklassenmethoden und Konstruktor online lesen:
https://riptutorial.com/de/java/topic/145/objektklassenmethoden-und-konstruktor

Kapitel 118: Objektklonen

Bemerkungen

Das Klonen kann schwierig sein, insbesondere wenn die Felder des Objekts andere Objekte enthalten. Es gibt Situationen, in denen Sie eine tiefe Kopie erstellen möchten, anstatt nur die Feldwerte zu kopieren (dh Verweise auf die anderen Objekte).

Unterm Strich ist Klon ist gebrochen , und Sie sollten sich zweimal überlegen , bevor Sie die Umsetzung Cloneable Schnittstelle und das Überschreiben der clone - Methode. Die clone Methode ist in der Object Klasse und nicht in der Cloneable Schnittstelle Cloneable funktioniert daher nicht als Schnittstelle, da es keine öffentliche clone Methode gibt. Das Ergebnis ist, dass der Vertrag für die Verwendung von clone dünn dokumentiert und schwach durchgesetzt wird. Eine Klasse, die den clone überschreibt, ist beispielsweise darauf angewiesen, dass alle übergeordneten Klassen den clone überschreiben. Sie werden dazu nicht erzwungen und wenn dies nicht der Fall ist, kann Ihr Code Ausnahmen auslösen.

Eine viel bessere Lösung für die Bereitstellung von Klonfunktionen ist das Bereitstellen eines *Kopierkonstruktors* oder einer *Kopierfabrik* . Siehe Joshua Blochs Effektives Java- Element 11: Überschreiben Sie den Klon mit Bedacht.

Examples

Klonen mit einem Kopierkonstruktor

Ein einfacher Weg, ein Objekt zu klonen, ist die Implementierung eines Kopierkonstruktors.

```
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
```

Klonen durch Implementierung einer klonbaren Schnittstelle

Klonen eines Objekts durch Implementieren der Schnittstelle 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
```

Klonen beim Durchführen einer flachen Kopie

Das Standardverhalten beim Klonen eines Objekts besteht darin, eine flache Kopie der Felder des Objekts auszuführen. In diesem Fall enthalten sowohl das Originalobjekt als auch das geklonte Objekt Verweise auf dieselben Objekte.

Dieses Beispiel zeigt dieses Verhalten.

```
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));
}
```

Klonen beim Durchführen einer tiefen Kopie

Um geschachtelte Objekte zu kopieren, muss eine tiefe Kopie ausgeführt werden, wie in diesem Beispiel gezeigt.

```
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));
}
```

Klonen mit einer Kopierfabrik

```
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)
    }

}
```

Objektklonen online lesen: https://riptutorial.com/de/java/topic/2830/objektklonen

Kapitel 119: Objektreferenzen

Bemerkungen

Dies sollte Ihnen helfen, eine "Null-Zeiger-Ausnahme" zu verstehen - man erhält eine davon, weil eine Objektreferenz null ist, der Programmcode jedoch erwartet, dass das Programm etwas in dieser Objektreferenz verwendet. Das verdient jedoch ein eigenes Thema ...

Examples

Objektreferenzen als Methodenparameter

In diesem Thema wird das Konzept einer *Objektreferenz* erläutert. Es richtet sich an Menschen, die mit der Programmierung in Java noch nicht vertraut sind. Sie sollten bereits mit einigen Begriffen und Bedeutungen vertraut sein: Klassendefinition, Hauptmethode, Objektinstanz und der Aufruf von Methoden "auf" einem Objekt sowie die Übergabe von Parametern an Methoden.

```
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;
  }
}
```

Um sich mit der Java-Programmierung auskennen zu können, sollten Sie in der Lage sein, dieses Beispiel jemand anderem außerhalb Ihres Kopfes zu erklären. Ihre Konzepte sind grundlegend für das Verständnis der Funktionsweise von Java.

Wie Sie sehen können, haben wir eine main , die ein Objekt auf die Variable instanziiert person , und ruft eine Methode , die festlegen name Feld in dem Objekt zu "Bob" . Dann ruft es eine andere Methode auf und übergibt person als einen von zwei Parametern; Der andere Parameter ist eine Ganzzahlvariable, die auf 5 gesetzt ist.

Die aufgerufene Methode setzt den name des übergebenen Objekts auf "Linda" und die ganzzahlige Variable auf 99, die zurückgegeben wird.

Was würde also gedruckt werden?

```
Linda 5
```

Warum wird die Änderung der person in main wirksam, die Änderung der Ganzzahl jedoch nicht?

Wenn der Aufruf erfolgt, übergibt die Hauptmethode eine *Objektreferenz* für eine person an die

Methode setPersonName . Jede Änderung, die setAnotherName an diesem Objekt vornimmt, ist Teil dieses Objekts. setAnotherName sind diese Änderungen immer noch Teil dieses Objekts, wenn die Methode zurückgegeben wird.

Eine andere Art, das Gleiche zu sagen: person zeigt auf ein Objekt (auf dem Haufen gespeichert, falls Sie interessiert sind). Alle Änderungen, die die Methode an diesem Objekt vornimmt, werden "an diesem Objekt" vorgenommen und haben keinen Einfluss darauf, ob die Methode, die die Änderung vornimmt, noch aktiv ist oder zurückgegeben wurde. Wenn die Methode zurückkehrt, werden alle an dem Objekt vorgenommenen Änderungen immer noch in diesem Objekt gespeichert.

Vergleichen Sie dies mit der übergebenen Ganzzahl. Da dies ein *primitives* int (und keine Integer-Objektinstanz) ist, wird es "by value" übergeben. Das bedeutet, dass der Wert der Methode zur Verfügung gestellt wird und kein Zeiger auf die ursprünglich übergebene Integer-Zahl. Die Methode kann sie für die Methode ändern eigene Zwecke, dies hat jedoch keine Auswirkungen auf die Variable, die beim Aufruf der Methode verwendet wird.

In Java werden alle Grundelemente nach Wert übergeben. Objekte werden als Referenz übergeben, das heißt, ein Zeiger auf das Objekt wird als Parameter an alle Methoden übergeben, die sie verwenden.

Eine weniger offensichtliche Sache bedeutet das: Eine aufgerufene Methode kann kein *neues* Objekt erstellen und als einen der Parameter zurückgeben. Die einzige Möglichkeit für eine Methode, ein Objekt zurückzugeben, das direkt oder indirekt durch den Methodenaufruf erstellt wird, ist der Rückgabewert der Methode. Lassen Sie uns zuerst sehen, wie das nicht funktionieren würde und dann, wie es funktionieren würde.

Fügen wir unserem kleinen Beispiel hier eine weitere Methode hinzu:

```
private static void getAnotherObjectNot(Person person) {
  person = new Person();
  person.setName("George");
}
```

Und zurück in der main , unter dem Aufruf von setAnotherName , rufen setAnotherName diese Methode und einen weiteren Aufruf von println auf:

```
getAnotherObjectNot(person);
System.out.println(person.getName());
```

Nun würde das Programm ausdrucken:

```
Linda 5
Linda
```

Was ist mit dem Objekt passiert, das George hatte? Nun, der Parameter, der übergeben wurde, war ein Zeiger auf Linda; Wenn die getAnotherObjectNot Methode ein neues Objekt erstellt hat, hat sie die Referenz auf das Linda-Objekt durch eine Referenz auf das George-Objekt ersetzt. Das Linda Objekt existiert noch (auf dem Heap), das main noch darauf zugreifen kann, aber die getAnotherObjectNot Methode wäre nicht in der Lage sein , etwas nach , dass mit ihm zu tun, weil es keinen Hinweis darauf hat. Es scheint, dass der Verfasser des Codes beabsichtigte, dass die Methode ein neues Objekt erstellt und zurückgibt, aber wenn dies der Fall war, funktionierte es nicht.

Wenn der Autor dies tun wollte, müsste er das neu erstellte Objekt aus der Methode zurückgeben, etwa wie folgt:

```
private static Person getAnotherObject() {
  Person person = new Person();
  person.setName("Mary");
  return person;
}
```

Dann nennen Sie es so:

```
Person mary;
mary = getAnotherObject();
System.out.println(mary.getName());
```

Und die gesamte Programmausgabe wäre jetzt:

```
Linda 5
Linda
Mary
```

Hier ist das gesamte Programm mit beiden Ergänzungen:

```
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;
  }
}
```

Objektreferenzen online lesen: https://riptutorial.com/de/java/topic/5454/objektreferenzen

Kapitel 120: Operatoren

Einführung

Operatoren in der Java-Programmiersprache sind spezielle Symbole, die bestimmte Operationen an einem, zwei oder drei Operanden ausführen und dann ein Ergebnis zurückgeben.

Bemerkungen

Ein *Operator* ist ein Symbol (oder Symbole), das ein Java-Programm anweist, eine *Operation* an einem, zwei oder drei *Operanden* auszuführen. Ein Operator und seine Operanden bilden einen *Ausdruck* (siehe Thema Ausdrücke). Die Operanden eines Operators sind selbst Ausdrücke.

In diesem Thema werden ungefähr 40 verschiedene Operatoren beschrieben, die von Java definiert werden. Das separate Thema "Ausdrücke" erläutert:

- wie Operatoren, Operanden und andere Dinge zu Ausdrücken kombiniert werden,
- wie die Ausdrücke bewertet werden und
- wie Ausdruckstypen, Konvertierungen und Ausdrucksauswertung funktionieren.

Examples

Der String-Verkettungsoperator (+)

Das + -Symbol kann drei verschiedene Operatoren in Java bedeuten:

- Wenn vor dem + kein Operand vorhanden ist, ist dies der unäre Plus-Operator.
- Wenn zwei Operanden vorhanden sind, sind beide numerisch. dann ist es der binäre Addition-Operator.
- Wenn zwei Operanden vorhanden sind und mindestens einer von ihnen ein String , handelt es sich um den binären Verkettungsoperator.

Im einfachen Fall verknüpft der Verkettungsoperator zwei Zeichenfolgen, um eine dritte Zeichenfolge zu erhalten. Zum Beispiel:

```
String s1 = "a String";
String s2 = "This is " + s1;    // s2 contains "This is a String"
```

Wenn einer der beiden Operanden kein String ist, wird er wie folgt in einen String konvertiert:

- Ein Operand, dessen Typ ein primitiver Typ ist, wird *wie* durch Aufruf von toString() für den geschachtelten Wert konvertiert.
- Ein Operand, dessen Typ ein Referenztyp ist, wird durch Aufrufen der toString() Methode des Operanden konvertiert. Wenn der Operand null ist oder wenn die toString() Methode null zurückgibt, wird stattdessen das Zeichenfolgenliteral "null" verwendet.

Zum Beispiel:

```
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"
```

Die Erklärung für das s5 Beispiel besteht darin, dass die toString() -Methode für Array-Typen von java.lang.Object geerbt wird und das Verhalten darin besteht, eine Zeichenfolge zu erzeugen, die aus dem Typnamen und dem Identitäts-Hashcode des Objekts besteht.

Der Verkettungsoperator wird angegeben, um ein neues String Objekt zu erstellen, außer wenn der Ausdruck ein konstanter Ausdruck ist. Im letzteren Fall wird der Ausdruck beim Kompilierungstyp ausgewertet, und sein Laufzeitwert entspricht einem String-Literal. Dies bedeutet, dass beim Aufteilen eines langen String-Literal wie folgt kein Laufzeit-Overhead entsteht:

```
String typing = "The quick brown fox " +
                "jumped over the " +
                "lazy dog";           // constant expression
```

Optimierung und Effizienz

Mit Ausnahme von Konstantenausdrücken erstellt jeder Zeichenfolgenverkettungsausdruck ein neues String Objekt. Betrachten Sie diesen Code:

```
public String stars(int count) {
    String res = "";
    for (int i = 0; i < count; i++) {
        res = res + "*";
    }
    return res;
}
```

In der obigen Methode wird bei jeder Iteration der Schleife ein neuer String , der um ein Zeichen länger ist als die vorherige Iteration. Jede Verkettung kopiert alle Zeichen in den Operanden - Strings die neuen bilden String . stars(N) werden also:

- Erstellen Sie N neue String Objekte und werfen Sie alle außer dem letzten weg.
- kopiere N * (N + 1) / 2 Zeichen und
- generiere O(N^2) Bytes Müll.

Dies ist für große N sehr teuer. In der Tat kann jeder Code, der Zeichenfolgen in einer Schleife verkettet, dieses Problem haben. Ein besserer Weg, dies zu schreiben, wäre wie folgt:

```
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();
}
```

Idealerweise sollten Sie die Kapazität des StringBuilder festlegen. Wenn dies jedoch nicht praktikabel ist, *vergrößert* die Klasse automatisch das Sicherungsarray, das der Builder zum Speichern von Zeichen verwendet. (Hinweis: Die Implementierung erweitert das Hintergrundarray exponentiell. Diese Strategie hält die Menge an Zeichen, die auf O(N) anstatt auf O(N^2) kopiert wird.)

Einige Leute wenden dieses Muster auf alle String-Verkettungen an. Dies ist jedoch nicht erforderlich, da JLS *es* einem Java-Compiler ermöglicht, Zeichenfolgenverkettungen innerhalb eines einzelnen Ausdrucks zu optimieren. Zum Beispiel:

```
String s1 = ...;
String s2 = ...;
String test = "Hello " + s1 + ". Welcome to " + s2 + "\n";
```

wird *normalerweise* vom Bytecode-Compiler auf so etwas optimiert;

```
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();
```

(Der JIT-Compiler kann dies weiter optimieren, wenn er davon ausgehen kann, dass s1 oder s2 nicht null .) Beachten Sie jedoch, dass diese Optimierung nur innerhalb eines einzelnen Ausdrucks zulässig ist.

Kurz gesagt, wenn Sie über die Effizienz von String-Verkettungen besorgt sind:

- Optimieren Sie von Hand, wenn Sie wiederholte Verkettungen in einer Schleife (oder ähnlichem) durchführen.
- Optimieren Sie einen einzelnen Verkettungsausdruck nicht manuell.

Die arithmetischen Operatoren (+, -, *, /,%)

Die Java-Sprache stellt 7 Operatoren zur Verfügung, die Arithmetik für Ganzzahl- und Fließkommawerte ausführen.

- Es gibt zwei Operatoren + :
  - Der binäre Additionsoperator fügt eine Zahl zu einer anderen hinzu. (Es gibt auch einen binären + -Operator, der eine String-Verkettung ausführt. Dies wird in einem separaten Beispiel beschrieben.)
  - Der Operator unary plus tut nichts außer das Auslösen einer numerischen Promotion (siehe unten).
- Es gibt zwei - Operatoren:
  - Der Operator für binäre Subtraktion subtrahiert eine Zahl von einer anderen.
  - Der unäre Minusoperator entspricht dem Abzug seines Operanden von Null.
- Der binäre Multiplikationsoperator (*) multipliziert eine Zahl mit einer anderen.
- Der Binärdivisionsoperator (/) teilt eine Zahl durch eine andere.
- Der Operator für binäre Rest [1] (%) berechnet den Rest, wenn eine Zahl durch eine andere geteilt wird.

1. Dies wird oft fälschlicherweise als "Modulus" -Operator bezeichnet. "Rest" ist der Begriff, der von der JLS verwendet wird. "Modul" und "Rest" sind nicht dasselbe.

Operanden- und Ergebnistypen sowie numerische Promotion

Die Operatoren benötigen numerische Operanden und erzeugen numerische Ergebnisse. Bei den Operandentypen kann es sich um einen beliebigen primitiven numerischen Typ handeln (dh byte , short , char , int , long , float oder double) oder um einen beliebigen numerischen Wrapper-Typ, der in java.lang definiert ist. dh (Byte , Character , Short , Integer , Long , Float oder Double .

Der Ergebnistyp wird basierend auf den Typen des Operanden oder der Operanden wie folgt bestimmt:

- Wenn einer der Operanden double oder Double , ist der Ergebnistyp double .
- Wenn einer der Operanden float oder Float , lautet der Ergebnistyp float .
- Andernfalls, wenn einer der Operanden long oder Long , ist der Ergebnistyp long .
- Ansonsten ist der Ergebnistyp int . Dies umfasst byte , short und char Operanden sowie `int.

Der Ergebnistyp der Operation bestimmt, wie die arithmetische Operation ausgeführt wird und wie die Operanden behandelt werden

- Wenn der Ergebnistyp double , werden die Operanden auf double erhöht, und die Operation wird unter Verwendung einer 64-Bit-IEE 754-Gleitkomma-Arithmetik ausgeführt.
- Wenn der Ergebnistyp " float , werden die Operanden zum " float , und die Operation wird unter Verwendung einer 32-Bit-Gleitkomma-Arithmetik (IEE 754 mit einfacher Genauigkeit)

ausgeführt.
- Wenn der Ergebnistyp long , werden die Operanden zu long heraufgestuft, und die Operation wird mit einer 64-Bit-Zweierkomplement-Binär-Ganzzahl-Arithmetik mit Vorzeichen ausgeführt.
- Wenn der Ergebnistyp " int , werden die Operanden zu " int , und die Operation wird unter Verwendung einer 32-Bit-Zweierkomplement-Binär-Ganzzahl-Arithmetik mit Vorzeichen ausgeführt.

Die Promotion erfolgt in zwei Schritten:

- Wenn es sich bei dem Operandentyp um einen Wrapper-Typ handelt, wird der Operandenwert *nicht* mit einem Wert des entsprechenden primitiven Typs *gefüllt* .
- Bei Bedarf wird der Primitivtyp auf den erforderlichen Typ hochgestuft:
  - Die Förderung von ganzen Zahlen zu int oder long ist verlustfrei.
  - Die Beförderung des float zum double ist verlustfrei.
  - Die Aufwertung einer Ganzzahl auf einen Gleitkommawert kann zu Genauigkeitsverlusten führen. Die Konvertierung wird unter Verwendung der Semantik "Round-to-next" von IEE 768 durchgeführt.

Die Bedeutung der Spaltung

Der Operator / teilt den linken Operanden n (den *Dividend*) und den rechten Operanden d (den *Divisor*) und erzeugt das Ergebnis q (den *Quotienten*).

Java Integer Division rundet gegen Null. Der JLS-Abschnitt 15.17.2 legt das Verhalten der Java-Ganzzahldivision wie folgt fest:

> Der für die Operanden n und d erzeugte Quotient ist ein ganzzahliger Wert q dessen Größe so groß wie möglich ist, während |d • q| ≤ |n| erfüllt wird |d • q| ≤ |n| . Außerdem ist q positiv, wenn |n| ≥ |d| und n und d haben dasselbe Vorzeichen, aber q ist negativ, wenn |n| ≥ |d| und n und d haben entgegengesetzte Vorzeichen.

Es gibt einige Sonderfälle:

- Wenn n MIN_VALUE ist und der Divisor -1 ist, tritt ein Ganzzahlüberlauf auf und das Ergebnis ist MIN_VALUE . In diesem Fall wird keine Ausnahme ausgelöst.
- Wenn d 0 ist, wird `ArithmeticException ausgelöst.

Bei der Java-Gleitkommadivision müssen mehr Randfälle berücksichtigt werden. Die Grundidee ist jedoch, dass das Ergebnis q der Wert ist, der am ehesten erfüllt ist, um d . q = n zu erfüllen d . q = n .

Gleitkommadivision führt niemals zu einer Ausnahme. Stattdessen führen Operationen, die durch Null dividieren, zu INF- und NaN-Werten. siehe unten.

Die Bedeutung von Rest

Im Gegensatz zu C und C ++ arbeitet der Restoperator in Java sowohl mit Ganzzahl- als auch mit Gleitkommaoperationen.

Für ganzzahlige Fälle ist das Ergebnis von a % b als Zahl r definiert, so dass (a / b) * b + r gleich a , wobei / , * und + die geeigneten Java-Ganzzahloperatoren sind. Dies gilt in allen Fällen, außer wenn b Null ist. In diesem Fall führt der Rest zu einer ArithmeticException .

Aus der obigen Definition folgt, dass a % b nur dann negativ sein kann, wenn a negativ ist, und es ist nur positiv, wenn a positiv ist. Außerdem ist die Größe von a % b immer kleiner als die Größe von b .

Die Floating-Point-Restoperation ist eine Verallgemeinerung des Ganzzahlfalls. Das Ergebnis von a % b ist der Rest r , der durch die mathematische Beziehung r = a - (b • q) wobei:

- q ist eine ganze Zahl,
- es ist nur negativ, wenn a / b negativ ist, positiv nur, wenn a / b positiv ist, und
- seine Größe ist so groß wie möglich, ohne die Größe des wahren mathematischen Quotienten

von a und b zu überschreiten.

Gleitpunktrest kann INF und NaN Werte in NaN erzeugen, beispielsweise wenn b Null ist; siehe unten. Es wird keine Ausnahme ausgelöst.

Wichtige Notiz:

> Das von % berechnete Ergebnis einer Gleitkomma- Math.IEEEremainder **ist nicht dasselbe** wie das Ergebnis, das von der durch IEEE 754 definierten Math.IEEEremainder erzeugt wird. Der Rest der IEEE 754 kann mit der Bibliotheksmethode Math.IEEEremainder berechnet werden.

Ganzzahlüberlauf

Java-Ganzzahlwerte für 32 und 64 Bit sind vorzeichenbehaftet und verwenden eine binäre Darstellung mit zwei Komplementen. Beispielsweise kann der Zahlenbereich als (32 Bit) int -2^{31} bis $+2^{31} - 1$ dargestellt werden.

Wenn Sie zwei N-Bit-Integer (N = 32 oder 64) addieren, subtrahieren oder mehrere N-Bits addieren, ist das Ergebnis der Operation möglicherweise zu groß, um als N-Bit-Integer dargestellt zu werden. In diesem Fall führt die Operation zu einem *Ganzzahlüberlauf* , und das Ergebnis kann wie folgt berechnet werden:

- Die mathematische Operation wird ausgeführt, um eine Zwillingskomplementdarstellung der gesamten Zahl zu erhalten. Diese Darstellung wird größer als N Bits sein.
- Die unteren 32 oder 64 Bits der Zwischendarstellung werden als Ergebnis verwendet.

Es ist zu beachten, dass der Integer-Überlauf unter keinen Umständen zu Ausnahmen führt.

Gleitkomma-INF- und NAN-Werte

Java verwendet IEE 754-Gleitkommadaten für float und double . Diese Darstellungen haben einige spezielle Werte für die Darstellung von Werten, die außerhalb der Domäne der reellen Zahlen liegen:

- Die Werte "unendlich" oder "INF" geben zu große Zahlen an. Der +INF Wert gibt Zahlen an, die zu groß und positiv sind. Der -INF Wert kennzeichnet zu große und negative Zahlen.
- "Unbestimmt" / "keine Zahl" oder NaN bezeichnen Werte, die aus bedeutungslosen Operationen resultieren.

Die INF-Werte werden durch Floating-Operationen erzeugt, die einen Überlauf verursachen, oder durch Division durch Null.

Die NaN-Werte werden erzeugt, indem Null durch Null dividiert oder Null berechnet wird.

Überraschenderweise ist es möglich, Arithmetik mit INF- und NaN-Operanden auszuführen, ohne Ausnahmen auszulösen. Zum Beispiel:

- Addiert man + INF und einen endlichen Wert, erhält man + INF.
- Das Hinzufügen von + INF und + INF ergibt + INF.
- Addieren von + INF und -INF ergibt NaN.
- Division durch INF ergibt entweder +0.0 oder -0.0.
- Alle Operationen mit einem oder mehreren NaN-Operanden ergeben NaN.

Ausführliche Informationen finden Sie in den entsprechenden Unterabschnitten von JLS 15 . Beachten Sie, dass dies weitgehend "akademisch" ist. Bei typischen Berechnungen bedeutet ein INF oder NaN , dass etwas schiefgegangen ist. Sie haben zB unvollständige oder falsche Eingabedaten oder die Berechnung wurde falsch programmiert.

Die Gleichheitsoperatoren (==,! =)

Die Operatoren == und != Sind binäre Operatoren, die abhängig davon, ob die Operanden gleich

sind, als true oder false ausgewertet werden. Der Operator == gibt true wenn die Operanden gleich sind, andernfalls false . Der Operator != Gibt false wenn die Operanden gleich und andernfalls true sind.

Diese Operatoren können mit primitiven Typen und Referenztypen verwendet werden, das Verhalten unterscheidet sich jedoch erheblich. Laut JLS gibt es drei verschiedene Gruppen dieser Operatoren:

- Die booleschen Operatoren == und != .
- Die numerischen Operatoren == und != .
- Die Referenzoperatoren == und != .

In allen Fällen ist der Ergebnistyp der Operatoren == und != Jedoch boolean .

Die numerischen Operatoren == und !=

Wenn einer (oder beide) der Operanden eines Operators == oder != Ein primitiver numerischer Typ ist (byte , short , char , int, long int, , float oder double), handelt es sich bei dem Operator um einen Zahlenvergleich. Der zweite Operand muss entweder ein numerischer Grundtyp oder ein numerischer Typ mit Rahmen sein.

Das Verhalten anderer numerischer Operatoren lautet wie folgt:

1. Wenn es sich bei einem der Operanden um einen Box-Typ handelt, ist er nicht im Boxmodus.
2. Wenn einer der Operanden jetzt ein byte , ein short oder ein char , wird er zu einem int .
3. Wenn die Typen der Operanden nicht gleich sind, wird der Operand mit dem Typ "Kleiner" zum Typ "Größer" befördert.
4. Der Vergleich wird dann wie folgt durchgeführt:
    - Wenn die heraufgestuften Operanden int oder long die Werte getestet, um zu sehen, ob sie identisch sind.
    - Wenn die beförderten Operanden float oder double sind, gilt float :
        - Die zwei Versionen von Null (+0.0 und -0.0) werden als gleich behandelt
        - Ein NaN Wert wird als nicht gleichwertig behandelt und
        - Andere Werte sind gleich, wenn ihre IEEE 754-Darstellungen identisch sind.

Hinweis: Sie müssen vorsichtig sein, wenn Sie == und != Verwenden, um Fließkommawerte zu vergleichen.

Die booleschen Operatoren == und !=

Wenn beide Operanden boolean sind oder einer boolean und der andere Boolean , sind diese Operatoren die Booleschen Operatoren == und != . Das Verhalten ist wie folgt:

1. Wenn einer der Operanden ein Boolean , ist er nicht in der Box.
2. Die ungepackten Operanden werden getestet und das boolesche Ergebnis wird gemäß der folgenden Wahrheitstabelle berechnet

EIN	B	A == B	A! = B
falsch	falsch	wahr	falsch
falsch	wahr	falsch	wahr
wahr	falsch	falsch	wahr
wahr	wahr	wahr	falsch

Es gibt zwei "Fallstricke", die es ratsam machen, == und != Sparsam mit den Wahrheitswerten zu verwenden:

- Wenn Sie mit == oder != Zwei Boolean Objekte vergleichen, werden die Referenzoperatoren verwendet. Dies kann zu einem unerwarteten Ergebnis führen. Siehe Pitfall: Verwenden Sie ==, um primitive Wrapper-Objekte wie Integer zu vergleichen

- Der Operator == kann leicht als = eingegeben werden. Bei den meisten Operandentypen führt dieser Fehler zu einem Kompilierungsfehler. Bei boolean und Boolean Operanden führt der Fehler jedoch zu falschem Laufzeitverhalten. siehe Pitfall - Mit '==' einen boolean testen

Die Referenzoperatoren == und !=

Wenn beide Operanden Objektreferenzen sind, prüfen die Operatoren == und != Ob die beiden Operanden **auf dasselbe Objekt verweisen** . Das ist oft nicht das, was Sie wollen. Um zu testen, ob zwei Objekte *nach Wert* gleich *sind* , sollte stattdessen die Methode .equals() verwendet werden.

```
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
```

Warnung: Die Verwendung von == und != Zum Vergleichen von String Werten ist in den meisten Fällen **falsch** . Siehe http://www.riptutorial.com/java/example/16290/pitfall-- using ---- to-compare-strings . Ein ähnliches Problem gilt für primitive Wrapper-Typen. Siehe http://www.riptutorial.com/java/example/8996/pitfall-- using ---- to- compare- primitive-wrappers- objects- such- as- integer .

Über die NaN-Randfälle

In JLS 15.21.1 heißt es:

> Wenn einer der Operanden NaN , ist das Ergebnis von == false aber das Ergebnis von != Ist true . Tatsächlich ist der Test x != x true dann true wenn der Wert von x NaN .

Dieses Verhalten ist (für die meisten Programmierer) unerwartet. Wenn Sie testen, ob ein NaN Wert sich selbst entspricht, lautet die Antwort "Nein, ist es nicht!". Mit anderen Worten: == ist für NaN Werte nicht *reflexiv* .

Dies ist jedoch keine Java-"Kuriosität", dieses Verhalten ist in den IEEE 754-Fließkomma-Standards angegeben, und Sie werden feststellen, dass es von den meisten modernen Programmiersprachen implementiert wird. (Weitere Informationen finden Sie unter http://stackoverflow.com/a/1573715/139985 ... Sie stellen fest, dass dies von jemandem geschrieben wurde, der "im Raum war, als die Entscheidungen getroffen wurden"!

Die Inkrement- / Dekrement-Operatoren (++ / --)

Variablen können mit den Operatoren ++ und -- um 1 erhöht oder dekrementiert werden.

Wenn die Operatoren ++ und -- auf Variablen folgen, werden sie als **Nachinkrement** bzw. **Nach-Dekrement** bezeichnet.

```
int a = 10;
a++; // a now equals 11
a--; // a now equals 10 again
```

Wenn die Operatoren ++ und -- vor den Variablen stehen, werden die Operationen als **Vorinkrement** bzw. **Vor-Dekrement** bezeichnet.

```
int x = 10;
--x; // x now equals 9
++x; // x now equals 10
```

Wenn der Operator der Variablen vorangeht, ist der Wert des Ausdrucks der Wert der Variablen, nachdem sie inkrementiert oder dekrementiert wurde. Wenn der Operator der Variablen folgt, ist der Wert des Ausdrucks der Wert der Variablen, bevor sie erhöht oder dekrementiert wird.

```
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
```

Achten Sie darauf, keine Nachinkremente oder Dekremente zu überschreiben. Dies geschieht, wenn Sie am Ende eines Ausdrucks einen Post-In / Decrement-Operator verwenden, der der In / Decrement-Variablen selbst neu zugewiesen wird. Das Ein- / Dekrement hat keine Auswirkungen. Obwohl die Variable auf der linken Seite korrekt inkrementiert wird, wird ihr Wert sofort mit dem zuvor ausgewerteten Ergebnis von der rechten Seite des Ausdrucks überschrieben:

```
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!)
```

Richtig:

```
int x = 0;
x = x++ + 1 + x;        // evaluates to x = 0 + 1 + 1
x++;                    // adds 1
System.out.println(x);  // prints 3
```

Der bedingte Operator (? :)

Syntax

{zu bewertende Bedingung} **?** *{Anweisung-ausgeführt auf-wahr}* **:** *{Anweisung-ausgeführt auf-falsch}*

Wie in der Syntax gezeigt, verwendet der Bedingungsoperator (auch als der ternäre Operator [1] bezeichnet) den ? (Fragezeichen) und : (Doppelpunkt) Zeichen, um einen bedingten Ausdruck von zwei möglichen Ergebnissen zu ermöglichen. Es kann verwendet werden, um längere if-else Blöcke zu ersetzen, um je nach Bedingung einen von zwei Werten zurückzugeben.

```
result = testCondition ? value1 : value2
```

Ist äquivalent zu

```
if (testCondition) {
    result = value1;
} else {
    result = value2;
}
```

Es kann gelesen werden als ***: Wenn testCondition wahr ist, setze das Ergebnis auf value1; Andernfalls setzen Sie das Ergebnis auf value2 ".***

Zum Beispiel:

```
// get absolute value using conditional operator
a = -10;
int absValue = a < 0 ? -a : a;
System.out.println("abs = " + absValue); // prints "abs = 10"
```

Ist äquivalent zu

```
// 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"
```

Gemeinsame Nutzung

Sie können den bedingten Operator für bedingte Zuweisungen verwenden (z. B. Nullprüfung).

```
String x = y != null ? y.toString() : ""; //where y is an object
```

Dieses Beispiel entspricht:

```
String x = "";

if (y != null) {
    x = y.toString();
}
```

Da der Bedingungsoperator über den Zuweisungsoperatoren die zweitniedrigste Priorität hat, ist es selten erforderlich, eine Klammer um die *Bedingung zu verwenden. In* Kombination mit anderen Operatoren ist jedoch eine Klammer um das gesamte Konstrukt des Bedingten Operators erforderlich:

```
// no parenthesis needed for expressions in the 3 parts
10 <= a && a < 19 ? b * 5 : b * 7

// parenthesis required
7 * (a > 0 ? 2 : 5)
```

Bedingte Operatoren können auch im dritten Teil verschachtelt werden, wobei sie eher wie Verketten oder wie eine switch-Anweisung funktionieren.

```
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))
```

Fußnote:

1 - Sowohl die Java-Sprachspezifikation als auch das Java-Lernprogramm rufen den Operator (? : :) Als

Bedingungsoperator auf . Das Tutorial sagt, dass es "auch als der ternäre Operator bekannt ist", da es (derzeit) der einzige von Java definierte ternäre Operator ist. Die Terminologie "Bedingter Operator" stimmt mit C und C ++ und anderen Sprachen mit einem äquivalenten Operator überein.

Die bitweisen und logischen Operatoren (~, &, |, ^)

Die Java-Sprache stellt 4 Operatoren bereit, die bitweise oder logische Operationen an ganzzahligen oder booleschen Operanden ausführen.

- Der Complement-Operator (~) ist ein unärer Operator, der eine bitweise oder logische Invertierung der Bits eines Operanden durchführt. siehe JLS 15.15.5. .
- Der Operator AND (&) ist ein binärer Operator, der ein bitweises oder logisches "und" von zwei Operanden ausführt. siehe JLS 15.22.2. .
- Der Operator OR (|) ist ein binärer Operator, der ein bitweises oder logisches "Inclusive or" von zwei Operanden ausführt. siehe JLS 15.22.2. .
- Der XOR (^) - Operator ist ein binärer Operator, der ein bitweises oder logisches "Exklusiv-Oder" von zwei Operanden ausführt. siehe JLS 15.22.2. .

Die logischen Operationen, die von diesen Operatoren ausgeführt werden, wenn die Operanden Boolesche Werte sind, können wie folgt zusammengefasst werden:

EIN	B	~ A	A & B	A | B	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Beachten Sie, dass für Integer-Operanden die obige Tabelle beschreibt, was für einzelne Bits geschieht. Die Operatoren bearbeiten tatsächlich alle 32 oder 64 Bits des Operanden oder der Operanden parallel.

Operandentypen und Ergebnistypen.

Die üblichen arithmetischen Konvertierungen gelten, wenn die Operanden Ganzzahlen sind. Häufige Anwendungsfälle für die bitweisen Operatoren

---

Der Operator ~ wird verwendet, um einen booleschen Wert umzukehren oder alle Bits in einem Ganzzahloperanden zu ändern.

Der Operator & dient zum "Ausblenden" einiger Bits in einem Ganzzahloperanden. Zum Beispiel:

```
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
```

Die | Operator wird verwendet, um die Wahrheitswerte von zwei Operanden zu kombinieren. Zum Beispiel:

```
int word2 = 0b01011111;
// Combine the bottom 2 bits of word1 with the top 30 bits of word2
int combined = (word & mask) | (word2 & ~mask);   // -> 0b01011110
```

Der ^ Operator wird zum Umschalten oder "Umdrehen" von Bits verwendet:

```
int word3 = 0b00101010;
int word4 = word3 ^ mask;             // -> 0b00101001
```

Weitere Beispiele für die Verwendung der bitweisen Operatoren finden Sie unter Bitmanipulation

Die Instanz des Operators

Dieser Operator prüft, ob das Objekt von einem bestimmten Klassen- / Schnittstellentyp ist.
Instanz des Operators lautet:

```
( Object reference variable ) instanceof  (class/interface type)
```

Beispiel:

```
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 );
   }
}
```

Dies würde folgendes Ergebnis ergeben:

```
true
```

Dieser Operator gibt immer noch true zurück, wenn das zu vergleichende Objekt mit dem Typ rechts kompatibel ist.

Beispiel:

```
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 );
   }
}
```

Dies würde folgendes Ergebnis ergeben:

```
true
```

Die Zuweisungsoperatoren (=, + =, - =, * =, / =,% =, << =, >> =, >>> =, & =, | = und ^ =)

Der linke Operand für diese Operatoren muss entweder eine nicht endgültige Variable oder ein Element eines Arrays sein. Der Operand für die rechte Hand muss mit dem linken Operanden *kompatibel* sein. Dies bedeutet, dass entweder die Typen gleich sein müssen oder der Typ des rechten Operanden durch Kombination von Boxing, Unboxing oder Expanding in den Typ des linken Operandentyps konvertierbar sein muss. (Für vollständige Details siehe JLS 5.2 .)

Die genaue Bedeutung der Operatoren "Operation and Assign" wird in JLS 15.26.2 wie folgt

angegeben:

> Ein zusammengesetzter Zuordnungsausdruck der Form E1 op= E2 ist äquivalent zu E1 = (T) ((E1) op (E2)) , wobei T der Typ von E1 , außer dass E1 nur einmal ausgewertet wird.

Beachten Sie, dass vor der endgültigen Zuweisung eine implizite Typumwandlung vorliegt.

1. =

Der einfache Zuweisungsoperator: weist dem linken Operanden den Wert des rechten Operanden zu.

> Beispiel: c = a + b addiert den Wert von a + b zum Wert von c und weist ihn c

2. +=

Der Operator "Hinzufügen und Zuweisen": fügt den Wert des Operanden für die linke Hand zum Wert des Operanden für die linke Hand hinzu und weist das Ergebnis dem linken Operanden zu. Wenn der linke Operand den Typ String , handelt es sich um einen Operator "Verketten und Zuweisen".

> Beispiel: c += a ungefähr c = c + a

3. -=

Der Operator "subtrahieren und zuordnen": subtrahiert den Wert des rechten Operanden vom Wert des linken Operanden und weist das Ergebnis dem linken Operanden zu.

> Beispiel: c -= a ungefähr c = c - a

4. *=

Der Operator "Multiplizieren und Zuweisen": multipliziert den Wert des Operanden für die rechte Hand mit dem Wert des Operanden für die linke Hand und weist das Ergebnis dem linken Operanden zu. .

> Beispiel: c *= a ungefähr c = c * a

5. /=

Der Operator "Teilen und Zuweisen": Dividiert den Wert des Operanden für die rechte Hand durch den Wert des Operanden für die linke Hand und weist das Ergebnis dem linken Operanden zu.

> Beispiel: c /*= a ungefähr c = c / a

6. %=

Der Operator "Modulus und Zuweisung": Berechnet den Modul des Werts des Operanden für die rechte Hand anhand des Werts des Operanden für die linke Hand und weist das Ergebnis dem linken Operanden zu.

> Beispiel: c %*= a ungefähr c = c % a

7. <<=

Der Operator "Links verschieben und zuweisen".

> Beispiel: c <<= 2 ungefähr c = c << 2

8. >>=

Der Operator "Arithmetik rechts verschieben und zuweisen".

> Beispiel: c >>= 2 ungefähr c = c >> 2

9. >>>=

Der Operator "logisch rechts verschieben und zuweisen".

Beispiel: c >>>= 2 ungefähr c = c >>> 2

10. &=

Der Operator "bitweise und und zuweisen".

Beispiel: c &= 2 ungefähr c = c & 2

11. |=

Der Operator "bitweise oder und zuweisen".

Beispiel: c |= 2 ungefähr c = c | 2

12. ^=

Der Operator "bitweises Exklusiv- oder Zuweisen".

Beispiel: c ^= 2 ungefähr c = c ^ 2

Die Bedingungs- und Bedingungsoperatoren (&& und ||)

Java bietet einen Bedingungs- und einen Bedingungs-Operator, die beide einen oder zwei Operanden vom Typ boolean annehmen und ein boolean Ergebnis erzeugen. Diese sind:

- `&&` - der bedingte AND-Operator
- `||` - die bedingten ODER-Operatoren. Die Auswertung von <left-expr> && <right-expr> entspricht dem folgenden Pseudocode:

```
{
   boolean L = evaluate(<left-expr>);
   if (L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return false;
   }
}
```

Die Bewertung von <left-expr> || <right-expr> entspricht dem folgenden Pseudocode:

```
{
   boolean L = evaluate(<left-expr>);
   if (!L) {
       return evaluate(<right-expr>);
   } else {
       // short-circuit the evaluation of the 2nd operand expression
       return true;
   }
}
```

Wie der obige Pseudocode zeigt, entspricht das Verhalten der Kurzschlussoperatoren der Verwendung von if / else Anweisungen.

Beispiel – Verwenden von && als Guard in einem Ausdruck

Das folgende Beispiel zeigt das am häufigsten verwendete Verwendungsmuster für den Operator && . Vergleichen Sie diese beiden Versionen einer Methode, um zu testen, ob eine bereitgestellte Integer Null ist.

```
public boolean isZero(Integer value) {
    return value == 0;
}

public boolean isZero(Integer value) {
    return value != null && value == 0;
}
```

Die erste Version funktioniert in den meisten Fällen, aber wenn das value Argument null , wird eine NullPointerException ausgelöst.

In der zweiten Version haben wir einen "Guard" -Test hinzugefügt. Der value != null && value == 0 wird ausgewertet, indem zuerst der value != null getestet wird. Wenn der null erfolgreich ist (dh als true bewertet wird), wird der value == 0 ausgewertet. Wenn die null - Test fehlschlägt, dann die Auswertung der value == 0 wird übersprungen (kurzgeschlossen), und wir *haben keine* bekommen NullPointerException .

Beispiel - Verwendung von &&, um eine kostspielige Berechnung zu vermeiden

Das folgende Beispiel zeigt, wie && verwendet werden kann, um eine relativ kostenintensive Berechnung zu vermeiden:

```
public boolean verify(int value, boolean needPrime) {
    return !needPrime | isPrime(value);
}

public boolean verify(int value, boolean needPrime) {
    return !needPrime || isPrime(value);
}
```

In der ersten Version sind beide Operanden von | wird immer ausgewertet, daher wird die (teure) isPrime Methode unnötig aufgerufen. Die zweite Version vermeidet den unnötigen Aufruf mit || anstelle von | .

Die Schichtoperatoren (<<, >> und >>>)

Die Java-Sprache bietet drei Operatoren für die bitweise Verschiebung von 32- und 64-Bit-Ganzzahlwerten. Dies sind alles binäre Operatoren, wobei der erste Operand der Wert ist, der verschoben werden soll, und der zweite Operand, wie weit verschoben werden soll.

- Der << oder *Linksverschiebungsoperator* verschiebt den durch den ersten Operanden gegebenen Wert um die Anzahl der durch den zweiten Operanden gegebenen Bitpositionen nach *links* . Die leeren Stellen am rechten Ende sind mit Nullen gefüllt.
- Der Operator ">>" oder *arithmetische Verschiebung* verschiebt den durch den ersten Operanden angegebenen Wert um die durch den zweiten Operanden angegebene Anzahl von Bitpositionen nach *rechts* . Die leeren Stellen am linken Ende werden durch Kopieren des am weitesten links befindlichen Bits gefüllt. Dieser Vorgang wird als *Zeichenerweiterung bezeichnet* .
- Der Operator ">>>" oder der *logische Rechtsverschiebung* verschiebt den durch den ersten Operanden angegebenen Wert um die durch den zweiten Operanden angegebene Anzahl von Bitpositionen nach *rechts* . Die leeren Stellen am linken Ende sind mit Nullen gefüllt.

Anmerkungen:

1. Diese Operatoren erfordern als ersten Operanden einen int oder long Wert und erzeugen einen

Wert mit demselben Typ wie der erste Operand. (Sie müssen eine explizite Typumwandlung verwenden, wenn Sie das Ergebnis einer Verschiebung einer byte , short oder char Variablen zuweisen.)

2. Wenn Sie einen Shift-Operator mit einem ersten Operanden verwenden, der ein byte , ein char oder ein short , wird er zu einem int und der Vorgang erzeugt ein int .)
3. Der zweite Operand wird *modulo um die Anzahl der Bits der Operation* reduziert, um den Betrag der Verschiebung anzugeben. Weitere **Informationen zum mathematischen Konzept** des **Mods** finden Sie unter Modul-Beispiele .
4. Die Bits, die durch die Operation nach links oder rechts verschoben werden, werden verworfen. (Java bietet keinen primitiven "Drehen" -Operator.)
5. Der arithmetische Verschiebungsoperator ist äquivalent, indem eine Zahl (Zweierkomplement) durch eine Potenz von 2 dividiert wird.
6. Der linke Verschiebungsoperator multipliziert eine Zahl (Zweierkomplement) mit einer Potenz von 2.

Die folgende Tabelle zeigt die Auswirkungen der drei Schichtbedienungen. (Die Zahlen wurden in binärer Schreibweise angegeben, um die Visualisierung zu erleichtern.)

Operand1	Operand2	<<	>>	>>>
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

Es gibt Beispiele für den Benutzer von Schichtoperatoren bei der Bit-Bearbeitung

Der Lambda-Operator (->)

Ab Java 8 wird der Operator Lambda (->) verwendet, um einen Lambda-Ausdruck einzuführen. Es gibt zwei gängige Syntaxtypen, wie in diesen Beispielen veranschaulicht:

Java SE 8

```
a -> a + 1               // a lambda that adds one to its argument
a -> { return a + 1; }   // an equivalent lambda using a block.
```

Ein Lambda-Ausdruck definiert eine anonyme Funktion oder richtiger eine Instanz einer anonymen Klasse, die eine *funktionale Schnittstelle* implementiert.

(Dieses Beispiel ist hier der Vollständigkeit halber enthalten. Die vollständige Behandlung finden Sie im Thema Lambda-Ausdrücke .)

Die relationalen Operatoren (<, <=,>,> =)

Die Operatoren < , <= , > und >= sind binäre Operatoren zum Vergleichen numerischer Typen. Die Bedeutung der Operatoren ist wie erwartet. Wenn zum Beispiel a und b als byte , short , char , int , long , float , double oder die entsprechenden Boxed-Typen deklariert sind:

```
- `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`.
```

Der Ergebnistyp für diese Operatoren ist in allen Fällen boolean .

Mit Vergleichsoperatoren können Zahlen mit unterschiedlichen Typen verglichen werden. Zum Beispiel:

```
int i = 1;
long l = 2;
if (i < l) {
    System.out.println("i is smaller");
}
```

Beziehungsoperatoren können verwendet werden, wenn eine oder beide Zahlen Instanzen von geschachtelten numerischen Typen sind. Zum Beispiel:

```
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");
}
```

Das genaue Verhalten wird wie folgt zusammengefasst:

1. Wenn es sich bei einem der Operanden um einen Box-Typ handelt, ist er nicht im Boxmodus.
2. Wenn einer der Operanden jetzt ein byte , ein short oder ein char , wird er zu einem int .
3. Wenn die Typen der Operanden nicht gleich sind, wird der Operand mit dem Typ "Kleiner" zum Typ "Größer" befördert.
4. Der Vergleich wird mit den resultierenden Werten int , long , float oder double .

Bei relationalen Vergleichen mit Fließkommazahlen ist Vorsicht geboten:

- Ausdrücke, die Fließkommazahlen berechnen, führen häufig zu Rundungsfehlern, da die Fließkommadaten des Computers eine begrenzte Genauigkeit aufweisen.
- Beim Vergleich eines Ganzzahlentyps mit einem Gleitkommatyp kann die Konvertierung der Ganzzahl in Gleitkommazahl ebenfalls zu Rundungsfehlern führen.

Schließlich unterstützt Java die Verwendung von relationalen Operatoren mit anderen als den oben aufgeführten Typen. Sie *können* diese Operatoren beispielsweise *nicht* verwenden, um Zeichenfolgen, Zahlenarrays usw. zu vergleichen.

Operatoren online lesen: https://riptutorial.com/de/java/topic/176/operatoren

Kapitel 121: Oracle Official Code Standard

Einführung

Der offizielle Oracle-Style-Guide für die Java-Programmiersprache ist ein Standard, der von Oracle-Entwicklern befolgt wird und von jedem anderen Java-Entwickler empfohlen wird. Es umfasst Dateinamen, Dateiorganisation, Einrückung, Kommentare, Deklarationen, Anweisungen, Leerzeichen, Namenskonventionen, Programmierpraktiken und enthält ein Codebeispiel.

Bemerkungen

- Die obigen Beispiele folgen strikt dem neuen offiziellen Style Guide von Oracle. Sie werden also *nicht* subjektiv von den Autoren dieser Seite verfasst.
- Der offizielle Style Guide wurde sorgfältig geschrieben, um rückwärtskompatibel mit dem ursprünglichen Style Guide und dem Großteil des Codes in freier Wildbahn zu sein.
- Der offizielle Styleguide wurde von Brian Goetz (Java Language Architect) und Mark Reinhold (Chefarchitekt der Java-Plattform) einem Peer- Review unterzogen.
- Die Beispiele sind nicht normativ; Sie möchten zwar die korrekte Formatierung des Codes veranschaulichen, es gibt jedoch andere Möglichkeiten, den Code richtig zu formatieren. Dies ist ein allgemeiner Grundsatz: Es gibt mehrere Möglichkeiten, den Code zu formatieren, wobei alle den offiziellen Richtlinien entsprechen.

Examples

Regeln der Namensgebung

Paketnamen

- Die Paketnamen sollten alle ohne Unterstriche oder andere Sonderzeichen in Kleinbuchstaben geschrieben werden.
- Paketnamen beginnen mit dem umgekehrten Berechtigungsbereich der Webadresse des Unternehmens des Entwicklers. Auf diesen Teil kann eine von der Projekt- / Programmstruktur abhängige Paketunterstruktur folgen.
- Verwenden Sie keine Pluralform. Folgen Sie der Konvention der Standard-API, die zum Beispiel java.lang.annotation und nicht java.lang.annotations .
- **Beispiele:** com.yourcompany.widget.button , com.yourcompany.core.api

Klassen-, Schnittstellen- und Aufzählungsnamen

- Klassen- und Enumnamen sollten normalerweise Nomen sein.
- Schnittstellennamen sollten in der Regel Substantive oder Adjektive sein, die auf ... enden.
- Verwenden Sie die Groß- und Kleinschreibung mit dem ersten Buchstaben in jedem Wort (z. B. CamelCase).
- ^[AZ][a-zA-Z0-9]*$ den regulären Ausdruck ^[AZ][a-zA-Z0-9]*$.
- Verwenden Sie ganze Wörter und vermeiden Sie die Verwendung von Abkürzungen, es sei denn, die Abkürzung wird häufiger als die Langform verwendet.
- Formatieren Sie eine Abkürzung als Wort, wenn sie Teil eines längeren Klassennamens ist.
- **Beispiele:** ArrayList , BigInteger , ArrayIndexOutOfBoundsException , Iterable .

Methodennamen

Methodennamen sollten normalerweise Verben oder andere Beschreibungen von Aktionen sein

- Sie sollten mit dem regulären Ausdruck ^[az][a-zA-Z0-9]*$.

- Verwenden Sie die Groß- und Kleinschreibung mit dem ersten Buchstaben.
- **Beispiele:** toString , hashCode

Variablen

Variablennamen sollten in Groß- und Kleinschreibung mit dem ersten Buchstaben geschrieben werden

- ^[az][a-zA-Z0-9]*$ den regulären Ausdruck ^[az][a-zA-Z0-9]*$
- Weitere Empfehlung: Variablen
- **Beispiele:** elements , currentIndex

Typvariablen

In einfachen Fällen, in denen nur wenige Typvariablen beteiligt sind, verwenden Sie einen einzelnen Großbuchstaben.

- Passen Sie den regulären Ausdruck ^[AZ][0-9]?$
- Wenn ein Buchstabe aussagekräftiger ist als ein anderer (z. B. K und V für Schlüssel und Werte in Karten oder R für einen Funktionsrückgabetyp), verwenden Sie diesen, andernfalls T
- Verwenden Sie in komplexen Fällen, in denen Variablen mit einem Buchstaben verwirrend werden, längere Namen, die in Großbuchstaben geschrieben sind, und verwenden Sie einen Unterstrich (_), um Wörter zu trennen.
- **Beispiele:** T , V , SRC_VERTEX

Konstanten

Konstanten (static final Endfelder, deren Inhalt unveränderlich ist, durch Sprachregeln oder durch Konventionen), sollten mit Großbuchstaben und Unterstrich (_) benannt werden, um Wörter zu trennen.

- ^[AZ][A-Z0-9]*(_[A-Z0-9]+)*$ den regulären Ausdruck ^[AZ][A-Z0-9]*(_[A-Z0-9]+)*$
- **Beispiele:** BUFFER_SIZE , MAX_LEVEL

Andere Richtlinien zur Benennung

- Vermeiden Sie das Ausblenden / Spiegeln von Methoden, Variablen und Typvariablen in äußeren Bereichen.
- Lassen Sie die Ausführlichkeit des Namens von der Größe des Bereichs abhängen. (Verwenden Sie beispielsweise beschreibende Namen für Felder großer Klassen und Kurznamen für lokale kurzlebige Variablen.)
- Wenn Sie öffentliche statische Member benennen, lassen Sie den Bezeichner selbstbeschreibend sein, wenn Sie der Meinung sind, dass er statisch importiert wird.
- Weiterführende Literatur: Naming Section (im offiziellen Java Style Guide)

Quelle: Java-Style-Richtlinien von Oracle

Java-Quelldateien

- Alle Zeilen müssen mit einem Zeilenvorschubzeichen (LF, ASCII-Wert 10) abgeschlossen werden und nicht beispielsweise CR oder CR + LF.
- Am Ende einer Zeile befindet sich möglicherweise kein Leerzeichen.
- Der Name einer Quelldatei muss mit dem Namen der enthaltenen Klasse, gefolgt von der Erweiterung .java , .java , auch für Dateien, die nur eine private .java enthalten. Dies gilt nicht für Dateien, die keine Klassendeklarationen enthalten, wie package-info.java .

Spezielle Charaktere

- Außer LF ist das einzige Leerzeichen Leerzeichen (ASCII-Wert 32). Beachten Sie, dass dies bedeutet, dass andere Leerzeichen (z. B. Zeichenfolgen- und Zeichenliterale) in Escape-Form geschrieben werden müssen.
- \' , \" , \\ , \t , \b , \r , \f und \n sollten gegenüber entsprechenden oktalen (zB \047) oder Unicode-Zeichen (zB \u0027) mit \u0027 bevorzugt werden.
- Sollte es zu Testzwecken erforderlich sein, gegen die obigen Regeln zu verstoßen, sollte der Test die erforderliche Eingabe programmgesteuert *generieren* .

Paket deklaration

```
package com.example.my.package;
```

Die Paketdeklaration sollte nicht Zeilenumbrüche enthalten, unabhängig davon, ob sie die empfohlene maximale Zeilenlänge überschreitet.

Anweisungen importieren

```
// 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;
```

- Importanweisungen sollten sortiert werden…
    - … In erster Linie durch nicht statische / statische Importe.
    - … Sekundär nach Paketursprung in folgender Reihenfolge
        - Java-Pakete
        - Javax-Pakete
        - externe Pakete (zB org.xml)
        - interne Pakete (zB com.sun)
    - … Tertiär nach Paket- und Klassenkennung in lexikographischer Reihenfolge
- Importanweisungen sollten nicht in Zeilen umbrochen werden, unabhängig davon, ob sie die empfohlene maximale Länge einer Zeile überschreiten.
- Es sollten keine ungenutzten Importe vorhanden sein.

Wildcard-Importe

- Platzhalterimporte sollten im Allgemeinen nicht verwendet werden.
- Beim Importieren einer großen Anzahl eng zusammengehöriger Klassen (z. B. beim Implementieren eines Besuchers über einen Baum mit Dutzenden verschiedener Knotenklassen) kann ein Platzhalterimport verwendet werden.
- In jedem Fall sollte nicht mehr als ein Platzhalterimport pro Datei verwendet werden.

Klassenstruktur

Reihenfolge der Teilnehmer

Die Teilnehmer sollten wie folgt bestellt werden:

1. Felder (in der Reihenfolge öffentlich, geschützt und privat)
2. Konstrukteure
3. Fabrikmethoden
4. Andere Methoden (in der Reihenfolge öffentlich, geschützt und privat)

Das Sortieren von Feldern und Methoden hauptsächlich nach ihren Zugriffsmodifizierern oder Bezeichnern ist nicht erforderlich.

Hier ist ein Beispiel für diese Reihenfolge:

```
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 + "]";
    }

}
```

Gruppierung der Schüler

- Verwandte Felder sollten zusammen gruppiert werden.
- Ein verschachtelter Typ kann direkt vor seiner ersten Verwendung deklariert werden. Andernfalls sollte es vor den Feldern angegeben werden.
- Konstruktoren und überladene Methoden sollten nach Funktionalität gruppiert und mit zunehmender Priorität geordnet werden. Dies impliziert, dass die Delegation zwischen diesen Konstrukten im Code nach unten fließt.
- Konstruktoren sollten ohne andere Mitglieder dazwischen gruppiert werden.
- Überladene Varianten einer Methode sollten ohne andere Mitglieder dazwischen gruppiert werden.

Modifikatoren

```
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();
}
```

- Modifikatoren sollten in der folgenden Reihenfolge angegeben werden
    - Zugriffsmodifizierer (public / private / protected)

        - abstract
        - static
        - final
        - transient
        - volatile
        - default
        - synchronized
        - native
        - strictfp

- Modifikatoren sollten nicht ausgeschrieben werden, wenn sie implizit sind. Zum Beispiel sollten Schnittstellenmethoden weder als public noch als abstract deklariert werden, und geschachtelte Enums und Schnittstellen sollten nicht als statisch deklariert werden.
- Methodenparameter und lokale Variablen sollten nicht als final deklariert werden, es sei denn, sie verbessern die Lesbarkeit oder dokumentieren eine tatsächliche Entwurfsentscheidung.
- Felder sollten als final deklariert werden, es sei denn, es besteht ein zwingender Grund, sie veränderbar zu machen.

Vertiefung

- Die Einrückungsebene besteht aus **vier Leerzeichen** .
- Für die Einrückung dürfen nur Leerzeichen verwendet werden. **Keine Registerkarten**
- Leere Zeilen dürfen nicht eingerückt werden. (Dies wird durch die Regel "Leerzeichen ohne Leerzeichen" impliziert.)
- case Leitungen sollten mit vier Räumen und Aussagen im Fall eingerückt werden sollen , mit weiteren vier Leerzeichen eingerückt werden.

```
switch (var) {
    case TWO:
        setChoice("two");
        break;
    case THREE:
        setChoice("three");
        break;
    default:
        throw new IllegalArgumentException();
}
```

Richtlinien zum Einrücken von Fortsetzungszeilen finden Sie unter Wrapping-Anweisungen .

Anweisungen einwickeln

- Quellcode und Kommentare sollten im Allgemeinen nicht mehr als 80 Zeichen pro Zeile und selten mehr als 100 Zeichen pro Zeile (einschließlich Einrückung) umfassen.

  Das Zeichenlimit muss von Fall zu Fall beurteilt werden. Was wirklich zählt, ist die semantische "Dichte" und Lesbarkeit der Linie. Durch das lange Versehen von Zeilen sind sie schwer lesbar. In ähnlicher Weise kann das Durchführen von "heroischen Versuchen", sie in 80 Säulen einzupassen, das Lesen schwer machen. Die hier skizzierte Flexibilität soll es den Entwicklern ermöglichen, diese Extreme zu vermeiden und die Nutzung von Monitor-Immobilien nicht zu maximieren.

- URLs oder Beispielbefehle sollten nicht umbrochen werden.

```
// 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(", "));
```

- Ein Wrapping auf einem höheren syntaktischen Level wird dem Wrapping auf einem niedrigeren syntaktischen Level vorgezogen.
- Es sollte höchstens eine Anweisung pro Zeile enthalten.
- Eine Fortsetzungszeile sollte auf eine der folgenden vier Arten eingerückt werden
    - **Variante 1** : Mit 8 zusätzlichen Leerzeichen relativ zur Einrückung der vorherigen Zeile.
    - **Variante 2** : Mit 8 zusätzlichen Leerzeichen relativ zur Startspalte des umschlossenen Ausdrucks.
    - **Variante 3** : An vorherigem Geschwisterausdruck ausgerichtet (solange klar ist, dass es sich um eine Fortsetzungszeile handelt)
    - **Variante 4** : Ausgerichtet mit dem vorherigen Methodenaufruf in einem verketteten Ausdruck.

Wrapping Method Declarations

```
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 {
    …
```

```
}
```

- Methodendeklarationen können formatiert werden, indem die Argumente vertikal oder durch eine neue Zeile und +8 zusätzliche Leerzeichen aufgelistet werden
- Wenn eine Throws-Klausel umbrochen werden soll, setzen Sie den Zeilenumbruch vor die Throws-Klausel und stellen Sie sicher, dass sie sich aus der Argumentliste heraushebt, entweder durch Einrücken von +8 relativ zur Funktionsdeklaration oder +8 relativ zur vorherigen Zeile.

Ausdrücke einpacken

- Wenn sich eine Zeile der maximalen Zeichengrenze nähert, sollten Sie sie immer in mehrere Anweisungen / Ausdrücke unterteilen, anstatt die Zeile zu umbrechen.
- Pause vor den Operatoren.
- Pause vor dem. in verketteten Methodenaufrufen.

```
popupMsg("Inbox notification: You have "
        + newMsgs + " new messages");

// Don't! Looks like two arguments
popupMsg("Inbox notification: You have " +
         newMsgs + " new messages");
```

Whitespace

Vertikaler Whitespace

- Eine einzelne Leerzeile sollte verwendet werden, um…
    - Paket deklaration
    - Klassenerklärungen
    - Konstrukteure
    - Methoden
    - Statische Initialisierer
    - Instanzinitialisierer
- … Und kann verwendet werden, um logische Gruppen von zu trennen
    - Anweisungen importieren
    - Felder
    - Aussagen
- Mehrere aufeinanderfolgende Leerzeilen sollten nur zum Trennen von Gruppen verwandter Mitglieder und nicht als Standardabstand zwischen Mitgliedern verwendet werden.

Horizontales Leerzeichen

- Ein einzelner Raum sollte verwendet werden…
    - So trennen Sie Schlüsselwörter von benachbarten öffnenden oder schließenden Klammern und Klammern
    - Vor und nach allen binären Operatoren und Operator-ähnlichen Symbolen wie Pfeilen in Lambda-Ausdrücken und Doppelpunkt in erweiterten for-Schleifen (jedoch nicht vor dem Doppelpunkt eines Labels)
    - Nach // beginnt ein Kommentar.
    - Nach Kommas trennen Argumente und Semikolons die Teile einer for-Schleife.
    - Nach der schließenden Klammer eines Casts.

- In Variablendeklarationen wird das Ausrichten von Typen und Variablen nicht empfohlen.

Variablendeklarationen

- Eine Variable pro Deklaration (und höchstens eine Deklaration pro Zeile)
- Eckige Klammern von Arrays sollten am Typ (String[] args) und nicht an der Variablen (String args[]) liegen.
- Deklarieren Sie eine lokale Variable direkt vor ihrer ersten Verwendung und initialisieren Sie sie so nahe wie möglich an der Deklaration.

Anmerkungen

Deklarationsanmerkungen sollten in einer separaten Zeile von der kommentierten Deklaration stehen.

```
@SuppressWarnings("unchecked")
public T[] toArray(T[] typeHolder) {
    ...
}
```

Wenige oder kurze Anmerkungen, die eine einzeilige Methode kommentieren, können jedoch in derselben Zeile wie die Methode stehen, wenn dadurch die Lesbarkeit verbessert wird. Zum Beispiel kann man schreiben:

```
@Nullable String getName() { return name; }
```

Aus Gründen der Konsistenz und Lesbarkeit sollten entweder alle Anmerkungen in derselben Zeile oder jede Anmerkung in einer separaten Zeile stehen.

```
// Bad.
@Deprecated @SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}

// Even worse.
@Deprecated @SafeVarargs
@CustomAnnotation public final Tuple<T> extend(T... elements) {
    ...
}

// Good.
@Deprecated
@SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}

// Good.
@Deprecated @SafeVarargs @CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}
```

Lambda-Ausdrücke

```
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));
```

- Expression-Lambdas werden gegenüber einzeiligen Block-Lambdas bevorzugt.
- Methodenreferenzen sollten im Allgemeinen gegenüber Lambda-Ausdrücken bevorzugt werden.
- Bei Methodenreferenzen für gebundene Instanzen oder bei Methoden mit mehr als eins kann ein Lambda-Ausdruck leichter verständlich und daher bevorzugt sein. Vor allem, wenn das Verhalten der Methode nicht aus dem Kontext ersichtlich ist.
- Die Parametertypen sollten weggelassen werden, es sei denn, sie verbessern die Lesbarkeit.
- Wenn sich ein Lambda-Ausdruck über mehrere Zeilen erstreckt, sollten Sie eine Methode erstellen.

Redundante Klammern

```
return flag ? "yes" : "no";

String cmp = (flag1 != flag2) ? "not equal" : "equal";

// Don't do this
return (flag ? "yes" : "no");
```

- Redundante Gruppierungsklammern (dh Klammern, die die Auswertung nicht beeinflussen) können verwendet werden, wenn sie die Lesbarkeit verbessern.
- Redundante Gruppierungsklammern sollten normalerweise in kürzeren Ausdrücken mit gemeinsamen Operatoren weggelassen werden, aber in längeren Ausdrücken oder Ausdrücken mit Operatoren, deren Vorrang und Assoziativität ohne Klammern unklar sind. Zu letzteren gehören ternäre Ausdrücke mit nicht-trivialen Bedingungen.
- Der gesamte Ausdruck, der auf ein return Schlüsselwort folgt, darf nicht in Klammern stehen.

Literale

```
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 Literale sollten das Suffix Großbuchstabe L .
- Hexadezimal-Literale sollten die Großbuchstaben A - F .
- Alle anderen numerischen Präfixe, Infixe und Suffixe sollten aus Kleinbuchstaben bestehen.

Hosenträger

```
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");
        }
    }
}
```

- Öffnende Klammern sollten am Ende der aktuellen Zeile und nicht in einer eigenen Zeile stehen.
- Vor einer schließenden Klammer sollte eine neue Zeile stehen, es sei denn, der Block ist leer (siehe Kurzformen unten).
- Klammern werden auch empfohlen, wenn die Sprache sie optional macht, z. B. einzeilige Wenn- und Schleifenkörper.
    - Wenn ein Block mehr als eine Zeile umfasst (einschließlich Kommentare), muss er geschweifte Klammern enthalten.
    - Wenn einer der Blöcke in einer if / else Anweisung geschweifte Klammern enthält, muss der andere Block ebenfalls.
    - Wenn der Block in einem umschließenden Block den letzten Platz erreicht, muss er geschweifte Klammern haben.
- Die Schlüsselwörter else , catch und while in do…while Schleifen stehen in derselben Zeile wie die schließende Klammer des vorhergehenden Blocks.

Kurz Formen

```
enum Response { YES, NO, MAYBE }
public boolean isReference() { return true; }
```

Die obigen Empfehlungen sollen die Einheitlichkeit verbessern (und damit die Bekanntheit / Lesbarkeit erhöhen). In manchen Fällen sind „Kurzformen", die von den obigen Richtlinien abweichen, genauso lesbar und können stattdessen verwendet werden. Diese Fälle umfassen beispielsweise einfache Aufzählungsdeklarationen und triviale Methoden und Lambda-Ausdrücke.

Oracle Official Code Standard online lesen: https://riptutorial.com/de/java/topic/2697/oracle-official-code-standard

Kapitel 122: Ortszeit

Syntax

- LocalTime Zeit = LocalTime.now (); // Initialisiert mit der aktuellen Systemuhr
- LocalTime Zeit = LocalTime.MIDNIGHT; // 00:00
- LocalTime Zeit = LocalTime.NOON; // 12:00
- LocalTime-Zeit = LocalTime.of (12, 12, 45); // 12:12:45

Parameter

Methode	Ausgabe
`LocalTime.of(13, 12, 11)`	13:12:11
`LocalTime.MIDNIGHT`	00:00
`LocalTime.NOON`	12:00
`LocalTime.now()`	Aktuelle Uhrzeit ab Systemuhr
`LocalTime.MAX`	Die maximal unterstützte Ortszeit 23: 59: 59.999999999
`LocalTime.MIN`	Die minimale unterstützte Ortszeit 00:00
`LocalTime.ofSecondOfDay(84399)`	23:59:59, Ermittelt die Zeit vom Tageswert
`LocalTime.ofNanoOfDay(2000000000)`	00:00:02, Ermittelt die Zeit vom Tageswert in Nanometern

Bemerkungen

Als Klassenname bezeichnet LocalTime eine Zeit ohne Zeitzone. Es ist kein Datum. Es ist ein einfaches Etikett für eine bestimmte Zeit.

Die Klasse ist wertbasiert und die Methode equals sollte bei Vergleichen verwendet werden.

Diese Klasse stammt aus dem Paket java.time.

Examples

Zeitänderung

Sie können Stunden, Minuten, Sekunden und Nanosekunden hinzufügen:

```
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)
```

Zeitzonen und deren Zeitunterschied

```
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");
    }
}
```

Zeit zwischen zwei LocalTime

Es gibt zwei äquivalente Methoden zur Berechnung der Zeiteinheit zwischen zwei LocalTime : (1) until(Temporal, TemporalUnit) -Methode und (2) TemporalUnit.between(Temporal, Temporal) .

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;

public class AmountOfTime {

    public static void main(String[] args) {

        LocalTime start = LocalTime.of(1, 0, 0); // hour, minute, second
        LocalTime end = LocalTime.of(2, 10, 20); // hour, minute, second

        long halfDays1 = start.until(end, ChronoUnit.HALF_DAYS); // 0
        long halfDays2 = ChronoUnit.HALF_DAYS.between(start, end); // 0

        long hours1 = start.until(end, ChronoUnit.HOURS); // 1
        long hours2 = ChronoUnit.HOURS.between(start, end); // 1

        long minutes1 = start.until(end, ChronoUnit.MINUTES); // 70
        long minutes2 = ChronoUnit.MINUTES.between(start, end); // 70

        long seconds1 = start.until(end, ChronoUnit.SECONDS); // 4220
        long seconds2 = ChronoUnit.SECONDS.between(start, end); // 4220

        long millisecs1 = start.until(end, ChronoUnit.MILLIS); // 4220000
        long millisecs2 = ChronoUnit.MILLIS.between(start, end); // 4220000

        long microsecs1 = start.until(end, ChronoUnit.MICROS); // 4220000000
        long microsecs2 = ChronoUnit.MICROS.between(start, end); // 4220000000

        long nanosecs1 = start.until(end, ChronoUnit.NANOS); // 4220000000000
        long nanosecs2 = ChronoUnit.NANOS.between(start, end); // 4220000000000
```

```
        // Using others ChronoUnit will be thrown UnsupportedTemporalTypeException.
        // The following methods are examples thereof.
        long days1 = start.until(end, ChronoUnit.DAYS);
        long days2 = ChronoUnit.DAYS.between(start, end);
    }
}
```

Intro

LocalTime ist eine unveränderliche Klasse und Thread-Safe, die zur Darstellung der Zeit verwendet wird und oft als Stunde-Minute-Sekunde angegeben wird. Die Zeit wird in Nanosekundengenauigkeit dargestellt. Beispielsweise kann der Wert "13: 45.30.123456789" in einer LocalTime gespeichert werden.

Diese Klasse speichert kein Datum oder keine Zeitzone. Es ist stattdessen eine Beschreibung der Ortszeit, die auf einer Wanduhr angezeigt wird. Es kann keinen Zeitpunkt auf der Zeitleiste ohne zusätzliche Informationen wie Offset oder Zeitzone darstellen. Dies ist eine wertbasierte Klasse, für Vergleiche sollte die Methode equals verwendet werden.

Felder

MAX - Die maximal unterstützte LocalTime, '23: 59: 59.999999999 '. Mitternacht, Minute, Mittag

Wichtige statische Methoden

jetzt (), jetzt (Uhr), jetzt (ZoneId-Zone), analysieren (Zeichenfolge-Text)

Wichtige Instanzmethoden

isAfter (LocalTime other), isBefore (LocalTime other), Minus (TemporalAmount AnzahlToSubtract), Minus (Long MengeToSubtract, TemporalUnit-Einheit), plus (TemporalAmount MengeToAdd), plus (Long MengeToAdd, TemporalUnit-Einheit)

```
ZoneId zone = ZoneId.of("Asia/Kolkata");
LocalTime now = LocalTime.now();
LocalTime now1 = LocalTime.now(zone);
LocalTime then = LocalTime.parse("04:16:40");
```

Der Zeitunterschied kann auf eine der folgenden Arten berechnet werden

```
long timeDiff = Duration.between(now, now1).toMinutes();
long timeDiff1 = java.time.temporal.ChronoUnit.MINUTES.between(now2, now1);
```

Sie können auch Stunden, Minuten oder Sekunden von jedem LocalTime-Objekt hinzufügen / subtrahieren.

minusStunden (lange StundenToSubtract), minusMinuten (lange StundenToMinute), minusNanos (lange nanosToSubtract), minusSekunden (lange SekundenToSubtract), plusStunden (lange StundenToSubtract), plusMinutes (lange StundenToMinutes), plusNanos ().

```
now.plusHours(1L);
now1.minusMinutes(20L);
```

Ortszeit online lesen: https://riptutorial.com/de/java/topic/3065/ortszeit

Kapitel 123: Pakete

Einführung

Paket in Java wird verwendet, um Klasse und Schnittstellen zu gruppieren. Dies hilft dem Entwickler, Konflikte zu vermeiden, wenn sehr viele Klassen vorhanden sind. Wenn wir dieses Paket verwenden, können die Klassen eine Klasse / Schnittstelle mit demselben Namen in verschiedenen Paketen erstellen. Durch die Verwendung von Paketen können wir das Stück erneut in eine andere Klasse importieren. Es gibt viele *eingebaute Pakete* in Java wie> 1.java.util> 2.java.lang> 3.java.io Wir können unsere eigenen *benutzerdefinierten Pakete definieren* .

Bemerkungen

Pakete bieten Zugriffsschutz.

> Die Paketanweisung muss die erste Zeile des Quellcodes sein. Es kann nur ein Paket in einer Quelldatei enthalten.

Mit Hilfe von Paketen können Konflikte zwischen verschiedenen Modulen vermieden werden.

Examples

Verwenden von Packages zum Erstellen von Klassen mit demselben Namen

Erste Testklasse:

```
package foo.bar

public class Test {

}
```

Auch Test.class in einem anderen Paket

```
package foo.bar.baz

public class Test {

}
```

Dies ist in Ordnung, da die beiden Klassen in unterschiedlichen Paketen vorhanden sind.

Package Protected Scope verwenden

Wenn Sie in Java keinen Zugriffsmodifizierer angeben, ist der Standardbereich für Variablen die auf Paket geschützte Ebene. Das bedeutet, dass Klassen auf die Variablen anderer Klassen innerhalb desselben Pakets zugreifen können, als ob diese Variablen öffentlich verfügbar wären.

```
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
}
```

Diese Methode funktioniert nicht für eine Klasse in einem anderen Paket:

```
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
}
```

Pakete online lesen: https://riptutorial.com/de/java/topic/8273/pakete

Kapitel 124: Parallele Programmierung mit dem Fork / Join-Framework

Examples

Gabel- / Join-Aufgaben in Java

Das Fork / Join-Framework in Java ist ideal für ein Problem, das in kleinere Teile aufgeteilt und parallel gelöst werden kann. Die grundlegenden Schritte eines Gabel / Join-Problems sind:

- Teilen Sie das Problem in mehrere Teile
- Löse die einzelnen Teile parallel zueinander
- Kombinieren Sie jede der Unterlösungen zu einer Gesamtlösung

Eine ForkJoinTask ist die Schnittstelle, die ein solches Problem definiert. Es wird allgemein erwartet, dass Sie eine seiner abstrakten Implementierungen (normalerweise die RecursiveTask) subclassieren, anstatt die Schnittstelle direkt zu implementieren.

In diesem Beispiel werden wir eine Sammlung von ganzen Zahlen zusammenfassen, die sich teilen, bis Chargengrößen von maximal zehn erreicht werden.

```
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;
        }
    }
}
```

Eine Instanz dieser Aufgabe kann jetzt an eine Instanz von ForkJoinPool übergeben werden .

```
// 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();
```

Parallele Programmierung mit dem Fork / Join-Framework online lesen: https://riptutorial.com/de/java/topic/4245/parallele-programmierung-mit-dem-fork---join-framework

Kapitel 125: Polymorphismus

Einführung

Polymorphismus ist eines der wichtigsten OOP-Konzepte (objektorientierte Programmierung). Das Polymorphismuswort wurde von den griechischen Wörtern "Poly" und "Morphen" abgeleitet. Poly bedeutet "viele" und Morphen bedeutet "Formen" (viele Formen).

Es gibt zwei Möglichkeiten, Polymorphie durchzuführen. **Method Overloading** und **Method Overriding** .

Bemerkungen

Interfaces sind ein weiterer Weg, um Polymorphismus in Java zu erreichen, abgesehen von der Klassenvererbung. Schnittstellen definieren eine Liste von Methoden, die die API des Programms bilden. Klassen müssen eine interface implement interface indem sie alle ihre Methoden überschreiben.

Examples

Methodenüberladung

Methodenüberladung , auch **Funktionsüberladung genannt** , ist die Fähigkeit einer Klasse, mehrere Methoden mit demselben Namen zu haben, vorausgesetzt, sie unterscheiden sich entweder in der Anzahl oder im Typ der Argumente.

Der Compiler überprüft die ***Methodensignatur*** auf Methodenüberladung.

Die Methodensignatur besteht aus drei Dingen:

1. Methodenname
2. Anzahl der Parameter
3. Arten von Parametern

Wenn diese drei für zwei Methoden in einer Klasse gleich sind, gibt der Compiler einen **doppelten Methodenfehler aus** .

Diese Art von Polymorphismus wird als *statischer* Polymorphismus oder als *Kompilierzeitpolymorphismus bezeichnet,* da die geeignete aufzurufende Methode vom Compiler während der Kompilierzeit auf Grundlage der Argumentliste festgelegt wird.

```
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));
    }

}
```

Dies führt zu:

```
2
6
4.000000
```

Überladene Methoden können statisch oder nicht statisch sein. Dies wirkt sich auch nicht auf das Überladen von Methoden aus.

```
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
    }
}
```

Wenn Sie den Rückgabetyp der Methode ändern, können wir sie nicht als Methodenüberladung abrufen.

```
public class Polymorph {

void methodOverloaded(){
    //No argument and No return type
}

int methodOverloaded(){
    //No argument and int return type
    return 0;
}
```

Überschreiben der Methode

Das Überschreiben von Methoden ist die Fähigkeit von Subtypen, das Verhalten ihrer Supertypen

neu zu definieren (zu überschreiben).

In Java wird dies in Unterklassen übersetzt, die die in der Oberklasse definierten Methoden überschreiben. In Java sind alle nicht primitiven Variablen tatsächlich references , die mit Zeigern auf die Position des tatsächlichen Objekts im Arbeitsspeicher verwandt sind. Die references nur einen Typ, den Typ, mit dem sie deklariert wurden. Sie können jedoch auf ein Objekt des deklarierten Typs oder eines seiner Subtypen zeigen.

Wenn eine Methode für eine reference aufgerufen wird, wird die entsprechende **Methode des tatsächlichen Objekts aufgerufen, auf das gezeigt wird .**

```
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
    }
}
```

Regeln, die zu beachten sind

Um eine Methode in der Unterklasse zu überschreiben, **MUSS** die überschreibende Methode (dh die in der Unterklasse) **HABEN :**

- gleicher Name
- gleicher Rückgabetyp bei Primitiven (eine Unterklasse ist für Klassen zulässig, dies wird auch als kovariante Rückgabetypen bezeichnet)
- gleicher Typ und Reihenfolge der Parameter
- Es kann nur die Ausnahmen werfen, die in der Throws-Klausel der Methode der Superklasse deklariert sind, oder Ausnahmen, die Unterklassen der deklarierten Ausnahmen sind. Es kann sich auch entscheiden, KEINE Ausnahme auszulösen. Die Namen der Parametertypen spielen keine Rolle. Zum Beispiel ist void methodX (int i) mit void methodX (int k) identisch.
- Endgültige oder statische Methoden können nicht überschrieben werden. Nur, dass wir nur den Methodenkörper ändern können.

Hinzufügen von Verhalten durch Hinzufügen von Klassen, ohne vorhandenen Code zu berühren

```
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");
    }
}
```

Erläuterung

a) Die MakeThingsFly Klasse kann mit FlyingMachine Typen vom Typ FlyingMachine .

b) Die Methode letTheMachinesFly funktioniert auch ohne Änderung (!), wenn Sie eine neue Klasse hinzufügen, beispielsweise PropellerPlane :

```
public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
        for (FlyingMachine flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
```

```
    }
}
```

Das ist die Kraft des Polymorphismus. Sie können das Open-Closed-Prinzip damit implementieren.

Virtuelle Funktionen

Virtuelle Methoden sind Methoden in Java, die nicht statisch sind und das Schlüsselwort Final nicht enthalten. Alle Methoden sind standardmäßig in Java virtuell. Virtuelle Methoden spielen eine wichtige Rolle in Polymorphism, da untergeordnete Klassen in Java die Methoden ihrer übergeordneten Klassen überschreiben können, wenn die zu überschreibende Funktion nicht statisch ist und dieselbe Methodensignatur aufweist.

Es gibt jedoch einige Methoden, die nicht virtuell sind. Wenn die Methode beispielsweise als privat oder mit dem Schlüsselwort final deklariert ist, ist die Methode nicht virtuell.

Betrachten Sie das folgende modifizierte Beispiel für die Vererbung mit virtuellen Methoden aus diesem StackOverflow-Beitrag. Wie funktionieren virtuelle Funktionen in C # und 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");

    }
}
```

Wenn wir die Klasse B aufrufen und hello () und boo () aufrufen, erhalten wir als Ergebnis die Ausgabe "No" und "Say haha", da B die gleichen Methoden von A überschreibt Wenn Sie die Methode überschreiben, ist es wichtig zu verstehen, dass die Methoden in Klasse A standardmäßig alle Virtual sind.

Außerdem können wir virtuelle Methoden mit dem abstrakten Schlüsselwort implementieren. Methoden, die mit dem Schlüsselwort "abstract" deklariert werden, haben keine Methodendefinition, dh der Körper der Methode ist noch nicht implementiert. Betrachten Sie das Beispiel von oben noch einmal, außer die boo () -Methode ist als abstrakt deklariert:

```
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");

    }
}
```

Wenn wir boo () von B aufrufen, ist die Ausgabe immer noch "Say haha", da B die abstrakte Methode boo () erbt und boo () die Ausgabe "Say haha" ausgibt.

Verwendete Quellen und weitere Lesungen:

Wie funktionieren virtuelle Funktionen in C # und Java?

Sehen Sie sich diese großartige Antwort an, die umfassendere Informationen zu virtuellen Funktionen enthält:

Können Sie virtuelle Funktionen / Methoden in Java schreiben?

Polymorphismus und verschiedene Arten des Überschreibens

Aus dem Java- Tutorial

> Die Wörterbuchdefinition für Polymorphismus bezieht sich auf ein Prinzip in der Biologie, bei dem ein Organismus oder eine Spezies viele verschiedene Formen oder Stufen haben kann. Dieses Prinzip kann auch auf objektorientierte Programmierung und Sprachen wie die Java-Sprache angewendet werden. ***Unterklassen einer Klasse können ihre eigenen eindeutigen Verhalten definieren und dabei einige Funktionen der übergeordneten Klasse gemeinsam nutzen.***

Sehen Sie sich dieses Beispiel an, um die verschiedenen Arten des Überschreibens zu verstehen.

1. Die Basisklasse stellt keine Implementierung bereit und die Unterklasse muss die vollständige Methode überschreiben. (Abstrakt)
2. Die Basisklasse stellt eine Standardimplementierung bereit und eine Unterklasse kann das Verhalten ändern
3. Die Unterklasse fügt der Basisklassenimplementierung eine Erweiterung hinzu, indem sie super.methodName() als erste Anweisung super.methodName()
4. Die Basisklasse definiert die Struktur des Algorithmus (Template-Methode) und die Unterklasse überschreibt einen Teil des Algorithmus

Code-Auszug:

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

Ausgabe:

```
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:
```

Polymorphismus online lesen: https://riptutorial.com/de/java/topic/980/polymorphismus

Kapitel 126: Primitive Datentypen

Einführung

Die 8 primitiven Datentypen byte , short , int , long , char , boolean , float und double speichern die meisten numerischen Rohdaten in Java-Programmen.

Syntax

- int aInt = 8; // Der definierende (Nummer) Teil dieser int-Deklaration wird als Literal bezeichnet.
- int hexInt = 0x1a; // = 26; Sie können Literale mit Hexadezimalwerten vor *0x definieren* .
- int binInt = 0b11010; // = 26; Sie können auch binäre Literale definieren. vorangestellt mit *0b* .
- long goodLong = 10000000000L; // Integer-Literale sind standardmäßig vom Typ int. Durch das Hinzufügen des L am Ende des Literals sagen Sie dem Compiler, dass das Literal lang ist. Andernfalls würde der Compiler einen Fehler "Integer number too large" ausgeben.
- double aDouble = 3,14; // Fließkomma-Literale haben standardmäßig den doppelten Typ.
- Float aFloat = 3.14F; // Standardmäßig wäre dieses Literal doppelt gewesen (und hat den Fehler "Inkompatible Typen" verursacht), aber durch Hinzufügen eines F sagen wir dem Compiler, dass es sich um einen Float handelt.

Bemerkungen

Java verfügt über 8 *primitive Datentypen* , nämlich boolean , byte , short , char , int , long , float und double . (Alle anderen Typen sind *Referenztypen* . Dies umfasst alle Array-Typen und integrierte Objekttypen / -klassen, die in der Java-Sprache eine besondere Bedeutung haben; z. B. String , Class und Throwable und ihre Unterklassen.)

Das Ergebnis aller Operationen (Addition, Subtraktion, Multiplikation usw.) eines primitiven Typs ist mindestens ein int also einen short zu einem short hinzufügen, wird ein int erzeugt, ebenso wie das Hinzufügen eines byte zu einem byte oder eines char zu einem char . Wenn Sie das Ergebnis davon einem Wert desselben Typs zuordnen möchten, müssen Sie es umwandeln. z.B

```
byte a = 1;
byte b = 2;
byte c = (byte) (a + b);
```

Wenn die Operation nicht umgesetzt wird, führt dies zu einem Kompilierungsfehler.

Dies ist auf den folgenden Teil der Java-Sprachspezifikation, §2.11.1, zurückzuführen :

> Ein Compiler codiert Lasten von Literalwerten der Typen byte und short mit Java Virtual Machine-Anweisungen, die diese Werte zur Kompilierzeit oder zur Laufzeit auf Werte des Typs int . Ladungen von Literalwerten der Typen boolean und char werden mit Anweisungen codiert, die das Literal zur Kompilierzeit oder zur Laufzeit auf einen Wert vom Typ int auf null erweitern. [..]. Daher werden die meisten Operationen mit Werten der tatsächlichen Typen boolean , byte , char und short von Anweisungen korrekt ausgeführt, die mit Werten des Rechentyps int .

Der Grund dafür ist auch in diesem Abschnitt angegeben:

> Angesichts der **Ein-Byte-Opcode-Größe** der Java Virtual Machine wird durch die Codierung von Typen in Opcodes Druck auf das Design des Befehlssatzes ausgeübt. Wenn jede getippte Anweisung alle Laufzeitdatentypen der Java Virtual Machine unterstützt, gibt es mehr Anweisungen, als in einem byte dargestellt werden könnten. [...] Mit

separaten Anweisungen können Sie bei Bedarf zwischen nicht unterstützten und unterstützten Datentypen konvertieren.

Examples

Das int Primitive

Ein primitiver Datentyp wie int enthält Werte direkt in der verwendeten Variable, während eine mit Integer deklarierte Variable einen Verweis auf den Wert enthält.

Gemäß Java-API : "Die Integer-Klasse hüllt einen Wert des primitiven Typs int in ein Objekt ein. Ein Objekt vom Typ Integer enthält ein einzelnes Feld, dessen Typ int ist."

Standardmäßig ist int eine 32-Bit-Ganzzahl mit Vorzeichen. Es kann einen Minimalwert von -2^{31} und einen Maximalwert von $2^{31} - 1$ speichern.

```
int example = -42;
int myInt = 284;
int anotherInt = 73;

int addedInts = myInt + anotherInt; // 284 + 73 = 357
int subtractedInts = myInt - anotherInt; // 284 - 73 = 211
```

Wenn Sie eine Nummer außerhalb dieses Bereichs speichern möchten, sollten Sie stattdessen long verwenden. Das Überschreiten des Wertebereichs von int führt zu einem Integer-Überlauf, wodurch der Wert, der den Bereich überschreitet, an der gegenüberliegenden Stelle des Bereichs hinzugefügt wird (Positiv wird negativ und umgekehrt). Der Wert ist ((value - MIN_VALUE) % RANGE) + MIN_VALUE oder ((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
```

Die maximalen und minimalen Werte von int finden Sie unter:

```
int high = Integer.MAX_VALUE;    // high == 2147483647
int low = Integer.MIN_VALUE;     // low == -2147483648
```

Der Standardwert eines int ist *0*

```
int defaultInt;    // defaultInt == 0
```

Das kurze Primitiv

Ein short ist eine vorzeichenbehaftete 16-Bit-Ganzzahl. Es hat einen Mindestwert von -2^{15} (-32.768) und einen Maximalwert von 2^{15} -1 (32.767).

```
short example = -48;
short myShort = 987;
short anotherShort = 17;

short addedShorts = (short) (myShort + anotherShort); // 1,004
short subtractedShorts = (short) (myShort - anotherShort); // 970
```

Die maximalen und minimalen Werte von short finden Sie unter:

```
short high = Short.MAX_VALUE;        // high == 32767
```

```
short low = Short.MIN_VALUE;         // low == -32768
```

Der Standardwert eines short ist *0*

```
short defaultShort;    // defaultShort == 0
```

Das lange Primitiv

Standardmäßig ist long eine 64-Bit-Ganzzahl mit Vorzeichen (in Java 8 kann sie entweder vorzeichenbehaftet oder vorzeichenlos sein). Signiert, kann es einen Minimalwert von -2^{63} und einen Maximalwert von $2^{63} - 1$ speichern und unsigniert einen Minimalwert von 0 und einen Maximalwert von $2^{64} - 1$

```
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
```

Die maximalen und minimalen Werte von long finden Sie unter:

```
long high = Long.MAX_VALUE;    // high == 9223372036854775807L
long low = Long.MIN_VALUE;     // low == -9223372036854775808L
```

Der Standardwert von long ist *0L*

```
long defaultLong;    // defaultLong == 0L
```

Hinweis: Der Buchstabe "L", der am Ende eines long Buchstabens angehängt wird, unterscheidet nicht zwischen Groß- und Kleinschreibung, es ist jedoch ratsam, Kapital zu verwenden, da es einfacher ist, sich von Ziffer 1 zu unterscheiden:

```
2L == 2l;            // true
```

Warnung: Java speichert Integer-Objektinstanzen im Bereich von -128 bis 127. Die Begründung wird hier erläutert: https://blogs.oracle.com/darcy/entry/boxing_and_caches_integer_valueof

Folgende Ergebnisse können gefunden werden:

```
Long val1 = 127L;
Long val2 = 127L;

System.out.println(val1 == val2); // true

Long val3 = 128L;
Long val4 = 128L;

System.out.println(val3 == val4); // false
```

Verwenden Sie den folgenden Code (ab Java 1.7), um 2 Object Long-Werte richtig zu vergleichen:

```
Long val3 = 128L;
Long val4 = 128L;

System.out.println(Objects.equal(val3, val4)); // true
```

Das Vergleichen eines primitiven long mit einem object long führt nicht zu einem falschen Negativ wie beim Vergleich von 2 Objekten mit == do.

Das boolesche Primitiv

Ein boolean kann einen von zwei Werten speichern, entweder " true oder " 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
```

Der Standardwert eines boolean Werts ist *false*

```
boolean defaultBoolean;    // defaultBoolean == false
```

Das Byte-Primitiv

Ein byte ist eine vorzeichenbehaftete 8-Bit-Ganzzahl. Es kann einen Minimalwert von -2^7 (-128) und einen Maximalwert von $2^7 - 1$ (127) speichern.

```
byte example = -36;
byte myByte = 96;
byte anotherByte = 7;

byte addedBytes = (byte) (myByte + anotherByte); // 103
byte subtractedBytes = (byte) (myBytes - anotherByte); // 89
```

Die maximalen und minimalen byte Werte finden Sie unter:

```
byte high = Byte.MAX_VALUE;        // high == 127
byte low = Byte.MIN_VALUE;         // low == -128
```

Der Standardwert eines byte ist *0*

```
byte defaultByte;    // defaultByte == 0
```

Das Float-Primitiv

Ein float ist eine 32-Bit-Gleitkommazahl mit einfacher Genauigkeit (32 Bit). Standardmäßig werden Dezimalzahlen als Doppelte interpretiert. Um einen float zu erstellen, hängen Sie einfach ein f an das Dezimal-Literal an.

```
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
```

Floats behandeln die fünf gängigen arithmetischen Operationen: Addition, Subtraktion, Multiplikation, Division und Modul.

Hinweis: Die folgenden Informationen können aufgrund von Fließkommafehlern geringfügig abweichen. Einige Ergebnisse wurden aus Gründen der Übersichtlichkeit und Lesbarkeit gerundet (dh das Druckergebnis des Additionsbeispiels war tatsächlich 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
```

Aufgrund der Art und Weise, wie Gleitkommazahlen gespeichert werden (dh in binärer Form), haben viele Zahlen keine exakte Darstellung.

```
float notExact = 3.1415926f;
System.out.println(notExact); // 3.1415925
```

Während die Verwendung von float für die meisten Anwendungen geeignet ist, sollten weder float noch double verwendet werden, um exakte Darstellungen von Dezimalzahlen (wie Geldbeträge) oder Zahlen zu speichern, bei denen eine höhere Genauigkeit erforderlich ist. Stattdessen sollte die BigDecimal Klasse verwendet werden.

Der Standardwert eines float ist *0.0f.*

```
float defaultFloat;    // defaultFloat == 0.0f
```

Ein float ist auf einen Fehler von 1: 10 Millionen genau.

Hinweis: Float.POSITIVE_INFINITY , Float.NEGATIVE_INFINITY , Float.NaN sind float Werte. NaN steht für Ergebnisse von Operationen, die nicht bestimmt werden können, z. B. das Aufteilen von 2 unendlichen Werten. Außerdem sind 0f und -0f unterschiedlich, aber == ergibt true:

```
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
```

Das Doppelprimitiv

Ein double ist eine 64-Bit-Gleitkommazahl mit 64-Bit-Genauigkeit.

```
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
```

Aufgrund der Art und Weise, wie Gleitkommazahlen gespeichert werden, haben viele Zahlen keine exakte Darstellung.

```
double notExact = 1.32 - 0.42; // result should be 0.9
System.out.println(notExact); // 0.9000000000000001
```

Während die Verwendung von double für die meisten Anwendungen in Ordnung ist, sollten weder float noch double für genaue Zahlen wie z. B. Währung verwendet werden. Stattdessen sollte die BigDecimal Klasse verwendet werden

Der Standardwert eines double ist *0.0d*

```
public double defaultDouble;    // defaultDouble == 0.0
```

Hinweis: Double.POSITIVE_INFINITY , Double.NEGATIVE_INFINITY , Double.NaN sind double Werte. NaN steht für Ergebnisse von Operationen, die nicht bestimmt werden können, z. B. das Aufteilen von 2 unendlichen Werten. Außerdem sind 0d und -0d verschieden, aber == ergibt true:

```
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
```

Das char primitive

A char speichern kann ein einziges 16-Bit - Unicode - Zeichen. Ein Zeichenliteral wird in einfache Anführungszeichen gesetzt

```
char myChar = 'u';
char myChar2 = '5';
char myChar3 = 65; // myChar3 == 'A'
```

Es hat einen Mindestwert von \u0000 (0 in der Dezimaldarstellung, auch *Nullzeichen genannt*) und einen Höchstwert von \uffff (65.535).

Der Standardwert eines char wird \u0000 .

```
char defaultChar;    // defaultChar == \u0000
```

Um einen Char of ' Wert zu definieren ' muss eine Escape-Sequenz (Zeichen mit vorangestelltem Backslash) verwendet werden:

```
char singleQuote = '\'';
```

Es gibt auch andere Escape-Sequenzen:

```
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
```

Sie können ein deklarieren char eines Unicode - Zeichen.

```
char heart = '\u2764';
System.out.println(Character.toString(heart)); // Prints a line containing "❤".
```

Es ist auch möglich, ein char hinzuzufügen. Um beispielsweise durch jeden Kleinbuchstaben zu iterieren, können Sie Folgendes tun:

```
for (int i = 0; i <= 26; i++) {
    char letter = (char) ('a' + i);
    System.out.println(letter);
}
```

Negative Darstellung

Java und die meisten anderen Sprachen speichern negative Integralzahlen in einer Darstellung, die als *2-Komplement-* Notation bezeichnet wird.

Für eine eindeutige binäre Darstellung eines Datentyps mit n Bits werden Werte wie folgt codiert:

Die niedrigstwertigen n-1 Bits speichern eine positive ganze Zahl x in integraler Darstellung. Der signifikanteste Wert speichert ein Bit mit dem Wert s . Der durch diese Bits dargestellte Wert ist

$x - s * 2^{n-1}$

dh wenn das höchstwertige Bit 1 ist, ist ein Wert, der nur um 1 größer ist als die Anzahl, die Sie mit den anderen Bits darstellen könnten ($2^{n-2} + 2^{n-3} + ... + 2^{1} + 2^{0} = 2^{n-1} - 1$) wird abgezogen, wodurch eine eindeutige binäre Darstellung für jeden Wert von -2^{n-1} ($s = 1; x = 0$) bis $2^{n-1} - 1$ ($s = 0; x = 2^{n-1} - 1$).

Dies hat auch den schönen Nebeneffekt, dass Sie die binären Darstellungen hinzufügen können, als ob sie positive binäre Zahlen wären:

$$v1 = x1 - s1 * 2^{n-1}$$
$$v2 = x2 - s2 * 2^{n-1}$$

s1	s2	x1 + x2 Überlauf	Zusatzergebnis
0	0	Nein	x1 + x2 = v1 + v2
0	0	Ja	zu groß, um mit Datentyp dargestellt zu werden (Überlauf)
0	1	Nein	$x1 + x2 - 2^{n-1} = x1 + x2 - s2 * 2^{n-1} = v1 + v2$
0	1	Ja	$(x1 + x2) \bmod 2^{n-1} = x1 + x2 - 2^{n-1} = v1 + v2$
1	0	*	siehe oben (Swap-Summanden)
1	1	Nein	zu klein, um mit Datentyp dargestellt zu werden ($x1 + x2 - 2^{n} < -2^{n-1}$; Unterlauf)
1	1	Ja	$(x1 + x2) \bmod 2^{n-1} - 2^{n-1} = (x1 + x2 - 2^{n-1}) - 2^{n-1} = (x1 - s1 * 2^{n-1}) + (x2 - s2 * 2^{n-1}) = v1 + v2$

Beachten Sie, dass diese Tatsache das Finden der binären Darstellung des additiven Inversen (dh des negativen Werts) vereinfacht:

Beachten Sie, dass das Hinzufügen des bitweisen Komplements zu der Zahl dazu führt, dass alle Bits 1 sind. Fügen Sie 1 hinzu, um den Wert zum Überlauf zu bringen, und Sie erhalten das neutrale Element 0 (alle Bits 0).

Der negative Wert einer Zahl i kann also berechnet werden (wobei hier eine mögliche Aufstiegsmöglichkeit nach int ignoriert wird).

```
(~i) + 1
```

Beispiel: negativen Wert von 0 (byte) nehmen:

Das Ergebnis der Negierung von 0 ist 11111111 . Addiert man 1, erhält man einen Wert von 100000000 (9 Bit). Da ein byte nur 8 Bits byte kann, wird der Wert ganz links abgeschnitten und das Ergebnis lautet 00000000

Original	Verarbeiten	Ergebnis
0 (00000000)	Negieren	-0 (11111111)
11111111	Addiere 1 zu binär	100000000
100000000	Auf 8 Bit abschneiden	00000000 (-0 entspricht 0)

Speicherverbrauch von Primitiven vs. Boxed Primitiven

Primitive	Boxed Type	Speichergröße von primitiv / boxed
boolean	Boolean	1 Byte / 16 Byte
Byte	Byte	1 Byte / 16 Byte
kurz	Kurz	2 Bytes / 16 Bytes
verkohlen	Verkohlen	2 Bytes / 16 Bytes
int	Ganze Zahl	4 Bytes / 16 Bytes
lange	Lange	8 Bytes / 16 Bytes
schweben	Schweben	4 Bytes / 16 Bytes
doppelt	Doppelt	8 Bytes / 16 Bytes

Boxed-Objekte benötigen immer 8 Byte für die Typ- und Speicherverwaltung. Da die Größe der Objekte immer ein Vielfaches von 8 ist, *benötigen* Boxed-Typen *insgesamt 16 Byte* . Darüber *hinaus* beinhaltet jede Verwendung eines geschachtelten Objekt Speicher eine Referenz , die für weitere 4 oder 8 Bytes ausmacht, in Abhängigkeit von dem JVM und JVM - Optionen.

Bei datenintensiven Vorgängen kann der Speicherverbrauch die Leistung erheblich beeinflussen. Der Speicherbedarf steigt bei Verwendung von Arrays noch weiter: Für ein float[5] -Array werden nur 32 Byte benötigt. Ein Float[5] der 5 verschiedene Nicht-Null-Werte speichert, erfordert insgesamt 112 Byte (bei 64 Bit ohne komprimierte Zeiger erhöht sich dieser Wert auf 152 Byte).

Boxed Value Caches

Der Platzaufwand der Boxtypen kann durch die Boxed Value Caches bis zu einem gewissen Grad gemindert werden. Einige der geschachtelten Typen implementieren einen Cache von Instanzen. In der Integer Klasse werden beispielsweise Instanzen zwischengespeichert, um Zahlen im Bereich von -128 bis +127 . Dies reduziert jedoch nicht die zusätzlichen Kosten, die durch die zusätzliche Speicherumleitung entstehen.

Wenn Sie eine Instanz eines geschachtelten Typs erstellen, indem Sie entweder autoboxing oder die statische valueOf(primitive) -Methode aufrufen, versucht das Laufzeitsystem, einen zwischengespeicherten Wert zu verwenden. Wenn Ihre Anwendung viele Werte in dem zwischengespeicherten Bereich verwendet, kann dies den Speichernachteil bei der Verwendung von Boxed-Typen erheblich reduzieren. Wenn Sie Boxed Value-Instanzen "von Hand" erstellen, ist es valueOf besser, valueOf anstelle von new . (Die new Operation erstellt immer eine neue Instanz.) Wenn sich jedoch die Mehrheit Ihrer Werte *nicht* im zwischengespeicherten Bereich befindet, kann es schneller sein, new aufzurufen und die Cache-Suche zu speichern.

Grundelemente konvertieren

In Java können wir zwischen ganzzahligen Werten und Gleitkommawerten konvertieren. Auch kann , da jedes Zeichen in eine Zahl in der Unicode - Codierung entspricht, char können Typen zu und von den Ganzzahl- und Gleitkomma-Typen umgewandelt werden. boolean ist der einzige primitive Datentyp, der nicht in einen oder aus einem anderen primitiven Datentyp konvertiert werden kann.

Es gibt zwei Arten von Konvertierungen: *Konvertierung erweitern* und *Konvertierung einschränken* .

Eine *Erweiterungskonvertierung* liegt vor, wenn ein Wert eines Datentyps in einen Wert eines anderen Datentyps konvertiert wird, der mehr Bits als der frühere belegt. In diesem Fall tritt kein Datenverlust auf.

Dementsprechend ist eine *Einengungskonvertierung,* wenn ein Wert eines Datentyps in einen Wert

eines anderen Datentyps konvertiert wird, der weniger Bits als der vorherige besetzt. In diesem Fall kann es zu Datenverlust kommen.

Java führt *Konvertierungen* automatisch aus. Wenn Sie jedoch eine *engere Konvertierung* durchführen möchten (wenn Sie sicher sind, dass kein Datenverlust auftritt), können Sie Java zwingen, die Konvertierung mit einem Sprachkonstrukt durchzuführen, das als cast .

Erweiterung der Konvertierung:

```
int a = 1;
double d = a;    // valid conversion to double, no cast needed (widening)
```

Enge Umwandlung:

```
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
```

Primitive Typen Cheatsheet

Tabelle mit Größe und Wertebereich aller Grundtypen:

Datentyp	numerische Darstellung	Wertebereich	Standardwert
boolean	n / a	falsch und wahr	falsch
Byte	8 Bit signiert	-2^7 bis $2^7 - 1$	0
		-128 bis +127	
kurz	16 Bit signiert	-2^{15} bis $2^{15} - 1$	0
		-32.768 bis +32.767	
int	32-Bit signiert	-2^{31} bis $2^{31} - 1$	0
		-2.147.483.648 bis +2.147.483.647	
lange	64-Bit signiert	-2^{63} bis $2^{63} - 1$	0L
		-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	
schweben	32-Bit-Gleitkommazahl	1.401298464e-45 bis 3.402823466e + 38 (positiv oder negativ)	0,0F
doppelt	64-Bit-Gleitkommazahl	4.94065645841246544e-324d bis 1.79769313486231570e + 308d (positiv oder negativ)	0,0D

Datentyp	numerische Darstellung	Wertebereich	Standardwert
verkohlen	16-Bit unsigniert	0 bis $2^{16} - 1$	0
		0 bis 65.535	

Anmerkungen:

1. Die Java-Sprachspezifikation schreibt vor, dass signierte Integraltypen (byte by long) eine binäre Zweierkomplementdarstellung verwenden, und die Fließkommatypen verwenden standardmäßige binäre Fließkommadaten nach IEE 754.
2. Java 8 und höher stellen Methoden bereit, um vorzeichenlose arithmetische Operationen für int und long auszuführen. Während diese Methoden es einem Programm ermöglichen , Werte der jeweiligen Typen als unsigniert zu *behandeln* , bleiben die Typen mit Vorzeichen signierte Typen.
3. Der kleinste Fließpunkt, der oben angezeigt wird, ist *subnormal* . dh sie haben weniger Genauigkeit als ein *normaler* Wert. Die kleinsten Normalzahlen sind 1,175494351e-38 und 2,2250738585072014e-308
4. Ein char stellt üblicherweise eine Unicode / UTF-16 - *Code* - *Einheit*.
5. Obwohl ein boolean nur ein Informationsbit enthält, variiert seine Größe im Speicher abhängig von der Implementierung der Java Virtual Machine (siehe Boolean-Typ).

Primitive Datentypen online lesen: https://riptutorial.com/de/java/topic/148/primitive-datentypen

Kapitel 127: Protokollierung (java.util.logging)

Examples

Verwenden des Standard-Loggers

Dieses Beispiel zeigt die Verwendung der Standardprotokollierungs-API.

```
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()});
    }

}
```

Protokollierungsstufen

Java Logging Api verfügt über 7 Stufen . Die Ebenen in absteigender Reihenfolge sind:

- SEVERE (höchster Wert)
- WARNING
  - INFO
  - CONFIG
  - FINE
  - FINER
  - FINEST (niedrigster Wert)

Die Standardstufe ist INFO (dies hängt jedoch vom System ab und verwendet eine virtuelle Maschine).

Hinweis : Es gibt auch die Pegel OFF (kann zum Deaktivieren der Protokollierung verwendet werden) und ALL (die Seite OFF).

Codebeispiel dafür:

```
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");
    }
}
```

Wenn diese Klasse standardmäßig ausgeführt wird, werden nur Nachrichten mit einer höheren Stufe als 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
```

Komplexe Meldungen protokollieren (effizient)

Schauen wir uns ein Protokollbeispiel an, das Sie in vielen Programmen sehen können:

```
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
}
```

Das obige Beispiel sieht perfekt aus, aber viele Programmierer vergessen, dass Java VM eine Stack-Maschine ist. Das bedeutet, dass alle Parameter der Methode **vor der** Ausführung der Methode berechnet werden.

Diese Tatsache ist für die Protokollierung in Java von entscheidender Bedeutung, insbesondere

für die Protokollierung von FINER in niedrigen Stufen wie FINE , FINER , FINEST die standardmäßig deaktiviert sind. Schauen wir uns den Java-Bytecode für die takeOrder() -Methode an.

Das Ergebnis für javap -c LoggingComplex.class lautet javap -c LoggingComplex.class so:

```
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
```

Zeile 39 führt die eigentliche Protokollierung aus. Alle vorherigen Arbeiten (Laden von Variablen, Erstellen neuer Objekte, Verketten von Strings in der format) können nichts ausrichten, wenn die Protokollierungsstufe höher als FINE (und standardmäßig) eingestellt ist. Eine solche Protokollierung kann sehr ineffizient sein und unnötigen Speicher und Prozessorressourcen beanspruchen.

Deshalb sollten Sie fragen, ob die gewünschte Ebene aktiviert ist.

Der richtige Weg sollte sein:

```
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
}
```

Seit Java 8:

Die Logger-Klasse verfügt über zusätzliche Methoden, die einen Supplier<String> als Parameter verwenden, der einfach von einem Lambda bereitgestellt werden kann:

```
public void takeOrder() {
    // making some stuff
    logger.fine(() -> String.format("User %s ordered %d things (%d in total)",
```

```
            username, orders, total));
    // some other stuff
}
```

Die Methode " get() Suppliers get() in diesem Fall das Lambda) wird nur aufgerufen, wenn die entsprechende Ebene aktiviert ist und die if Konstruktion nicht mehr benötigt wird.

Protokollierung (java.util.logging) online lesen:
https://riptutorial.com/de/java/topic/2010/protokollierung--java-util-logging-

Kapitel 128: Referenzdatentypen

Examples

Einen Referenztyp instanziieren

```
Object obj = new Object(); // Note the 'new' keyword
```

Woher:

- Object ist ein Referenztyp.
- obj ist die Variable, in der die neue Referenz gespeichert werden soll.
- Object() ist der Aufruf eines Konstruktors von Object .

Was geschieht:

- Speicherplatz wird für das Objekt reserviert.
- Der Konstruktor Object() wird aufgerufen, um diesen Speicherplatz zu initialisieren.
- Die Speicheradresse wird in obj gespeichert, so dass sie *auf* das neu erstellte Objekt *verweist* .

Dies unterscheidet sich von Primitiven:

```
int i = 10;
```

Wo der Istwert 10 in i gespeichert ist.

Dereferenzierung

Dereferenzierung geschieht mit dem . Operator:

```
Object obj = new Object();
String text = obj.toString(); // 'obj' is dereferenced.
```

Die Dereferenzierung *folgt* der in einer Referenz gespeicherten Speicheradresse an den Ort im Speicher, an dem sich das tatsächliche Objekt befindet. Wenn ein Objekt gefunden wurde, wird die angeforderte Methode aufgerufen (in diesem Fall toString).

Wenn eine Referenz den Wert null , führt die Dereferenzierung zu einer NullPointerException :

```
Object obj = null;
obj.toString(); // Throws a NullpointerException when this statement is executed.
```

null bedeutet das Fehlen eines Wertes, dh die *Nachverfolgung* der Speicheradresse führt zu nichts. Es gibt also kein Objekt, auf dem die angeforderte Methode aufgerufen werden kann.

Referenzdatentypen online lesen: https://riptutorial.com/de/java/topic/1046/referenzdatentypen

Kapitel 129: Referenztypen

Examples

Unterschiedliche Referenztypen

java.lang.ref Paket java.lang.ref stellt Referenzobjektklassen bereit, die eine begrenzte Interaktion mit dem Garbage Collector unterstützen.

Java hat vier Hauptreferenztypen. Sie sind:

- Starke Referenz
- Schwache Referenz
- Weiche Referenz
- Phantomreferenz

1. Starke Referenz

Dies ist die übliche Form zum Erstellen von Objekten.

```
MyObject myObject = new MyObject();
```

Der Variablenhalter hält einen starken Verweis auf das erstellte Objekt. Solange diese Variable aktiv ist und diesen Wert enthält, wird die MyObject Instanz nicht vom Garbage Collector erfasst.

2. Schwache Referenz

Wenn Sie ein Objekt nicht länger behalten möchten und den für ein Objekt zugewiesenen Speicher so schnell wie möglich löschen / freigeben müssen, gehen Sie auf diese Weise vor.

```
WeakReference myObjectRef = new WeakReference(MyObject);
```

Eine schwache Referenz ist einfach eine Referenz, die nicht stark genug ist, um ein Objekt im Speicher zu halten. Schwache Referenzen ermöglichen es Ihnen, die Fähigkeit des Müllsammlers zu nutzen, um die Erreichbarkeit für Sie zu bestimmen, so dass Sie dies nicht selbst tun müssen.

Wenn Sie das von Ihnen erstellte Objekt benötigen, verwenden Sie einfach die .get() Methode:

```
myObjectRef.get();
```

Der folgende Code veranschaulicht dies beispielhaft:

```
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. Weiche Referenz

Weiche Referenzen sind etwas stärker als schwache Referenzen. Sie können ein weich referenziertes Objekt wie folgt erstellen:

```
SoftReference myObjectRef = new SoftReference(MyObject);
```

Sie können den Speicher stärker als den schwachen Bezug halten. Wenn Sie über genügend Speicher / Ressourcen verfügen, bereinigt der Garbage Collector die weichen Referenzen nicht so begeistert wie schwache Referenzen.

Weiche Verweise sind beim Zwischenspeichern von Nutzen. Sie können weich referenzierte Objekte als Cache erstellen, in denen sie aufbewahrt werden, bis der Speicher voll ist. Wenn Ihr Speicher nicht genügend Ressourcen bereitstellen kann, entfernt der Garbage Collector weiche Referenzen.

```
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. Phantomreferenz

Dies ist der schwächste Referenzierungstyp. Wenn Sie mit Phantom Reference einen Objektverweis erstellt haben, gibt die get() Methode immer null zurück!

Die Verwendung dieser Referenzierung ist folgende: "Phantomreferenzobjekte, die in eine Warteschlange eingereiht werden, nachdem der Collector feststellt, dass ihre Referenzen andernfalls zurückgefordert werden können. Phantomreferenzen werden am häufigsten verwendet, um Bereinigungsaktionen vor der Mortem-Aktion flexibler zu gestalten, als dies mit der Java-Finalisierungsmechanismus. " - Aus der Phantomreferenz Javadoc von Oracle.

Sie können ein Objekt der Phantom-Referenz wie folgt erstellen:

```
PhantomReference myObjectRef = new PhantomReference(MyObject);
```

Referenztypen online lesen: https://riptutorial.com/de/java/topic/4017/referenztypen

Kapitel 130: Reflexions-API

Einführung

Reflection wird häufig von Programmen verwendet, bei denen das Laufzeitverhalten von Anwendungen, die in der JVM ausgeführt werden, untersucht oder geändert werden muss. Die Java Reflection-API wird zu diesem Zweck verwendet, um Klassen, Schnittstellen, Felder und Methoden zur Laufzeit zu untersuchen, ohne deren Namen zur Kompilierzeit zu kennen. Außerdem können neue Objekte instanziiert und Methoden mithilfe von Reflektion aufgerufen werden.

Bemerkungen

Performance

Beachten Sie, dass Reflektionen die Leistung beeinträchtigen können. Verwenden Sie sie nur, wenn Ihre Aufgabe nicht ohne Reflektion abgeschlossen werden kann.

Aus dem Java-Tutorial The Reflection API :

> Da bei der Reflektion dynamisch aufgelöste Typen beteiligt sind, können bestimmte Java Virtual Machine-Optimierungen nicht durchgeführt werden. Folglich haben reflektierende Operationen eine langsamere Leistung als ihre nicht reflektierenden Pendants und sollten in Codeabschnitten vermieden werden, die in leistungsempfindlichen Anwendungen häufig genannt werden.

Examples

Einführung

Grundlagen

Die Reflection-API ermöglicht die Überprüfung der Klassenstruktur des Codes zur Laufzeit und den dynamischen Aufruf von Code. Dies ist sehr mächtig, aber es ist auch gefährlich, da der Compiler nicht statisch feststellen kann, ob dynamische Aufrufe gültig sind.

Ein einfaches Beispiel wäre, die öffentlichen Konstruktoren und Methoden einer bestimmten Klasse abzurufen:

```
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
```

Mit diesen Informationen ist es möglich, das Objekt zu instanziieren und verschiedene Methoden dynamisch aufzurufen.

Reflexion und generische Typen

Generische Typinformationen sind verfügbar für:

- Methodenparameter mit getGenericParameterTypes() .
- Rückgabetypen für Methoden mithilfe von getGenericReturnType() .
- **öffentliche** Felder mit getGenericType .

Das folgende Beispiel zeigt, wie die generischen Typinformationen in allen drei Fällen extrahiert werden:

```
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;
    }

}
```

Daraus ergibt sich folgende Ausgabe:

```
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
```

Methode aufrufen

Mit Reflection kann eine Methode eines Objekts zur Laufzeit aufgerufen werden.

Das Beispiel zeigt, wie die Methoden eines String Objekts String werden.

```
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!"
```

Felder abrufen und einstellen

Mit der Reflection-API können Sie zur Laufzeit den Wert eines Felds ändern oder abrufen. Sie können es beispielsweise in einer API verwenden, um verschiedene Felder basierend auf einem Faktor wie dem Betriebssystem abzurufen. Sie können auch Modifikatoren wie final entfernen, um endgültige Felder zu ändern.

Dazu müssen Sie die Methode Class # getField () wie folgt verwenden :

```
// 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);
```

Felder zu bekommen ist viel einfacher. Wir können Field # get () und seine Varianten verwenden, um seinen Wert zu erhalten:

```
// 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);
```

Beachten Sie dies:

Wenn Sie die Klasse # getDeclaredField verwenden , verwenden Sie sie, um ein Feld in der Klasse selbst abzurufen :

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

class Hacked {
    public String someState;
}
```

Hier wird HackMe#iAmDeclared als Feld deklariert. HackMe#someState ist jedoch kein deklariertes Feld, da es von seiner Superklasse Hacked geerbt wird.

Konstruktor aufrufen

Das Konstruktorobjekt abrufen

Sie können die Constructor Klasse wie Constructor vom Class Objekt abrufen:

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

Die constructors hat eine Constructor für jeden in der Klasse deklarierten öffentlichen Konstruktor.

Wenn Sie die genauen Parametertypen des Konstruktors kennen, auf den Sie zugreifen möchten, können Sie den spezifischen Konstruktor filtern. Das nächste Beispiel gibt den öffentlichen Konstruktor der angegebenen Klasse zurück, der einen Integer Parameter als Parameter annimmt:

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

Wenn kein Konstruktor mit den angegebenen Konstruktorargumenten NoSuchMethodException wird eine NoSuchMethodException ausgelöst.

Neue Instanz mit Konstruktorobjekt

```
Class myClass = MyObj.class // get a class object
```

```
Constructor constructor = myClass.getConstructor(Integer.class);
MyObj myObj = (MyObj) constructor.newInstance(Integer.valueOf(123));
```

Die Konstanten einer Aufzählung erhalten

Diese Aufzählung als Beispiel geben:

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

In Java ist eine Enumenklasse wie jede andere Klasse, hat jedoch bestimmte Konstanten für die Enummenwerte. Darüber hinaus verfügt es über ein Feld, das ein Array mit allen Werten und zwei statischen Methoden mit dem Namen valueOf(String) values() und valueOf(String) .
Wir können dies sehen, wenn wir mit Reflection alle Felder in dieser Klasse drucken

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

Die Ausgabe wird sein:

```
NORDEN
OSTEN
SÜDEN
WEST
Grad
ENUM $ WERTE
```

So könnten wir Enum-Klassen wie jede andere Klasse mit Reflection untersuchen. Die Reflection-API bietet jedoch drei aufzählungsspezifische Methoden.

Aufzählungsprüfung

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

Gibt true für Klassen zurück, die einen Aufzählungstyp darstellen.

Werte abrufen

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

Gibt ein Array aller Aufzählungswerte wie Compass.values () zurück, ohne dass eine Instanz erforderlich ist.

Enum ständige Prüfung

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

```
}
```

Listet alle Klassenfelder auf, die Listenwerte sind.

Bekommen Sie der Klasse den (vollständig qualifizierten) Namen

Bei einem String den Namen einer Klasse enthält, ist es Class - Objekt zugegriffen werden kann , mit Class.forName :

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

Java SE 1.2

Es kann angegeben werden, ob die Klasse initialisiert werden soll (zweiter Parameter von forName) und welcher ClassLoader werden soll (dritter Parameter):

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

Rufen Sie überladene Konstruktoren mit Reflektion auf

Beispiel: Rufen Sie verschiedene Konstruktoren auf, indem Sie relevante Parameter übergeben

```
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"});

    }
}
```

Ausgabe:

```
Default constructor
```

```
Constructor :String => StackOverFlow
```

Erläuterung:

1. Instanz der Klasse mit Class.forName : Der Standardkonstruktor wird Class.forName
2. Rufen Sie getDeclaredConstructor der Klasse auf, indem Sie den Typ der Parameter als Class array
3. Erstellen newInstance nach dem newInstance des Konstruktors newInstance indem Sie den Parameterwert als Object array

Missbrauch der Reflection-API zum Ändern von privaten und endgültigen Variablen

Reflexion ist nützlich, wenn sie richtig für den richtigen Zweck verwendet wird. Mit Reflektion können Sie auf private Variablen zugreifen und endgültige Variablen erneut initialisieren.

Unten ist der Code-Ausschnitt, der **nicht** empfohlen wird.

```
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++;
    }
}
```

Ausgabe:

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

Erläuterung:

Im normalen Szenario kann nicht auf private Variablen außerhalb der deklarierten Klasse (ohne Getter- und Setter-Methoden) zugegriffen werden. final Variablen können nach der Initialisierung nicht neu zugeordnet werden.

Reflection durchbricht beide Barrieren, um private und letzte Variablen zu ändern, wie oben erläutert.

field.setAccessible(true) ist der Schlüssel zum Erreichen der gewünschten Funktionalität.

Rufen Sie den Konstruktor der verschachtelten Klasse auf

Wenn Sie eine Instanz einer inneren verschachtelten Klasse erstellen möchten, müssen Sie ein Klassenobjekt der umgebenden Klasse als zusätzlichen Parameter mit Class # getDeclaredConstructor angeben .

```
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"});
    }
}
```

Wenn die verschachtelte Klasse statisch ist, benötigen Sie diese einschließende Instanz nicht.

Dynamische Proxies

Dynamische Proxies haben nicht wirklich viel mit Reflection zu tun, aber sie sind Teil der API. Es ist im Grunde eine Möglichkeit, eine dynamische Implementierung einer Schnittstelle zu erstellen. Dies kann beim Erstellen von Mockup-Services hilfreich sein.
Ein dynamischer Proxy ist eine Instanz einer Schnittstelle, die mit einem sogenannten Aufrufhandler erstellt wird, der alle Methodenaufrufe abfängt und deren manuellen Aufruf ermöglicht.

```
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();
    }
}
```

Das Ergebnis dieses Codes ist folgendes:

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

Böse Java-Hacks mit Reflection

Mit der Reflection-API können Sie auch in der JDK-Standardbibliothek Werte von privaten und endgültigen Feldern ändern. Dies kann verwendet werden, um das Verhalten einiger bekannter Klassen zu manipulieren, wie wir sehen werden.

Was ist nicht möglich

Beginnen wir mit der einzigen Einschränkung. Das einzige Feld, das wir mit Reflection nicht ändern können. Das ist der Java SecurityManager . Es ist in java.lang.System als deklariert

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

Es wird jedoch nicht in der Systemklasse aufgeführt, wenn wir diesen Code ausführen

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

Das liegt an der fieldFilterMap in sun.reflect.Reflection , die die Karte selbst und das Sicherheitsfeld in der System.class und sie vor dem Zugriff mit Reflection schützt. Daher konnten wir den SecurityManager nicht deaktivieren.

Verrückte Saiten

Jeder Java-String wird von der JVM als Instanz der String Klasse dargestellt. In einigen Situationen spart die JVM jedoch Heapspeicherplatz, indem dieselbe Instanz für Strings verwendet wird. Dies geschieht für String-Literale und auch für Strings, die durch Aufruf von String.intern() "interniert" wurden. Wenn Sie also mehrmals "hello" in Ihrem Code haben, handelt es sich immer um dieselbe Objektinstanz.

Strings sollen unveränderlich sein, aber es ist möglich, "böse" Reflexion zu verwenden, um sie zu ändern. Das folgende Beispiel zeigt , wie wir die Zeichen in einem String ändern kann durch seine Ersetzung value Feld.

```
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");
    }
}
```

Also wird dieser Code "Sie stinken!"

1 = 42

Die gleiche Idee könnte mit der Integer-Klasse verwendet werden

```
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));
    }
}
```

Alles ist wahr

Und gemäß diesem Stackoverflow-Beitrag können wir durch Nachdenken etwas wirklich Böses tun.

```
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);
    }
}
```

Beachten Sie, dass das, was wir hier tun, dazu führt, dass sich die JVM auf unerklärliche Weise verhält. Das ist sehr gefährlich.

Reflexions-API online lesen: https://riptutorial.com/de/java/topic/629/reflexions-api

Kapitel 131: Reguläre Ausdrücke

Einführung

Ein regulärer Ausdruck ist eine spezielle Zeichenfolge, die beim Zuordnen oder Finden anderer Zeichenketten oder Zeichenfolgen hilft, wobei eine spezielle, in einem Muster enthaltene Syntax verwendet wird. Java unterstützt die Verwendung regulärer Ausdrücke über das Paket java.util.regex . In diesem Thema werden Entwickler vorgestellt und anhand von Beispielen erläutert, wie reguläre Ausdrücke in Java verwendet werden müssen.

Syntax

- Pattern patternName = Pattern.compile (Regex);
- Matcher MatcherName = MusterName.Matcher (TextToSearch);
- matcherName.matches () // Gibt "true" zurück, wenn textToSearch genau mit dem regulären Ausdruck übereinstimmt
- matcherName.find () // Durchsucht textToSearch nach der ersten Instanz eines Teilstrings, der der Regex entspricht. Nachfolgende Aufrufe durchsuchen den Rest der Zeichenfolge.
- matcherName.group (groupNum) // Gibt den Teilstring innerhalb einer Erfassungsgruppe zurück
- matcherName.group (groupName) // Gibt den Teilstring innerhalb einer benannten Erfassungsgruppe zurück (Java 7+)

Bemerkungen

Importe

Sie müssen die folgenden Importe hinzufügen, bevor Sie Regex verwenden können:

```
import java.util.regex.Matcher
import java.util.regex.Pattern
```

Fallstricke

In Java wird ein Backslash mit einem doppelten Backslash geschützt, daher sollte ein Backslash in der Regex-Zeichenfolge als doppelter Backslash eingegeben werden. Wenn Sie einem doppelten Backslash entgehen müssen (um einen einzelnen Backslash mit dem Regex abgleichen zu können, müssen Sie ihn als vierfachen Backslash eingeben.

Wichtige Symbole erklärt

Charakter	Beschreibung
*	Stimmt mit dem vorhergehenden Zeichen oder Unterausdruck mindestens 0 mal überein
+	Ordnen Sie den vorhergehenden Buchstaben oder Unterausdruck mindestens ein Mal zu
?	Stimmt mit dem vorhergehenden Zeichen oder Unterausdruck 0 oder 1 überein

Lesen Sie weiter

Das Thema "Regex" enthält weitere Informationen zu regulären Ausdrücken.

Examples

Capture-Gruppen verwenden

Wenn Sie einen Teil der Zeichenfolge aus der Eingabezeichenfolge extrahieren müssen, können Sie **Erfassungsgruppen** für Regex verwenden.

In diesem Beispiel beginnen wir mit einer einfachen Regex: Telefonnummer:

```
\d{3}-\d{3}-\d{4}
```

Wenn dem Regex Klammern hinzugefügt werden, wird jeder Satz von Klammern als *Erfassungsgruppe betrachtet* . In diesem Fall verwenden wir so genannte nummerierte Erfassungsgruppen:

```
(\d{3})-(\d{3})-(\d{4})
^-----^ ^-----^ ^-----^
Group 1 Group 2 Group 3
```

Bevor wir es in Java verwenden können, dürfen wir nicht vergessen, die Regeln von Strings zu befolgen und die Backslashes zu umgehen, was folgendes Muster ergibt:

```
"(\\d{3})-(\\d{3})-(\\d{4})"
```

Wir müssen zuerst das Regex-Muster kompilieren, um ein Pattern zu erstellen, und dann brauchen wir einen Matcher , um unsere Eingabezeichenfolge mit dem Muster Matcher :

```
Pattern phonePattern = Pattern.compile("(\\d{3})-(\\d{3})-(\\d{4})");
Matcher phoneMatcher = phonePattern.matcher("abcd800-555-1234wxyz");
```

Als Nächstes muss der Matcher die erste Untersequenz finden, die der Regex entspricht:

```
phoneMatcher.find();
```

Mit der Gruppenmethode können wir nun die Daten aus der Zeichenfolge extrahieren:

```
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"
```

Hinweis: Matcher.group() kann anstelle von Matcher.group(0) .

Java SE 7

In Java 7 wurden benannte Capture-Gruppen eingeführt. Benannte Erfassungsgruppen funktionieren genauso wie nummerierte Erfassungsgruppen (jedoch mit einem Namen anstelle einer Zahl), obwohl es geringfügige Syntaxänderungen gibt. Die Verwendung benannter Capture-Gruppen verbessert die Lesbarkeit.

Wir können den obigen Code ändern, um benannte Gruppen zu verwenden:

```
(?<AreaCode>\d{3})-(\d{3})-(\d{4})
^-----------------^ ^-----^ ^-----^
AreaCode            Group 2 Group 3
```

Um den Inhalt von "AreaCode" zu erhalten, können wir stattdessen Folgendes verwenden:

```
String aCode = phoneMatcher.group("AreaCode"); //"800"
```

Regex mit benutzerdefiniertem Verhalten verwenden, indem das Muster mit Flags kompiliert wird

Ein Pattern kann mit Flags kompiliert werden. Wenn der Regex als literaler String , verwenden Sie Inline-Modifizierer:

```
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".
```

Fluchtfiguren

Allgemein

Um reguläre Ausdrücke (?+| Usw.) in ihrer wörtlichen Bedeutung zu verwenden, müssen sie mit Escapezeichen versehen werden. Im normalen regulären Ausdruck geschieht dies durch einen Backslash \ . Da dies jedoch in Java-Zeichenfolgen eine besondere Bedeutung hat, müssen Sie einen doppelten Backslash \\ .

Diese beiden Beispiele funktionieren nicht:

```
"???".replaceAll ("?", "!"); //java.util.regex.PatternSyntaxException
"???".replaceAll ("\?", "!"); //Invalid escape sequence
```

Dieses Beispiel funktioniert

```
"???".replaceAll ("\\?", "!"); //"!!!"
```

Aufteilen einer durch Pipe getrennten Zeichenfolge

Dies liefert nicht das erwartete Ergebnis:

```
"a|b".split ("|"); // [a, |, b]
```

Dies gibt das erwartete Ergebnis zurück:

```
"a|b".split ("\\|"); // [a, b]
```

**Backslash abfangen **

Dies gibt einen Fehler:

```
"\\".matches("\\"); // PatternSyntaxException
"\\".matches("\\\"); // Syntax Error
```

Das funktioniert:

```
"\\".matches("\\\\"); // true
```

Übereinstimmung mit einem Regex-Literal.

Wenn Sie Zeichen finden müssen, die Teil der Syntax für reguläre Ausdrücke sind, können Sie das gesamte Muster oder einen Teil des Musters als Regex-Literal markieren.

\Q markiert den Anfang des Regex-Literal. \E markiert das Ende des Regex-Literal.

```
// 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
```

Eine einfachere Möglichkeit, dies zu tun, ohne sich an die \Q Pattern.quote() und \E Escape-Sequenzen zu erinnern, ist die Verwendung von Pattern.quote()

```
"[123".matches(Pattern.quote("[") + "123"); // returns true
```

Stimmt nicht mit einer bestimmten Zeichenfolge überein

Um etwas zu finden, das *keine* bestimmte Zeichenfolge enthält, kann ein negatives Lookahead verwendet werden:

Regex-Syntax: (?!string-to-not-match)

Beispiel:

```
//not matching "popcorn"
String regexString = "^(?!popcorn).*$";
System.out.println("[popcorn] " + ("popcorn".matches(regexString) ? "matched!" : "nope!"));
System.out.println("[unicorn] " + ("unicorn".matches(regexString) ? "matched!" : "nope!"));
```

Ausgabe:

```
[popcorn] nope!
[unicorn] matched!
```

Einen Backslash abgleichen

Wenn Sie einen umgekehrten Schrägstrich in Ihrem regulären Ausdruck abgleichen möchten, müssen Sie ihn deaktivieren.

Backslash ist ein Escape-Zeichen in regulären Ausdrücken. Sie können '\\' verwenden, um auf einen einzelnen Backslash in einem regulären Ausdruck zu verweisen.

Backslash ist jedoch *auch* ein Escape-Zeichen in Java-Literal-Strings. Um aus einem String-Literal einen regulären Ausdruck zu erstellen, müssen Sie jeden *seiner* Backslashes mit Escapezeichen versehen. In einem String-Literal kann mit '\\\\' ein regulärer Ausdruck mit '\\' erstellt werden, der wiederum mit '\' übereinstimmen kann.

Stellen Sie sich beispielsweise vor, Zeichenfolgen wie "C: \ dir \ myfile.txt" zu finden. Ein regulärer Ausdruck ([A-Za-z]):\\(.*) Stimmt überein und stellt den Laufwerksbuchstaben als Erfassungsgruppe bereit. Beachten Sie den doppelten Backslash.

Um dieses Muster in einem Java-String-Literal auszudrücken, muss jeder der umgekehrten Schrägstriche im regulären Ausdruck mit Escapezeichen versehen werden.

```
    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:.
    }
```

Wenn Sie *zwei* umgekehrte Schrägstriche abgleichen möchten, verwenden Sie acht in einer Literal-Zeichenfolge, um vier im regulären Ausdruck darzustellen, und zwei.

```
    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'.
    }
```

Reguläre Ausdrücke online lesen: https://riptutorial.com/de/java/topic/135/regulare-ausdrucke

Kapitel 132: Rekursion

Einführung

Rekursion tritt auf, wenn eine Methode sich selbst aufruft. Eine solche Methode wird als **rekursiv bezeichnet** . Eine rekursive Methode kann prägnanter sein als ein gleichwertiger nichtrekursiver Ansatz. Bei einer tiefen Rekursion kann eine iterative Lösung jedoch manchmal weniger endlichen Stack-Platz eines Threads beanspruchen.

Dieses Thema enthält Beispiele für Rekursionen in Java.

Bemerkungen

Eine rekursive Methode entwerfen

Beachten Sie beim Entwurf einer rekursiven Methode, dass Sie Folgendes benötigen:

- **Basisfall**. Dadurch wird festgelegt, wann Ihre Rekursion angehalten wird und das Ergebnis ausgegeben wird. Der Basisfall in dem faktoriellen Beispiel ist:

```
if (n <= 1) {
    return 1;
}
```

- **Rekursiver Aufruf**. In dieser Anweisung rufen Sie die Methode mit einem geänderten Parameter erneut auf. Der rekursive Aufruf im obigen faktoriellen Beispiel lautet:

```
else {
    return n * factorial(n - 1);
}
```

Ausgabe

In diesem Beispiel berechnen Sie die n-te Faktornummer. Die ersten Fakultäten sind:

```
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

...
```

Java- und Tail-Call-Beseitigung

Aktuelle Java-Compiler (bis einschließlich Java 9) führen keine Tail-Call-Eliminierung durch. Dies kann sich auf die Leistung rekursiver Algorithmen auswirken. Wenn die Rekursion tief genug ist, kann dies zu Abstürzen von StackOverflowError führen. siehe Deep Rekursion ist in Java problematisch

Examples

Die Grundidee der Rekursion

Was ist Rekursion:

Rekursion ist im Allgemeinen, wenn eine Funktion sich direkt oder indirekt aufruft. Zum Beispiel:

```
// This method calls itself "infinitely"
public void useless() {
    useless();  // method calls itself (directly)
}
```

Bedingungen für das Anwenden einer Rekursion auf ein Problem:

Es gibt zwei Voraussetzungen für die Verwendung rekursiver Funktionen zur Lösung eines bestimmten Problems:

1. Es muss eine Basisbedingung für das Problem geben, die der Endpunkt für die Rekursion sein wird. Wenn eine rekursive Funktion die Basisbedingung erreicht, führt sie keine weiteren (tieferen) rekursiven Aufrufe aus.
2. Jede Rekursionsebene sollte ein kleineres Problem versuchen. Die rekursive Funktion unterteilt das Problem somit in immer kleinere Teile. Vorausgesetzt, dass das Problem endlich ist, wird sichergestellt, dass die Rekursion beendet wird.

In Java gibt es eine dritte Voraussetzung: Es sollte nicht notwendig sein, zu tief zu rekursieren, um das Problem zu lösen. siehe Deep Rekursion ist in Java problematisch

Beispiel

Die folgende Funktion berechnet die Fakultäten anhand der Rekursion. Beachten Sie, wie sich die Methode factorial innerhalb der Funktion aufruft. Bei jedem Aufruf von selbst wird der Parameter n um 1 reduziert. Wenn n den Wert 1 (Basisbedingung) erreicht, wird die Funktion nicht tiefer rekonstieren.

```
public int factorial(int n) {
    if (n <= 1) { // the base condition
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Dies ist keine praktische Methode zum Berechnen von Fakultäten in Java, da Integer-Überlauf oder Aufruf-Stack-Überlauf (dh `StackOverflowError` Ausnahmen) für große Werte von `n` nicht berücksichtigt werden.

Berechnung der N-ten Fibonacci-Zahl

Die folgende Methode berechnet die Nte Fibonacci-Zahl unter Verwendung der Rekursion.

```
public int fib(final int n) {
    if (n > 2) {
        return fib(n - 2) + fib(n - 1);
    }
    return 1;
}
```

Das Verfahren implementiert einen Basisfall (n <= 2) und einen rekursiven Fall (n> 2). Dies veranschaulicht die Verwendung der Rekursion zum Berechnen einer rekursiven Beziehung.

Dieses Beispiel ist zwar veranschaulichend, aber auch ineffizient: Jede einzelne Instanz der Methode ruft die Funktion zweimal auf, was zu einem exponentiellen Wachstum der Anzahl der Aufrufe der Funktion führt, wenn N zunimmt. Die obige Funktion ist O (2^N), aber eine äquivalente iterative Lösung hat die Komplexität O (N). Darüber hinaus gibt es einen Ausdruck "geschlossener Form", der in O (N) Gleitkommamultiplikationen ausgewertet werden kann.

Berechnen der Summe der Zahlen von 1 bis N

Die folgende Methode berechnet die Summe der Ganzzahlen von 0 bis N unter Verwendung der Rekursion.

```
public int sum(final int n) {
    if (n > 0) {
        return n + sum(n - 1);
    } else {
        return n;
    }
}
```

Diese Methode ist O (N) und kann mithilfe der Tail Call-Optimierung auf eine einfache Schleife reduziert werden. Tatsächlich gibt es einen Ausdruck in *geschlossener Form* , der die Summe in O(1) -Operationen berechnet.

Berechnung der N-ten Potenz einer Zahl

Die folgende Methode berechnet den Wert von num , der mit exp auf exp erhöht wurde:

```
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);
}
```

Dies veranschaulicht die oben genannten Prinzipien: Die rekursive Methode implementiert einen Basisfall (zwei Fälle, n = 0 und n = 1), der die Rekursion beendet, und einen rekursiven Fall, der die Methode erneut aufruft. Diese Methode ist O (N) und kann mithilfe der Tail Call-Optimierung auf eine einfache Schleife reduziert werden.

Umkehren einer Zeichenfolge mit Rekursion

Nachfolgend finden Sie einen rekursiven Code zum Umkehren einer Zeichenfolge

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

Durchlaufen einer Baumdatenstruktur mit Rekursion

Betrachten Sie die Knotenklasse mit 3 Elementdaten, linkem unterem Zeiger und rechtem unterem Zeiger wie unten.

```
public class Node {
    public int data;
    public Node left;
    public Node right;

    public Node(int data){
        this.data = data;
    }
}
```

Wir können den Baum durchqueren, der durch Verbinden der Objekte mehrerer Node-Klassen wie unten erstellt wurde. Der Durchlauf wird als Durchlauf des Baums in der Reihenfolge bezeichnet.

```
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
        }
    }
```

Wie oben gezeigt, können wir mithilfe der **Rekursion die** Baumdatenstruktur durchlaufen , ohne eine andere Datenstruktur zu verwenden, die mit dem **iterativen** Ansatz nicht möglich ist.

Arten der Rekursion

Rekursion kann je nach dem Ort des rekursiven Methodenaufrufs entweder als **Head-Rekursion** oder als **Tail-Rekursion** kategorisiert werden.

Bei der **Rekursion "head"** kommt der rekursive Aufruf, wenn er auftritt, vor einer anderen Verarbeitung in der Funktion (denken Sie daran, dass dies am Anfang oder Kopf der Funktion geschieht).

Bei der **Tail-Rekursion** ist es genau umgekehrt: Die Verarbeitung erfolgt vor dem rekursiven Aufruf. Die Wahl zwischen den beiden rekursiven Stilen mag beliebig erscheinen, aber die Wahl kann den Unterschied ausmachen.

Eine Funktion mit einem Pfad mit einem einzelnen rekursiven Aufruf am Anfang des Pfads verwendet die sogenannte Rekursion "head". Die faktorielle Funktion eines vorherigen Exponats verwendet die Rekursion des Kopfes. Sobald er feststellt, dass eine Rekursion erforderlich ist, ruft er sich zuerst mit dem dekrementierten Parameter auf. Eine Funktion mit einem einzelnen rekursiven Aufruf am Ende eines Pfads verwendet die Tail-Rekursion.

```
public void tail(int n)
{
   if(n == 1)
      return;
   else
      System.out.println(n);

   tail(n-1);
}
```

```
public void head(int n)
   {
     if(n == 0)
        return;
     else
        head(n-1);

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

Wenn der rekursive Aufruf am Ende einer Methode auftritt, wird dies als tail recursion . Die Schwanzrekursion similar to a loop . Die method executes all the statements before jumping into

the next recursive call .

Wenn der rekursive Aufruf am beginning of a method, it is called a head recursion auftritt beginning of a method, it is called a head recursion . Die method saves the state before jumping into the next recursive call .

Referenz: Der Unterschied zwischen Rekursion von Kopf und Schwanz

StackOverflowError & Rekursion in Schleife

Wenn ein rekursiver Aufruf "zu tief" geht, führt dies zu einem StackOverflowError . Java ordnet jedem Methodenaufruf im Stack seines Threads einen neuen Frame zu. Der Platz des Stapels jedes Threads ist jedoch begrenzt. Zu viele Frames auf dem Stack führen zum Stack Overflow (SO).

Beispiel

```
public static void recursion(int depth) {
    if (depth > 0) {
        recursion(depth-1);
    }
}
```

Das Aufrufen dieser Methode mit großen Parametern (z. B. recursion(50000) wahrscheinlich zu einem Stapelüberlauf. Der genaue Wert hängt von der Thread-Stack-Größe ab, die wiederum von der Threadkonstruktion, den Befehlszeilenparametern wie -Xss oder der -Xss ist Standardgröße für die JVM.

Problemumgehung

Eine Rekursion kann in eine Schleife konvertiert werden, indem die Daten für jeden rekursiven Aufruf in einer Datenstruktur gespeichert werden. Diese Datenstruktur kann auf dem Heapspeicher statt auf dem Threadstapel gespeichert werden.

Im Allgemeinen können die Daten, die zum Wiederherstellen des Zustands eines Methodenaufrufs erforderlich sind, in einem Stapel gespeichert werden, und eine While-Schleife kann verwendet werden, um die rekursiven Aufrufe zu "simulieren". Zu den Daten, die möglicherweise erforderlich sind, gehören:

- das Objekt, für das die Methode aufgerufen wurde (nur Instanzmethoden)
- die Methodenparameter
- lokale Variablen
- die aktuelle Position in der Ausführung oder der Methode

Beispiel

Die folgende Klasse ermöglicht das rekursive Drucken einer Baumstruktur bis zu einer angegebenen Tiefe.

```
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(")");
        }
    }

}
```

z.B

```
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();
```

Drucke

```
(((...)20(...))10((...)30))
```

Dies könnte in die folgende Schleife umgewandelt werden:

```
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();
```

Hinweis: Dies ist nur ein Beispiel für den allgemeinen Ansatz. Häufig können Sie einen viel besseren Weg finden, einen Frame darzustellen und / oder die Frame-Daten zu speichern.

Die tiefe Rekursion ist in Java problematisch

Betrachten Sie die folgende naive Methode zum Hinzufügen zweier positiver Zahlen mithilfe der Rekursion:

```
public static int add(int a, int b) {
    if (a == 0) {
        return b;
    } else {
        return add(a - 1, b + 1);  // TAIL CALL
    }
}
```

Das ist algorithmisch korrekt, hat aber ein großes Problem. Wenn Sie add mit einem großen a aufrufen add stürzt es mit einem StackOverflowError auf einer beliebigen Java-Version bis mindestens Java 9 ab.

In einer typischen funktionalen Programmiersprache (und vielen anderen Sprachen) optimiert der Compiler die Rekursion des Endes . Der Compiler würde feststellen, dass der Aufruf zum add (an der markierten Zeile) ein Tail-Aufruf ist , und die Rekursion als Schleife neu schreiben. Diese Umwandlung wird als Tail-Call-Eliminierung bezeichnet.

Java-Compiler der aktuellen Generation führen jedoch keine Tail Call-Eliminierung durch. (Dies ist kein einfaches Versehen. Hierfür gibt es wesentliche technische Gründe; siehe unten.)

Stattdessen führt jeder rekursive Aufruf von add dass ein neuer Frame auf dem Stack des Threads zugewiesen wird. Wenn Sie beispielsweise add(1000, 1) aufrufen, werden 1000 rekursive Anrufe benötigt, um bei der Antwort 1001 anzukommen.

Das Problem ist, dass die Größe des Java-Thread-Stacks beim Erstellen des Threads festgelegt wird. (Dazu gehört der "main" -Thread in einem Singlethread-Programm.) Wenn zu viele Stack-Frames zugewiesen werden, läuft der Stack über. Die JVM erkennt dies und gibt einen StackOverflowError .

Ein Ansatz, um damit umzugehen, besteht darin, einfach einen größeren Stapel zu verwenden. Es gibt JVM-Optionen, die die Standardgröße eines Stacks steuern. Sie können auch die Stackgröße als Thread Konstruktorparameter angeben. Leider "schiebt" dies nur den Stapelüberlauf. Wenn Sie eine Berechnung durchführen müssen, die einen noch größeren Stack erfordert, wird der StackOverflowError zurückgegeben.

Die eigentliche Lösung besteht darin, rekursive Algorithmen zu identifizieren, bei denen eine tiefe Rekursion wahrscheinlich ist, und die *Endanrufoptimierung manuell* auf Quellcodeebene durchzuführen. Zum Beispiel kann unsere add Methode wie folgt umgeschrieben werden:

```
public static int add(int a, int b) {
    while (a != 0) {
       a = a - 1;
       b = b + 1;
    }
    return b;
}
```

(Offensichtlich gibt es bessere Möglichkeiten, zwei Ganzzahlen hinzuzufügen. Das Obige dient lediglich zur Veranschaulichung der Auswirkung der manuellen Anrufbeseitigung.)

Warum die Rückrufbeseitigung (noch) nicht in Java implementiert ist

Es gibt eine Reihe von Gründen, warum das Hinzufügen der Rückrufbeseitigung zu Java nicht einfach ist. Zum Beispiel:

- Ein Teil des Codes könnte auf StackOverflowError angewiesen sein, um beispielsweise die Größe eines Rechenproblems zu beschränken.
- Sandbox-Sicherheitsmanager sind häufig darauf angewiesen, den Aufrufstapel zu analysieren, wenn sie entscheiden, ob nicht privilegierter Code eine privilegierte Aktion ausführen soll.

Wie John Rose in "Tail Calls in der VM" erklärt :

> *"Die Auswirkungen des Entfernens des Stackframes des Aufrufers sind für einige APIs sichtbar, insbesondere für die Überprüfung der Zugriffskontrolle und für die Stapelverfolgung. Es ist, als hätte der Anrufer des Anrufers den Angerufenen direkt angerufen. Alle Rechte, die der Anrufer besitzt, werden verworfen, nachdem die Kontrolle an den Server übergeben wurde callee. Die Verknüpfung und die Zugänglichkeit der callee-Methode werden jedoch vor der Übertragung der Kontrolle berechnet und berücksichtigen den Aufrufer, der den Anruf ausführt. "*

Mit anderen Worten, die Beseitigung von Rückrufen könnte dazu führen, dass eine Zugriffskontrollmethode fälschlicherweise der Meinung ist, dass eine sicherheitsrelevante API von vertrauenswürdigem Code aufgerufen wurde.

Rekursion online lesen: https://riptutorial.com/de/java/topic/914/rekursion

Kapitel 133: Remote Method Invocation (RMI)

Bemerkungen

RMI erfordert 3 Komponenten: Client, Server und eine gemeinsam genutzte Remote-Schnittstelle. Die gemeinsam genutzte Remote-Schnittstelle definiert den Client-Server-Vertrag durch Angabe der Methoden, die ein Server implementieren muss. Die Schnittstelle muss für den Server sichtbar sein, damit die Methoden implementiert werden können. Die Schnittstelle muss für den Client sichtbar sein, damit er weiß, welche Methoden ("Dienste") der Server bereitstellt.
Jedes Objekt, das eine Remote-Schnittstelle implementiert, übernimmt die Rolle eines Servers. Somit ist eine Client-Server-Beziehung, in der der Server auch Methoden im Client aufrufen kann, eine Server-Server-Beziehung. Dies wird als *Rückruf bezeichnet,* da der Server den "Client" zurückrufen kann. Aus diesem Grund ist es akzeptabel, den Designation *Client* für die Server zu verwenden, die als solche funktionieren.

Die gemeinsam genutzte Remote-Schnittstelle ist eine beliebige Schnittstelle, die Remote . Ein Objekt, das als Server fungiert, durchläuft Folgendes:

1. Implementiert die freigegebene Remote-Schnittstelle entweder explizit oder implizit durch Erweiterung von UnicastRemoteObject das Remote implementiert.
2. Exportiert, entweder implizit, wenn UnicastRemoteObject erweitert UnicastRemoteObject , oder explizit durch UnicastRemoteObject#exportObject an UnicastRemoteObject#exportObject .
3. In eine Registry eingebunden, entweder direkt über die Registry oder indirekt über die Naming . Dies ist nur für den Aufbau der Erstkommunikation notwendig, da weitere Stubs direkt über RMI übergeben werden können.

In der Projektkonfiguration sind die Client- und Serverprojekte völlig unabhängig voneinander, aber jedes gibt ein freigegebenes Projekt in seinem Erstellungspfad an. Das freigegebene Projekt enthält die Remote-Schnittstellen.

Examples

Client-Server: Aufrufen von Methoden in einer JVM von einer anderen

Die gemeinsam genutzte Remote-Schnittstelle:

```
package remote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteServer extends Remote {

    int stringToInt(String string) throws RemoteException;
}
```

Der Server, der die freigegebene Remote-Schnittstelle implementiert:

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

Der Client, der eine Methode auf dem Server (remote) aufruft:

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

Ausgabe:

```
 Server erhalten: "120"
```

```
Kunde erhielt: 120
```

Callback: Aufrufen von Methoden auf einem "Client"

Überblick

In diesem Beispiel senden 2 Clients Informationen über einen Server. Ein Client sendet dem Server eine Nummer, die an den zweiten Client weitergeleitet wird. Der zweite Client halbiert die Nummer und sendet sie über den Server an den ersten Client zurück. Der erste Client macht das Gleiche. Der Server stoppt die Kommunikation, wenn die von einem der Clients zurückgegebene Nummer weniger als 10 ist. Der Rückgabewert vom Server an die Clients (die Nummer, die er in eine String-Darstellung umgewandelt hat) führt dann den Prozess zurück.

1. Ein Anmeldeserver bindet sich an eine Registry.
2. Ein Client sucht den Login-Server und ruft die login Methode mit seinen Informationen auf. Dann:
    - Der Anmeldeserver speichert die Clientinformationen. Es enthält den Stub des Clients mit den Rückmeldemethoden.
    - Der Anmeldeserver erstellt einen Serverstub ("Verbindung" oder "Sitzung") und gibt ihn an den zu speichernden Client zurück. Es enthält den Stub des Servers mit seinen Methoden, einschließlich einer logout (in diesem Beispiel nicht verwendet).
3. Ein Client ruft das passInt des Servers mit dem Namen des Empfängerclients und einem int .
4. Der Server ruft die half des Empfängerclients mit diesem int . Dadurch wird eine Hin- und Her-Kommunikation (Anrufe und Rückrufe) eingeleitet, bis der Server angehalten wird.

Die gemeinsam genutzten Remote-Schnittstellen

Der Login-Server:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteLogin extends Remote {

    RemoteConnection login(String name, RemoteClient client) throws RemoteException;
}
```

Der Server:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteConnection extends Remote {

    void logout() throws RemoteException;

    String passInt(String name, int i) throws RemoteException;
}
```

Der Kunde:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;
```

```
public interface RemoteClient extends Remote {

    void half(int i) throws RemoteException;
}
```

Die Implementierungen

Der Login-Server:

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

Der Server:

```
package callbackServer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.server.Unreferenced;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
```

```
public class Connection implements RemoteConnection, Unreferenced {

    RemoteClient client;
    String name;

    public Connection(String name, RemoteClient client) {

        this.client = client;
        this.name = name;
        try {
            UnicastRemoteObject.exportObject(this, Registry.REGISTRY_PORT);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void unreferenced() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void logout() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String passInt(String recipient, int i) {

        System.out.println("Server received from " + name + ":" + i);
        if (i < 10)
            return String.valueOf(i);
        RemoteClient client = LoginServer.clients.get(recipient);
        try {
            client.half(i);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
        return String.valueOf(i);
    }
}
```

Der Kunde:

```
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 + "\"");
    }
}
```

Das Beispiel ausführen:

1. Führen Sie den Login-Server aus.
2. Führen Sie einen Client mit den Argumenten Client2 Client1 1097 .
3. Führen Sie einen Client mit den Argumenten Client1 Client2 1098 .

Die Ausgänge erscheinen in 3 Konsolen, da es 3 JVMs gibt. hier sind sie zusammengeworfen:

```
Client2 ist angemeldet
Client1 angemeldet
Server erhielt von Client1: 120
Server erhalten von Client2: 60
Server erhielt von Client1: 30
Server erhalten von Client2: 15
Server erhalten von Client1: 7
Client1 erhielt: "7"
```

```
Client2 erhalten: "15"
Client1 erhielt: "30"
Client2 erhalten: "60"
```

Einfaches RMI-Beispiel mit Client- und Server-Implementierung

Dies ist ein einfaches RMI-Beispiel mit fünf Java-Klassen und zwei Paketen (*Server* und *Client)* .

Server-Paket

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();
    }
}
```

Client-Paket

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);
        }
    }
}
```

Testen Sie Ihre Bewerbung

- Starten Sie die Hauptmethode von Server.java. Ausgabe:

```
Server : Registry created.
Server : PersonList registered
```

- Starten Sie die Hauptmethode von PersonTest.java. Ausgabe:

```
Peter Pan
Pippi Langstrumpf
```

Remote Method Invocation (RMI) online lesen: https://riptutorial.com/de/java/topic/171/remote-method-invocation--rmi-

Kapitel 134: Ressourcen (auf Klassenpfad)

Einführung

Java ermöglicht das Abrufen von dateibasierten Ressourcen, die in einer JAR-Datei neben kompilierten Klassen gespeichert sind. Dieses Thema konzentriert sich darauf, diese Ressourcen zu laden und für Ihren Code verfügbar zu machen.

Bemerkungen

Eine *Ressource* besteht aus dateiähnlichen Daten mit einem pfadähnlichen Namen, der sich im Klassenpfad befindet. Die häufigste Verwendung von Ressourcen ist das Bündeln von Anwendungsbildern, Sounds und schreibgeschützten Daten (z. B. Standardkonfiguration).

Auf die Ressourcen kann mit den Methoden ClassLoader.getResource und ClassLoader.getResourceAsStream zugegriffen werden. Der häufigste Anwendungsfall besteht darin, Ressourcen in demselben Paket wie die Klasse abzulegen, in der sie gelesen werden. Die Methoden Class.getResource und Class.getResourceAsStream dienen diesem allgemeinen Anwendungsfall.

Der einzige Unterschied zwischen einer getResource-Methode und der getResourceAsStream-Methode besteht darin, dass erstere eine URL zurückgibt, während letztere diese URL öffnet und einen InputStream zurückgibt.

Die Methoden von ClassLoader akzeptieren einen pfadähnlichen Ressourcennamen als Argument und durchsuchen jede Position im Classpath des ClassLoader nach einem Eintrag, der diesem Namen entspricht.

- Wenn ein Classpath-Speicherort eine JAR-Datei ist, wird ein Jar-Eintrag mit dem angegebenen Namen als Übereinstimmung betrachtet.
- Wenn ein Classpath-Speicherort ein Verzeichnis ist, wird eine relative Datei unter diesem Verzeichnis mit dem angegebenen Namen als Übereinstimmung betrachtet.

Der Ressourcenname ähnelt dem Pfadteil einer relativen URL. Auf *allen Plattformen* werden Schrägstriche (/) als Verzeichnisseparatoren verwendet. Es darf nicht mit einem Schrägstrich beginnen.

Die entsprechenden Methoden von Class sind ähnlich, außer:

- Der Ressourcenname kann mit einem Schrägstrich beginnen. In diesem Fall wird der anfängliche Schrägstrich entfernt und der Rest des Namens an die entsprechende Methode von ClassLoader übergeben.
- Wenn der Ressourcenname nicht mit einem Schrägstrich beginnt, wird er als relativ zu der Klasse behandelt, deren getResource- oder getResourceAsStream-Methode aufgerufen wird. Der tatsächliche Ressourcenname wird zu *Paket* / *Name* , wobei *Paket* der Name des Pakets ist, zu dem die Klasse gehört, wobei jeder Punkt durch einen Schrägstrich ersetzt wird, und *Name* das ursprüngliche Argument für die Methode ist.

Zum Beispiel:

```
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);
    }
}
```

Ressourcen sollten in benannten Paketen und nicht im Stammverzeichnis einer JAR-Datei abgelegt werden. Aus demselben Grund werden Klassen in Paketen platziert: Um Kollisionen zwischen mehreren Anbietern zu vermeiden. Wenn sich zum Beispiel mehrere .jar-Dateien im Klassenpfad befinden und mehr als eine von ihnen einen Eintrag config.properties im Stammverzeichnis enthält, geben Aufrufe der Methoden getResource oder getResourceAsStream die Datei config.properties aus dem ersten Verzeichnis zurück der Klassenpfad Dies ist kein vorhersagbares Verhalten in Umgebungen, in denen die Klassenpfadreihenfolge nicht direkt von der Anwendung gesteuert wird, wie z. B. Java EE.

Alle getResource- und getResourceAsStream-Methoden geben null wenn die angegebene Ressource nicht vorhanden ist. Da der Anwendung zum Erstellungszeitpunkt Ressourcen hinzugefügt werden müssen, sollten deren Speicherorte bekannt sein, wenn der Code geschrieben wird. Ein Fehler beim Finden einer Ressource zur Laufzeit ist in der Regel das Ergebnis eines Programmierfehlers.

Ressourcen sind schreibgeschützt. Es ist nicht möglich, auf eine Ressource zu schreiben. Anfänger begehen häufig den Fehler, anzunehmen, dass die Ressource bei der Entwicklung in einer IDE (wie Eclipse) eine separate physische Datei ist. Daher ist es sicher, dass sie im allgemeinen Fall als separate physische Datei behandelt wird. Dies ist jedoch nicht korrekt. Anwendungen werden fast immer als Archive wie JAR- oder WAR-Dateien verteilt, und in solchen Fällen ist eine Ressource keine separate Datei und kann nicht beschrieben werden. (Die getFile-Methode der URL-Klasse ist *keine* Lösung für dieses Problem; sie gibt trotz ihres Namens lediglich den Pfadteil einer URL zurück, der keinesfalls als gültiger Dateiname garantiert ist.)

Es gibt keine sichere Möglichkeit, Ressourcen zur Laufzeit aufzulisten. Da wiederum die Entwickler dafür verantwortlich sind, Ressourcendateien zur Buildzeit hinzuzufügen, sollten sie ihre Pfade bereits kennen. Es gibt zwar Problemumgehungen, sie sind jedoch nicht zuverlässig und scheitern schließlich.

Examples

Laden eines Bildes von einer Ressource

So laden Sie ein gebündeltes Bild:

```
package com.example;

public class ExampleApplication {
    private Image getIcon() throws IOException {
        URL imageURL = ExampleApplication.class.getResource("icon.png");
        return ImageIO.read(imageURL);
    }
}
```

Standardkonfiguration wird geladen

So lesen Sie die Standardkonfigurationseigenschaften:

```
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;
    }
}
```

Laden der gleichnamigen Ressource aus mehreren JARs

Eine Ressource mit demselben Pfad und Namen kann in mehr als einer JAR-Datei im Klassenpfad vorhanden sein. Häufige Fälle sind Ressourcen, die einer Konvention folgen oder Teil einer Verpackungsspezifikation sind. Beispiele für solche Ressourcen sind

- META-INF / MANIFEST.MF
- META-INF / beans.xml (CDI-Spezifikation)
- ServiceLoader-Eigenschaften, die Implementierungsanbieter enthalten

Um auf *alle* diese Ressourcen in verschiedenen Gläsern zugreifen zu *können* , muss ein ClassLoader verwendet werden, der über eine Methode dafür verfügt. Die zurückgegebene Enumeration kann bequem mit einer Collections-Funktion in eine List konvertiert werden.

```
Enumeration<URL> resEnum = MyClass.class.getClassLoader().getResources("META-
INF/MANIFEST.MF");
ArrayList<URL> resources = Collections.list(resEnum);
```

Ressourcen mit einem Classloader finden und lesen

Das Laden von Ressourcen in Java umfasst die folgenden Schritte:

1. Die Class oder den ClassLoader finden, die die Ressource finden.
2. Die Ressource finden
3. Abrufen des Byte-Streams für die Ressource.
4. Lesen und Verarbeiten des Byte-Streams.
5. Den Bytestrom schließen.

Die letzten drei Schritte werden normalerweise ausgeführt, indem die URL an eine Bibliotheksmethode oder einen Konstruktor übergeben wird, um die Ressource zu laden. In diesem Fall verwenden Sie normalerweise eine getResource Methode. Es ist auch möglich, die Ressourcendaten im Anwendungscode zu lesen. In diesem Fall verwenden Sie normalerweise getResourceAsStream .

Absolute und relative Ressourcenpfade

Ressourcen, die aus dem Klassenpfad geladen werden können, werden durch einen *Pfad angegeben* . Die Syntax des Pfads ähnelt einem UNIX / Linux-Dateipfad. Es besteht aus einfachen Namen, die durch Schrägstrich (/) getrennt sind. Ein *relativer Pfad* beginnt mit einem Namen und ein *absoluter Pfad* beginnt mit einem Trennzeichen.

Wie in den Classpath-Beispielen beschrieben, definiert der Classpath einer JVM einen Namespace, indem die Namespaces der Verzeichnisse und JAR- oder ZIP-Dateien im Classpath eingeblendet werden. Wenn ein absoluter Pfad aufgelöst wird, interpretieren die Klassenladeprogramme den Anfang / als die Wurzel des Namespaces. Im Gegensatz dazu *kann* ein relativer Pfad im Namensraum relativ zu einem beliebigen „Ordner" gelöst *werden.* Der verwendete Ordner hängt von dem Objekt ab, mit dem Sie den Pfad auflösen.

Eine Klasse oder einen Klassenlader erhalten

Eine Ressource kann entweder über ein Class Objekt oder ein ClassLoader Objekt ClassLoader werden. Ein Class Objekt kann relative Pfade auflösen, sodass Sie normalerweise einen dieser Pfade verwenden, wenn Sie über eine (Klassen-) relative Ressource verfügen. Es gibt verschiedene Möglichkeiten, ein Class Objekt zu erhalten. Zum Beispiel:

- Eine *Klasse wörtliche* wird Ihnen das Class - Objekt für jede Klasse , die Sie in Java - Quellcode nennen kann; zB String.class gibt Ihnen das Class Objekt für den String Typ.
- Das Object.getClass() gibt Ihnen das Class Objekt für den Typ eines beliebigen Objekts. "hello".getClass() Beispiel "hello".getClass() ist eine andere Möglichkeit, Class vom String Typ zu erhalten.
- Die Class.forName(String) -Methode Class.forName(String) falls erforderlich) eine Klasse dynamisch und gibt ihr Class Objekt zurück. zB Class.forName("java.lang.String") .

Ein ClassLoader Objekt wird normalerweise durch Aufrufen von getClassLoader() für ein Class Objekt abgerufen. Es ist auch möglich, den Standard-Classloader der JVM mithilfe der statischen ClassLoader.getSystemClassLoader() -Methode zu erhalten.

Die get Methoden

Sobald Sie eine haben Class oder ClassLoader - Instanz können Sie eine Ressource finden, eine der folgenden Methoden verwenden:

Methoden	Beschreibung
`ClassLoader.getResource(path)` `ClassLoader.getResources(path)`	Gibt eine URL zurück, die den Ort der Ressource mit dem angegebenen Pfad darstellt.
`ClassLoader.getResources(path)` `Class.getResources(path)`	Gibt eine `Enumeration<URL>` , die die URLs `foo.bar` , die zum `foo.bar` der `foo.bar` Ressource verwendet werden können. siehe unten.
`ClassLoader.getResourceAsStream(path)` `Class.getResourceStream(path)`	Gibt einen `InputStream` aus dem Sie den Inhalt der `foo.bar` Ressource als Folge von Bytes lesen können.

Anmerkungen:

- Der Hauptunterschied zwischen den ClassLoader und Class Versionen der Methoden besteht in der Art und Weise, wie relative Pfade interpretiert werden.
    - Die Class lösen einen relativen Pfad im "Ordner" auf, der dem Klassenpaket entspricht.
    - Die ClassLoader Methoden behandeln relative Pfade so, als wären sie absolut. dh sie lösen sie im "Root-Ordner" des Classpath-Namespaces auf.
- Wenn die angeforderte Ressource (oder Ressourcen) nicht gefunden werden können, geben die methods return getResource und getResourceAsStream null zurück , and the methods return an empty getResources methods return an empty Enumeration` zurück.
- Die zurückgegebenen URLs können mit URL.toStream() . Sie können file: URLs oder andere herkömmliche URLs sein. Wenn sich die Ressource jedoch in einer JAR-Datei befindet, handelt es sich um jar: URLs, die die JAR-Datei und eine bestimmte Ressource darin angeben.
- Wenn Ihr Code eine getResourceAsStream Methode (oder URL.toStream()) verwendet, um einen InputStream zu erhalten, ist er für das Schließen des Stream-Objekts verantwortlich. Wenn der Stream nicht geschlossen wird, kann dies zu einem Ressourcenleck führen.

Ressourcen (auf Klassenpfad) online lesen:
https://riptutorial.com/de/java/topic/2433/ressourcen--auf-klassenpfad-

Kapitel 135: RSA-Verschlüsselung

Examples

Ein Beispiel für ein Hybrid-Kryptosystem bestehend aus OAEP und GCM

Im folgenden Beispiel werden Daten mithilfe eines Hybrid-Kryptosystems bestehend aus AES GCM und OAEP unter Verwendung ihrer Standardparametergrößen und einer AES-Schlüsselgröße von 128 Bit verschlüsselt.

OAEP ist weniger anfällig für Padding-Orakel-Angriffe als PKCS # 1 v1.5-Padding. GCM ist auch vor Auffüllangriffen geschützt.

Die Entschlüsselung kann durchgeführt werden, indem zuerst die Länge des eingekapselten Schlüssels und dann der gekapselte Schlüssel abgerufen wird. Der gekapselte Schlüssel kann dann mit dem privaten RSA-Schlüssel entschlüsselt werden, der ein Schlüsselpaar mit dem öffentlichen Schlüssel bildet. Danach kann der AES / GCM-verschlüsselte Chiffretext zum ursprünglichen Klartext entschlüsselt werden.

Das Protokoll besteht aus:

1. ein Längenfeld für den RSAPrivateKey Schlüssel (RSAPrivateKey vermisst eine getKeySize() - Methode);
2. der umwickelte / gekapselte Schlüssel, der die gleiche Größe wie die RSA-Schlüsselgröße in Bytes hat;
3. der GCM-Chiffretext und das 128-Bit-Authentifizierungs-Tag (automatisch von Java hinzugefügt).

Anmerkungen:

- Um diesen Code korrekt zu verwenden, sollten Sie einen RSA-Schlüssel mit mindestens 2048 Bit angeben. Größer ist besser (aber langsamer, insbesondere während der Entschlüsselung).
- Um AES-256 zu verwenden, sollten Sie zunächst die unlimitierten Kryptographiedateien installieren.
- Anstatt ein eigenes Protokoll zu erstellen, sollten Sie stattdessen ein Containerformat verwenden, z. B. die Cryptographic Message Syntax (CMS / PKCS # 7) oder PGP.

Hier ist das Beispiel:

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

Natürlich ist Verschlüsselung ohne Entschlüsselung nicht sehr nützlich. Beachten Sie, dass dies zu minimalen Informationen führt, wenn die Entschlüsselung fehlschlägt.

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

RSA-Verschlüsselung online lesen: https://riptutorial.com/de/java/topic/1889/rsa-verschlusselung

Kapitel 136: Sammlungen

Einführung

Das Collections-Framework in java.util bietet eine Reihe generischer Klassen für Datensätze mit Funktionen, die von regulären Arrays nicht bereitgestellt werden können.

Collections Framework enthält Schnittstellen für Collection<O> mit den Haupt-Subschnittstellen List<O> und Set<O> und der Mapping-Collection Map<K,V> . Collections sind die Root-Schnittstelle und werden von vielen anderen Collection-Frameworks implementiert.

Bemerkungen

Sammlungen sind Objekte, in denen Sammlungen anderer Objekte gespeichert werden können. Sie können den Typ der in einer Sammlung gespeicherten Daten mit Generics angeben.

Sammlungen verwenden im Allgemeinen die Namespaces java.util oder java.util.concurrent .

Java SE 1.4

Java 1.4.2 und folgende unterstützen keine Generics. Daher können Sie nicht die Typparameter angeben, die eine Auflistung enthält. Abgesehen von der Typensicherheit müssen Sie auch Casts verwenden, um den korrekten Typ aus einer Sammlung zurückzuholen.

Neben Collection<E> gibt es mehrere Haupttypen von Collections, von denen einige Untertypen haben.

- List<E> ist eine geordnete Sammlung von Objekten. Es ähnelt einem Array, definiert jedoch keine Größenbegrenzung. Implementierungen werden normalerweise intern größer, um neuen Elementen Rechnung zu tragen.
- Set<E> ist eine Sammlung von Objekten, die keine Duplikate zulässt.
    - SortedSet<E> ist ein Set<E> , das auch die Elementreihenfolge angibt.
- Map<K,V> ist eine Sammlung von Schlüssel / Wert-Paaren.
    - SortedMap<K,V> ist eine Map<K,V> , in der auch die Elementreihenfolge angegeben ist.

Java SE 5

Java 5 fügt einen neuen Sammlungstyp hinzu:

- Queue<E> ist eine Sammlung von Elementen, die in einer bestimmten Reihenfolge verarbeitet werden sollen. Die Implementierung gibt an, ob dies FIFO oder LIFO ist. Dies macht die Stack Klasse überflüssig.

Java SE 6

Java 6 fügt einige neue Untertypen von Sammlungen hinzu.

- NavigableSet<E> ist ein Set<E> mit integrierten Navigationsmethoden.
- NavigableMap<K,V> ist eine Map<K,V> mit integrierten Navigationsmethoden.
- Deque<E> ist eine Queue<E> , die von beiden Enden gelesen werden kann.

Beachten Sie, dass die obigen Elemente alle Schnittstellen sind. Um sie verwenden zu können, müssen Sie die entsprechenden implementierenden Klassen wie ArrayList , HashSet , HashMap oder PriorityQueue .

Für jeden Auflistungstyp gibt es mehrere Implementierungen mit unterschiedlichen Leistungskennzahlen und Anwendungsfällen.

Beachten Sie, dass das Liskov-Substitutionsprinzip für die Erfassungsuntertypen gilt. Das heißt, ein SortedSet<E> kann an eine Funktion übergeben werden, die ein Set<E> erwartet. Weitere Informationen zur Verwendung von Auflistungen mit Klassenvererbung finden Sie auch im Abschnitt Generics über Bounded Parameters .

Wenn Sie eigene Sammlungen erstellen möchten, kann es einfacher sein, eine der abstrakten

Klassen (z. B. AbstractList) zu erben, anstatt die Schnittstelle zu implementieren.

Java SE 1.2

Vor 1.2 mussten Sie stattdessen die folgenden Klassen / Schnittstellen verwenden:

- Vector statt ArrayList
- Dictionary anstelle von Map . Beachten Sie, dass das Wörterbuch auch eine abstrakte Klasse und keine Schnittstelle ist.
- Hashtable statt HashMap

Diese Klassen sind veraltet und sollten in modernem Code nicht verwendet werden.

Examples

ArrayList deklarieren und Objekte hinzufügen

Wir können eine ArrayList erstellen (nach der List Schnittstelle):

```
List aListOfFruits = new ArrayList();
```

Java SE 5

```
List<String> aListOfFruits = new ArrayList<String>();
```

Java SE 7

```
List<String> aListOfFruits = new ArrayList<>();
```

Verwenden Sie nun die Methode add , um einen String hinzuzufügen:

```
aListOfFruits.add("Melon");
aListOfFruits.add("Strawberry");
```

Im obigen Beispiel enthält die ArrayList den String "Melon" an Index 0 und den String "Strawberry" an Index 1.

Wir können auch mehrere Elemente mit der addAll(Collection<? extends E> c) -Methode hinzufügen

```
List<String> aListOfFruitsAndVeggies = new ArrayList<String>();
aListOfFruitsAndVeggies.add("Onion");
aListOfFruitsAndVeggies.addAll(aListOfFruits);
```

Jetzt ist "Onion" bei aListOfFruitsAndVeggies auf 0 aListOfFruitsAndVeggies , "Melon" bei Index 1 und "Strawberry" bei Index 2.

Erstellen von Sammlungen aus vorhandenen Daten

Standardsammlungen

Java Collections-Framework

Eine einfache Methode zum Arrays.asList einer List aus einzelnen Datenwerten besteht in der Verwendung der java.utils.Arrays Methode Arrays.asList :

```
List<String> data = Arrays.asList("ab", "bc", "cd", "ab", "bc", "cd");
```

Alle Standardimplementierungsimplementierungen stellen Konstruktoren bereit, die eine andere

Collection als Argument verwenden und der neuen Collection zum Zeitpunkt der Konstruktion alle Elemente hinzufügen:

```
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
```

Google Guava Collections-Framework

Ein weiteres großartiges Framework ist Google Guava , das eine erstaunliche Google Guava ist (mit praktischen statischen Methoden) zum Erstellen verschiedener Typen von Standardsammlungen. Lists und 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");
```

Mapping-Sammlungen

Java Collections-Framework

In ähnlicher Weise kann für Karten bei einer Map<String, Object> map eine neue Karte mit allen Elementen wie folgt erstellt werden:

```
Map<String, Object> map1 = new HashMap<>(map);
SortedMap<String, Object> map2 = new TreeMap<>(map);
```

Apache Commons Collections Framework

Mit Apache Commons Sie Map mit Array in ArrayUtils.toMap sowie 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"}});
```

Jedes Element des Arrays muss entweder ein Map.Entry oder ein Array sein, das mindestens zwei Elemente enthält, wobei das erste Element als Schlüssel und das zweite als Wert verwendet wird.

Google Guava Collections-Framework

Dienstprogrammklasse aus dem Google Guava Framework heißt 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

Stream ,

```
Stream.of("xyz", "abc").collect(Collectors.toList());
```

oder

```
Arrays.stream("xyz", "abc").collect(Collectors.toList());
```

Listen beitreten

Die folgenden Möglichkeiten können zum Verbinden von Listen verwendet werden, ohne die Quellliste (n) zu ändern.

Erste Ansatz. Hat mehr Zeilen aber leicht verständlich

```
List<String> newList = new ArrayList<String>();
newList.addAll(listOne);
newList.addAll(listTwo);
```

Zweiter Ansatz. Hat eine Zeile weniger aber weniger lesbar.

```
List<String> newList = new ArrayList<String>(listOne);
newList.addAll(listTwo);
```

Dritter Ansatz. Erfordert die Bibliothek von Drittanbieter- Apache-Commons-Collections .

```
ListUtils.union(listOne,listTwo);
```

Java SE 8

Bei Verwendung von Streams kann das gleiche durch erreicht werden

```
List<String> newList = Stream.concat(listOne.stream(),
listTwo.stream()).collect(Collectors.toList());
```

Verweise. Schnittstellenliste

Elemente aus einer Liste innerhalb einer Schleife entfernen

Es ist schwierig, Elemente aus einer Liste zu entfernen, wenn sie sich innerhalb einer Schleife befinden. Dies liegt daran, dass der Index und die Länge der Liste geändert werden.

In Anbetracht der folgenden Liste werden hier einige Beispiele aufgeführt, die zu einem unerwarteten Ergebnis führen, und einige, die zu einem korrekten Ergebnis führen.

```
List<String> fruits = new ArrayList<String>();
fruits.add("Apple");
fruits.add("Banana");
```

```
fruits.add("Strawberry");
```

FALSCH

In Iteration von `for` *Überspringen "Banane"* entfernen *:*

Das Codebeispiel druckt nur Apple und Strawberry . Banana wird übersprungen , da es zum Index bewegt 0 einmal von Apple gelöscht wird, aber zur gleichen Zeit i zu erhöht wird 1 .

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

Entfernen in der erweiterten `for` Anweisung *Throws Exception:*

Wegen der Iteration über die Sammlung und deren gleichzeitige Änderung.

> Löst aus: java.util.ConcurrentModificationException

```
for (String fruit : fruits) {
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruits.remove(fruit);
    }
}
```

RICHTIG

While-Schleife mit einem `Iterator` entfernen

```
Iterator<String> fruitIterator = fruits.iterator();
while(fruitIterator.hasNext()) {
    String fruit = fruitIterator.next();
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruitIterator.remove();
    }
}
```

Die Iterator Schnittstelle verfügt nur für diesen Fall über eine remove() Methode. Diese Methode ist jedoch in der Dokumentation als "optional" markiert und kann eine UnsupportedOperationException .

> Löst aus: UnsupportedOperationException - wenn der Entfernungsvorgang von diesem Iterator nicht unterstützt wird

Es ist daher ratsam, die Dokumentation zu überprüfen, um sicherzustellen, dass diese Operation unterstützt wird (in der Praxis, sofern es sich nicht um eine unveränderliche Sammlung handelt, die über eine Bibliothek eines Drittanbieters erhalten wird, oder wenn eine der Methoden Collections.unmodifiable...() verwendet wird. Die Operation wird fast immer unterstützt.

Während der Verwendung eines Iterator eine ConcurrentModificationException ausgelöst, wenn der modCount der List der modCount des Iterator geändert wird. Dies kann in demselben Thread oder in einer Multithread-Anwendung geschehen, die dieselbe Liste verwendet.

Ein modCount ist eine int Variable, die zählt, wie oft diese Liste strukturell geändert wurde.

Eine strukturelle Änderung bedeutet im Wesentlichen, dass eine add() oder remove() Operation für das Collection Objekt aufgerufen wird (von Iterator vorgenommene Änderungen werden nicht gezählt). Wenn der Iterator erstellt wird, speichert er diesen modCount und prüft bei jeder Iteration der List , ob der aktuelle modCount ist und wann der Iterator erstellt wurde. Wenn der modCount Wert modCount wird, wird eine ConcurrentModificationException modCount .

Für die oben deklarierte Liste wird eine Operation wie folgt keine Ausnahme auslösen:

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.set(0, "Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());
}
```

Iterator jedoch nach der Initialisierung eines Iterator ein neues Element zur List hinzufügen, wird eine ConcurrentModificationException :

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.add("Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());    //ConcurrentModificationException here
}
```

Rückwärts iterieren

```
for (int i = (fruits.size() - 1); i >=0; i--) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
    }
}
```

Dies überspringt nichts. Der Nachteil dieses Ansatzes ist, dass die Ausgabe umgekehrt ist. In den meisten Fällen, in denen Sie Elemente entfernen, spielt dies jedoch keine Rolle. Sie sollten dies niemals mit LinkedList tun.

Iterieren vorwärts, Anpassen des Schleifenindex

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println (fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
         fruits.remove(i);
         i--;
    }
}
```

Dies überspringt nichts. Wenn das i te Element aus der List , wird das ursprünglich beim Index i+1 positionierte Element zum neuen i ten Element. Daher kann die Schleife i dekrementieren, damit die nächste Iteration das nächste Element verarbeiten kann, ohne zu überspringen.

Verwenden einer Liste "Sollte entfernt werden"

```
ArrayList shouldBeRemoved = new ArrayList();
for (String str : currentArrayList) {
    if (condition) {
        shouldBeRemoved.add(str);
    }
}
```

```
currentArrayList.removeAll(shouldBeRemoved);
```

Mit dieser Lösung kann der Entwickler überprüfen, ob die korrekten Elemente sauberer entfernt wurden.

Java SE 8

In Java 8 sind folgende Alternativen möglich. Diese sind sauberer und unkomplizierter, wenn das Entfernen nicht in einer Schleife erfolgen muss.

Stream filtern

Eine List kann gestreamt und gefiltert werden. Mit einem geeigneten Filter können Sie alle unerwünschten Elemente entfernen.

```
List<String> filteredList =
    fruits.stream().filter(p -> !"Apple".equals(p)).collect(Collectors.toList());
```

Beachten Sie, dass im Gegensatz zu allen anderen Beispielen hier, in diesem Beispiel eine neue produziert List Instanz und hält die ursprüngliche List unverändert.

removeIf

Spart den Aufwand für das Erstellen eines Streams, wenn nur ein Satz von Elementen entfernt werden muss.

```
fruits.removeIf(p -> "Apple".equals(p));
```

Unveränderbare Sammlung

Manchmal ist es nicht ratsam, eine interne Sammlung offenzulegen, da dies aufgrund seiner veränderlichen Eigenschaft zu einer Sicherheitsanfälligkeit durch schädlichen Code führen kann. Um "schreibgeschützte" Sammlungen bereitzustellen, stellt java seine unveränderlichen Versionen zur Verfügung.

Eine nicht veränderbare Sammlung ist oft eine Kopie einer veränderbaren Sammlung, die garantiert, dass die Sammlung selbst nicht geändert werden kann. Versuche, es zu ändern, führen zu einer Ausnahme für die Ausnahme "UnsupportedOperationException".

Es ist wichtig zu beachten, dass Objekte, die sich in der Sammlung befinden, noch geändert werden können.

```
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);
    }
}
```

Der folgende Versuch, eine nicht veränderbare Sammlung zu ändern, löst eine Ausnahme aus:

```
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);
    }
}
```

Ausgabe:

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
    at App.main(App.java:12)
```

Iteration über Sammlungen

Iteration über Liste

```
List<String> names  = new ArrayList<>(Arrays.asList("Clementine", "Duran", "Mike"));
```

Java SE 8

```
names.forEach(System.out::println);
```

Wenn wir Parallelität brauchen, verwenden Sie

```
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());
}
```

Iteration über 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());
}
```

Iteration über Map

```
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());
}
```

Unveränderliche leere Sammlungen

Manchmal ist es angebracht, eine unveränderliche leere Sammlung zu verwenden. Die Collections Klasse stellt Methoden bereit, um solche Sammlungen auf effiziente Weise abzurufen:

```
List<String> anEmptyList = Collections.emptyList();
Map<Integer, Date> anEmptyMap = Collections.emptyMap();
Set<Number> anEmptySet = Collections.emptySet();
```

Diese Methoden sind generisch und konvertieren die zurückgegebene Sammlung automatisch in den Typ, dem sie zugewiesen ist. Das heißt, ein Aufruf von zB emptyList() kann auf jede Art von zugeordnet werden List und ebenfalls für emptySet() und emptyMap() .

Die von diesen Methoden zurückgegebenen Auflistungen sind unveränderlich, da sie UnsupportedOperationException UnsupportedOperationException wenn Sie versuchen, Methoden aufzurufen, die ihren Inhalt ändern (add , put usw.). Diese Auflistungen sind in erster Linie als Ersatz für leere Methodenergebnisse oder andere Standardwerte nützlich, anstatt null oder Objekte mit new erstellen.

Sammlungen und Grundwerte

Sammlungen in Java funktionieren nur für Objekte. Dh es gibt keine Map<int, int> in Java. Stattdessen müssen primitive Werte in Objekte *verpackt* werden, wie in der Map<Integer, Integer> . Java-Auto-Boxing ermöglicht die transparente Verwendung dieser Sammlungen:

```
Map<Integer, Integer> map = new HashMap<>();
map.put(1, 17); // Automatic boxing of int to Integer objects
int a = map.get(1); // Automatic unboxing.
```

Leider ist der Aufwand *erheblich* . Eine HashMap<Integer, Integer> erfordert etwa 72 Bytes pro Eintrag (z. B. bei 64-Bit-JVM mit komprimierten Zeigern und unter Annahme von Ganzzahlen größer als 256 und einer Belastung von 50% der Map). Da die tatsächlichen Daten nur 8 Byte umfassen, führt dies zu einem erheblichen Overhead. Darüber hinaus erfordert es zwei Ebenen der Indirektion (Karte -> Eintrag -> Wert), es ist unnötig langsam.

Es gibt mehrere Bibliotheken mit optimierten Sammlungen für primitive Datentypen (die nur ~ 16 Bytes pro Eintrag bei 50% Last erfordern, dh 4x weniger Speicher und eine Ebene der Dereferenzierung weniger) Werte in Java.

Übereinstimmende Elemente mit Iterator aus Listen entfernen.

Oben habe ich ein Beispiel zum Entfernen von Elementen aus einer Liste innerhalb einer Schleife bemerkt und ich dachte an ein anderes Beispiel, das diesmal mit Hilfe der Iterator Schnittstelle nützlich sein könnte.
Dies ist eine Demonstration eines Tricks, der hilfreich sein kann, wenn Sie doppelte Elemente in Listen behandeln, die Sie entfernen möchten.

Hinweis: Dies wird nur zum **Entfernen von Elementen aus einer Liste innerhalb eines Schleifenbeispiels** hinzugefügt:

Definieren wir also unsere Listen wie gewohnt

```
    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));
```

Die folgende Methode nimmt zwei Collection-Objekte auf und führt den Zauber aus, die Elemente in unserer removeNameList entfernen, die mit Elementen in nameList .

```
private static void removeNames(Collection<String> collection1, Collection<String>
collection2) {
      //get Iterator.
    Iterator<String> iterator = collection1.iterator();

    //Loop while collection has items
    while(iterator.hasNext()){
        if (collection2.contains(iterator.next()))
            iterator.remove(); //remove the current Name or Item
    }
}
```

Aufrufen der Methode und Übergeben der nameList und der removeNameList wie folgt:
removeNames(nameList,removeNameList);
Erzeugt die folgende Ausgabe:

> Array-Liste vor dem Entfernen von Namen: **James Smith Sonny Huckle Berry Finn Allan**
> Array-Liste nach dem Entfernen von Namen: **James Smith Finn Allan**

Eine einfache, übersichtliche Verwendung für Sammlungen, die sich als nützlich erweisen, um sich wiederholende Elemente in Listen zu entfernen.

Erstellen Sie Ihre eigene Iterable-Struktur für die Verwendung mit Iterator oder für jede Schleife.

Um sicherzustellen, dass unsere Sammlung mithilfe eines Iterators oder für jede Schleife wiederholt werden kann, müssen wir die folgenden Schritte ausführen:

1. Das, worauf wir iterieren wollen, muss Iterable und Iterable iterator() Iterable .
2. Entwerfen Sie einen java.util.Iterator indem Sie hasNext() , next() und remove() überschreiben.

Ich habe unten eine einfache generische Liste für verknüpfte Listen hinzugefügt, die die oben genannten Entitäten verwendet, um die verknüpfte Liste iterierbar zu machen.

```
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 + " ");
        }
    }
}
```

Ausgabe

```
using Iterator:
1 2 4 3
using for-each:
1 2 4 3
```

Dies wird in Java 7+ ausgeführt. Sie können es auch auf Java 5 und Java 6 ausführen, indem Sie Folgendes ersetzen:

```
LinkedList<Integer> list = new LinkedList<>(1);
```

mit

```
LinkedList<Integer> list = new LinkedList<Integer>(1);
```

oder einfach jede andere Version durch Einfügen der kompatiblen Änderungen.

Pitfall: Ausnahmen für gleichzeitige Änderungen

Diese Ausnahme tritt auf, wenn eine Auflistung geändert wird, während sie mit anderen Methoden als den vom Iterator-Objekt bereitgestellten Methoden durchlaufen wird. Wir haben zum Beispiel eine Liste von Hüten und möchten alle entfernen, die über Ohrenklappen verfügen:

```
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);
    }
}
```

Wenn wir diesen Code ausführen, wird ***ConcurrentModificationException*** ausgelöst, da der Code die Auflistung ändert, während er durchlaufen wird. Die gleiche Ausnahme kann auftreten, wenn einer der mehreren Threads, die mit derselben Liste arbeiten, versucht, die Auflistung zu ändern, während andere iterieren. Das gleichzeitige Ändern von Sammlungen in mehreren Threads ist eine natürliche Sache, sollte jedoch mit den üblichen Werkzeugen aus der Toolbox für gleichzeitige Programmierung behandelt werden, z. B. Synchronisationssperren, spezielle Sammlungen, die für die gleichzeitige Änderung übernommen wurden, und die geklonte Sammlung von der ursprünglichen Konfiguration abändern.

Untersammlungen

Liste subList (int fromIndex, int bisIndex)

FromIndex ist hier inklusive und toIndex ist exklusiv.

```
List list = new ArrayList();
List list1 = list.subList(fromIndex,toIndex);
```

1. Wenn die Liste nicht im gegebenen Bereich vorhanden ist, wird IndexOutofBoundException ausgelöst.
2. Alle Änderungen, die an list1 vorgenommen wurden, wirken sich auf dieselben Änderungen in der Liste aus. Dies wird als gesicherte Sammlungen bezeichnet.
3. Wenn der fromnIndex größer ist als der toIndex (fromIndex> bisIndex), wird IllegalArgumentException ausgelöst.

Beispiel:

```
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);
```

Ausgabe:
Vor der Unterliste [Hello1, Hello2]
Nach Unterlistenänderungen [Hello1, Hello3, Hello2]

Teilmenge festlegen (fromIndex, toIndex)

FromIndex ist hier inklusive und toIndex ist exklusiv.

```
Set set = new TreeSet();
Set set1 = set.subSet(fromIndex,toIndex);
```

Der zurückgegebene Satz löst eine IllegalArgumentException aus, wenn versucht wird, ein Element außerhalb seines Bereichs einzufügen.

Map subMap (vonKey, toKey)

fromKey ist inklusive und toKey ist exklusiv

```
Map map = new TreeMap();
Map map1 = map.get(fromKey,toKey);
```

Wenn fromKey größer als toKey ist oder wenn diese Map selbst einen eingeschränkten Bereich hat und fromKey oder toKey außerhalb der Grenzen des Bereichs liegt, wird IllegalArgumentException ausgelöst.

Alle Sammlungen unterstützen gesicherte Sammlungen, dh Änderungen, die an der Untersammlung vorgenommen wurden, haben dieselben Änderungen an der Hauptsammlung.

Sammlungen online lesen: https://riptutorial.com/de/java/topic/90/sammlungen

Kapitel 137: Sammlungen auswählen

Einführung

Java bietet eine Vielzahl von Collections an. Die Auswahl der zu verwendenden Sammlung kann schwierig sein. Im Abschnitt mit den Beispielen finden Sie ein einfach zu befolgendes Flussdiagramm, in dem Sie die richtige Sammlung für den Job auswählen.

Examples

Flussdiagramm für Java-Sammlungen

Verwenden Sie das folgende Flussdiagramm, um die richtige Sammlung für den Job auszuwählen.

Dieses Flussdiagramm basiert auf [http://i.stack.imgur.com/aSDsG.png) .

Sammlungen auswählen online lesen: https://riptutorial.com/de/java/topic/10846/sammlungen-auswahlen

Kapitel 138: Scanner

Syntax

- Scanner Scanner = neuer Scanner (Quellquelle);
- Scanner Scanner = neuer Scanner (System.in);

Parameter

Parameter	Einzelheiten
Quelle	Quelle kann entweder String, File oder InputStream sein

Bemerkungen

Die Scanner Klasse wurde in Java 5 eingeführt. Die reset() Methode wurde in Java 6 hinzugefügt. In Java 7 wurden einige neue Konstruktoren hinzugefügt, um die Interoperabilität mit der (dann) neuen Path Schnittstelle zu Path .

Examples

Systemeingabe mit Scanner lesen

```
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);
```

Das Scannerobjekt wird initialisiert, um Eingaben von der Tastatur zu lesen. Für die unten stehende Eingabe von keyboar wird die Ausgabe als Reading from keyboard

```
Reading
from
keyboard
  //space
```

Dateieingabe mit Scanner lesen

```
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();
```

```
}
```

Hier wird ein Scanner Objekt erstellt, indem ein File Objekt mit dem Namen einer Textdatei als Eingabe übergeben wird. Diese Textdatei wird vom File-Objekt geöffnet und in den folgenden Zeilen vom Scanner-Objekt eingelesen. scanner.hasNext() prüft, ob eine nächste scanner.hasNext() in der Textdatei enthalten ist. Wenn Sie dies mit einer while Schleife Names.txt , können Sie jede Names.txt in der Datei Names.txt . Um die Daten selbst abzurufen, können Methoden wie nextLine() , nextInt() , nextBoolean() usw. verwendet werden. Im obigen Beispiel wird scanner.nextLine() verwendet. nextLine() bezieht sich auf die folgende Zeile in einer Textdatei. Durch das Kombinieren mit einem scanner können Sie den Inhalt der Zeile drucken. Um ein .close() zu schließen, verwenden Sie .close() .

Mit try mit Ressourcen (ab Java 7) kann der oben erwähnte Code wie folgt elegant geschrieben werden.

```
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!");
}
```

Lesen Sie die gesamte Eingabe als Zeichenfolge mit dem Scanner

Sie können Scanner , um den gesamten Text in der Eingabe als Scanner zu lesen, indem Sie \Z (gesamte Eingabe) als Trennzeichen verwenden. So können Sie beispielsweise den gesamten Text einer Textdatei in einer Zeile lesen:

```
String content = new Scanner(new File("filename")).useDelimiter("\\Z").next();
System.out.println(content);
```

Denken Sie daran, dass Sie den Scanner schließen und die IoException abfangen müssen, die möglicherweise IoException , wie im Beispiel Lesen der Dateieingabe mit Scanner beschrieben .

Verwenden von benutzerdefinierten Trennzeichen

Sie können benutzerdefinierte Begrenzer (reguläre Ausdrücke) mit Scanner und .useDelimiter(",") , um zu bestimmen, wie die Eingabe gelesen wird. Dies funktioniert ähnlich wie String.split(...) . Beispielsweise können Sie Scanner , um aus einer Liste mit durch Kommas getrennten Werten in einem String zu lesen:

```
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();
}
```

Dadurch können Sie jedes Element in der Eingabe einzeln lesen. Beachten Sie, dass Sie dies **nicht** zum Parsen von CSV-Daten verwenden sollten. Verwenden Sie stattdessen eine geeignete CSV-Parser-Bibliothek. Weitere Möglichkeiten finden Sie unter CSV-Parser für Java .

Allgemeines Muster, das am häufigsten nach Aufgaben gefragt wird

Im Folgenden wird beschrieben, wie Sie die Klasse java.util.Scanner richtig verwenden, um die Benutzereingaben von System.in richtig zu lesen (manchmal als stdin , insbesondere in C, C ++ und anderen Sprachen sowie in Unix und Linux). Es demonstriert idiomatisch die häufigsten Dinge, die gefordert werden.

```
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);
    }
```

```
}
```

Liest ein int von der Kommandozeile

```
import java.util.Scanner;

Scanner s = new Scanner(System.in);
int number = s.nextInt();
```

Wenn Sie ein int aus der Befehlszeile lesen möchten, verwenden Sie einfach dieses Snippet. Zunächst müssen Sie ein Scanner-Objekt erstellen, das auf System.in wartet, das standardmäßig die Befehlszeile ist, wenn Sie das Programm von der Befehlszeile aus starten. Danach lesen Sie mit Hilfe des Scanner-Objekts das erste int, das der Benutzer in die Befehlszeile übergibt, und speichern es in der Variablennummer. Jetzt können Sie mit dem gespeicherten int machen, was Sie wollen.

Scanner vorsichtig schließen

Es kann vorkommen, dass Sie einen Scanner mit dem Parameter "System.in" als Parameter für den Konstruktor verwenden. Dann müssen Sie wissen, dass beim Schließen des Scanners auch der InputStream geschlossen wird. Als Nächstes wird versucht, die Eingabe darauf zu lesen Scannerobjekt) löst eine java.util.NoSuchElementException oder eine java.lang.IllegalStateException

Beispiel:

```
    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();
```

Scanner online lesen: https://riptutorial.com/de/java/topic/551/scanner

Kapitel 139: Schnittstellen

Einführung

Eine *Schnittstelle* ist ein Referenztyp, der einer Klasse ähnelt und mit dem interface deklariert werden kann. Schnittstellen können nur Konstanten, Methodensignaturen, Standardmethoden, statische Methoden und verschachtelte Typen enthalten. Methodentexte sind nur für Standardmethoden und statische Methoden vorhanden. Wie abstrakte Klassen können Schnittstellen nicht instanziiert werden. Sie können nur von Klassen implementiert oder von anderen Schnittstellen erweitert werden. Die Schnittstelle ist eine gängige Methode, um eine vollständige Abstraktion in Java zu erreichen.

Syntax

- öffentliche Schnittstelle Foo {void foo (); / * andere Methoden * /}
- öffentliche Schnittstelle Foo1 erweitert Foo {Leerverbinder (); / * andere Methoden * /}
- öffentliche Klasse Foo2 implementiert Foo, Foo1 {/ * Implementierung von Foo und Foo1 * /}

Examples

Eine Schnittstelle deklarieren und implementieren

Deklaration einer Schnittstelle mit dem interface :

```
public interface Animal {
    String getSound(); // Interface methods are public by default
}
```

Annotation überschreiben

```
@Override
public String getSound() {
    // Code goes here...
}
```

Dies zwingt den Compiler, zu prüfen, ob wir überschreiben, und verhindert, dass das Programm eine neue Methode definiert oder die Methodensignatur durcheinander bringt.

Schnittstellen werden mit dem Schlüsselwort implements implements .

```
public class Cat implements Animal {

    @Override
    public String getSound() {
        return "meow";
    }
}

public class Dog implements Animal {

    @Override
    public String getSound() {
        return "woof";
    }
}
```

Im Beispiel Klassen Cat und Dog **müssen** definieren getSound() Methode als Methoden einer Schnittstelle von Natur aus abstrakt sind (mit Ausnahme von Standardmethoden).

Verwenden der Schnittstellen

```
Animal cat = new Cat();
Animal dog = new Dog();

System.out.println(cat.getSound()); // prints "meow"
System.out.println(dog.getSound()); // prints "woof"
```

Implementierung mehrerer Schnittstellen

Eine Java-Klasse kann mehrere Schnittstellen implementieren.

```
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");
    }
}
```

Beachten Sie, wie die Cat Klasse die geerbten abstract Methoden in beiden Schnittstellen implementieren **muss** . Beachten Sie außerdem, wie eine Klasse praktisch beliebig viele Schnittstellen implementieren kann (aufgrund der JVM-Einschränkung gibt es eine Grenze von **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
```

Hinweis:

1. Alle in einer Schnittstelle deklarierten Variablen sind public static final
2. Alle in einer Schnittstellenmethode deklarierten Methoden sind public abstract (Diese Anweisung ist nur über Java 7 gültig. In Java 8 dürfen Sie Methoden in einer Schnittstelle haben, die nicht abstrakt sein müssen; solche Methoden werden als Standardmethoden bezeichnet.)
3. Schnittstellen können nicht als final deklariert werden
4. Wenn mehr als eine Schnittstelle eine Methode mit identischer Signatur deklariert, wird sie effektiv als nur eine Methode behandelt, und Sie können nicht unterscheiden, welche Schnittstellenmethode implementiert ist
5. Bei der Kompilierung wird für jede Schnittstelle eine entsprechende **InterfaceName.class-**Datei generiert

Eine Schnittstelle erweitern

Eine Schnittstelle kann eine andere Schnittstelle über das Schlüsselwort extend extends .

```
public interface BasicResourceService {
    Resource getResource();
}

public interface ExtendedResourceService extends BasicResourceService {
    void updateResource(Resource resource);
}
```

Jetzt muss eine Klasse, die ExtendedResourceService implementiert, sowohl getResource() als auch updateResource() implementieren.

Mehrere Schnittstellen erweitern

Im Gegensatz zu Klassen, die extends Schlüsselwort kann verwendet werden , um mehrere Schnittstellen zu erweitern (durch Kommas getrennt) so dass für Kombinationen von Schnittstellen in eine neue Schnittstelle

```
public interface BasicResourceService {
    Resource getResource();
}

public interface AlternateResourceService {
    Resource getAlternateResource();
}

public interface ExtendedResourceService extends BasicResourceService,
AlternateResourceService {
    Resource updateResource(Resource resource);
}
```

In diesem Fall muss eine Klasse, die ExtendedResourceService implementiert, getResource() , getAlternateResource() und updateResource() .

Verwenden von Schnittstellen mit Generics

Angenommen, Sie möchten eine Schnittstelle definieren, die das Veröffentlichen / Konsumieren von Daten in und aus verschiedenen Kanaltypen (z. B. AMQP, JMS usw.) ermöglicht. Sie möchten jedoch die Implementierungsdetails ausschalten können.

Definieren wir eine grundlegende E / A-Schnittstelle, die in mehreren Implementierungen wiederverwendet werden kann:

```
public interface IO<IncomingType, OutgoingType> {

    void publish(OutgoingType data);
    IncomingType consume();
    IncomingType RPCSubmit(OutgoingType data);

}
```

Jetzt kann ich diese Schnittstelle instanziieren, aber da es für diese Methoden keine Standardimplementierungen gibt, ist eine Implementierung erforderlich, wenn wir sie instanziieren:

```
    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
```

Wir können auch etwas nützlicheres mit dieser Schnittstelle machen, nehmen wir an, wir wollen es verwenden, um einige grundlegende RabbitMQ-Funktionen einzubinden:

```
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);
    }

}
```

Angenommen, ich möchte diese E / A-Schnittstelle jetzt verwenden, um Besuche auf meiner Website seit dem letzten Neustart des Systems zu zählen und dann die Gesamtzahl der Besuche anzeigen zu können. Sie können beispielsweise Folgendes tun:

```
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);
    }

}
```

Jetzt nutzen wir den 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
```

Wenn Sie mehrere Schnittstellen implementieren, können Sie dieselbe Schnittstelle nicht zweimal implementieren. Das gilt auch für generische Schnittstellen. Daher ist der folgende Code ungültig und führt zu einem Kompilierungsfehler:

```
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); }
}
```

Nützlichkeit von Schnittstellen

Schnittstellen können in vielen Fällen äußerst hilfreich sein. Angenommen, Sie hatten eine Liste von Tieren und wollten die Liste durchlaufen, wobei jeweils der Ton gedruckt wurde.

```
{cat, dog, bird}
```

Eine Möglichkeit, dies zu tun, wäre die Verwendung von Schnittstellen. Dies würde ermöglichen, dass dieselbe Methode für alle Klassen aufgerufen wird

```
public interface Animal {
    public String getSound();
}
```

Jede Klasse, implements Animal muss auch eine getSound() -Methode enthalten. Sie kann jedoch unterschiedliche Implementierungen haben

```
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";
    }
}
```

Wir haben jetzt drei verschiedene Klassen, von denen jede eine getSound() -Methode hat. Da alle diese Klassen implement die Animal - Schnittstelle, die das erklärt getSound() Methode, jede Instanz eines Animal haben kann getSound() auf sie genannt

```
Animal dog = new Dog();
Animal cat = new Cat();
Animal bird = new Bird();

dog.getSound(); // "Woof"
cat.getSound(); // "Meow"
bird.getSound(); // "Chirp"
```

Da jedes Animal ein Animal , könnten wir die Tiere sogar in eine Liste setzen, sie durchlaufen und ihre Geräusche ausdrucken

```
Animal[] animals = { new Dog(), new Cat(), new Bird() };
for (Animal animal : animals) {
    System.out.println(animal.getSound());
}
```

Da die Reihenfolge des Arrays Dog , Cat und Bird lautet, wird *"Woof Meow Chirp"* auf die Konsole gedruckt.

Schnittstellen können auch als Rückgabewert für Funktionen verwendet werden. Zum Beispiel die Rückgabe eines Dog wenn die Eingabe *"Hund" ist* , Cat wenn die Eingabe *"Katze" ist* , und Bird wenn es *"Vogel" ist* , und dann das Geräusch dieses Tieres gedruckt werden könnte

```
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
```

Schnittstellen sind auch für die Erweiterbarkeit hilfreich. Wenn Sie einen neuen Animal Typ hinzufügen möchten, müssen Sie nichts an den Operationen ändern, die Sie für sie ausführen.

Schnittstellen in einer abstrakten Klasse implementieren

Eine in einer interface definierte Methode ist standardmäßig public abstract . Wenn eine abstract class eine interface implementiert, müssen alle Methoden, die in der interface definiert sind, nicht von der abstract class implementiert werden. Dies liegt daran, dass eine class , die als abstract deklariert ist, abstrakte Methodendeklarationen enthalten kann. Es ist daher die Aufgabe der ersten konkreten Unterklasse, irgendwelche abstract Methoden zu implementieren, die von beliebigen Schnittstellen und / oder der abstract class geerbt werden.

```
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.");
    }
}
```

Ab Java 8 ist es möglich, dass eine interface default von Methoden deklariert. default bedeutet, dass die Methode nicht abstract ist. Konkrete Unterklassen müssen die Methode nicht implementieren, erben jedoch die default , sofern sie nicht überschrieben werden.

Standardmethoden

In Java 8 eingeführt, bieten Standardmethoden eine Möglichkeit, eine Implementierung innerhalb einer Schnittstelle anzugeben. Dies könnte verwendet werden, um die typische Klasse "Base" oder "Abstract" zu vermeiden, indem eine teilweise Implementierung einer Schnittstelle bereitgestellt wird und die Hierarchie der Unterklassen eingeschränkt wird.

Beobachtermuster-Implementierung

Beispielsweise ist es möglich, das Observer-Listener-Muster direkt in die Benutzeroberfläche zu implementieren, wodurch die implementierenden Klassen flexibler werden.

```
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);
        }
    }
}
```

Jetzt kann jede Klasse durch das Implementieren der Observable-Schnittstelle "beobachtbar" gemacht werden, während sie frei ist, Teil einer anderen Klassenhierarchie zu sein.

```
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();
    }
}
```

Diamantproblem

Dem Compiler in Java 8 ist das Diamantproblem bekannt, das verursacht wird, wenn eine Klasse Schnittstellen implementiert, die eine Methode mit derselben Signatur enthalten.

Um dies zu lösen, muss eine implementierende Klasse die gemeinsam genutzte Methode überschreiben und ihre eigene Implementierung bereitstellen.

```
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"
    }
}
```

Es gibt immer noch das Problem, Methoden mit demselben Namen und Parametern mit unterschiedlichen Rückgabetypen zu haben, die nicht kompiliert werden können.

Verwenden Sie Standardmethoden, um Kompatibilitätsprobleme zu beheben

Die Standardmethodenimplementierungen sind sehr praktisch, wenn einer Schnittstelle in einem vorhandenen System eine Methode hinzugefügt wird, in der die Schnittstellen von mehreren Klassen verwendet werden.

Um ein Aufbrechen des gesamten Systems zu vermeiden, können Sie eine Standardmethodenimplementierung bereitstellen, wenn Sie einer Schnittstelle eine Methode hinzufügen. Auf diese Weise wird das System immer noch kompiliert und die eigentlichen Implementierungen können Schritt für Schritt durchgeführt werden.

Weitere Informationen finden Sie im Thema Standardmethoden .

Modifikatoren in Schnittstellen

Im Oracle Java Style Guide heißt es:

> Modifikatoren sollten nicht ausgeschrieben werden, wenn sie implizit sind.

(Siehe Modifikatoren in Oracle Official Code Standard für den Kontext und einen Link zum eigentlichen Oracle-Dokument.)

Diese Stilanleitung gilt insbesondere für Schnittstellen. Betrachten wir das folgende Code-Snippet:

```
interface I {
    public static final int VARIABLE = 0;

    public abstract void method();

    public static void staticMethod() { ... }
    public default void defaultMethod() { ... }
}
```

Variablen

Alle Schnittstellenvariablen sind implizit *Konstanten* mit impliziten public (zugänglich für alle), static (sind über Schnittstellennamen zugänglich) und final (müssen während der Deklaration initialisiert werden):

```
~~public static final~~ int VARIABLE = 0;
```

Methoden

1. Alle Methoden, *die keine Implementierung bieten,* sind implizit public und abstract .

```
~~public abstract~~ void method();
```

Java SE 8

2. Alle Methoden mit static oder default *müssen implementiert werden* und sind implizit public .

```
~~public~~ static void staticMethod() { ... }
```

Nachdem alle oben genannten Änderungen angewendet wurden, erhalten Sie Folgendes:

```
interface I {
    int VARIABLE = 0;

    void method();

    static void staticMethod() { ... }
    default void defaultMethod() { ... }
}
```

Verstärken Sie die Parameter des beschränkten Typs

Begrenzte Typparameter ermöglichen das Festlegen von Einschränkungen für generische Typargumente:

```
class SomeClass {

}

class Demo<T extends SomeClass> {

}
```

Ein Typparameter kann jedoch nur an einen einzelnen Klassentyp gebunden werden.

Ein Schnittstellentyp kann an einen Typ gebunden sein, der bereits eine Bindung hatte. Dies wird mit dem Symbol & :

```
interface SomeInterface {

}

class GenericClass<T extends SomeClass & SomeInterface> {

}
```

Dies stärkt die Bindung und erfordert möglicherweise Argumente, die von mehreren Typen abgeleitet werden.

Mehrere Schnittstellentypen können an einen Typparameter gebunden werden:

```
class Demo<T extends SomeClass & FirstInterface & SecondInterface> {

}
```

Sollte aber mit Vorsicht angewendet werden. Mehrere Schnittstellenbindungen sind in der Regel ein Zeichen für einen Codegeruch , was darauf hindeutet, dass ein neuer Typ erstellt werden sollte, der als Adapter für die anderen Typen fungiert:

```
interface NewInterface extends FirstInterface, SecondInterface {

}

class Demo<T extends SomeClass & NewInterface> {

}
```

Schnittstellen online lesen: https://riptutorial.com/de/java/topic/102/schnittstellen

Kapitel 140: Serialisierung

Einführung

Java stellt einen Mechanismus bereit, der Objektserialisierung, bei dem ein Objekt als Folge von Bytes dargestellt werden kann, die die Daten des Objekts sowie Informationen über den Objekttyp und die im Objekt gespeicherten Datentypen enthält.

Nachdem ein serialisiertes Objekt in eine Datei geschrieben wurde, kann es aus der Datei gelesen und deserialisiert werden, d. H. Die Typinformationen und Bytes, die das Objekt und seine Daten darstellen, können verwendet werden, um das Objekt im Speicher wiederherzustellen.

Examples

Grundlegende Serialisierung in Java

Was ist Serialisierung?

Serialisierung ist der Prozess, bei dem der Status eines Objekts (einschließlich seiner Referenzen) in eine Folge von Bytes konvertiert wird, sowie der Prozess, bei dem diese Bytes zu einem späteren Zeitpunkt in ein Live-Objekt umgewandelt werden. Die Serialisierung wird verwendet, wenn Sie das Objekt beibehalten möchten. Es wird auch von Java RMI verwendet, um Objekte zwischen JVMs zu übergeben, entweder als Argumente in einem Methodenaufruf von einem Client an einen Server oder als Rückgabewerte von einem Methodenaufruf oder als von Remotemethoden ausgelöste Ausnahmen. Im Allgemeinen wird die Serialisierung verwendet, wenn das Objekt über die Lebensdauer der JVM hinaus existieren soll.

java.io.Serializable ist eine Markierungsschnittstelle (hat keinen Körper). Es wird nur verwendet, um Java-Klassen als serialisierbar zu "kennzeichnen".

Die Serialisierungslaufzeit serialVersionUID jeder serialisierbaren Klasse eine Versionsnummer zu, die als serialVersionUID bezeichnet wird. Diese wird während der *De-* Serialisierung verwendet, um zu überprüfen, ob der Sender und der Empfänger eines serialisierten Objekts Klassen für dieses Objekt geladen haben, die hinsichtlich der Serialisierung kompatibel sind. Wenn der Empfänger eine Klasse für das Objekt geladen hat, die eine andere serialVersionUID als die Klasse des entsprechenden Senders, führt die Deserialisierung zu einer InvalidClassException . Eine serialisierbare Klasse kann ihre eigene serialVersionUID explizit deklarieren, indem sie ein Feld mit dem Namen serialVersionUID , das static, final, und vom Typ long :

```
ANY-ACCESS-MODIFIER static final long serialVersionUID = 1L;
```

So machen Sie eine Klasse für die Serialisierung geeignet

Um ein Objekt zu persistieren, muss die entsprechende Klasse die Schnittstelle java.io.Serializable implementieren.

```
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;
    }
}
```

Wie schreibe ich ein Objekt in eine Datei?

Nun müssen wir dieses Objekt in ein Dateisystem schreiben. Wir verwenden dazu java.io.ObjectOutputStream .

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

So erstellen Sie ein Objekt aus dem serialisierten Status

Das gespeicherte Objekt kann zu einem späteren Zeitpunkt mit java.io.ObjectInputStream wie java.io.ObjectInputStream aus dem Dateisystem gelesen werden:

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

Die serialisierte Klasse ist in binärer Form. Die Deserialisierung kann problematisch sein, wenn sich die Klassendefinition ändert: Weitere Informationen finden Sie im Kapitel Versionierung von serialisierten Objekten der Java-Serialisierungsspezifikation .

Durch das Serialisieren eines Objekts wird der gesamte Objektgraph, von dem es die Wurzel ist, serialisiert und bei Vorhandensein von zyklischen Graphen korrekt ausgeführt. Eine reset() - Methode wird bereitgestellt, um den ObjectOutputStream zu zwingen, ObjectOutputStream zu vergessen, die bereits serialisiert wurden.

Transientenfelder - Serialisierung

Serialisierung mit Gson

Die Serialisierung mit Gson ist einfach und gibt korrektes JSON aus.

```
public class Employe {

    private String firstName;
    private String lastName;
    private int age;
    private BigDecimal salary;
    private List<String> skills;

    //getters and setters
}
```

(Serialisierung)

```
//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"]}
```

Beachten Sie, dass Sie Objekte mit Zirkelverweisen nicht serialisieren können, da dies zu einer unendlichen Rekursion führt.

(Deserialisierung)

```
//it's very simple...
//Assuming that json is the previous String object....

Employe obj2 = gson.fromJson(json, Employe.class); // obj2 is just like obj
```

Serialisierung mit Jackson 2

Die folgende Implementierung zeigt, wie ein Objekt in die entsprechende JSON-Zeichenfolge serialisiert werden kann.

```
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;
    }
}
```

Serialisierung:

```
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
}
```

Ausgabe:

```
{
  "idx" : 1,
  "name" : "abc"
}
```

Sie können den Standarddrucker auslassen, wenn Sie ihn nicht benötigen.

Die hier verwendete Abhängigkeit ist wie folgt:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.6.3</version>
</dependency>
```

Benutzerdefinierte Serialisierung

In diesem Beispiel möchten wir eine Klasse erstellen, die eine Konsole generiert und ausgibt, eine Zufallszahl zwischen zwei Ganzzahlen, die während der Initialisierung als Argumente übergeben werden.

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

Wenn wir diese Klasse serialisierbar machen wollen, gibt es einige Probleme. Der Thread ist eine der bestimmten Klassen auf Systemebene, die nicht serialisierbar sind. Also müssen wir den Thread als **transient** erklären. Auf diese Weise können wir die Objekte dieser Klasse serialisieren, haben jedoch noch ein Problem. Wie Sie im Konstruktor sehen können, legen wir die Min- und Max-Werte unseres Randomizers fest. Danach starten wir den Thread, der für das Generieren und Drucken des Zufallswerts verantwortlich ist. Beim Wiederherstellen des persistenten Objekts durch Aufrufen von ***readObject () wird*** der Konstruktor daher nicht erneut ausgeführt, da kein neues Objekt erstellt wird. In diesem Fall müssen wir eine **benutzerdefinierte Serialisierung entwickeln**, indem wir zwei Methoden innerhalb der Klasse bereitstellen. Diese Methoden sind:

```
private void writeObject(ObjectOutputStream out) throws IOException;
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

Indem wir unsere Implementierung in ***readObject ()*** hinzufügen, können wir unseren Thread initiieren und starten:

```
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();
}
}
```

Hier ist das Wichtigste für unser Beispiel:

```
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();
            }

}
}
```

Wenn Sie main ausführen, werden Sie sehen, dass für jede RangeRandom-Instanz zwei Threads ausgeführt werden. Dies liegt daran, dass sich die ***Thread.start () -*** Methode jetzt sowohl im Konstruktor als auch in ***readObject () befindet*** .

Versionierung und serialVersionUID

Wenn Sie die Schnittstelle java.io.Serializable implementieren, um eine Klasse serialisierbar zu machen, sucht der Compiler nach einem static final serialVersionUID namens serialVersionUID vom Typ long . Wenn für die Klasse dieses Feld nicht explizit deklariert ist, erstellt der Compiler ein solches Feld und weist ihm einen Wert zu, der aus einer implementierungsabhängigen Berechnung von serialVersionUID . Diese Berechnung hängt von verschiedenen Aspekten der Klasse ab und folgt den von Sun angegebenen Objekt-Serialisierungsspezifikationen . Es ist jedoch nicht garantiert, dass der Wert bei allen Compiler-Implementierungen gleich ist.

Dieser Wert wird zur Überprüfung der Kompatibilität der Klassen in Bezug auf die Serialisierung verwendet. Dies erfolgt während der Deserialisierung eines gespeicherten Objekts. Die Serialisierungslaufzeit stellt serialVersionUID dass die aus den serialVersionUID gelesene serialVersionUID und die in der Klasse deklarierte serialVersionUID identisch sind. Ist dies nicht der Fall, wird eine InvalidClassException .

Es wird dringend empfohlen, das statische, letzte Feld des Typs long und den Namen 'serialVersionUID' in allen Klassen, in denen Sie Serializable machen möchten, explizit zu deklarieren und zu initialisieren, anstatt sich auf die Standardberechnung des Werts für dieses Feld zu verlassen, auch wenn Sie dies nicht tun Versionierung verwenden. **Die 'serialVersionUID'-Berechnung ist äußerst empfindlich und kann von einer Compilerimplementierung zur anderen variieren. Daher kann es InvalidClassException , dass Sie die InvalidClassException auch für dieselbe Klasse InvalidClassException , nur weil Sie verschiedene Compilerimplementierungen auf der Sender- und Empfängerseite des Serialisierungsprozesses verwendet haben.**

```
public class Example implements Serializable {
    static final long serialVersionUID = 1L /*or some other value*/;
    //...
}
```

Solange serialVersionUID ist, kann Java Serialization verschiedene Versionen einer Klasse verarbeiten. Kompatible und inkompatible Änderungen sind:

Kompatible Änderungen

- **Felder hinzufügen:** Wenn die wiederherzustellende Klasse ein Feld enthält, das im Stream nicht vorkommt, wird dieses Feld im Objekt mit dem Standardwert für seinen Typ initialisiert. Wenn eine klassenspezifische Initialisierung erforderlich ist, kann die Klasse eine readObject-Methode bereitstellen, mit der das Feld auf nicht standardmäßige Werte initialisiert werden kann.
- **Klassen hinzufügen:** Der Stream enthält die Typhierarchie jedes Objekts im Stream. Beim Vergleich dieser Hierarchie im Stream mit der aktuellen Klasse können zusätzliche Klassen erkannt werden. Da es im Stream keine Informationen gibt, von denen aus das Objekt initialisiert werden kann, werden die Felder der Klasse mit den Standardwerten initialisiert.
- **Entfernen von Klassen:** Wenn Sie die Klassenhierarchie im Stream mit der der aktuellen Klasse vergleichen, kann dies feststellen, dass eine Klasse gelöscht wurde. In diesem Fall werden die dieser Klasse entsprechenden Felder und Objekte aus dem Stream gelesen. Primitive Felder werden verworfen, aber die von der gelöschten Klasse referenzierten Objekte werden erstellt, da auf sie später im Stream verwiesen werden kann. Sie werden Müll

gesammelt, wenn der Stream Müll gesammelt oder zurückgesetzt wird.

- **WriteObject / readObject-Methoden hinzufügen:** Wenn die Version, die den Stream liest, über diese Methoden verfügt, wird erwartet, dass readObject wie üblich die erforderlichen Daten liest, die von der Standardserialisierung in den Stream geschrieben werden. Es sollte defaultReadObject zuerst aufrufen, bevor optionale Daten gelesen werden. Es wird erwartet, dass die writeObject-Methode wie üblich defaultWriteObject zum Schreiben der erforderlichen Daten aufruft und möglicherweise optionale Daten schreibt.
- **Hinzufügen von java.io.Serializable:** Dies entspricht dem Hinzufügen von Typen. Es gibt keine Werte im Stream für diese Klasse, daher werden die Felder mit den Standardwerten initialisiert. Die Unterstützung für Unterklassen nicht serialisierbarer Klassen erfordert, dass der Supertyp der Klasse über einen No-Arg-Konstruktor verfügt und die Klasse selbst mit Standardwerten initialisiert wird. Wenn der no-arg-Konstruktor nicht verfügbar ist, wird die InvalidClassException ausgelöst.
- **Ändern des Zugriffs auf ein Feld:** Die Zugriffsmodifizierer public, package, protected und private haben keinen Einfluss auf die Möglichkeit der Serialisierung, den Feldern Werte zuzuweisen.
- **Ändern eines Felds von statisch in nicht statisch oder vorübergehend in nicht transient:** Wenn Sie sich bei der Berechnung der serialisierbaren Felder auf die Standardserialisierung verlassen, entspricht diese Änderung dem Hinzufügen eines Felds zur Klasse. Das neue Feld wird in den Stream geschrieben, aber frühere Klassen ignorieren den Wert, da die Serialisierung statischen oder transienten Feldern keine Werte zuweist.

Inkompatible Änderungen

- **Löschen von Feldern:** Wenn ein Feld in einer Klasse gelöscht wird, enthält der geschriebene Datenstrom seinen Wert nicht. Wenn der Stream von einer früheren Klasse gelesen wird, wird der Wert des Felds auf den Standardwert gesetzt, da im Stream kein Wert verfügbar ist. Dieser Standardwert kann jedoch die Fähigkeit der früheren Version zur Erfüllung ihres Vertrags beeinträchtigen.
- **Klassen in der Hierarchie nach oben oder unten verschieben:** Dies ist nicht zulässig, da die Daten im Stream in der falschen Reihenfolge angezeigt werden.
- **Ändern eines nicht statischen Felds in ein statisches Feld oder ein nicht-transientes Feld in einen vorübergehenden Zustand:** Wenn Sie sich auf die Standardserialisierung verlassen, entspricht diese Änderung dem Löschen eines Felds aus der Klasse. Diese Version der Klasse schreibt diese Daten nicht in den Stream, sodass sie nicht von früheren Versionen der Klasse gelesen werden können. Wie beim Löschen eines Felds wird das Feld der früheren Version mit dem Standardwert initialisiert, was dazu führen kann, dass die Klasse auf unerwartete Weise versagt.
- **Ändern des deklarierten Typs eines primitiven Felds:** Jede Version der Klasse schreibt die Daten mit ihrem deklarierten Typ. Frühere Versionen der Klasse, die versuchen, das Feld zu lesen, schlagen fehl, da der Datentyp im Stream nicht mit dem Typ des Felds übereinstimmt.
- Ändern der writeObject- oder readObject-Methode, sodass die Standardfelddaten nicht mehr geschrieben oder gelesen werden, oder geändert, sodass versucht wird, sie zu schreiben oder zu lesen, wenn die vorherige Version dies nicht tat. Die Standardfelddaten müssen entweder dauerhaft im Stream angezeigt werden oder nicht.
- Das Ändern einer Klasse von Serializable in Externalizable oder umgekehrt ist eine inkompatible Änderung, da der Stream Daten enthält, die mit der Implementierung der verfügbaren Klasse nicht kompatibel sind.
- Ändern einer Klasse von einem Nicht-Enummentyp in einen Enummentyp oder umgekehrt, da der Stream Daten enthält, die mit der Implementierung der verfügbaren Klasse nicht kompatibel sind.
- Das Entfernen von Serializable oder Externalizable ist eine inkompatible Änderung, da beim Schreiben nicht mehr die Felder bereitgestellt werden, die von älteren Versionen der Klasse benötigt werden.
- Das Hinzufügen der writeReplace- oder readResolve-Methode zu einer Klasse ist nicht kompatibel, wenn das Verhalten ein Objekt erzeugen würde, das mit einer älteren Version der Klasse nicht kompatibel ist.

Benutzerdefinierte JSON-Deserialisierung mit Jackson

Wir verwenden Rest API als JSON-Format und entpacken es dann in ein POJO. Jacksons org.codehaus.jackson.map.ObjectMapper funktioniert einfach "out of the box" und wir machen in

den meisten Fällen wirklich nichts. Manchmal benötigen wir jedoch einen benutzerdefinierten Deserializer, um unsere benutzerdefinierten Anforderungen zu erfüllen. In diesem Lernprogramm werden Sie durch den Prozess des Erstellens Ihres eigenen benutzerdefinierten Deserializers geführt.

Nehmen wir an, wir haben folgende Entitäten.

```
public class User {
    private Long id;
    private String name;
    private String email;

    //getter setter are omitted for clarity
}
```

Und

```
public class Program {
    private Long id;
    private String name;
    private User createdBy;
    private String contents;

    //getter setter are omitted for clarity
}
```

Lassen Sie uns zuerst ein Objekt serialisieren / marshallen.

```
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 (Programm); System.out.println (json);
```

Der obige Code erzeugt folgende JSON-

```
{
    "id": 1,
    "name": "Program @# 1",
    "createdBy": {
        "id": 1,
        "name": "Bazlur Rahman",
        "email": "example@example.com"
    },
    "contents": "Some contents"
}
```

Jetzt kann man ganz leicht das Gegenteil tun. Wenn wir über diese JSON verfügen, können Sie die Objektgenerierung mithilfe von ObjectMapper wie folgt aufheben:

Nehmen wir an, dies ist nicht der Fall. Wir haben eine andere JSON als eine API, die nicht zu unserer Program passt.

```
{
"id": 1,
"name": "Program @# 1",
"ownerId": 1
"contents": "Some contents"
}
```

Schauen Sie sich die JSON-Zeichenfolge an, wie Sie sehen, sie hat ein anderes Feld, das owenerId ist.

Wenn Sie diesen JSON wie zuvor serialisieren möchten, haben Sie Ausnahmen.

Es gibt zwei Möglichkeiten, Ausnahmen zu vermeiden und diese serialisiert zu haben:

Ignoriere die unbekannten Felder

Ignoriere die onwerId . Fügen Sie die folgende Anmerkung in der Programmklasse hinzu

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Program {}
```

Schreiben Sie benutzerdefinierte Deserializer

Es gibt jedoch Fälle, in denen Sie dieses owerId Feld tatsächlich benötigen. Angenommen, Sie möchten es als ID der User Klasse angeben.

In diesem Fall müssen Sie einen benutzerdefinierten Deserialisierer schreiben.

Wie Sie sehen, müssen Sie zunächst vom JsonNode aus auf den JonsParser . Dann können JsonNode mit der Methode get() problemlos Informationen aus einem JsonNode extrahieren. und Sie müssen sich um den Feldnamen kümmern. Es sollte der genaue Name sein. Rechtschreibfehler verursachen Ausnahmen.

Und schließlich müssen Sie Ihren ProgramDeserializer beim ObjectMapper registrieren.

```
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);
```

Alternativ können Sie Annotation verwenden, um den Deserializer direkt zu registrieren.

```
@JsonDeserialize(using = ProgramDeserializer.class)
public class Program {
}
```

Serialisierung online lesen: https://riptutorial.com/de/java/topic/767/serialisierung

Kapitel 141: ServiceLoader

Bemerkungen

ServiceLoader können Instanzen von Klassen abgerufen werden, die einen bestimmten Typ (= Dienst) erweitern und in einer in einer .jar Datei gepackten Datei angegeben sind. Der Dienst, der erweitert / implementiert wird, ist häufig eine Schnittstelle, dies ist jedoch nicht erforderlich.

Die Erweiterungs- / Implementierungsklassen müssen einen Null-Argument-Konstruktor für den ServiceLoader bereitstellen, um sie zu instanziieren.

Um vom ServiceLoader eine Textdatei mit dem Namen des vollständig qualifizierten Typnamens des implementierten META-INF/services Verzeichnis META-INF/services in der JAR-Datei gespeichert werden. Diese Datei enthält einen vollständig qualifizierten Namen einer Klasse, die den Dienst pro Zeile implementiert.

Examples

Logger Service

Das folgende Beispiel zeigt, wie eine Klasse für die Protokollierung über den ServiceLoader instanziiert wird.

Bedienung

```
package servicetest;

import java.io.IOException;

public interface Logger extends AutoCloseable {

    void log(String message) throws IOException;
}
```

Implementierungen des Dienstes

Die folgende Implementierung schreibt die Nachricht einfach in 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() {
    }

}
```

Die folgende Implementierung schreibt die Nachrichten in eine Textdatei:

```
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

Die Datei META-INF/services/servicetest.Logger listet die Namen der Logger Implementierungen auf.

```
servicetest.logger.ConsoleLogger
servicetest.logger.FileLogger
```

Verwendungszweck

Die folgende main Methode schreibt eine Nachricht an alle verfügbaren Logger. Die Logger werden mit ServiceLoader instanziiert.

```
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);
        }
    }
}
```

Einfaches ServiceLoader-Beispiel

Der ServiceLoader ist ein einfacher und benutzerfreundlicher integrierter Mechanismus zum dynamischen Laden von Schnittstellenimplementierungen. Mit dem Service Loader, der Mittel zur Instantiierung bereitstellt (nicht jedoch die Verdrahtung), kann in Java SE ein einfacher Abhängigkeitseinspritzungsmechanismus erstellt werden. Mit der ServiceLoader-Schnittstelle und -Implementierung wird die Trennung zur Selbstverständlichkeit und die Programme können bequem erweitert werden. Tatsächlich sind viele Java-APIs auf Basis des ServiceLoader implementiert

Die grundlegenden Konzepte sind

- Betrieb an *Schnittstellen* von Diensten
- Abrufen der Implementierung (en) des Dienstes über ServiceLoader
- Bereitstellung von Diensten

Beginnen accounting-api.jar mit der Schnittstelle und legen Sie sie in ein jar, das zum Beispiel accounting-api.jar

```
package example;

public interface AccountingService {

  long getBalance();
}
```

Jetzt bieten wir eine Implementierung dieses Dienstes in einem Jar mit dem Namen accounting-impl.jar , der eine Implementierung des Dienstes enthält

```
package example.impl;
import example.AccountingService;

public interface DefaultAccountingService implements AccouningService {

  public long getBalance() {
    return balanceFromDB();
  }

  private long balanceFromDB(){
    ...
  }
}
```

accounting-impl.jar enthält die accounting-impl.jar eine Datei, die accounting-impl.jar , dass diese Jar-Datei eine Implementierung von AccountingService bereitstellt. Die Datei muss einen Pfad haben, der mit META-INF/services/ beginnt, und muss denselben Namen wie der *vollständig qualifizierte* Name der Schnittstelle haben:

- META-INF/services/example.AccountingService

Der Inhalt der Datei ist der *vollständig qualifizierte* Name der Implementierung:

```
example.impl.DefaultAccountingService
```

Da sich beide Gläser im Klassenpfad des Programms befinden, das den AccountingService , kann eine Instanz des Diensts mithilfe des ServiceLauncher abgerufen werden

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
AccountingService service = loader.next();
long balance = service.getBalance();
```

Da die ServiceLoader eine ist Iterable unterstützt es mehrere Implementierungsanbieter, wobei das Programm von wählen:

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
for(AccountingService service : loader) {
   //...
}
```

Beachten Sie, dass beim Aufruf von next() eine neue Instanz erstellt wird. Wenn Sie eine Instanz wiederverwenden möchten, müssen Sie die iterator() Methode des ServiceLoader oder die for-each-Schleife wie oben gezeigt verwenden.

ServiceLoader online lesen: https://riptutorial.com/de/java/topic/5433/serviceloader

Kapitel 142: Sets

Examples

Ein HashSet mit Werten deklarieren

Sie können eine neue Klasse erstellen, die von HashSet erbt:

```
Set<String> h = new HashSet<String>() {{
    add("a");
    add("b");
}};
```

Einzeilige Lösung:

```
Set<String> h = new HashSet<String>(Arrays.asList("a", "b"));
```

Guave verwenden:

```
Sets.newHashSet("a", "b", "c")
```

Streams verwenden:

```
Set<String> set3 = Stream.of("a", "b", "c").collect(toSet());
```

Typen und Verwendung von Sets

Im Allgemeinen sind Sets eine Art Sammlung, in der eindeutige Werte gespeichert werden. Die Eindeutigkeit wird durch die Methoden equals() und hashCode() .

Die Sortierung wird durch den Satztyp bestimmt.

HashSet - Zufällige Sortierung

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 - LinkedHashSet

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 - Mit compareTo() oder 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"]
```

Initialisierung

Ein Set ist eine Collection, die keine doppelten Elemente enthalten kann. Es modelliert die mathematische Mengenabstraktion.

Set hat seine Implementierung in verschiedenen Klassen wie HashSet , TreeSet , LinkedHashSet .

Zum Beispiel:

HashSet:

```
Set<T> set = new HashSet<T>();
```

Hier kann T String , Integer oder ein beliebiges anderes **Objekt sein** . **HashSet** ermöglicht eine schnelle Suche nach O (1), sortiert jedoch nicht die hinzugefügten Daten und verliert die Einfügereihenfolge der Elemente.

TreeSet:

Es speichert die Daten in einer sortierten Weise, wobei für grundlegende Operationen, die O (lg (n)) benötigen, eine gewisse Geschwindigkeit geopfert wird. Die Einfügereihenfolge der Artikel wird nicht beibehalten.

```
TreeSet<T> sortedSet = new TreeSet<T>();
```

LinkedHashSet:

Es ist eine verknüpfte HashSet von HashSet Once, bei der die Elemente in der Reihenfolge, in der sie hinzugefügt wurden, HashSet . Der Inhalt wird nicht sortiert. O (1) grundlegende Operationen werden bereitgestellt, jedoch sind für das HashSet der unterstützenden verknüpften Liste höhere Kosten als für HashSet .

```
LinkedHashSet<T> linkedhashset = new LinkedHashSet<T>();
```

Grundlagen des Sets

Was ist ein Set?

Ein Satz ist eine Datenstruktur, die einen Satz von Elementen mit der wichtigen Eigenschaft enthält, dass keine zwei Elemente im Satz gleich sind.

Arten von Set:

1. **HashSet:** Ein Satz, der durch eine Hashtabelle (tatsächlich eine HashMap-Instanz) gesichert wird.
2. **Linked HashSet:** Ein durch Hash-Tabelle und verknüpfte Liste gesichertes Set mit einer vorhersagbaren Iterationsreihenfolge
3. **TreeSet:** Eine NavigableSet-Implementierung basierend auf einer TreeMap.

Set erstellen

```
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
```

Elemente zu einem Set hinzufügen

Elemente können mit der add() -Methode zu einem Set hinzugefügt werden

```
 set.add(12); //  - Adds element 12 to the set
 set.add(13); //  - Adds element 13 to the set
```

Unser Set nach Ausführung dieser Methode:

```
set = [12,13]
```

Löschen Sie alle Elemente eines Sets

```
set.clear();  //Removes all objects from the collection.
```

Nach diesem Set wird:

```
set = []
```

Prüfen Sie, ob ein Element Teil des Sets ist

Die Existenz eines Elements in der Gruppe kann mit der Methode contains() überprüft werden

```
set.contains(0);  //Returns true if a specified object is an element within the set.
```

Ausgabe: False

Prüfen Sie, ob ein Set leer ist

isEmpty() Methode isEmpty() kann verwendet werden, um zu überprüfen, ob ein Set leer ist.

```
set.isEmpty();  //Returns true if the set has no elements
```

Ausgabe: True

Entfernen Sie ein Element aus dem Set

```
 set.remove(0); // Removes first occurrence of a specified object from the collection
```

Überprüfen Sie die Größe des Sets

```
set.size(); //Returns the number of elements in the collection
```

Ausgabe: 0

Erstellen Sie eine Liste aus einem vorhandenen Set

Verwenden einer neuen Liste

```
List<String> list = new ArrayList<String>(listOfElements);
```

Verwenden der List.addAll () -Methode

```
    Set<String> set = new HashSet<String>();
    set.add("foo");
    set.add("boo");

    List<String> list = new ArrayList<String>();
    list.addAll(set);
```

Java 8 Steam API verwenden

```
List<String> list = set.stream().collect(Collectors.toList());
```

Duplikate mit Set beseitigen

Angenommen , Sie eine Sammlung haben elements , und Sie wollen eine weitere Sammlung erstellen , um die gleichen Elemente enthalten , aber mit allen **Duplikate eliminiert**:

```
Collection<Type> noDuplicates = new HashSet<Type>(elements);
```

Beispiel :

```
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);
```

Ausgabe :

```
noDuplicates = [Marco, Emily, John, Jenny]
```

Sets online lesen: https://riptutorial.com/de/java/topic/3102/sets

Kapitel 143: Sichere Objekte

Syntax

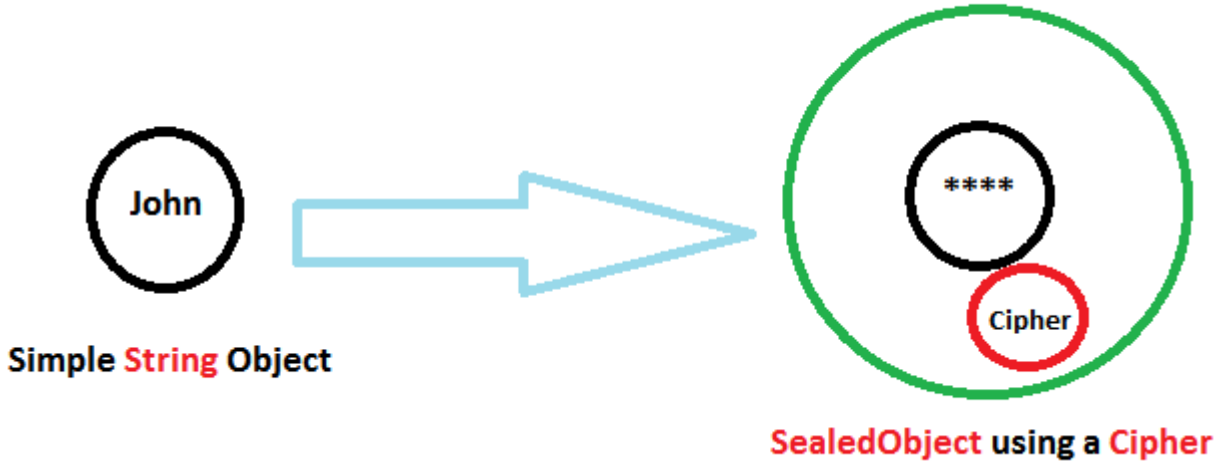
- SealedObject sealedObject = neues SealedObject (obj, cipher);
- SignedObject signedObject = new SignedObject (obj, signingKey, signingEngine);

Examples

SealedObject (javax.crypto.SealedObject)

Mit dieser Klasse kann ein Programmierer ein Objekt erstellen und seine Vertraulichkeit mit einem kryptographischen Algorithmus schützen.

Bei jedem serialisierbaren Objekt kann ein **SealedObject erstellt werden** , das das ursprüngliche Objekt in serialisiertem Format (z. B. eine "tiefe Kopie") einkapselt und seinen serialisierten Inhalt unter Verwendung eines kryptographischen Algorithmus wie AES, DES zum **Verschließen** (verschlüsselt) seine Vertraulichkeit. Der verschlüsselte Inhalt kann später entschlüsselt werden (mit dem entsprechenden Algorithmus unter Verwendung des richtigen Entschlüsselungsschlüssels) und deserialisiert werden, wodurch das ursprüngliche Objekt entsteht.

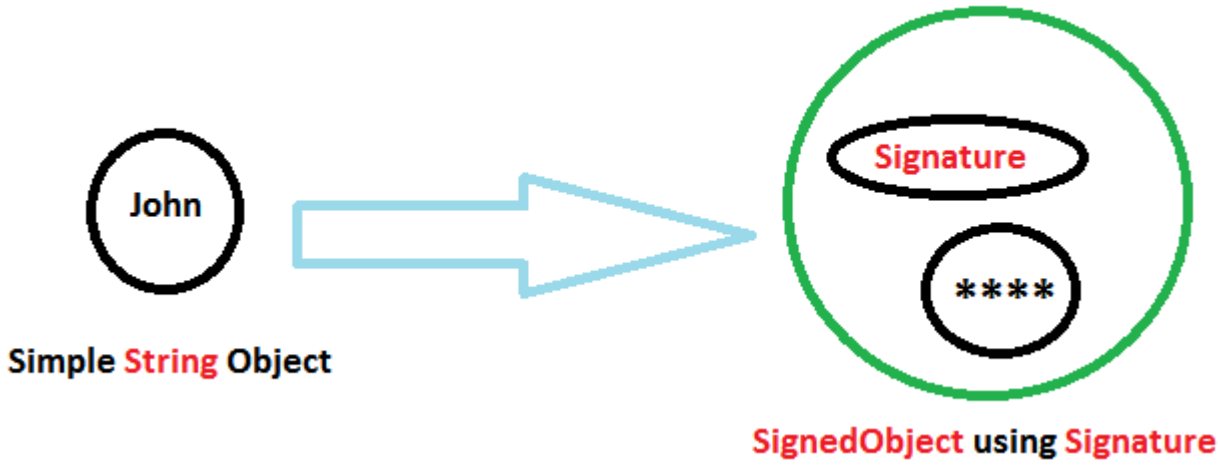



```
Serializable obj = new String("John");
// Generate key
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128);
SecretKey aesKey = kgen.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, aesKey);
SealedObject sealedObject = new SealedObject(obj, cipher);
System.out.println("sealedObject-" + sealedObject);
System.out.println("sealedObject Data-" + sealedObject.getObject(aesKey));
```

SignedObject (java.security.SignedObject)

SignedObject ist eine Klasse zum Erstellen von authentischen Laufzeitobjekten, deren Integrität nicht beeinträchtigt werden kann, ohne entdeckt zu werden.

Insbesondere enthält ein SignedObject ein anderes serialisierbares Objekt, das (zu) signierte Objekt und seine Signatur.




```
//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());
```

Sichere Objekte online lesen: https://riptutorial.com/de/java/topic/5528/sichere-objekte

Kapitel 144: Sicherheit & Kryptographie

Examples

Kryptographische Hashes berechnen

Um die Hashwerte relativ kleiner Datenblöcke mit verschiedenen Algorithmen zu berechnen:

```
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)));
```

Produziert diese Ausgabe:

```
MD5    hash: E99E768582F6DD5A3BA2D9C849DF736E
SHA1   hash: 0135FAA6323685BA8A8FF8D3F955F0C36949D8FB
SHA256 hash: 8D35C97BCD902B96D1B551741BBE8A7F50BB5A690B4D0225482EAA63DBFB9DED
```

Abhängig von Ihrer Implementierung der Java-Plattform sind möglicherweise weitere Algorithmen verfügbar.

Kryptografisch zufällige Daten erzeugen

So erstellen Sie Stichproben kryptographisch zufälliger Daten:

```
final byte[] sample = new byte[16];

new SecureRandom().nextBytes(sample);

System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

Erzeugt eine Ausgabe ähnlich wie:

```
Sample: E4F14CEA2384F70B706B53A6DF8C5EFE
```

Beachten Sie, dass der Aufruf von nextBytes() blockieren kann, während Entropie gesammelt wird, abhängig vom verwendeten Algorithmus.

So legen Sie den Algorithmus und den Anbieter fest:

```
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));
```

Erzeugt eine Ausgabe ähnlich wie:

```
Provider: SUN version 1.8
Algorithm: SHA1PRNG
Sample: C80C44BAEB352FD29FBBE20489E4C0B9
```

Generieren Sie öffentliche / private Schlüsselpaare

So erstellen Sie Schlüsselpaare mit unterschiedlichen Algorithmen und Schlüsselgrößen:

```
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();
```

Zusätzliche Algorithmen und Schlüsselgrößen sind möglicherweise in Ihrer Implementierung der Java-Plattform verfügbar.

So legen Sie eine Zufallsquelle fest, die beim Generieren der Schlüssel verwendet werden soll:

```
final KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");

generator.initialize(2048, SecureRandom.getInstance("SHA1PRNG", "SUN"));

final KeyPair pair = generator.generateKeyPair();
```

Digitale Signaturen berechnen und überprüfen

So berechnen Sie eine Signatur:

```
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();
```

Beachten Sie, dass der Signaturalgorithmus mit dem zur Generierung des Schlüsselpaares verwendeten Algorithmus kompatibel sein muss.

So überprüfen Sie eine Signatur:

```
final PublicKey publicKey = keyPair.getPublic();
final Signature verifier = Signature.getInstance("SHA1withRSA");

verifier.initVerify(publicKey);
verifier.update(data);

System.out.println("Signature: " + verifier.verify(signature));
```

Produziert diese Ausgabe:

```
Signature: true
```

Verschlüsseln und Entschlüsseln von Daten mit öffentlichen / privaten Schlüsseln

So verschlüsseln Sie Daten mit einem öffentlichen Schlüssel:

```
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));
```

Erzeugt eine Ausgabe ähnlich wie:

```
Message: Hello
Encrypted: 5641FBB9558ECFA9ED...
```

Beachten Sie, dass Sie beim Erstellen des Cipher Objekts eine Transformation angeben müssen, die mit dem verwendeten Schlüsseltyp kompatibel ist. (Eine Liste der unterstützten Transformationen finden Sie unter Namen der JCA-Standardalgorithmen .) Bei RSA-Verschlüsselungsdaten muss die Länge von message.getBytes() kleiner als die Schlüsselgröße sein. Siehe diese SO-Antwort für Details.

So entschlüsseln Sie die Daten:

```
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);
```

Erzeugt die folgende Ausgabe:

```
Decrypted: Hello
```

Sicherheit & Kryptographie online lesen: https://riptutorial.com/de/java/topic/7529/sicherheit--amp--kryptographie

Kapitel 145: Sicherheit & Kryptographie

Einführung

Sicherheitspraktiken in Java können in zwei große, vage definierte Kategorien unterteilt werden. Java-Plattformsicherheit und sichere Java-Programmierung.

Die Sicherheitspraktiken der Java-Plattform betreffen die Verwaltung der Sicherheit und Integrität der JVM. Dazu gehören Themen wie das Verwalten von JCE-Providern und Sicherheitsrichtlinien.

Sichere Java-Programmierverfahren beziehen sich auf die besten Möglichkeiten, sichere Java-Programme zu schreiben. Dazu gehören Themen wie die Verwendung von Zufallszahlen und Kryptografie sowie die Vermeidung von Sicherheitslücken.

Bemerkungen

Während Beispiele klar gemacht werden sollten, sind einige Themen, die behandelt werden müssen:

1. Das Konzept / die Struktur des JCE-Anbieters
2. Listenpunkt

Examples

Die JCE

Die Java Cryptography Extension (JCE) ist ein in die JVM integriertes Framework, mit dem Entwickler Kryptografie in ihren Programmen einfach und sicher verwenden können. Dies geschieht durch die Bereitstellung einer einfachen, tragbaren Schnittstelle für Programmierer, während ein System von JCE-Providern verwendet wird, um die zugrunde liegenden kryptographischen Operationen sicher zu implementieren.

Schlüssel und Schlüsselverwaltung

Während das JCE kryptografische Operationen und die Schlüsselgenerierung sichert, ist es Sache des Entwicklers, die Schlüssel tatsächlich zu verwalten. Weitere Informationen müssen hier bereitgestellt werden.

Eine allgemein anerkannte bewährte Methode für die Handhabung von Schlüsseln zur Laufzeit besteht darin, sie nur als byte Arrays und niemals als Zeichenfolgen zu speichern. Dies liegt daran, dass Java-Zeichenfolgen nicht veränderbar sind und nicht manuell im Speicher gelöscht oder gelöscht werden können. Während ein Verweis auf eine Zeichenfolge entfernt werden kann, bleibt die genaue Zeichenfolge im Speicher, bis der Speicherbereich des Speichers gesammelt und erneut verwendet wird. Ein Angreifer verfügt über ein großes Fenster, in dem er den Speicher des Programms ablegen und den Schlüssel leicht finden kann. Im Gegensatz dazu sind byte Arrays veränderbar, und ihr Inhalt kann überschrieben werden. Es ist eine gute Idee, Ihre Schlüssel zu löschen, sobald Sie sie nicht mehr benötigen.

Häufige Java-Schwachstellen

Benötigt Inhalt

Bedenken hinsichtlich der Vernetzung

Benötigt Inhalt

Zufälligkeit und Du

Benötigt Inhalt

Für die meisten Anwendungen ist die Klasse java.utils.Random eine perfekte Quelle für "zufällige" Daten. Wenn Sie ein zufälliges Element aus einem Array auswählen oder eine zufällige Zeichenfolge generieren oder einen temporären "eindeutigen" Bezeichner erstellen müssen, sollten Sie wahrscheinlich Random .

Viele kryptografische Systeme sind jedoch für ihre Sicherheit auf Zufälligkeit angewiesen, und die von Random bereitgestellte Random einfach nicht hoch genug. Für jede kryptografische Operation, die eine zufällige Eingabe erfordert, sollten SecureRandom stattdessen SecureRandom verwenden.

Hashing und Validierung

Weitere Informationen werden benötigt.

Eine kryptografische Hashfunktion ist ein Mitglied einer Klasse von Funktionen mit drei wesentlichen Eigenschaften. Konsistenz, Einzigartigkeit und Irreversibilität.

Konsistenz: Bei gleichen Daten liefert eine Hash-Funktion immer den gleichen Wert. Das heißt, wenn X = Y ist, ist f (x) für die Hashfunktion f immer gleich f (y).

Eindeutigkeit: Keine zwei Eingaben für eine Hash-Funktion führen zu derselben Ausgabe. Das heißt, falls X! = Y, f (x)! = F (y) für beliebige Werte von X und Y.

Irreversibilität: Es ist unpraktisch schwierig, wenn nicht unmöglich, eine Hash-Funktion "umzukehren". Das heißt, wenn nur f (X) gegeben ist, sollte es nicht möglich sein, das ursprüngliche X zu finden, bevor jeder mögliche Wert von X durch die Funktion f (rohe Kraft) gesetzt wird. Es sollte keine Funktion f1 geben, so dass f1 (f (X)) = X ist.

Vielen Funktionen fehlt mindestens eines dieser Attribute. Zum Beispiel ist bekannt, dass MD5 und SHA1 Kollisionen haben, dh zwei Eingänge, die den gleichen Ausgang haben, so dass ihnen die Eindeutigkeit fehlt. Einige Funktionen, die derzeit als sicher gelten, sind SHA-256 und SHA-512.

Sicherheit & Kryptographie online lesen: https://riptutorial.com/de/java/topic/9371/sicherheit--amp--kryptographie

Kapitel 146: Sicherheitsmanager

Examples

Aktivieren des SecurityManagers

Java Virtual Machines (JVMs) können mit einem installierten SecurityManager ausgeführt werden. Der SecurityManager legt fest, welche Aufgaben der in der JVM ausgeführte Code ausführen darf. Dies hängt von Faktoren ab, von denen der Code geladen wurde und aus welchen Zertifikaten der Code signiert wurde.

Der SecurityManager kann installiert werden, indem beim Starten der JVM die Systemeigenschaft java.security.manager in der Befehlszeile festgelegt wird:

```
java -Djava.security.manager <main class name>
```

oder programmgesteuert aus Java-Code heraus:

```
System.setSecurityManager(new SecurityManager())
```

Der Standard-Java SecurityManager erteilt Berechtigungen auf der Grundlage einer Richtlinie, die in einer Richtliniendatei definiert ist. Wenn keine Richtliniendatei angegeben ist, wird die Standardrichtliniendatei unter $JAVA_HOME/lib/security/java.policy verwendet.

Sandboxing-Klassen, die von einem ClassLoader geladen werden

Der ClassLoader muss eine ProtectionDomain bereitstellen, die die Quelle des Codes identifiziert:

```
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;
    }
}
```

Durch das Überschreiben von findClass anstelle von loadClass das Delegationsmodell beibehalten. Der PluginClassLoader fragt zunächst das System und den übergeordneten Classloader nach Klassendefinitionen ab.

Erstellen Sie eine Richtlinie:

```
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;
    }

}
```

Legen Sie abschließend die Richtlinie und einen SecurityManager fest (Standardimplementierung ist in Ordnung):

```
    Policy.setPolicy(new PluginSecurityPolicy());
    System.setSecurityManager(new SecurityManager());
```

Regeln zum Verweigern von Richtlinien implementieren

Es ist gelegentlich wünschenswert , einer bestimmten ProtectionDomain eine bestimmte Permission zu *verweigern* , *unabhängig* von anderen Berechtigungen, die von der Domäne angefordert werden. Dieses Beispiel zeigt nur einen möglichen Ansatz, um diese Anforderung zu erfüllen. Sie führt eine „negative" Berechtigungsklasse, zusammen mit einem Wrapper, der die Standard ermöglicht Policy als Repository solcher Berechtigungen wiederverwendet werden.

Anmerkungen:

- Die standardmäßige Richtliniendatei-Syntax und der Mechanismus für die Zuweisung von Berechtigungen im Allgemeinen bleiben hiervon unberührt. Dies bedeutet, dass *Ablehnungsregeln* in Richtliniendateien immer noch als *Zuschüsse* ausgedrückt werden.
- Der Richtlinien-Wrapper soll die standardmäßige, dateibasierte Policy (die als com.sun.security.provider.PolicyFile) spezifisch kapseln.
- Verweigerte Berechtigungen werden nur auf Richtlinienebene als solche verarbeitet. Wenn Sie

einer Domäne statisch zugewiesen sind, werden sie standardmäßig von dieser Domäne als normale "positive" Berechtigungen behandelt.

Die DeniedPermission Klasse

```
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/>
 * &amp;nbsp;&amp;nbsp;&amp;nbsp;&amp;nbsp;<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/>
 *
&amp;nbsp;&amp;nbsp;&amp;nbsp;&amp;nbsp;<em>&amp;lt;target_class_name&amp;gt;:&amp;lt;target_name&amp;gt;(:&amp;lt;target_actions&amp;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;
    }

}
```

Die `DenyingPolicy` Klasse

```
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();
    }

}
```

Demo

```
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");
    }

}
```

Weisen Sie einige Berechtigungen zu:

```
grant codeBase "file:///path/to/classes/bin/-"
    permission java.util.PropertyPermission "*", "read,write";
    permission com.example.DeniedPermission "java.util.PropertyPermission:foo.bar:read";
};
```

Führen Sie zuletzt den Main und beobachten Sie, wie er aufgrund der "deny" DeniedPermission (DeniedPermission), die den grant (seine PropertyPermission) überschreibt, DeniedPermission . Beachten Sie, dass ein setProperty("foo.baz", "xyz") stattdessen erfolgreich war, da die verweigerte Berechtigung nur die Aktion "Lesen" und nur für die Eigenschaft "foo.bar" umfasst.

Sicherheitsmanager online lesen: https://riptutorial.com/de/java/topic/5712/sicherheitsmanager

Kapitel 147: Sichtbarkeit (Kontrolle des Zugriffs auf Mitglieder einer Klasse)

Syntax

- Name des öffentlichen Typs [= Wert];
- Name des privaten Typs [= Wert];
- geschützter Typname [= Wert];
- Typname [= Wert];
- öffentlicher Klassenname {
- Klassenname{

Bemerkungen

Aus dem Java-Tutorial :

Zugriffsstufenmodifizierer bestimmen, ob andere Klassen ein bestimmtes Feld verwenden oder eine bestimmte Methode aufrufen können. Es gibt zwei Ebenen der Zugriffskontrolle:

- Auf der obersten Ebene - public oder *package-private* (kein expliziter Modifikator).
- Auf Mitgliedsebene - public , private , protected oder *package-private* (kein expliziter Modifikator).

Eine Klasse kann mit dem Modifizierer public deklariert werden. In diesem Fall ist diese Klasse für alle Klassen überall sichtbar. Wenn eine Klasse keinen Modifikator hat (der Standard, auch als *package-private bezeichnet*), ist sie nur in ihrem eigenen Paket sichtbar.

Auf Mitgliedsebene können Sie ebenso wie bei Klassen der obersten Ebene und mit derselben Bedeutung den Modifizierer public oder no modifier (*package-private*) verwenden. Für Mitglieder gibt es zwei zusätzliche Zugriffsmodifizierer: private und protected . Der private Modifizierer gibt an, dass auf das Member nur in seiner eigenen Klasse zugegriffen werden kann. Der protected Modifier gibt an, dass auf das Mitglied nur innerhalb seines eigenen Pakets (wie bei *package-private*) und zusätzlich auf eine Unterklasse seiner Klasse in einem anderen Paket zugegriffen werden kann.

Die folgende Tabelle zeigt den Zugriff auf Mitglieder, die von jedem Modifikator zugelassen werden.

Zugangsebenen:

Modifikator	Klasse	Paket	Unterklasse	Welt
public	Y	Y	Y	Y
protected	Y	Y	Y	N
kein Modifikator	Y	Y	N	N
private	Y	N	N	N

Examples

Interface-Mitglieder

```
public interface MyInterface {
    public void foo();
```

```
    int bar();

    public String TEXT = "Hello";
    int ANSWER = 42;

    public class X {
    }

    class Y {
    }
}
```

Die Mitglieder der Benutzeroberfläche sind immer öffentlich sichtbar, auch wenn das public Schlüsselwort nicht angegeben wird. Daher sind sowohl foo() auch bar() , TEXT , ANSWER , X und Y öffentlich sichtbar. Der Zugriff kann jedoch immer noch durch die enthaltende Schnittstelle eingeschränkt sein. Da MyInterface öffentlich MyInterface ist, kann auf seine Mitglieder von überall her zugegriffen werden. Wenn MyInterface jedoch MyInterface hatte, waren seine Mitglieder nur aus demselben Paket heraus erreichbar.

Sichtbarkeit der Öffentlichkeit

Sichtbar für die Klasse, das Paket und die Unterklasse.

Sehen wir uns ein Beispiel mit der Klasse Test an.

```
public class Test{
    public int number = 2;

    public Test(){

    }
}
```

Versuchen wir nun, eine Instanz der Klasse zu erstellen. In diesem Beispiel können **wir** auf die number zugreifen, da sie public .

```
public class Other{

    public static void main(String[] args){
        Test t = new Test();
        System.out.println(t.number);
    }

}
```

Private Sichtbarkeit

private Sichtbarkeit kann auf eine Variable nur von ihrer Klasse zugegriffen werden. Sie werden häufig in **Verbindung** mit public Getter und Setzern verwendet.

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

Paket-Sichtbarkeit

Ohne Modifikator lautet die Standardeinstellung für die Paketsichtbarkeit. *In der Java-Dokumentation* "zeigt [Paketsichtbarkeit] an, ob Klassen in demselben Paket wie die Klasse (unabhängig von ihrer Abstammung) Zugriff auf das Member haben." In diesem Beispiel von javax.swing

```
package javax.swing;
public abstract class JComponent extends Container … {
    …
    static boolean DEBUG_GRAPHICS_LOADED;
    …
}
```

DebugGraphics befindet sich im selben Paket, sodass auf DEBUG_GRAPHICS_LOADED .

```
package javax.swing;
public class DebugGraphics extends Graphics {
    …
    static {
        JComponent.DEBUG_GRAPHICS_LOADED = true;
    }
    …
}
```

Dieser Artikel enthält einige Hintergrundinformationen zu diesem Thema.

Geschützte Sicht

Geschützte Sichtbarkeitsursachen bedeuten, dass dieses Member zusammen mit seinen Unterklassen für sein Paket sichtbar ist.

Als Beispiel:

```
package com.stackexchange.docs;
public class MyClass{
    protected int variable; //This is the variable that we are trying to access
    public MyClass(){
        variable = 2;
    };
```

```
}
```

Jetzt erweitern wir diese Klasse und versuchen, auf eines ihrer protected Mitglieder zuzugreifen.

```
package some.other.pack;
import com.stackexchange.docs.MyClass;
public class SubClass extends MyClass{
    public SubClass(){
        super();
        System.out.println(super.variable);
    }
}
```

Sie können auch auf ein protected Mitglied zugreifen, ohne es zu erweitern, wenn Sie von demselben Paket aus darauf zugreifen.

Beachten Sie, dass dieser Modifikator nur für Member einer Klasse und nicht für die Klasse selbst funktioniert.

Zusammenfassung der Zugriffsmodifizierer für Klassenmitglieder

Zugriffsmodifizierer	Sichtweite	Erbe
Privatgelände	Nur Klasse	Kann nicht vererbt werden
Kein Modifikator / Paket	Im Paket	Verfügbar, wenn Unterklasse im Paket
Geschützt	Im Paket	Verfügbar in Unterklasse
Öffentlichkeit	Überall	Verfügbar in Unterklasse

Es gab einmal einen private protected (beide Schlüsselwörter gleichzeitig) Modifizierer, der auf Methoden oder Variablen angewendet werden konnte, um sie von einer Unterklasse außerhalb des Pakets aus zugänglich zu machen, sie jedoch für die Klassen in diesem Paket als privat zu definieren. Dies wurde jedoch in der Version von Java 1.0 entfernt .

Sichtbarkeit (Kontrolle des Zugriffs auf Mitglieder einer Klasse) online lesen: https://riptutorial.com/de/java/topic/134/sichtbarkeit--kontrolle-des-zugriffs-auf-mitglieder-einer-klasse-

Kapitel 148: Singletons

Einführung

Ein Singleton ist eine Klasse, die immer nur eine Instanz hat. Weitere Informationen zum Singleton- *Entwurfsmuster* finden Sie im Thema Singleton im Design Patterns- Tag.

Examples

Enum Singleton

Java SE 5

```
public enum Singleton {
    INSTANCE;

    public void execute (String arg) {
        // Perform operation here
    }
}
```

Enums haben private Konstruktoren, sind endgültig und bieten geeignete Serialisierungsmaschinen. Sie sind auch sehr knapp und fadensicher initialisiert.

Die JVM bietet die Garantie, dass die Aufzählungswerte nicht mehr als jeweils einmal instanziiert werden, was dem Aufzählungszeichen eine enorme Abwehr gegen Reflexionsangriffe gibt.

Wobei das Enumerationsmuster *nicht* schützt, wenn andere Entwickler dem Quellcode physikalisch mehr Elemente hinzufügen. Wenn Sie diesen Implementierungsstil für Ihre Singletons wählen, müssen Sie daher unbedingt eindeutig dokumentieren, dass diesen Aufzählungen keine neuen Werte hinzugefügt werden sollen.

Dies ist die empfohlene Methode zum Implementieren des Singleton-Musters, wie von Joshua Bloch in Effective Java erläutert .

Fadensicheres Singleton mit doppeltem Karo-Verschluss

Dieser Singleton-Typ ist threadsicher und verhindert unnötiges Sperren, nachdem die Singleton-Instanz erstellt wurde.

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;
    }
}
```

Es muss hervorgehoben werden - in Versionen vor Java SE 5 ist die obige Implementierung falsch und sollte vermieden werden. Es ist nicht möglich, doppelt geprüftes Sperren vor Java 5 korrekt in Java zu implementieren.

Singleton ohne Enum (eifrige Initialisierung)

```
public class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Es kann argumentiert werden , dass dieses Beispiel *wirksam* faul Initialisierung ist. Abschnitt 12.4.1 der Java-Sprachspezifikation lautet:

> Eine Klasse oder ein Schnittstellentyp T wird unmittelbar vor dem ersten Auftreten eines der folgenden Ereignisse initialisiert:
>
> - T ist eine Klasse und eine Instanz von T wird erstellt
> - T ist eine Klasse und eine von T deklarierte statische Methode wird aufgerufen
> - Ein durch T deklariertes statisches Feld wird zugewiesen
> - Ein von T deklariertes statisches Feld wird verwendet und das Feld ist keine konstante Variable
> - T ist eine Klasse der obersten Ebene, und eine in T eingebettete Assert-Anweisung wird ausgeführt.

Solange es keine anderen statischen Felder oder statischen Methoden in der Klasse gibt, wird die Singleton Instanz daher erst initialisiert, wenn die Methode getInstance() zum ersten Mal aufgerufen wird.

Fadensichere Lazy-Initialisierung mit der Holder-Klasse | Bill Pugh Singleton Implementierung

```
public class Singleton {
    private static class InstanceHolder {
        static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return InstanceHolder.INSTANCE;
    }

    private Singleton() {}
}
```

Dies initialisiert die INSTANCE Variable beim ersten Aufruf von Singleton.getInstance() , wobei die Thread-Sicherheitsgarantien der Sprache für die statische Initialisierung genutzt werden, ohne dass eine zusätzliche Synchronisierung erforderlich ist.

Diese Implementierung wird auch als Bill Pugh Singleton Pattern bezeichnet. [Wiki]

Singleton erweitern (Singleton-Vererbung)

In diesem Beispiel stellt die Singleton getMessage() die getMessage() Methode getMessage() , die "Hello world!" getMessage() Botschaft.

Die Unterklassen UppercaseSingleton und LowercaseSingleton überschreiben die getMessage () - Methode, um die entsprechende Darstellung der Nachricht bereitzustellen.

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

Singletons online lesen: https://riptutorial.com/de/java/topic/130/singletons

Kapitel 149: SortedMap

Einführung

Einführung in die sortierte Karte.

Examples

Einführung in die sortierte Karte.

Kernpunkt :-

- SortedMap-Schnittstelle erweitert Map.
- Einträge werden in aufsteigender Reihenfolge gespeichert.

Methoden der sortierten Karte:

- Komparatorvergleicher ().
- Objekt firstKey ().
- SortedMap headMap (Objektende).
- Objekt lastKey ().
- SortedMap SubMap (Objektanfang, Objektende).
- SortedMap tailMap (Objektstart).

Beispiel

```
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"));
   }
```

SortedMap online lesen: https://riptutorial.com/de/java/topic/10748/sortedmap

Kapitel 150: Stack-Walking-API

Einführung

Vor Java 9 war der Zugriff auf die sun.reflect.Reflection auf eine interne Klasse sun.reflect.Reflection . Speziell die Methode sun.reflect.Reflection::getCallerClass . Einige Bibliotheken verlassen sich auf diese Methode, die veraltet ist.

Ein alternativer Standard - API wird nun in JDK 9 über die vorgesehen java.lang.StackWalker Klasse und wird , indem man lazy Zugang zu dem Stapelrahmen sein , effizient gestaltet. Einige Anwendungen können diese API verwenden, um den Ausführungsstapel zu durchsuchen und nach Klassen zu filtern.

Examples

Drucken Sie alle Stack-Frames des aktuellen Threads

Folgendes druckt alle Stack-Frames des aktuellen Threads:

```
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 }
```

Ausgabe:

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

Aktuelle Anruferklasse drucken

Im Folgenden wird die aktuelle Anruferklasse gedruckt. Beachten Sie, dass in diesem Fall die `StackWalker` mit der Option erstellt werden muss `RETAIN_CLASS_REFERENCE` , so dass die Class in den zurückgehalten werden `StackFrame` Objekte. Andernfalls würde eine Ausnahme auftreten.

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

Ausgabe:

```
class test.FooHelper
```

Reflektion und andere verborgene Frames anzeigen

Bei einigen anderen Optionen können Stack-Spuren Implementierungs- und / oder Reflektionsframes enthalten. Dies kann für Debugging-Zwecke hilfreich sein. Zum Beispiel können wir die Add `SHOW_REFLECT_FRAMES` Option zum `StackWalker` Beispiel bei der Erstellung, so dass der Rahmen für die reflektierenden Methoden als auch gedruckt werden:

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

Ausgabe:

```
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
```

Beachten Sie, dass Zeilennummern für einige Reflektionsmethoden möglicherweise nicht verfügbar sind, sodass StackFrame.getLineNumber() negative Werte zurückgeben kann.

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

Kapitel 151: Standardmethoden

Einführung

Mit der in Java 8 eingeführten **Standardmethode** können Entwickler einer Schnittstelle neue Methoden hinzufügen, ohne die vorhandenen Implementierungen dieser Schnittstelle zu beschädigen. Es bietet Flexibilität, damit die Schnittstelle eine Implementierung definieren kann, die standardmäßig verwendet wird, wenn eine Klasse, die diese Schnittstelle implementiert, keine Implementierung dieser Methode bereitstellt.

Syntax

- public default void methodName () {/ * Methodenkörper * /}

Bemerkungen

Standardmethoden

- Kann innerhalb einer Schnittstelle verwendet werden, um ein Verhalten einzuführen, ohne vorhandene Unterklassen zur Implementierung zu zwingen.
- Kann von Unterklassen oder von einem Subinterface überschrieben werden.
- In der Klasse java.lang.Object dürfen keine Methoden überschrieben werden.
- Wenn eine Klasse, die mehr als eine Schnittstelle implementiert, Standardmethoden mit identischen Methodensignaturen von den einzelnen Schnittstellen erbt, muss sie ihre eigene Schnittstelle überschreiben und so angeben, als wären sie keine Standardmethoden (als Teil der Auflösung der Mehrfachvererbung).
- Obwohl beabsichtigt wird, ein Verhalten einzuführen, ohne vorhandene Implementierungen zu unterbrechen, werden vorhandene Unterklassen mit einer statischen Methode mit derselben Methodensignatur wie die neu eingeführte Standardmethode weiterhin beschädigt. Dies gilt jedoch auch für die Einführung einer Instanzmethode in einer Superklasse.

Statische Methoden

- Kann innerhalb einer Schnittstelle verwendet werden, die hauptsächlich als Hilfsmittel für Standardmethoden verwendet werden soll.
- Kann nicht von Unterklassen oder von einem Subinterface überschrieben werden (ist für sie ausgeblendet). Wie schon bei statischen Methoden kann jedoch auch jede Klasse oder jedes Interface eine eigene haben.
- Es ist nicht zulässig, Instanzmethoden in der Klasse java.lang.Object zu überschreiben (wie dies auch für Unterklassen der Fall ist).

Die folgende Tabelle fasst die Interaktion zwischen Unterklasse und Oberklasse zusammen.

–	SUPER_CLASS-INSTANZ-METHODE	SUPER_CLASS-STATIC-METHODE
SUB_CLASS-INSTANCE-METHODE	*überschreibt*	*generiert-compiletime-error*
SUB_CLASS-STATIC-METHODE	*generiert-compiletime-error*	*versteckt sich*

Nachfolgend finden Sie eine Tabelle, die die Interaktion zwischen Schnittstelle und Implementierungsklasse zusammenfasst.

-	INTERFACE-DEFAULT-METHODE	SCHNITTSTELLE-STATISCHE-METHODE
IMPL_CLASS-INSTANCE-METHODE	*überschreibt*	*versteckt sich*
IMPL_CLASS-STATIC-METHODE	*generiert-compiletime-error*	*versteckt sich*

Verweise :

- http://www.journaldev.com/2752/java-8-interface-changes-static-method-default-method
- https://docs.oracle.com/javase/tutorial/java/IandI/override.html

Examples

Grundlegende Verwendung von Standardmethoden

```
/**
 * 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" );
    }
}
```

Die folgenden aussagen

```
    new WithDefault().printString();
    new OverrideDefault().printString();
```

Erzeugt diese Ausgabe:

```
default implementation
overridden implementation
```

Zugriff auf andere Schnittstellenmethoden innerhalb der Standardmethode

Sie können auch auf andere Schnittstellenmethoden innerhalb Ihrer Standardmethode zugreifen.

```
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;
    }
}
```

Die folgende Anweisung wird *3* drucken:

```
System.out.println(new Sum().calculateSum());
```

Standardmethoden können auch zusammen mit statischen Methoden der Schnittstelle verwendet werden:

```
public interface Summable {
    static int getA() {
        return 1;
    }

    static int getB() {
        return 2;
    }

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {}
```

Die folgende Anweisung wird auch 3 drucken:

```
System.out.println(new Sum().calculateSum());
```

Zugriff auf überschriebene Standardmethoden aus der implementierenden Klasse

In Klassen wird super.foo() nur in Superklassen super.foo() . Wenn Sie eine Standardimplementierung von einem Superinterface aus aufrufen möchten, müssen Sie sich super mit dem Namen der Schnittstelle qualifizieren: 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
    }
}
```

Warum Standardmethoden verwenden?

Die einfache Antwort ist, dass Sie eine vorhandene Schnittstelle weiterentwickeln können, ohne vorhandene Implementierungen zu beschädigen.

Zum Beispiel haben Sie eine Swim , die Sie vor 20 Jahren veröffentlicht haben.

```
public interface Swim {
    void backStroke();
}
```

Wir haben großartige Arbeit geleistet, unsere Benutzeroberfläche ist sehr beliebt, es gibt viele Implementierungen auf der ganzen Welt und Sie haben keine Kontrolle über den Quellcode.

```
public class FooSwimmer implements Swim {
    public void backStroke() {
         System.out.println("Do backstroke");
    }
}
```

Nach 20 Jahren haben Sie sich entschieden, der Benutzeroberfläche neue Funktionen hinzuzufügen. Es sieht jedoch so aus, als ob unsere Benutzeroberfläche eingefroren ist, da bestehende Implementierungen dadurch beschädigt werden.

Glücklicherweise führt Java 8 eine brandneue Funktion namens Default-Methode ein.

Wir können jetzt der Swim Oberfläche eine neue Methode hinzufügen.

```
public interface Swim {
    void backStroke();
    default void sideStroke() {
        System.out.println("Default sidestroke implementation. Can be overridden");
    }
}
```

Jetzt können alle vorhandenen Implementierungen unserer Schnittstelle weiterhin funktionieren. Am wichtigsten ist jedoch, dass sie die neu hinzugefügte Methode zu ihrer eigenen Zeit implementieren können.

Einer der Hauptgründe für diese Änderung und eine ihrer größten Anwendungen liegt im Java Collections-Framework. Oracle konnte der vorhandenen Iterable-Schnittstelle keine foreach Methode hinzufügen, ohne den gesamten vorhandenen Code zu zerstören, der Iterable implementiert hat. Durch das Hinzufügen von Standardmethoden erbt die vorhandene Iterable-Implementierung die Standardimplementierung.

Klasse, abstrakte Klasse und Schnittstellenmethode Vorrang

Implementierungen in Klassen, einschließlich abstrakter Deklarationen, haben Vorrang vor allen Schnittstellenstandardwerten.

- Die abstrakte Klassenmethode hat Vorrang vor der Schnittstellenstandardmethode .

```
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 {
}
```

Die folgende Aussage

```
new FooSwimmer().backStroke();
```

Wird herstellen

```
AbstractSwimmer.backStroke
```

- Klassenmethode hat Vorrang vor Schnittstellenstandardmethode

```
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");
    }
}
```

Die folgende Aussage

```
new FooSwimmer().backStroke();
```

Wird herstellen

```
FooSwimmer.backStroke
```

Standardmethode Mehrfachvererbungskollision

Betrachten Sie das nächste Beispiel:

```
public interface A {
    default void foo() { System.out.println("A.foo"); }
}

public interface B {
    default void foo() { System.out.println("B.foo"); }
```

```
}
```

Hier sind zwei Schnittstellen, die die default foo mit derselben Signatur deklarieren.

Wenn Sie versuchen extend diese beiden Schnittstellen in der neuen Schnittstelle zu erweitern, müssen Sie zwei auswählen, da Java Sie zwingt, diese Kollision explizit aufzulösen.

Zunächst können Sie die Methode foo mit derselben Signatur wie abstract deklarieren, wodurch das Verhalten von A und B außer Kraft gesetzt wird.

```
public interface ABExtendsAbstract extends A, B {
    @Override
    void foo();
}
```

Und wenn Sie ABExtendsAbstract in der class implement ABExtendsAbstract Sie eine foo Implementierung bereitstellen:

```
public class ABExtendsAbstractImpl implements ABExtendsAbstract {
    @Override
    public void foo() { System.out.println("ABImpl.foo"); }
}
```

Oder **zweitens**, können Sie eine völlig neue schaffen default Sie können den Code der A und B foo Methoden auch wiederverwenden, indem Sie von der Implementierung der Klasse auf überschriebene Standardmethoden zugreifen .

```
public interface ABExtends extends A, B {
    @Override
    default void foo() { System.out.println("ABExtends.foo"); }
}
```

Und wenn Sie ABExtends in der class implement ABExtends Sie not foo Implementierung bereitstellen:

```
public class ABExtendsImpl implements ABExtends {}
```

Standardmethoden online lesen: https://riptutorial.com/de/java/topic/113/standardmethoden

Kapitel 152: Steckdosen

Einführung

Ein Socket ist ein Endpunkt einer bidirektionalen Kommunikationsverbindung zwischen zwei im Netzwerk laufenden Programmen.

Examples

Vom Sockel lesen

```
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
}
```

Steckdosen online lesen: https://riptutorial.com/de/java/topic/9918/steckdosen

Kapitel 153: Streams

Einführung

Ein Stream repräsentiert eine Folge von Elementen und unterstützt verschiedene Arten von Operationen, um Berechnungen an diesen Elementen durchzuführen. Mit Java 8 verfügt die Collection Schnittstelle über zwei Methoden zum Generieren eines Stream : stream() und parallelStream() . Stream Operationen sind entweder Zwischen- oder Endgeräte. Zwischenoperationen geben einen Stream sodass mehrere Zwischenoperationen verkettet werden können, bevor der Stream geschlossen wird. Terminalvorgänge sind entweder ungültig oder liefern ein Nicht-Stream-Ergebnis.

Syntax

- collection.stream ()
- Arrays.stream (Array)
- Stream.iterate (ersterWert, aktuellerWert -> nächsterWert)
- Stream.generate (() -> Wert)
- Stream.of (elementOfT [, elementOfT, ...])
- Stream.empty ()
- StreamSupport.stream (iterable.spliterator (), false)

Examples

Streams verwenden

Ein Stream ist eine Folge von Elementen, auf die sequentielle und parallele Aggregatoperationen ausgeführt werden können. Jeder Stream kann möglicherweise eine unbegrenzte Datenmenge durchfließen. Infolgedessen werden Daten, die von einem Stream empfangen werden, bei ihrer Ankunft einzeln verarbeitet, im Gegensatz zur gesamten Stapelverarbeitung der Daten. In Kombination mit Lambda-Ausdrücken bieten sie eine präzise Möglichkeit, Operationen an Datensequenzen mithilfe eines funktionalen Ansatzes auszuführen.

Beispiel: (siehe Arbeit auf 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);
```

Ausgabe:

```
APFEL
BANANE
ORANGE
BIRNE
```

Die durch den obigen Code ausgeführten Operationen können wie folgt zusammengefasst werden:

1. Erstellen Sie einen Stream<String> - Stream String Stream.of(values) Stream<String> enthält eine sequenziert geordneten Stream von Obst String - Elemente unter Verwendung der statischen Factory - Methode Stream.of(values) .
2. Die filter() Operation behält nur Elemente bei, die mit einem bestimmten Prädikat übereinstimmen (die Elemente, die beim Prädikat getestet werden, geben true zurück). In diesem Fall bleiben die Elemente erhalten, die ein "a" . Das Prädikat wird als Lambda-Ausdruck angegeben .

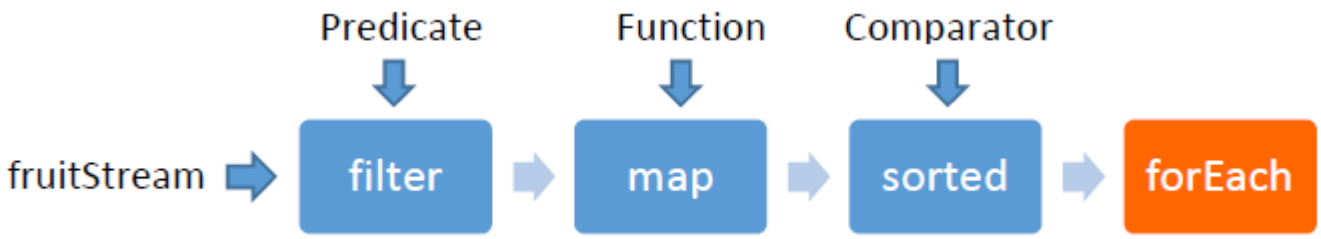
3. Die map() Operation transformiert jedes Element mithilfe einer angegebenen Funktion, die als Mapper bezeichnet wird. In diesem Fall wird jeder Obst- String mithilfe der Methodenreferenz String::toUppercase seiner String Version in Großbuchstaben String::toUppercase .

   > **Beachten Sie,** dass die Operation map() einen Stream mit einem anderen generischen Typ zurückgibt, wenn die Zuordnungsfunktion einen anderen Typ als den Eingabeparameter zurückgibt. Zum Beispiel gibt ein Aufruf von .map(String::isEmpty) in einem Stream<Boolean> Stream<String> einen Stream<Boolean>

4. Die sorted() Operation sortiert die Elemente des Stream entsprechend ihrer natürlichen Reihenfolge (lexikographisch im Fall von String).

5. Schließlich führt die Operation forEach(action) eine Aktion aus, die auf jedes Element des Stream einwirkt und an einen Consumer weitergibt . In diesem Beispiel wird jedes Element einfach auf die Konsole gedruckt. Diese Operation ist eine Terminal-Operation, die es unmöglich macht, sie erneut zu bearbeiten.

   > **Beachten Sie,** dass auf dem Stream definierte Operationen *aufgrund* der Terminaloperation ausgeführt werden. Ohne eine Terminaloperation wird der Stream nicht verarbeitet. Streams können nicht wiederverwendet werden. Sobald eine Terminaloperation aufgerufen wird, wird das Stream Objekt unbrauchbar.



Operationen (wie oben gezeigt) werden miteinander verkettet und bilden so eine Abfrage der Daten.

Streams schließen

> **Beachten Sie, dass ein Stream Allgemeinen nicht geschlossen werden muss.** Es müssen nur Streams geschlossen werden, die auf E / A-Kanälen arbeiten. Die meisten Stream Typen arbeiten nicht mit Ressourcen und müssen daher nicht geschlossen werden.

Die Stream Schnittstelle erweitert AutoCloseable . Streams können durch Aufrufen der close Methode oder mithilfe von try-with-resource-Anweisungen geschlossen werden.

Ein Beispiel für den Fall, dass ein Stream geschlossen werden soll, ist das Erstellen eines Stream aus Zeilen aus einer Datei:

```
try (Stream<String> lines = Files.lines(Paths.get("somePath"))) {
    lines.forEach(System.out::println);
}
```

Das Stream Interface deklariert auch die Stream.onClose() -Methode, mit der Sie Runnable Handler registrieren können, die beim Runnable des Streams aufgerufen werden. Ein Beispiel für einen Anwendungsfall ist, wo Code, der einen Stream erzeugt, wissen muss, wann er zur Bereinigung verbraucht wird.

```
public Stream<String>streamAndDelete(Path path) throws IOException {
    return Files.lines(path).onClose(() -> someClass.deletePath(path));
}
```

Der run-Handler wird nur ausgeführt, wenn die close() -Methode explizit oder implizit durch eine

try-with-resources-Anweisung aufgerufen wird.

Bearbeitungsauftrag

Die Verarbeitung eines Stream Objekts kann sequentiell oder parallel sein .

In einem **sequentiellen** Modus werden die Elemente in der Reihenfolge der Quelle des Stream . Wenn der Stream bestellt wird (z. B. eine SortedMap Implementierung oder eine List), SortedMap die Verarbeitung garantiert mit der Reihenfolge der Quelle überein. In anderen Fällen sollte jedoch darauf geachtet werden, nicht von der Reihenfolge abzuhängen (siehe: keySet() die Java HashMap keySet() Iterationsreihenfolge konsistent?).

Beispiel:

```
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
```

Live auf Ideone

Der parallele Modus ermöglicht die Verwendung mehrerer Threads auf mehreren Kernen, es gibt jedoch keine Garantie für die Reihenfolge, in der Elemente verarbeitet werden.

Wenn mehrere Methoden für einen sequentiellen Stream aufgerufen werden, muss nicht jede Methode aufgerufen werden. Wenn beispielsweise ein Stream gefiltert wird und die Anzahl der Elemente auf eins reduziert wird, erfolgt kein nachfolgender Aufruf einer Methode wie sort . Dies kann die Leistung eines sequentiellen Stream erhöhen - eine Optimierung, die mit einem parallelen Stream nicht möglich ist.

Beispiel:

```
// parallel
long howManyOddNumbersParallel = integerList.parallelStream()
                                            .filter(e -> (e % 2) == 1)
                                            .count();

System.out.println(howManyOddNumbersParallel); // Output: 2
```

Live auf Ideone

Unterschiede zu Containern (oder Sammlungen)

Während einige Aktionen sowohl für Container als auch für Streams ausgeführt werden können, dienen sie letztendlich unterschiedlichen Zwecken und unterstützen unterschiedliche Operationen. Container konzentrieren sich mehr darauf, wie die Elemente gespeichert werden und wie auf diese Elemente effizient zugegriffen werden kann. Auf der anderen Seite bietet ein Stream keinen direkten Zugriff und keine Manipulation auf seine Elemente. Es ist mehr der Gruppe von Objekten als einer kollektiven Entität gewidmet und führt Operationen an dieser Entität als Ganzes durch. Stream und Collection sind separate Abstraktionen auf hoher Ebene für diese unterschiedlichen Zwecke.

Sammeln Sie Elemente eines Streams in eine Sammlung

Sammeln mit toList() und toSet()

Elemente aus einem Stream können mit der Stream.collect Operation leicht in einem Container Stream.collect werden:

```
System.out.println(Arrays
    .asList("apple", "banana", "pear", "kiwi", "orange")
    .stream()
    .filter(s -> s.contains("a"))
    .collect(Collectors.toList())
);
// prints: [apple, banana, pear, orange]
```

Andere Sammlungsinstanzen, wie z. B. ein Set , können mit anderen integrierten Collectors - Methoden erstellt werden. Beispielsweise sammelt Collectors.toSet() die Elemente eines Stream in einem Set .

Explizite Kontrolle über die Implementierung von List oder Set

Gemäß der Dokumentation von Collectors#toList() und Collectors#toSet() gibt es keine Gewähr für Typ, Veränderlichkeit, Serialisierbarkeit oder Collectors#toSet() der zurückgegebenen List oder des Set .

Für die explizite Steuerung der zurückzugebenden Implementierung kann stattdessen Collectors#toCollection(Supplier) verwendet werden, bei der der angegebene Lieferant eine neue und leere Sammlung zurückgibt.

```
// 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<>()))
);
```

Elemente mit toMap sammeln

Der Collector sammelt Elemente in einer Karte, wobei der Schlüssel die Schüler-ID und der Wert den Schülerwert ist.

```
  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);
```

Ausgabe :

```
{1=test1, 2=test2, 3=test3}
```

Die Collectors.toMap hat eine andere Implementierung: Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper, BinaryOperator<U> mergeFunction) Die mergeFunction wird hauptsächlich verwendet, um entweder einen neuen Wert

auszuwählen oder den alten Wert beizubehalten, wenn der Schlüssel wiederholt wird, wenn ein neues Mitglied in der Map aus einer Liste hinzugefügt wird.

Die mergeFunction sieht häufig wie folgt aus: (s1, s2) -> s1 , um den Wert zu erhalten, der dem wiederholten Schlüssel entspricht, oder (s1, s2) -> s2 , um einen neuen Wert für den wiederholten Schlüssel zu setzen.

Elemente zur Karte der Sammlungen sammeln

Beispiel: von ArrayList nach Map <String, List <>>

Häufig muss eine Liste mit einer Liste aus einer Primärliste erstellt werden. Beispiel: Von einem Schüler der Liste müssen wir eine Liste der Themen für jeden Schüler erstellen.

```
    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);
```

Ausgabe:

```
{ Robert=[LITERATURE],
Sascha=[ENGLISH, MATH, SCIENCE, LITERATURE],
Davis=[MATH, SCIENCE, GEOGRAPHY] }
```

Beispiel: von ArrayList nach 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);
```

Ausgabe:

```
{ 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]} }
```

Spickzettel

Tor	Code
Sammle zu einer List	Collectors.toList()
In einer ArrayList mit vorab zugewiesener Größe sammeln	Collectors.toCollection(() -> new ArrayList<>(size))
Sammeln Sie zu einem Set	Collectors.toSet()
Sammeln Sie zu einem Set mit besserer Iterationsleistung	Collectors.toCollection(() -> new LinkedHashSet<>())
Sammeln Sie zu einem Set<String> Groß- und Kleinschreibung nicht Set<String>	Collectors.toCollection(() -> new TreeSet<>(String.CASE_INSENSITIVE_ORDER))
In einem EnumSet<AnEnum> (beste Leistung für Enums)	Collectors.toCollection(() -> EnumSet.noneOf(AnEnum.class))
Sammeln Sie auf einer Map<K,V> mit eindeutigen Schlüsseln	Collectors.toMap(keyFunc,valFunc)
Ordnen Sie MyObject.getter () einem eindeutigen MyObject zu	Collectors.toMap(MyObject::getter, Function.identity())
Ordnen Sie MyObject.getter () mehreren MyObjects zu	Collectors.groupingBy(MyObject::getter)

Unendlich Streams

Es ist möglich, einen Stream zu generieren, der nicht beendet wird. Wenn Sie eine Terminal-Methode für einen infinite Stream aufrufen, tritt der Stream in eine Endlosschleife ein. Die limit Methode eines Stream kann verwendet werden, um die Anzahl der Terme des Stream zu begrenzen, die Java verarbeitet.

In diesem Beispiel wird ein Stream aller natürlichen Zahlen generiert, beginnend mit der Zahl 1. Jeder nachfolgende Begriff des Stream ist um eins höher als der vorherige. Beim Aufruf der Limit-Methode dieses Stream werden nur die ersten fünf Terme des Stream berücksichtigt und gedruckt.

```
// 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);
```

Ausgabe:

```
1
2
3
4
```

```
5
```

Eine andere Methode zum Erzeugen eines unendlichen Streams besteht in der Verwendung der Stream.generate- Methode. Diese Methode benötigt ein Lambda vom Typ Supplier .

```
// Generate an infinite stream of random numbers
Stream<Double> infiniteRandomNumbers = Stream.generate(Math::random);

// Print out only the first 10 random numbers
infiniteRandomNumbers.limit(10).forEach(System.out::println);
```

Verbrauchende Streams

Ein Stream wird nur dann durchlaufen, wenn eine *Terminaloperation* wie count() , collect() oder forEach() . Andernfalls wird keine Operation für den Stream ausgeführt.

Im folgenden Beispiel wird dem Stream keine Terminaloperation hinzugefügt, sodass die filter() Operation nicht aufgerufen wird und keine Ausgabe erzeugt wird, da peek() keine *Terminaloperation ist* .

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).peek(System.out::println);
```

Live auf Ideone

Dies ist eine Stream Sequenz mit einem gültigen *Terminalbetrieb* , daher wird eine Ausgabe erzeugt.

Sie können auch forEach anstelle von peek :

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).forEach(System.out::println);
```

Live auf Ideone

Ausgabe:

```
2
4
6
8
```

Nachdem der Terminalbetrieb ausgeführt wurde, wird der Stream verbraucht und kann nicht wiederverwendet werden.

Obwohl ein bestimmtes Stream-Objekt nicht wiederverwendet werden kann, ist es einfach, eine wiederverwendbare Iterable zu erstellen, die an eine Stream-Pipeline delegiert. Dies kann nützlich sein, um eine modifizierte Ansicht eines Live-Datensatzes zurückzugeben, ohne Ergebnisse in einer temporären Struktur sammeln zu müssen.

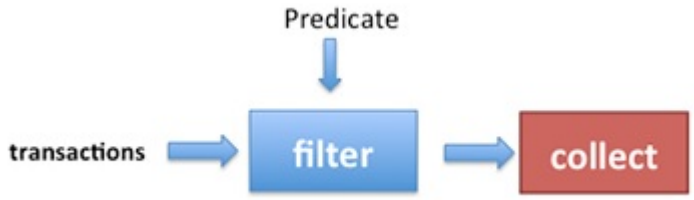
```
List<String> list = Arrays.asList("FOO", "BAR");
Iterable<String> iterable = () -> list.stream().map(String::toLowerCase).iterator();

for (String str : iterable) {
    System.out.println(str);
}
for (String str : iterable) {
    System.out.println(str);
}
```

Ausgabe:

```
foo
Bar
foo
Bar
```

Dies funktioniert, weil Iterable eine einzige abstrakte Methode Iterator<T> iterator() deklariert. Das macht es zu einer funktionellen Schnittstelle, die von einem Lambda implementiert wird, das bei jedem Aufruf einen neuen Stream erstellt.

Im Allgemeinen funktioniert ein Stream wie in der folgenden Abbildung dargestellt:



HINWEIS : Argumentprüfungen werden immer ausgeführt, auch ohne *Terminaloperation* :

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

Live auf Ideone

Ausgabe:

```
Wir haben eine NullPointerException erhalten, da null als Argument an filter ()
übergeben wurde.
```

Frequenzkarte erstellen

Der groupingBy(classifier, downstream) -Kollektor ermöglicht das Sammeln von Stream Elementen in einer Map indem jedes Element in einer Gruppe klassifiziert und ein Downstream-Vorgang für die Elemente ausgeführt wird, die in derselben Gruppe klassifiziert sind.

Ein klassisches Beispiel für dieses Prinzip ist die Verwendung einer Map um die Vorkommen von Elementen in einem Stream . In diesem Beispiel ist der Klassifizierer einfach die Identitätsfunktion, die das Element unverändert zurückgibt. Der Downstream-Vorgang zählt die Anzahl der gleichen Elemente mithilfe von counting() .

```
Stream.of("apple", "orange", "banana", "apple")
      .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
      .entrySet()
      .forEach(System.out::println);
```

Die nachgelagerte Operation ist selbst ein Collector (Collectors.counting()), der Elemente des Typs String verarbeitet und ein Ergebnis des Typs Long . Das Ergebnis des Aufrufs der collect Methode ist Map<String, Long> .

Dies würde die folgende Ausgabe erzeugen:

```
Banane = 1
orange = 1
Apfel = 2
```

Paralleler Stream

Hinweis: Bevor Sie sich für einen Stream , sollten Sie sich das Verhalten von ParallelStream vs. Sequential Stream anschauen.

Wenn Sie Stream Vorgänge gleichzeitig ausführen möchten, können Sie eine dieser Methoden verwenden.

```
List<String> data = Arrays.asList("One", "Two", "Three", "Four", "Five");
Stream<String> aParallelStream = data.stream().parallel();
```

Oder:

```
Stream<String> aParallelStream = data.parallelStream();
```

Rufen Sie einen Terminaloperator auf, um die für den parallelen Stream definierten Vorgänge auszuführen:

```
aParallelStream.forEach(System.out::println);
```

(Eine mögliche) Ausgabe vom parallelen Stream :

```
Drei
Vier
Ein
Zwei
Fünf
```

Die Reihenfolge kann sich ändern, da alle Elemente parallel verarbeitet werden (was sie *möglicherweise* schneller macht). Verwenden Sie parallelStream wenn die Bestellung keine Rolle spielt.

Auswirkungen auf die Leistung

Wenn es sich um Netzwerke handelt, können parallele Stream die Gesamtleistung einer Anwendung beeinträchtigen, da alle parallelen Stream einen gemeinsamen Fork-Join-Thread-Pool für das Netzwerk verwenden.

Auf der anderen Seite können parallele Stream die Leistung in vielen anderen Fällen erheblich verbessern, abhängig von der Anzahl der verfügbaren Kerne in der laufenden CPU.

Konvertieren eines optionalen Streams in einen Wertestrom

Möglicherweise müssen Sie einen Stream , der Optional in einen Stream von Werten konvertieren und nur Werte aus dem vorhandenen Optional . (dh ohne null und nicht mit 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]
```

Stream erstellen

Alle java Collection<E> verfügen über die Methoden stream() und parallelStream() , aus denen ein

Stream<E> erstellt werden kann:

```
Collection<String> stringList = new ArrayList<>();
Stream<String> stringStream = stringList.parallelStream();
```

Ein Stream<E> kann mit einem von zwei Verfahren aus einem Array erstellt werden:

```
String[] values = { "aaa", "bbbb", "ddd", "cccc" };
Stream<String> stringStream = Arrays.stream(values);
Stream<String> stringStreamAlternative = Stream.of(values);
```

Der Unterschied zwischen Arrays.stream() und Stream.of() besteht darin, dass Stream.of() einen varargs -Parameter besitzt, sodass er folgendermaßen verwendet werden kann:

```
Stream<Integer> integerStream = Stream.of(1, 2, 3);
```

Es gibt auch primitive Stream , die Sie verwenden können. Zum Beispiel:

```
IntStream intStream = IntStream.of(1, 2, 3);
DoubleStream doubleStream = DoubleStream.of(1.0, 2.0, 3.0);
```

Diese primitiven Streams können auch mit der Arrays.stream() -Methode erstellt werden:

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

Es ist möglich, einen Stream aus einem Array mit einem angegebenen Bereich zu erstellen.

```
int[] values= new int[]{1, 2, 3, 4, 5};
IntStream intStram = Arrays.stream(values, 1, 3);
```

Beachten Sie, dass jeder primitive Stream mit der boxed Methode in einen Boxed-Type-Stream konvertiert werden kann:

```
Stream<Integer> integerStream = intStream.boxed();
```

Dies kann in einigen Fällen hilfreich sein, wenn Sie die Daten erfassen möchten, da der primitive Stream keine collect , für die ein Collector als Argument verwendet wird.

Wiederverwenden von Zwischenvorgängen einer Stromkette

Der Stream wird geschlossen, wenn der Terminalbetrieb aufgerufen wird. Wiederverwenden des Stroms von Zwischenvorgängen, wenn nur der Terminalbetrieb nur variiert. Wir könnten einen Stream-Anbieter erstellen, um einen neuen Stream mit allen bereits eingerichteten Zwischenvorgängen aufzubauen.

```
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[] Arrays können mit Streams in List<Integer> konvertiert werden

```
int[] ints = {1,2,3};
List<Integer> list = IntStream.of(ints).boxed().collect(Collectors.toList());
```

Suchen von Statistiken zu numerischen Streams

Java 8 stellt Klassen genannt IntSummaryStatistics , DoubleSummaryStatistics und LongSummaryStatistics , die ein Statusobjekt für die Statistik wie geben sammeln count , min , max , sum und 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);
```

Was führt zu:

Java SE 8

```
IntSummaryStatistics{count=10, sum=55, min=1, max=10, average=5.500000}
```

Holen Sie sich ein Stück Strom

Beispiel: Holen Sie sich einen Stream mit 30 Elementen, der das 21. bis einschließlich 50. Element einer Sammlung enthält.

```
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);
```

Anmerkungen:

- IllegalArgumentException wird ausgelöst, wenn n negativ ist oder maxSize negativ ist
- Sowohl skip(long) als auch limit(long) sind Zwischenoperationen
- Wenn ein Stream weniger als n Elemente enthält, gibt skip(n) einen leeren Stream zurück
- Sowohl skip(long) als auch limit(long) sind billige Vorgänge bei sequentiellen Stream-Pipelines, können jedoch bei geordneten parallelen Pipelines recht teuer sein

Streams verketten

Variablendeklaration für Beispiele:

```
Collection<String> abc = Arrays.asList("a", "b", "c");
Collection<String> digits = Arrays.asList("1", "2", "3");
Collection<String> greekAbc = Arrays.asList("alpha", "beta", "gamma");
```

Beispiel 1 - Verketten Sie zwei Stream

```
final Stream<String> concat1 = Stream.concat(abc.stream(), digits.stream());

concat1.forEach(System.out::print);
// prints: abc123
```

Beispiel 2 - Verketten Sie mehr als zwei Stream

```
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
```

Alternativ zur Vereinfachung der verschachtelten concat() die Stream auch mit flatMap() verkettet werden:

```
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
```

Seien Sie beim Stream von Stream aus wiederholten Verkettungen vorsichtig, da der Zugriff auf ein Element eines tief verketteten Stream zu tiefen Aufrufketten oder sogar zu einer StackOverflowException .

IntStream to String

Java verfügt nicht über einen *Char-Stream* . Wenn Sie also mit String s arbeiten und einen Stream von Character IntStream , besteht die Option, einen IntStream von Codepunkten mithilfe der String.codePoints() -Methode String.codePoints() . So ist IntStream wie IntStream :

```
public IntStream stringToIntStream(String in) {
  return in.codePoints();
}
```

Es ist etwas komplizierter, die Konvertierung anders durchzuführen, z. B. IntStreamToString. Das kann wie folgt gemacht werden:

```
public String intStreamToString(IntStream intStream) {
  return intStream.collect(StringBuilder::new, StringBuilder::appendCodePoint,
StringBuilder::append).toString();
}
```

Sortieren mit 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);
```

Ausgabe:

```
[Sydney, London, New York, Amsterdam, Mumbai, California]
[Amsterdam, California, London, Mumbai, New York, Sydney]
```

Es ist auch möglich, einen anderen Vergleichsmechanismus zu verwenden, da es eine überladene `sorted` Version gibt, die einen Vergleicher als Argument verwendet.

Sie können auch einen Lambda-Ausdruck zum Sortieren verwenden:

```
List<String> sortedData2 = data.stream().sorted((s1,s2) ->
s2.compareTo(s1)).collect(Collectors.toList());
```

Dies würde [Sydney, New York, Mumbai, London, California, Amsterdam] ausgeben.

Sie können Comparator.reverseOrder() , um einen Komparator zu haben, der die reverse der natürlichen Reihenfolge durchführt.

```
List<String> reverseSortedData =
data.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
```

Ströme von Primitiven

Java bietet spezialisierte Stream für drei Arten von IntStream : IntStream (für int), LongStream (für long s) und DoubleStream (für double s). Sie sind nicht nur optimierte Implementierungen für ihre jeweiligen Grundelemente, sondern bieten auch mehrere spezifische Terminalmethoden, typischerweise für mathematische Operationen. Z.B:

```
IntStream is = IntStream.of(10, 20, 30);
double average = is.average().getAsDouble(); // average is 20.0
```

Sammeln Sie die Ergebnisse eines Streams in einem Array

Analog, um eine Sammlung für einen Stream durch collect() zu erhalten, kann ein Array mit der Stream.toArray() -Methode erhalten werden:

```
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 ist eine spezielle Art von Methodenreferenz: eine Konstruktorreferenz.

Das erste Element finden, das einem Prädikat entspricht

Es ist möglich, das erste Element eines Stream zu finden, das einer Bedingung entspricht.

In diesem Beispiel finden wir die erste Integer deren Quadrat über 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
```

Dieser Ausdruck gibt ein OptionalInt mit dem Ergebnis zurück.

Beachten Sie, dass Java bei einem unendlichen Stream jedes Element so lange überprüft, bis es

ein Ergebnis findet. Bei einem endlichen Stream gibt Java ein leeres OptionalInt zurück, wenn ihm die Elemente ausgehen, das Ergebnis jedoch nicht gefunden wird.

Verwenden von IntStream zum Durchlaufen von Indizes

Stream von Elementen erlauben normalerweise keinen Zugriff auf den Indexwert des aktuellen Elements. Verwenden Sie IntStream.range(start, endExclusive) um über ein Array oder eine ArrayList zu iterieren, während Sie auf Indizes 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);
```

Die range(start, endExclusive) -Methode gibt einen anderen ÌntStream und der mapToObj(mapper) gibt einen Stream von String .

Ausgabe:

```
# 1 Jon
# 2 Darin
# 3 Bauke
# 4 Hans
# 5 Marc
```

Dies ist der Verwendung einer normalen for Schleife mit einem Zähler sehr ähnlich, jedoch mit dem Vorteil des Pipelining und der Parallelisierung:

```
for (int i = 0; i < names.length; i++) {
    String newName = String.format("#%d %s", i + 1, names[i]);
    System.out.println(newName);
}
```

Flachen von Streams mit flatMap ()

Ein Stream von Elementen, die wiederum streamfähig sind, kann zu einem einzigen kontinuierlichen Stream :

Das Array mit der Liste der Elemente kann in eine einzige Liste konvertiert werden.

```
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]
```

Map mit der Liste der Elemente als Werte kann auf eine kombinierte Liste reduziert werden

```
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 der Map kann in einen einzigen kontinuierlichen Stream

```
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]
```

Erstellen Sie eine Karte basierend auf einem Stream

Einfacher Fall ohne doppelte Schlüssel

```
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}
```

Um die Dinge deklarativer zu gestalten, können wir in Function interface - Function.identity() statische Methode verwenden. Wir können dieses Lambda- element -> element durch Function.identity() ersetzen.

Fall, in dem möglicherweise doppelte Schlüssel vorhanden sind

Der Javadoc für Collectors.toMap gibt Folgendes an:

> Wenn die zugeordneten Schlüssel Duplikate enthalten (gemäß Object.equals(Object)), wird eine IllegalStateException ausgelöst, wenn der Auflistungsvorgang ausgeführt wird. Wenn die zugeordneten Tasten Duplikate enthalten können, verwenden toMap(Function, Function, BinaryOperator) stattdessen 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}
```

Der an Collectors.toMap(...) BinaryOperator generiert den Wert, der bei einer Kollision gespeichert werden soll. Es kann:

- den alten Wert zurückgeben, sodass der erste Wert im Stream Vorrang hat,
- den neuen Wert zurückgeben, damit der letzte Wert im Stream Vorrang hat, oder
- kombinieren Sie die alten und neuen Werte

Gruppierung nach Wert

Sie können Collectors.groupingBy wenn Sie das Äquivalent einer kaskadierten "group by" - Operation der Datenbank ausführen müssen. Zur Veranschaulichung wird im Folgenden eine Karte erstellt, in der die Namen der Personen den Nachnamen zugeordnet werden:

```
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]}
```

Live auf Ideone

Zufällige Strings mit Streams erzeugen

Es ist manchmal nützlich, zufällige Strings zu erstellen, z. B. als Sitzungs-ID für einen Web-Service oder als Initialkennwort nach der Registrierung für eine Anwendung. Dies kann leicht mit Stream s erreicht werden.

Zuerst müssen wir einen Zufallszahlengenerator initialisieren. Zur Erhöhung der Sicherheit für den erzeugten String s, ist es eine gute Idee zu verwenden SecureRandom .

Hinweis : Das Erstellen eines SecureRandom ist ziemlich teuer. SecureRandom es sich, dies nur einmal zu tun und setSeed() eine seiner setSeed() Methoden aufzurufen, um es neu zu bestimmen.

```
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
```

Bei der Erstellung von zufälligen String s möchten wir normalerweise, dass sie nur bestimmte Zeichen (z. B. nur Buchstaben und Ziffern) verwenden. Daher können wir eine Methode erstellen, die einen boolean der später zum Filtern des 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);
}
```

Als Nächstes können wir das RNG verwenden, um einen zufälligen String mit einer bestimmten Länge zu generieren, der den Zeichensatz enthält, der unsere useThisCharacter Prüfung besteht.

```
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;
}
```

Verwenden von Streams zum Implementieren mathematischer Funktionen

Stream und vor allem IntStream sind eine elegante Möglichkeit, Summationsterme (∑) zu implementieren. Die Bereiche des Stream können als Grenzen der Summation verwendet werden.

Madhavas Annäherung an Pi ist beispielsweise durch die Formel gegeben (Quelle: wikipedia):

$$\pi = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-3)^{-k}}{2k+1} = \sqrt{12}\sum_{k=0}^{\infty}\frac{(-\frac{1}{3})^{k}}{2k+1} = \sqrt{12}\left(\frac{1}{1\cdot 3^0} - \frac{1}{3\cdot 3^1} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots\right)$$

Dies kann mit einer beliebigen Genauigkeit berechnet werden. ZB für 101 Begriffe:

```
double pi = Math.sqrt(12) *
            IntStream.rangeClosed(0, 100)
                     .mapToDouble(k -> Math.pow(-3, -1 * k) / (2 * k + 1))
                     .sum();
```

Hinweis: Mit der double Genauigkeit genügt die Auswahl einer oberen Grenze von 29, um ein Ergebnis zu Math.Pi , das sich von Math.Pi .

Verwenden von Streams und Methodenreferenzen zum Schreiben selbstdokumentierender Prozesse

Methodenreferenzen machen ausgezeichneten selbstdokumentierenden Code, und die Verwendung von Methodenreferenzen mit Stream s macht komplizierte Prozesse leicht lesbar und verständlich. Betrachten Sie den folgenden Code:

```
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;
}
```

Diese letzte mit Stream und Methodenreferenzen neu geschriebene Methode ist viel lesbarer und jeder Schritt des Prozesses ist schnell und einfach zu verstehen. Sie ist nicht nur kürzer, sondern zeigt auf einen Blick, welche Schnittstellen und Klassen in jedem Schritt für den Code verantwortlich sind:

```
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());
}
```

Verwenden von Streams of Map.Entry zum Erhalten der Anfangswerte nach dem Mapping

Wenn Sie einen Stream haben, den Sie Map.Entry<K,V> müssen, die ursprünglichen Werte aber auch beibehalten möchten, können Sie den Stream einer Map.Entry<K,V>

```
public static <K, V> Function<K, Map.Entry<K, V>> entryMapper(Function<K, V> mapper){
    return (k)->new AbstractMap.SimpleEntry<>(k, mapper.apply(k));
}
```

Dann können Sie Ihren Konverter verwenden, um Stream zu verarbeiten, die sowohl auf die ursprünglichen als auch auf die zugeordneten Werte zugreifen können:

```
Set<K> mySet;
Function<K, V> transformer = SomeClass::transformerMethod;
Stream<Map.Entry<K, V>> entryStream = mySet.stream()
    .map(entryMapper(transformer));
```

Sie können diesen Stream wie gewohnt weiterverarbeiten. Dies vermeidet den Aufwand für das Erstellen einer Zwischensammlung.

Stream-Betriebskategorien

Streamoperationen lassen sich in zwei Hauptkategorien, Zwischen- und Terminaloperationen, und zwei Unterkategorien, zustandslos und zustandsbehaftet, unterteilen.

Zwischenoperationen:

Eine Zwischenoperation ist immer *faul* , wie eine einfache Stream.map . Sie wird erst aufgerufen, wenn der Stream tatsächlich verbraucht ist. Dies kann leicht überprüft werden:

```
Arrays.asList(1, 2 ,3).stream().map(i -> {
    throw new RuntimeException("not gonna happen");
    return i;
});
```

Zwischenoperationen sind die allgemeinen Bausteine eines Streams, die nach der Quelle verkettet sind, und normalerweise folgt eine Terminaloperation, die die Streamkette auslöst.

Terminalbetrieb

Terminalvorgänge lösen den Verbrauch eines Streams aus. Einige der häufigsten sind Stream.forEach oder Stream.collect . Sie werden normalerweise nach einer Kette von Zwischenoperationen angeordnet und sind fast immer *eifrig* .

Zustandslose Operationen

Zustandslosigkeit bedeutet, dass jeder Artikel ohne den Kontext anderer Artikel verarbeitet wird. Zustandslose Operationen ermöglichen eine speichereffiziente Verarbeitung von Streams. Vorgänge wie Stream.map und Stream.filter , für die keine Informationen zu anderen Elementen des Streams erforderlich sind, werden als zustandslos betrachtet.

Stateful Operationen

Statefulness bedeutet, dass die Operation für jedes Element von (einigen) anderen Elementen des Streams abhängt. Dies erfordert, dass ein Zustand erhalten bleibt. Statefulness-Operationen können mit langen oder unendlichen Streams brechen. Vorgänge wie Stream.sorted erfordern die Stream.sorted des gesamten Streams, bevor ein Element Stream.sorted wird, das einen ausreichend langen Stream von Elementen Stream.sorted . Dies kann durch einen langen Stream (**auf eigenes Risiko**) gezeigt werden:

```
// works - stateless stream
long BIG_ENOUGH_NUMBER = 999999999;
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).forEach(System.out::println);
```

Dies führt aufgrund von Stream.sorted von Stream.sorted zu einem Out-of-Memory.

```
// Out of memory - stateful stream
IntStream.iterate(0, i -> i +
1).limit(BIG_ENOUGH_NUMBER).sorted().forEach(System.out::println);
```

Konvertierung eines Iterators in einen Stream

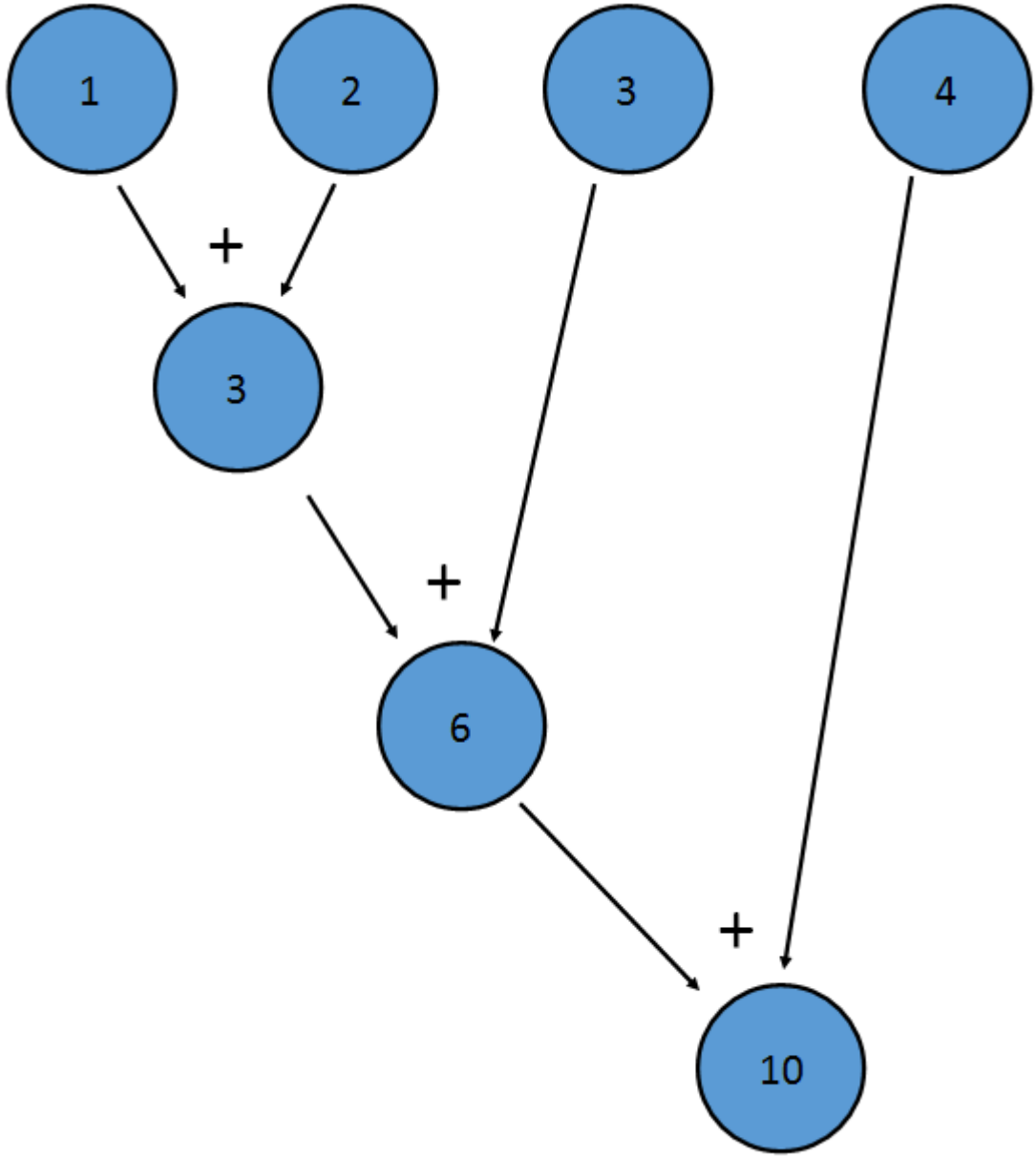
Verwenden Sie Spliterators.spliterator() oder Spliterators.spliteratorUnknownSize() , um einen Iterator in einen Stream zu konvertieren:

```
Iterator<String> iterator = Arrays.asList("A", "B", "C").iterator();
Spliterator<String> spliterator = Spliterators.spliteratorUnknownSize(iterator, 0);
Stream<String> stream = StreamSupport.stream(spliterator, false);
```

Reduktion mit Streams

Reduktion ist der Vorgang, bei dem ein binärer Operator auf jedes Element eines Streams angewendet wird, um einen Wert zu erhalten.

Die sum() Methode eines IntStream ist ein Beispiel für eine Reduzierung. es wird zusätzlich auf jeden Begriff des Streams angewendet, was zu einem endgültigen Wert führt:



Dies ist äquivalent zu (((1+2)+3)+4)

Die reduce eines Streams ermöglicht das Erstellen einer benutzerdefinierten Reduzierung. Es ist möglich, die reduce Methode zum Implementieren der sum() -Methode zu verwenden:

```
IntStream istr;

//Initialize istr

OptionalInt istr.reduce((a,b)->a+b);
```

Die Optional Version wird zurückgegeben, damit leere Streams entsprechend behandelt werden können.

Ein weiteres Beispiel für die Reduktion ist die Kombination einer Stream<LinkedList<T>> in einer einzelnen LinkedList<T> :

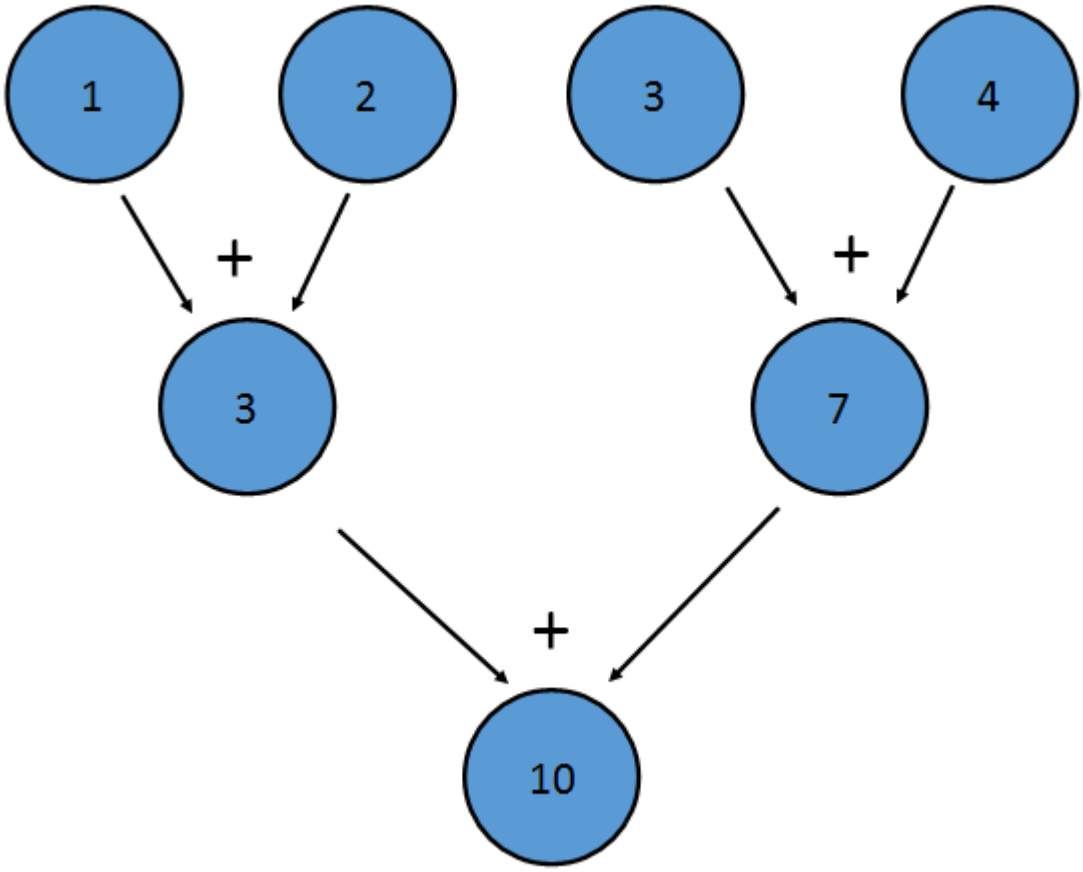
```
Stream<LinkedList<T>> listStream;

//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;
});
```

Sie können auch ein *Identitätselement angeben* . Beispielsweise ist das Identitätselement für die Addition 0, da x+0==x . Für die Multiplikation ist das Identitätselement 1, da x*1==x . Im obigen Fall handelt es sich bei dem Identitätselement um eine leere LinkedList<T> . Wenn Sie einer anderen Liste eine leere Liste hinzufügen, ändert sich die Liste, zu der Sie "hinzufügen", nicht:

```
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;
});
```

Wenn ein Identitätselement bereitgestellt wird, wird der Rückgabewert nicht in ein Optional Element eingeschlossen. Wenn dieses Element in einem leeren Stream aufgerufen wird, wird das Identitätselement von reduz reduce() zurückgegeben.

Der binäre Operator muss auch *assoziativ sein* , dh (a+b)+c==a+(b+c) . Dies liegt daran, dass die Elemente in beliebiger Reihenfolge reduziert werden können. Die obige Additionsreduktion könnte beispielsweise folgendermaßen durchgeführt werden:



Diese Reduktion entspricht dem Schreiben ((1+2)+(3+4)) . Die Eigenschaft der Assoziativität ermöglicht es auch, dass Java den Stream parallel reduziert. Ein Teil des Streams kann von jedem Prozessor reduziert werden, wobei das Ergebnis jedes Prozessors am Ende kombiniert wird.

Verknüpfen eines Streams mit einem einzelnen String

Ein häufig auftretender Anwendungsfall ist das Erstellen eines String aus einem Stream, bei dem die Stream-Elemente durch ein bestimmtes Zeichen getrennt werden. Collectors.joining() kann die Collectors.joining() -Methode verwendet werden, wie im folgenden Beispiel:

```
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);
```

Ausgabe:

```
     Apfel, Banane, Orange, Birne
```

Die Collectors.joining() -Methode kann auch für Prä- und Postfixes sorgen:

```
String result = fruitStream.filter(s -> s.contains("e"))
           .map(String::toUpperCase)
           .sorted()
           .collect(Collectors.joining(", ", "Fruits: ", "."));
```

```
System.out.println(result);
```

Ausgabe:

```
      Früchte: APPLE, ORANGE, BIRN.
```

Live auf Ideone

Streams online lesen: https://riptutorial.com/de/java/topic/88/streams

Kapitel 154: String Tokenizer

Einführung

Mit der Klasse java.util.StringTokenizer können Sie eine Zeichenfolge in Token aufteilen. Es ist ein einfacher Weg, die Zeichenfolge zu brechen.

Der Satz von Trennzeichen (die Zeichen, die Token trennen) kann entweder beim Erstellen oder auf Token-Basis angegeben werden.

Examples

StringTokenizer Nach Leerzeichen aufgeteilt

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

Ausgabe:

Apfel

Ball

Katze

Hund

StringTokenizer Nach Komma getrennt ','

```
public static void main(String args[]) {
        StringTokenizer st = new StringTokenizer("apple,ball cat,dog", ",");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
```

Ausgabe:

Apfel

Ball Katze

Hund

String Tokenizer online lesen: https://riptutorial.com/de/java/topic/10563/string-tokenizer

Kapitel 155: StringBuffer

Einführung

Einführung in die Java-StringBuffer-Klasse.

Examples

String-Pufferklasse

Schlüsselpunkte: -

- wird verwendet, um eine veränderliche (modifizierbare) Zeichenfolge zu erstellen.
- **Veränderlich** : - was geändert werden kann.
- ist threadsicher, dh mehrere Threads können nicht gleichzeitig darauf zugreifen.

Methoden: -

- public synchronized StringBuffer append (String s)
- public synchronisierter StringBuffer-Insert (int offset, String s)
- public synchronized StringBuffer replace (int startIndex, int endIndex, String str)
- public synchronized StringBuffer delete (int startIndex, int endIndex)
- public synchronized StringBuffer reverse ()
- öffentliche Kapazität ()
- public void sureCapacity (int minimumCapacity)
- öffentliches Zeichen charAt (int index)
- public int length ()
- public String substring (int beginIndex)
- public String-Teilzeichenfolge (int beginIndex, int endIndex)

Beispiel Zeigt den Unterschied zwischen der Implementierung von String und 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
 }
}
```

StringBuffer online lesen: https://riptutorial.com/de/java/topic/10757/stringbuffer

Kapitel 156: StringBuilder

Einführung

Die Java-StringBuilder-Klasse wird zum Erstellen einer veränderbaren (veränderbaren) Zeichenfolge verwendet. Die Java-StringBuilder-Klasse ist mit der StringBuffer-Klasse identisch, außer dass sie nicht synchronisiert ist. Es ist seit JDK 1.5 verfügbar.

Syntax

- neuer StringBuilder ()
- neuer StringBuilder (int. Kapazität)
- neuer StringBuilder (CharSequence-Sequenz)
- neuer StringBuilder (Builder für StringBuilder)
- neuer StringBuilder (String string)
- neuer StringJoiner (CharSequence-Trennzeichen)
- neuer StringJoiner (Trennzeichen CharSequence, Präfix CharSequence, Suffix CharSequence)

Bemerkungen

Das Erstellen eines neuen StringBuilder mit dem Typ char als Parameter würde dazu führen, dass der Konstruktor mit dem Argument int capacity und nicht mit dem 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

Wiederholen Sie einen String n-mal

Problem: Erstellen Sie einen String mit n Wiederholungen eines String s .

Der triviale Ansatz würde den String wiederholt verketten

```
final int n = ...
final String s = ...
String result = "";

for (int i = 0; i < n; i++) {
    result += s;
}
```

Dadurch werden n neue String-Instanzen mit 1 bis n Wiederholungen von s die zu einer Laufzeit von O(s.length() * n²) = O(s.length() * (1+2+...+(n-1)+n)) .

Um dies zu vermeiden StringBuilder sollte verwendet werden, die die ermöglicht die Erstellung von String in O(s.length() * n) statt:

```
final int n = ...
```

```
final String s = ...

StringBuilder builder = new StringBuilder();

for (int i = 0; i < n; i++) {
    builder.append(s);
}

String result = builder.toString();
```

Vergleich von StringBuffer, StringBuilder, Formatter und StringJoiner

Die Klassen StringBuffer , StringBuilder , Formatter und StringJoiner sind Java SE-Dienstprogrammklassen, die hauptsächlich zum Zusammenstellen von Zeichenfolgen aus anderen Informationen verwendet werden:

- Die StringBuffer Klasse ist seit Java 1.0 vorhanden und bietet verschiedene Methoden zum Erstellen und Ändern eines "Puffers", der eine Folge von Zeichen enthält.
- Die StringBuilder Klasse wurde in Java 5 hinzugefügt, um Leistungsprobleme mit der ursprünglichen StringBuffer Klasse zu StringBuffer . Die APIs für die beiden Klassen sind im Wesentlichen gleich. Der Hauptunterschied zwischen StringBuffer und StringBuilder besteht darin, dass ersterer threadsicher und synchronisiert ist und letzterer nicht.

Dieses Beispiel zeigt, wie StringBuilder verwendet werden kann:

```
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.
```

(Die StringBuffer Klasse wird auf dieselbe Weise verwendet: Ändern Sie einfach StringBuilder in StringBuffer oben).

Die Klassen StringBuffer und StringBuilder eignen sich sowohl zum Zusammenstellen als auch zum Ändern von Zeichenfolgen. dh sie bieten Methoden zum Ersetzen und Entfernen von Zeichen sowie zum Hinzufügen verschiedener Zeichen. Die verbleibenden zwei Klassen sind spezifisch für die Aufgabe des Zusammenstellens von Strings.

- Die Formatter Klasse wurde in Java 5 hinzugefügt und basiert auf der sprintf Funktion in der C-Standardbibliothek. Es dauert einen *Format* - String mit eingebetteten *Formatbezeich* und A - Sequenzen von anderen Argumenten, und erzeugt eine Zeichenfolge , die durch die Argumente in Text umzuwandeln und sie an der Stelle der Formatbezeich ersetzen. Die Details der Formatbezeichner geben an, wie die Argumente in Text umgewandelt werden.
- Die StringJoiner Klasse wurde in Java 8 hinzugefügt. Es ist ein Formatierungsprogramm für spezielle Zwecke, das eine Folge von Zeichenfolgen mit Trennzeichen zwischen ihnen formatiert. Es verfügt über eine *fließende* API und kann mit Java 8-Streams verwendet werden.

Hier einige typische Beispiele für die Verwendung von 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));
```

Die StringJoiner Klasse ist für die obige Task nicht ideal. StringJoiner wird hier ein Beispiel für das Formatieren eines Arrays von Strings angegeben.

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
for (String s : new String[]{"A", "B", "C"}) {
    sj.add(s);
}
System.out.println(sj);
// Prints "[A, B, C]"
```

Die Anwendungsfälle für die 4 Klassen können zusammengefasst werden:

- StringBuilder für alle Modifikationsaufgaben für String-Assemblys ODER-Strings geeignet.
- StringBuffer wird (nur) verwendet, wenn Sie eine Thread-sichere Version von StringBuilder benötigen.
- Formatter bietet eine reichhaltigere String-Formatierungsfunktion, ist jedoch nicht so effizient wie StringBuilder . Dies liegt daran, dass jeder Aufruf von Formatter.format(...) Folgendes umfasst:
    - Parsen des format - String,
    - Erstellen und Auffüllen eines *varargs*- Arrays und
    - autoboxing für alle primitiven Argumente.
- StringJoiner eine StringJoiner und effiziente Formatierung einer Folge von Strings mit Trennzeichen, eignet sich jedoch nicht für andere Formatierungsaufgaben.

StringBuilder online lesen: https://riptutorial.com/de/java/topic/1037/stringbuilder

Kapitel 157: sun.misc.Unsafe

Bemerkungen

Mit der Unsafe Klasse kann ein Programm Dinge Unsafe , die der Java-Compiler nicht zulässt. Bei normalen Programmen sollte Unsafe .

WARNUNGEN

1. Wenn Sie bei der Verwendung der Unsafe API einen Fehler machen, Unsafe Ihre Anwendungen dazu, dass die JVM abstürzt und / oder Symptome aufweist, die schwer zu diagnostizieren sind.
2. Die Unsafe API kann ohne vorherige Ankündigung geändert werden. Wenn Sie es in Ihrem Code verwenden, müssen Sie den Code möglicherweise ändern, wenn Sie die Java-Version ändern.

Examples

Instanziieren von sun.misc.Unsafe durch Reflektion

```
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 verfügt über einen privaten Konstruktor, und die statische Methode getUnsafe() wird durch eine Überprüfung des Klassenladers überwacht, um sicherzustellen, dass der Code mit dem primären Klassenlader geladen wurde. Daher besteht eine Methode zum Laden der Instanz darin, das statische Feld mithilfe von Reflektion zu erhalten.

Sun.misc.Unsafe über den Bootclasspath instanziieren

```
public class UnsafeLoader {
    public static Unsafe loadUnsafe() {
        return Unsafe.getUnsafe();
    }
}
```

Während dieses Beispiel kompiliert wird, schlägt es wahrscheinlich zur Laufzeit fehl, wenn die Unsafe-Klasse nicht mit dem primären Classloader geladen wurde. Um sicherzustellen, dass dies geschieht, sollte die JVM mit den entsprechenden Argumenten geladen werden, z.

```
java -Xbootclasspath:$JAVA_HOME/jre/lib/rt.jar:./UnsafeLoader.jar foo.bar.MyApp
```

Die Klasse foo.bar.MyApp kann dann UnsafeLoader.loadUnsafe() .

Instanz von unsicher erhalten

Unsafe wird als privates Feld gespeichert, auf das nicht direkt zugegriffen werden kann. Der Konstruktor ist privat und die einzige Methode für den Zugriff auf public static Unsafe getUnsafe() hat privilegierten Zugriff. Mithilfe von Reflektionen wird eine Problemumgehung durchgeführt, um private Felder zugänglich zu machen:

```
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;
}
```

Verwendung von unsicher

Einige Verwendungen von unsicher ist s:

Benutzen	API
Off-Heap / Direct-Speicherzuweisung, Neuzuweisung und Freigabe	`allocateMemory(bytes)` , `reallocateMemory(address, bytes)` und `freeMemory(address)`
Gedächtniszäune	`loadFence()` , `storeFence()` , `fullFence()`
Parkstrom-Thread	`park(isAbsolute, time)` , `unpark(thread)`
Direkter Feld- und / oder Speicherzugriff	`get*` und `put*` Methodenfamilie
Ungeprüfte Ausnahmen werfen	`throwException(e)`
CAS und Atomic Operations	`compareAndSwap*` Methodenfamilie
Speicher einrichten	`setMemory`
Flüchtige oder gleichzeitige Operationen	`putOrdered*` `get*Volatile` , `put*Volatile` , `putOrdered*`

Die Methoden von get und put sind relativ zu einem gegebenen Objekt. Wenn das Objekt null ist, wird es als absolute Adresse behandelt.

```
// 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);
```

Einige Methoden sind nur für int und longs definiert. Sie können diese Methoden für Floats und Doubles verwenden, indem Sie floatToRawIntBits , intBitsToFloat, doubleToRawLongBits , longBitsToDouble` verwenden

sun.misc.Unsafe online lesen: https://riptutorial.com/de/java/topic/6771/sun-misc-unsafe

Kapitel 158: Super Keyword

Examples

Super-Keyword-Verwendung mit Beispielen

Super Keyword spielt an drei Stellen eine wichtige Rolle

1. Konstruktorebene
2. Methodenebene
3. Variable Ebene

Konstruktorebene

super Schlüsselwort wird verwendet, um den Konstruktor der übergeordneten Klasse aufzurufen. Dieser Konstruktor kann ein Standardkonstruktor oder ein parametrisierter Konstruktor sein.

- Standardkonstruktor: super();
- Parametrisierter Konstruktor: 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();
  }
}
```

Hinweis : super() muss die erste Anweisung im Konstruktor sein, andernfalls wird die Kompilierungsfehlermeldung angezeigt.

Methodenebene

super Schlüsselwort kann auch bei Überschreiben von Methoden verwendet werden. super Schlüsselwort kann verwendet werden, um die übergeordnete Klassenmethode aufzurufen oder aufzurufen.

```
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();
   }
}
```

Hinweis: Wenn nicht Methode überschrieben , dann brauchen wir nicht verwenden super Schlüsselwort Elternklasse - Methode aufzurufen.

Variable Ebene

super wird verwendet, um auf die Instanzvariable der übergeordneten Klassenklasse zu verweisen. Im Falle der Vererbung kann die Basisklasse möglich sein, und die abgeleitete Klasse kann ähnliche Datenmitglieder haben. Um zwischen dem Datenmitglied der Basis- / Elternklasse und der abgeleiteten / Kindklasse zu unterscheiden, sind im Kontext der abgeleiteten Klasse die Basisklassendaten zu unterscheiden Mitgliedern muss ein super Schlüsselwort vorangestellt werden.

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

Anmerkung : Wenn wir kein super Schlüsselwort vor dem Namen des Basisklassendatenmitglieds schreiben, wird es als aktuelles Klassendatenmitglied bezeichnet, und das Basismitglied wird im Kontext der abgeleiteten Klasse ausgeblendet.

Super Keyword online lesen: https://riptutorial.com/de/java/topic/5764/super-keyword

Kapitel 159: ThreadLocal

Bemerkungen

Am besten für Objekte SimpleDateFormat , die beim Aufruf eines Anrufs auf Interna angewiesen sind, ansonsten aber staatenlos sind, wie SimpleDateFormat , Marshaller

Bei der Verwendung von Random ThreadLocal sollten Sie ThreadLocalRandom

Examples

ThreadLocal Java 8-Funktionsinitialisierung

```
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);
    }
}
```

Grundlegende ThreadLocal-Verwendung

Java ThreadLocal wird verwendet, um lokale Thread-Variablen zu erstellen. Es ist bekannt, dass Threads eines Objekts seine Variablen gemeinsam nutzen, sodass die Variable nicht threadsicher ist. Wir können die Synchronisierung für die Thread-Sicherheit verwenden, aber wenn wir die Synchronisation vermeiden möchten, können Sie mit ThreadLocal lokale Variablen für den Thread erstellen, dh, nur dieser Thread kann diese Variablen lesen oder schreiben, sodass die anderen Threads denselben Code ausführen kann nicht auf die ThreadLocal-Variablen zugreifen.

Dies kann verwendet werden, um ThreadLocal Variablen zu verwenden. in Situationen, in denen Sie einen Thread-Pool haben, wie zum Beispiel in einem Webdienst. Das Erstellen eines SimpleDateFormat Objekts für jede Anforderung ist beispielsweise sehr zeitaufwändig, und ein statisches Objekt kann nicht erstellt werden, da SimpleDateFormat nicht SimpleDateFormat ist. SimpleDateFormat können wir ein ThreadLocal erstellen, sodass Thread-sichere Vorgänge ausgeführt werden können, ohne dass SimpleDateFormat muss Zeit.

Der folgende Code zeigt, wie er verwendet werden kann:

Jeder Thread hat seine eigene ThreadLocal Variable. Mit den Methoden get() und set() können sie den Standardwert abrufen oder den lokalen Wert in Thread ändern.

ThreadLocal Instanzen sind normalerweise private statische Felder in Klassen, die einem Thread den ThreadLocal möchten.

Hier ein kleines Beispiel, das die Verwendung von ThreadLocal im Java-Programm zeigt und beweist, dass jeder Thread eine eigene Kopie der ThreadLocal Variablen hat.

```
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());
    }

}
```

Ausgabe:

```
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
```

Wie aus der Ausgabe hervorgeht, hat Thread-0 den Wert des Formatierers geändert, der Standardformatierer von Thread-2 ist jedoch mit dem initialisierten Wert identisch.

Mehrere Threads mit einem gemeinsamen Objekt

In diesem Beispiel haben wir nur ein Objekt, das jedoch von verschiedenen Threads gemeinsam genutzt wird. Eine normale Verwendung von Feldern zum Speichern des Zustands wäre nicht möglich, da der andere Thread dies auch sehen würde (oder wahrscheinlich nicht sehen würde).

```
public class Test {
    public static void main(String[] args) {
        Foo foo = new Foo();
        new Thread(foo, "Thread 1").start();
        new Thread(foo, "Thread 2").start();
    }
}
```

In Foo zählen wir von Null an. Anstatt den Status in einem Feld zu speichern, speichern wir unsere aktuelle Nummer im ThreadLocal-Objekt, das statisch zugänglich ist. Beachten Sie, dass die Synchronisierung in diesem Beispiel nicht mit der Verwendung von ThreadLocal zusammenhängt, sondern eine bessere Konsolenausgabe gewährleistet.

```
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) {
```

```
                }
            }
        }
    }
}
```

In der Ausgabe sehen wir, dass jeder Thread für sich selbst zählt und nicht den Wert des anderen verwendet:

```
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
```

ThreadLocal online lesen: https://riptutorial.com/de/java/topic/2001/threadlocal

Kapitel 160: ThreadPoolExecutor in MultiThreaded-Anwendungen verwenden.

Einführung

Beim Erstellen einer performanten und datengesteuerten Anwendung kann es sehr hilfreich sein, zeitintensive Aufgaben asynchron auszuführen und mehrere Aufgaben gleichzeitig auszuführen. In diesem Thema wird das Konzept der Verwendung von ThreadPoolExecutors zum gleichzeitigen Ausführen mehrerer ansynchroner Aufgaben beschrieben.

Examples

Ausführen von asynchronen Aufgaben, bei denen kein Rückgabewert mithilfe einer ausführbaren Klasseninstanz benötigt wird

Einige Anwendungen möchten möglicherweise so genannte "Fire & Forget" -Aufgaben erstellen, die periodisch ausgelöst werden können und keinen Wert zurückgeben müssen, der nach Abschluss der zugewiesenen Aufgabe zurückgegeben wird (z. B. alte Temp-Dateien löschen, Protokolle rotieren, automatisch speichern) Zustand).

In diesem Beispiel erstellen wir zwei Klassen: Eine, die die Runnable-Schnittstelle implementiert, und eine, die eine main () -Methode enthält.

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();
    }
}
```

Die Ausführung von AsyncExample1.main () führte zu folgender Ausgabe:

```
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
```

Anmerkungen zur Anmerkung: In der obigen Ausgabe sind einige Punkte zu beachten:

1. Die Aufgaben wurden nicht in einer vorhersagbaren Reihenfolge ausgeführt.
2. Da jede Aufgabe eine (pseudo) zufällige Zeit lang geschlafen hat, wurde sie nicht notwendigerweise in der Reihenfolge abgeschlossen, in der sie aufgerufen wurden.

Ausführen asynchroner Aufgaben, bei denen ein Rückgabewert mithilfe einer Instanz einer aufrufbaren Klasse erforderlich ist

Es ist häufig erforderlich, eine lang andauernde Aufgabe auszuführen und das Ergebnis dieser Aufgabe zu verwenden, sobald sie abgeschlossen ist.

In diesem Beispiel erstellen wir zwei Klassen: Eine, die die Callable <T> -Schnittstelle implementiert (wobei T der Typ ist, den wir zurückgeben möchten) und eine, die eine main () - Methode enthält.

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

Die Ausführung von AsyncExample2.main () führte zu folgender Ausgabe:

```
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
```

Bemerkungen der Anmerkung:

In der obigen Ausgabe sind einige Dinge zu beachten:

1. Bei jedem Aufruf von ExecutorService.submit () wurde eine Instanz von Future zurückgegeben, die zur späteren Verwendung in einer Liste gespeichert wurde
2. Future enthält eine Methode namens isDone (), mit der überprüft werden kann, ob unsere Aufgabe abgeschlossen wurde, bevor der Rückgabewert überprüft wird. Durch Aufrufen der Future.get () - Methode für eine noch nicht erledigte Zukunft wird der aktuelle Thread bis zum Abschluss der Task blockiert. Dadurch werden viele Vorteile zunichte gemacht, die sich aus der asynchronen Ausführung der Task ergeben.
3. Die Methode executorService.shutdown () wurde aufgerufen, bevor die Rückgabewerte der Future-Objekte geprüft wurden. Dies ist nicht erforderlich, wurde aber auf diese Weise gemacht, um zu zeigen, dass dies möglich ist. Die Methode executorService.shutdown () verhindert nicht den Abschluss von Aufgaben, die bereits an den ExecutorService übermittelt wurden, sondern verhindert, dass neue Aufgaben zur Warteschlange hinzugefügt werden.

Asynchrone Aufgaben inline mit Lambdas definieren

Ein gutes Softwaredesign maximiert häufig die Wiederverwendbarkeit von Code. In manchen Fällen kann es jedoch sinnvoll sein, asynchrone Aufgaben in Ihrem Code über Lambda-Ausdrücke zu definieren, um die Lesbarkeit des Codes zu verbessern.

In diesem Beispiel erstellen wir eine einzelne Klasse, die eine main () - Methode enthält. In dieser Methode verwenden wir Lambda-Ausdrücke, um Instanzen von Callable und Runnable <T> zu erstellen und auszuführen.

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);
    }
}
```

Bemerkungen der Anmerkung:

In der obigen Ausgabe sind einige Dinge zu beachten:

1. Lambda-Ausdrücke haben Zugriff auf Variablen und Methoden, die für den Gültigkeitsbereich verfügbar sind, in dem sie definiert sind, aber alle Variablen müssen endgültig (oder effektiv final) sein, um in einem Lambda-Ausdruck verwendet zu werden.
2. Wir müssen nicht angeben, ob unser Lambda-Ausdruck explizit ein Callable oder ein Runable <T> ist. Der Rückgabetyp wird automatisch vom Rückgabetyp abgeleitet.

ThreadPoolExecutor in MultiThreaded-Anwendungen verwenden. online lesen: https://riptutorial.com/de/java/topic/8646/threadpoolexecutor-in-multithreaded-anwendungen-verwenden-

Kapitel 161: TreeMap und TreeSet

Einführung

TreeMap und TreeSet sind grundlegende Java-Sammlungen, die in Java 1.2 hinzugefügt wurden. TreeMap ist eine **veränderliche** , **geordnete** Map Implementierung. In ähnlicher Weise ist TreeSet eine **veränderliche** , **geordnete** Set Implementierung.

TreeMap ist als rot-schwarzer Baum implementiert, der O(log n) Zugriffszeiten bietet. TreeSet wird mithilfe einer TreeMap mit Dummy-Werten implementiert.

Beide Sammlungen sind **nicht** threadsicher.

Examples

TreeMap eines einfachen Java-Typs

Zuerst erstellen wir eine leere Map und fügen einige Elemente ein:

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");
```

Sobald wir einige Elemente in der Karte haben, können wir einige Operationen ausführen:

```
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
```

Wir können die Kartenelemente auch mit einem Iterator oder einer foreach-Schleife durchlaufen. Beachten Sie, dass die Einträge gemäß ihrer natürlichen Reihenfolge gedruckt werden, nicht der Reihenfolge der Einfügung:

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 eines einfachen Java-Typs

Zuerst erstellen wir ein leeres Set und fügen einige Elemente ein:

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);
```

Sobald wir einige Elemente im Set haben, können wir einige Operationen ausführen:

```
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
```

Wir können die Kartenelemente auch mit einem Iterator oder einer foreach-Schleife durchlaufen. Beachten Sie, dass die Einträge gemäß ihrer natürlichen Reihenfolge gedruckt werden, nicht der Reihenfolge der Einfügung:

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 eines benutzerdefinierten Java-Typs

Da TreeMap s und TreeSet s Schlüssel / Elemente gemäß ihrer natürlichen Reihenfolge pflegen. TreeMap Schlüssel und TreeSet Elemente miteinander vergleichbar sein.

Angenommen, wir haben eine benutzerdefinierte Person :

```
public class Person  {

    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constuctors, getters, setters and various methods
}
```

Wenn wir es in einem TreeSet (oder einem Key in einer TreeMap) unverändert speichern:

```
TreeSet<Person2> set = ...
```

```
set.add(new Person(1,"first","last",Date.from(Instant.now())));
```

Dann stießen wir auf eine Ausnahme wie diese:

```
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)
```

Um dies zu beheben, nehmen wir an, dass wir Person nach der Reihenfolge ihrer IDs ordnen möchten (private int id). Wir können es auf zwei Arten tun:

1. Eine Lösung besteht darin, Person so zu ändern, dass die Comparable-Schnittstelle implementiert wird :

   ```
   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. Eine andere Lösung besteht darin, das TreeSet mit einem Comparator TreeSet :

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());
    }
});
```

Bei beiden Ansätzen gibt es jedoch zwei Einschränkungen:

1. Es ist **sehr wichtig** , keine Felder zu ändern, die zum TreeSet verwendet werden, nachdem eine Instanz in eine TreeSet / TreeMap eingefügt wurde. Wenn wir im obigen Beispiel die id einer Person ändern, die bereits in die Sammlung eingefügt wurde, kann dies zu unerwartetem Verhalten führen.

2. Es ist wichtig, den Vergleich richtig und konsistent durchzuführen. Wie im Javadoc :

   Der Implementierer muss sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) für alle x und y sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) . (Dies bedeutet, dass x.compareTo(y) eine Ausnahme y.compareTo(x) muss, wenn y.compareTo(x) eine Ausnahme y.compareTo(x) .)

   Der Implementierer muss außerdem sicherstellen, dass die Relation transitiv ist: (x.compareTo(y)>0 && y.compareTo(z)>0) impliziert x.compareTo(z)>0 .

   Schließlich muss der Implementierer sicherstellen, dass x.compareTo(y)==0 bedeutet,

dass sgn(x.compareTo(z)) == sgn(y.compareTo(z)) für alle z.

TreeMap- und TreeSet-Thread-Sicherheit

TreeMap und TreeSet sind **keine** Thread-sicheren Sammlungen, daher muss bei der Verwendung in Multithread-Programmen darauf geachtet werden.

Sowohl TreeMap als auch TreeSet sind sicher, wenn sie von mehreren Threads gleichzeitig gelesen werden. Wenn sie also von einem einzelnen Thread erstellt und gefüllt wurden (z. B. beim Start des Programms) und erst dann gelesen, aber nicht von mehreren Threads geändert werden, gibt es keinen Grund für die Synchronisierung oder das Sperren.

Wenn die Auflistung jedoch gleichzeitig gelesen und geändert wird oder von mehr als einem Thread gleichzeitig geändert wird, kann die Auflistung eine ConcurrentModificationException auslösen oder sich unerwartet verhalten. In diesen Fällen ist es unbedingt erforderlich, den Zugriff auf die Sammlung mit einem der folgenden Ansätze zu synchronisieren / sperren:

1. Verwenden von Collections.synchronizedSorted.. :

```
SortedSet<Integer> set = Collections.synchronizedSortedSet(new TreeSet<Integer>());
SortedMap<Integer,String> map = Collections.synchronizedSortedMap(new
TreeMap<Integer,String>());
```

   Dadurch wird eine SortedSet / SortedMap- Implementierung bereitgestellt , die von der eigentlichen Auflistung unterstützt und für ein Mutex-Objekt synchronisiert wird. Beachten Sie, dass dadurch alle Lese- und Schreibzugriffe auf die Sammlung mit einer einzigen Sperre synchronisiert werden, so dass selbst gleichzeitiges Lesen nicht möglich ist.

2. Durch manuelles Synchronisieren für ein Objekt wie die Sammlung selbst:

```
 TreeSet<Integer> set = new TreeSet<>();
```

   ...

```
//Thread 1
synchronized (set) {
    set.add(4);
}
```

   ...

```
//Thread 2
synchronized (set) {
    set.remove(5);
}
```

3. Verwenden Sie eine Sperre, z. B. eine 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();
```

Im Gegensatz zu den vorherigen Synchronisationsmethoden können bei Verwendung von ReadWriteLock mehrere Threads gleichzeitig aus der Map lesen.

TreeMap und TreeSet online lesen: https://riptutorial.com/de/java/topic/9905/treemap-und-treeset

Kapitel 162: Typumwandlung

Syntax

- TargetType target = (SourceType) -Quelle;

Examples

Nicht-numerisches primitives Casting

Der boolean Typ kann nicht in einen anderen primitiven Typ umgewandelt werden.

Ein char kann in / aus einem beliebigen numerischen Typ umgewandelt werden, indem die in Unicode angegebenen Codepunktezuordnungen verwendet werden. Ein char wird im Speicher als vorzeichenloser 16-Bit-Integerwert (2 Byte) dargestellt, sodass beim Umsetzen in byte (1 Byte) 8 dieser Bits gelöscht werden (dies ist für ASCII-Zeichen sicher). Die Utility-Methoden der Character Klasse verwenden int (4 Bytes) für die Übertragung zu / von Codepunktwerten, eine short Angabe (2 Bytes) würde jedoch auch zum Speichern eines Unicode-Codepunkts ausreichen.

```
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
```

Numerisches primitives Casting

Numerische Primitive können auf zwei Arten gegossen werden. *Implizite* Umwandlung erfolgt, wenn der Quelltyp einen kleineren Bereich als den Zieltyp hat.

```
//Implicit casting
byte byteVar = 42;
short shortVar = byteVar;
int intVar = shortVar;
long longVar = intvar;
float floatVar = longVar;
double doubleVar = floatVar;
```

Ein explizites Casting muss durchgeführt werden, wenn der Quelltyp einen größeren Bereich als den Zieltyp hat.

```
//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;
```

Wenn Sie Gleitkomma-Grundelemente (float , double) in Ganzzahl-Grundelemente konvertieren, wird die Anzahl **abgerundet** .

Objektguss

Wie bei Grundelementen können Objekte sowohl explizit als auch implizit umgewandelt werden.

Implizites Casting tritt ein, wenn der Quelltyp den Zieltyp erweitert oder implementiert (Casting in eine Superklasse oder Schnittstelle).

Ein explizites Casting muss durchgeführt werden, wenn der Quelltyp durch den Zieltyp erweitert oder implementiert wird (Casting auf einen Subtyp). Dies kann eine Laufzeitausnahme (ClassCastException) erzeugen, wenn das umgesetzte Objekt nicht vom ClassCastException (oder vom Untertyp des Ziels) ist.

```
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)
```

Grundlegende numerische Promotion

```
 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.
}
```

Testen, ob ein Objekt mit instanceof umgewandelt werden kann

Java stellt den Operator instanceof bereit, um zu testen, ob ein Objekt einen bestimmten Typ oder eine Unterklasse dieses Typs aufweist. Das Programm kann dann wählen, ob dieses Objekt entsprechend umgewandelt werden soll oder nicht.

```
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.
}
```

Typumwandlung online lesen: https://riptutorial.com/de/java/topic/1392/typumwandlung

Kapitel 163: Unit Testing

Einführung

Komponententests sind ein wesentlicher Bestandteil der testgetriebenen Entwicklung und ein wichtiges Merkmal für den Aufbau jeder robusten Anwendung. In Java werden Unit-Tests fast ausschließlich mit externen Bibliotheken und Frameworks durchgeführt, von denen die meisten über ein eigenes Dokumentations-Tag verfügen. Dieser Stub dient dazu, den Leser mit den verfügbaren Werkzeugen und deren Dokumentation vertraut zu machen.

Bemerkungen

Unit Test Frameworks

Es gibt zahlreiche Frameworks für Unit-Tests in Java. Die mit Abstand beliebteste Option ist JUnit. Es ist unter folgendem dokumentiert:

JUnit

JUnit4 - *Vorgeschlagenes Tag für JUnit4-Funktionen; noch nicht implementiert*

Andere Unit-Test-Frameworks sind vorhanden und verfügen über Dokumentation:

TestNG

Testgeräte für Einheiten

Es gibt mehrere andere Tools, die für die Prüfung von Einheiten verwendet werden:

Mockito - Spottendes Gerüst; ermöglicht das Nachahmen von Objekten. Nützlich, um das **erwartete** Verhalten einer externen Einheit im Test einer bestimmten Einheit nachzuahmen, um das Verhalten der externen Einheit nicht mit den Tests der jeweiligen Einheit zu verknüpfen.

JBehave - BDD Framework. Ermöglicht die Verknüpfung von Tests mit dem Benutzerverhalten (Validierung der Anforderungen / Szenarien). *Zum Zeitpunkt des Schreibens ist kein Dokument verfügbar. Hier ist ein externer Link .*

Examples

Was ist Unit Testing?

Dies ist ein bisschen eine Grundierung. Es wird meistens eingesetzt, weil die Dokumentation ein Beispiel haben muss, auch wenn sie als Stub-Artikel gedacht ist. Wenn Sie bereits Grundlagen zum Testen von Einheiten kennen, können Sie mit den Anmerkungen fortfahren, in denen bestimmte Rahmenbedingungen erwähnt werden.

Unit-Tests stellen sicher, dass sich ein bestimmtes Modul wie erwartet verhält. In großen Anwendungen ist die Gewährleistung der ordnungsgemäßen Ausführung von Modulen im Vakuum ein wesentlicher Bestandteil der Gewährleistung der Anwendungstreue.

Betrachten Sie das folgende (triviale) Pseudo-Beispiel:

```
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;
    }
  }
}
```

Dieses Beispiel ist also trivial; DataReader erhält die Daten aus einer Datei und leitet sie an den Capitalizer . Dieser konvertiert alle Zeichen in Großbuchstaben, die dann an den Consumer . Der DataReader ist jedoch eng mit unserer Anwendungsumgebung verbunden. DataReader verschieben wir den Test dieser Kette, bis wir bereit sind, eine Testversion bereitzustellen.

getVal() , irgendwo auf dem Weg in einer Version wurde aus unbekannten Gründen die getVal() Methode in Capitalizer geändert, toUpperCase() ein toUpperCase() String in einen toLowerCase() String zurückgegeben wurde:

```
  // Another team's module.
  class Capitalizer {
    ...

    public String getVal() {
      return dr.readVal().toLowerCase();
    }
  }
```

Dies bricht eindeutig das erwartete Verhalten. Aufgrund der mühsamen Prozesse, die mit der

Ausführung des DataReader , werden wir dies erst bei unserer nächsten DataReader feststellen. Tage / Wochen / Monate vergehen also mit diesem Fehler in unserem System, und der Produktmanager sieht dies und wendet sich sofort an Sie, den Teamleiter, der mit dem Consumer . "Warum passiert das? Was hast du geändert?" Offensichtlich bist du ahnungslos. Du hast keine Ahnung, was los ist. Sie haben keinen Code geändert, der dies berühren sollte. Warum ist es plötzlich kaputt?

Nach einer Diskussion zwischen den Teams und der Zusammenarbeit wird das Problem verfolgt und das Problem gelöst. Aber es wirft die Frage auf; Wie konnte das verhindert werden?

Es gibt zwei offensichtliche Dinge:

Tests müssen automatisiert werden

Unser Vertrauen auf manuelle Tests ließ diesen Fehler viel zu lange unbemerkt bleiben. Wir brauchen eine Möglichkeit, den Prozess zu automatisieren, bei dem Fehler **sofort** eingeführt werden. Keine 5 Wochen mehr. Nicht in 5 Tagen. Keine 5 Minuten von jetzt an. Jetzt sofort.

Sie müssen wissen, dass ich in diesem Beispiel einen **sehr banalen** Fehler zum Ausdruck gebracht habe, der eingeführt und nicht bemerkt wurde. In einer industriellen Anwendung, bei der Dutzende von Modulen ständig aktualisiert werden, können sich diese überall einschleichen. Sie fixieren etwas mit einem Modul, nur um zu erkennen, dass das Verhalten, auf das Sie "repariert" haben, auf eine andere Weise (intern oder extern) verwendet wurde.

Ohne strenge Validierung schleichen sich die Dinge in das System ein. Es ist möglich, dass, wenn man es weit genug vernachlässigt, dies so viel zusätzlichen Aufwand mit sich bringt, Änderungen zu korrigieren (und diese Fixes dann zu korrigieren usw.), dass ein Produkt tatsächlich mit der verbleibenden Arbeit **zunimmt**, je mehr Aufwand vorhanden ist. Sie möchten nicht in dieser Situation sein.

Tests müssen feinkörnig sein

Das zweite Problem, das in unserem obigen Beispiel festgestellt wurde, ist die Zeit, die zum Aufspüren des Fehlers benötigt wurde. Der Produktmanager hat Sie mit einem Ping versehen, als die Tester dies bemerkten, Sie haben untersucht und festgestellt, dass der Capitalizer scheinbar schlechte Daten zurückgibt, Sie haben das Capitalizer Team mit Ihren Befunden, Ihren Ermittlungen usw. etc. angerufen.

Der gleiche Punkt, den ich oben bezüglich der Menge und Schwierigkeit dieses trivialen Beispiels angesprochen habe, trifft hier zu. Natürlich kann jeder, der mit Java einigermaßen vertraut ist, das eingeführte Problem schnell finden. Es ist jedoch oft viel schwieriger, Probleme aufzuspüren und zu kommunizieren. Vielleicht hat Ihnen das Capitalizer Team einen JAR ohne Quelle zur Verfügung gestellt. Möglicherweise befinden sie sich auf der anderen Seite der Welt und die Kommunikationszeiten sind sehr begrenzt (z. B. für E-Mails, die einmal täglich gesendet werden). Es kann zu Fehlern führen, die Wochen oder länger dauern, bis sie aufgespürt wurden (und wiederum kann es mehrere davon für eine bestimmte Version geben).

Um dem entgegenzuwirken, wünschen wir uns strenge Tests auf einem möglichst **feinen** Niveau (Sie wollen auch grobkörnige Tests, um sicherzustellen, dass die Module ordnungsgemäß zusammenwirken, aber das ist hier nicht unser Schwerpunkt). Wir möchten genau festlegen, wie alle nach außen gerichteten Funktionen (mindestens) funktionieren, und Tests für diese Funktionen durchführen.

Komponententest eingeben

Stellen Sie sich vor, wir hätten einen Test, der insbesondere sicherstellt, dass die getVal() Methode von Capitalizer eine getVal() für eine bestimmte Eingabezeichenfolge getVal() . Stellen Sie sich außerdem vor, dass der Test ausgeführt wurde, bevor wir überhaupt Code geschrieben haben. Der Fehler in das System eingeführt (das heißt, toUpperCase() mit ersetzt toLowerCase()) würde keine Probleme verursachen , weil der Fehler würde nie in das System eingebracht **werden.** Wir würden es in einem Test fangen, der Entwickler würde (hoffentlich) ihren Fehler erkennen und eine alternative Lösung finden, wie die beabsichtigte Wirkung eingeführt werden kann.

Es wurden einige Auslassungen gemacht, **wie** diese Tests implementiert werden können, aber diese werden in der Framework-spezifischen Dokumentation (verlinkt in den Anmerkungen) behandelt.

Hoffentlich dient dies als Beispiel dafür, **warum** Unit-Tests so wichtig sind.

Unit Testing online lesen: https://riptutorial.com/de/java/topic/8155/unit-testing

Kapitel 164: Unveränderliche Klasse

Einführung

Unveränderliche Objekte sind Instanzen, deren Zustand sich nach der Initialisierung nicht ändert. Beispielsweise ist String eine unveränderliche Klasse, und sobald sie instanziiert wird, ändert sich der Wert nie.

Bemerkungen

Einige unveränderliche Klassen in Java:

1. java.lang.String
2. Die Wrapper-Klassen für die primitiven Typen: 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. Die meisten Aufzählungsklassen sind unveränderlich, dies hängt jedoch vom konkreten Fall ab.
4. java.math.BigInteger und java.math.BigDecimal (zumindest Objekte dieser Klassen selbst)
5. java.io.File. Beachten Sie, dass dies ein Objekt außerhalb der VM (eine Datei auf dem lokalen System) darstellt, das möglicherweise existiert oder nicht vorhanden ist und über einige Methoden verfügt, die den Status dieses externen Objekts ändern und abfragen. Das File-Objekt selbst bleibt jedoch unveränderlich.

Examples

Regeln zum Definieren unveränderlicher Klassen

Die folgenden Regeln definieren eine einfache Strategie zum Erstellen unveränderlicher Objekte.

1. Stellen Sie keine "Setter" -Methoden bereit - Methoden, die Felder oder Objekte ändern, auf die Felder verweisen.
2. Machen Sie alle Felder endgültig und privat.
3. Lassen Sie nicht zu, dass Unterklassen Methoden überschreiben. Am einfachsten ist es, die Klasse als final zu deklarieren. Ein komplexerer Ansatz besteht darin, den Konstruktor privat zu machen und Instanzen in Factory-Methoden zu erstellen.
4. Wenn die Instanzfelder Verweise auf veränderliche Objekte enthalten, dürfen diese Objekte nicht geändert werden:
5. Stellen Sie keine Methoden bereit, die die veränderbaren Objekte ändern.
6. Verweise auf die veränderlichen Objekte nicht freigeben. Speichern Sie niemals Verweise auf externe, veränderliche Objekte, die an den Konstruktor übergeben werden. Erstellen Sie ggf. Kopien und speichern Sie Verweise auf die Kopien. Erstellen Sie auf ähnliche Weise Kopien Ihrer internen veränderbaren Objekte, wenn dies erforderlich ist, um zu vermeiden, dass die Originale in Ihren Methoden zurückgegeben werden.

Beispiel ohne mutable refs

```
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);
    }
}
```

Beispiel mit veränderlichen Refs

In diesem Fall ist die Klasse Point veränderbar und einige Benutzer können den Objektstatus dieser Klasse ändern.

```
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;
    }
```

Was ist der Vorteil der Unveränderlichkeit?

Der Vorteil der Unveränderlichkeit liegt in der Parallelität. Es ist schwierig, die Korrektheit in veränderlichen Objekten aufrechtzuerhalten, da mehrere Threads versuchen könnten, den Zustand des gleichen Objekts zu ändern, was dazu führt, dass einige Threads einen unterschiedlichen Zustand des gleichen Objekts sehen, abhängig vom Zeitpunkt der Lese- und Schreibvorgänge Objekt.

Mit einem unveränderlichen Objekt kann sichergestellt werden, dass alle Threads, die das Objekt

betrachten, denselben Status sehen, da sich der Status eines unveränderlichen Objekts nicht ändert.

Unveränderliche Klasse online lesen:
https://riptutorial.com/de/java/topic/10561/unveranderliche-klasse

Kapitel 165: Unveränderliche Objekte

Bemerkungen

Unveränderliche Objekte haben einen festen Status (keine Setter), daher muss der Status zum Zeitpunkt der Objekterstellung bekannt sein.

Obwohl dies nicht technisch erforderlich ist, sollten alle Felder final . Dadurch wird die unveränderliche Klasse Thread-sicher (vgl. Java Concurrency in Practice, 3.4.1).

Die Beispiele zeigen mehrere Muster, die dabei helfen können.

Examples

Erstellen einer unveränderlichen Version eines Typs durch defensives Kopieren.

Einige grundlegende Typen und Klassen in Java sind grundsätzlich veränderbar. Zum Beispiel sind alle Array-Typen veränderbar, ebenso Klassen wie java.util.Data . Dies kann in Situationen schwierig sein, in denen ein unveränderlicher Typ vorgeschrieben ist.

Eine Möglichkeit, damit umzugehen, besteht darin, einen unveränderlichen Wrapper für den veränderbaren Typ zu erstellen. Hier ist ein einfacher Wrapper für ein Array von Ganzzahlen

```
public class ImmutableIntArray {
    private final int[] array;

    public ImmutableIntArray(int[] array) {
        this.array = array.clone();
    }

    public int[] getValue() {
        return this.clone();
    }
}
```

Diese Klasse verwendet *defensives Kopieren* , um den veränderbaren Zustand (das int[]) von jedem Code zu isolieren, der ihn möglicherweise verändert:

- Der Konstruktor erstellt mit clone() eine eindeutige Kopie des Parameter-Arrays. Wenn der Aufrufer des Konstruktors das Parameterarray anschließend geändert hat, hat dies keinen Einfluss auf den Status des ImmutableIntArray .
- Die getValue() -Methode verwendet auch clone() , um das zurückgegebene Array zu erstellen. Wenn der Aufrufer das Ergebnis-Array ändern würde, hat dies keinen Einfluss auf den Status des ImmutableIntArray .

Wir könnten auch ImmutableIntArray Methoden hinzufügen, um schreibgeschützte Operationen für das umschlossene Array auszuführen. z. B. Länge abrufen, Wert an einem bestimmten Index abrufen usw.

Beachten Sie, dass ein auf diese Weise implementierter unveränderlicher Wrapper-Typ nicht mit dem Originaltyp kompatibel ist. Sie können das erstere nicht einfach durch das Letztere ersetzen.

Das Rezept für eine unveränderliche Klasse

Ein unveränderliches Objekt ist ein Objekt, dessen Status nicht geändert werden kann. Eine unveränderliche Klasse ist eine Klasse, deren Instanzen nach Entwurf und Implementierung unveränderlich sind. Die Java-Klasse, die am häufigsten als Beispiel für Unveränderlichkeit dargestellt wird, ist java.lang.String .

Das folgende ist ein stereotypes Beispiel:

```
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;
    }
}
```

Eine Variante davon ist, den Konstruktor als private zu deklarieren und stattdessen eine public static Factory-Methode bereitzustellen.

Das *Standardrezept* für eine unveränderliche Klasse lautet wie folgt:

- Alle Eigenschaften müssen in den Konstruktoren oder Factory-Methoden festgelegt werden.
- Es sollte keine Setter geben.
- Wenn aus Gründen der Schnittstellenkompatibilität Setter erforderlich sind, sollten sie entweder nichts tun oder eine Ausnahme auslösen.
- Alle Eigenschaften sollten als private und final deklariert werden.
- Für alle Eigenschaften, die Verweise auf veränderbare Typen sind:
    - Die Eigenschaft sollte mit einer tiefen Kopie des über den Konstruktor übergebenen Werts initialisiert werden
    - Der Getter der Eigenschaft sollte eine tiefe Kopie des Eigenschaftswerts zurückgeben.
- Die Klasse sollte als final deklariert werden, um zu verhindern, dass jemand eine veränderliche Unterklasse einer unveränderlichen Klasse erstellt.

Ein paar andere Dinge zu beachten:

- Die Unveränderlichkeit verhindert nicht, dass das Objekt nullfähig ist. null kann null einer String Variablen zugewiesen werden.
- Wenn die Eigenschaften einer unveränderlichen Klasse als final deklariert werden, sind Instanzen von Natur aus Thread-sicher. Dies macht unveränderliche Klassen zu einem guten Baustein für die Implementierung von Multithread-Anwendungen.

Typische Designfehler, die verhindern, dass eine Klasse unveränderlich ist

Einige Setter verwenden, ohne alle erforderlichen Eigenschaften in den Konstruktoren festzulegen

```
public final class Person { // example of a bad immutability
    private final String name;
    private final String surname;
    public Person(String name) {
        this.name = name;
      }
    public String getName() { return name;}
    public String getSurname() { return surname;}
    public void setSurname(String surname) { this.surname = surname); }
}
```

Es ist leicht zu zeigen, dass die Person nicht unveränderlich ist:

```
Person person = new Person("Joe");
person.setSurname("Average"); // NOT OK, change surname field after creation
```

Um dies zu beheben, löschen Sie einfach setSurname() und setzen den Konstruktor wie folgt um:

```
public Person(String name, String surname) {
    this.name = name;
    this.surname = surname;
  }
```

Instanzvariablen werden nicht als privat und als final markiert

Schauen Sie sich die folgende Klasse an:

```
public final class Person {
    public String name;
    public Person(String name) {
        this.name = name;
     }
    public String getName() {
        return name;
    }

}
```

Der folgende Ausschnitt zeigt, dass die oben genannte Klasse nicht unveränderlich ist:

```
Person person = new Person("Average Joe");
person.name = "Magic Mike"; // not OK, new name for person after creation
```

Um dies zu beheben, kennzeichnen Sie einfach name property als private und final .

Offenlegen eines veränderlichen Objekts der Klasse in einem Getter

Schauen Sie sich die folgende Klasse an:

```
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 Klasse scheint auf den ersten Blick unveränderlich zu sein. Dies ist jedoch nicht der folgende Code:

```
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"
```

Dies geschah , weil eine Änderung der Referenzliste zurück von getNames() die aktuelle Liste von ändern können Names .

Um dies zu beheben, vermeiden einfach Referenzen Rückkehr , die Klasse veränderbare Objekte verweisen *entweder* durch defensive Kopien zu machen, wie folgt:

```
public List<String> getNames() {
   return new ArrayList<String>(this.names); // copies elements
}
```

oder indem Sie Getter so erstellen, dass nur andere *unveränderliche Objekte* und *Primitive* wie folgt zurückgegeben werden:

```
public String getName(int index) {
    return names.get(index);
}
public int size() {
    return names.size();
}
```

Einfügen eines Konstruktors mit Objekten, die außerhalb der unveränderlichen Klasse geändert werden können

Dies ist eine Variation des vorherigen Fehlers. Schauen Sie sich die folgende Klasse an:

```
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();
    }
}
```

Als Names Klasse zuvor scheint auch die NewNames Klasse auf den ersten Blick unveränderlich zu sein. NewNames ist jedoch nicht der NewNames , sondern der folgende Ausschnitt zeigt das Gegenteil:

```
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"
```

Um dies zu beheben, wie beim vorherigen Fehler, erstellen Sie einfach defensive Kopien des Objekts, ohne es direkt der unveränderlichen Klasse zuzuweisen. Der Konstruktor kann also wie folgt geändert werden:

```
    public Names(List<String> names) {
        this.names = new ArrayList<String>(names);
    }
```

Die Methoden der Klasse überschreiben lassen

Schauen Sie sich die folgende Klasse an:

```
public class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
      }
    public String getName() { return name;}
}
```

Person scheint auf den ersten Blick unveränderlich zu sein, es wird jedoch eine neue Unterklasse von Person definiert:

```
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;
    }
}
```

Jetzt kann die Mutabilität von Person (un) durch Polymorphismus ausgenutzt werden, indem die neue Unterklasse verwendet wird:

```
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
```

Um dies zu beheben, markieren Sie die Klasse *entweder* als final sodass sie nicht erweitert werden kann, *oder* deklarieren Sie alle Konstruktoren als private .

Unveränderliche Objekte online lesen:
https://riptutorial.com/de/java/topic/2807/unveranderliche-objekte

Kapitel 166: Varargs (variables Argument)

Bemerkungen

Mit einem Argument der Methode varargs können Aufrufer dieser Methode mehrere Argumente des angegebenen Typs angeben, die jeweils als separates Argument dienen. Sie wird in der Methodendeklaration durch drei ASCII-Punkte (...) nach dem Basistyp angegeben.

Die Methode selbst erhält diese Argumente als einzelnes Array, dessen Elementtyp der Typ des varargs-Arguments ist. Das Array wird automatisch erstellt (obwohl Aufrufer weiterhin ein explizites Array übergeben dürfen, anstatt mehrere Werte als separate Methodenargumente zu übergeben).

Regeln für varargs:

1. Varargs muss das letzte Argument sein.
2. Es kann nur eine Varargs in der Methode geben.

Sie müssen die obigen Regeln befolgen, andernfalls gibt das Programm einen Kompilierungsfehler aus.

Examples

Angabe eines varargs-Parameters

```
void doSomething(String... strings) {
    for (String s : strings) {
        System.out.println(s);
    }
}
```

Die drei Punkte nach dem Typ des letzten Parameters geben an, dass das letzte Argument als Array oder als Folge von Argumenten übergeben werden kann. Varargs können nur in der endgültigen Argumentposition verwendet werden.

Arbeiten mit Varargs-Parametern

Wenn Sie varargs als Parameter für eine Methodendefinition verwenden, können Sie entweder ein Array oder eine Folge von Argumenten übergeben. Wenn eine Folge von Argumenten übergeben wird, werden diese automatisch in ein Array umgewandelt.

Dieses Beispiel zeigt, wie ein Array und eine Folge von Argumenten an die Methode printVarArgArray() werden und wie sie im Code innerhalb der Methode identisch behandelt werden:

```
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);
    }
}
```

Ausgabe:

```
10,20,
5,6,5,8,6,31
```

Wenn Sie die Methode auf diese Weise definieren, werden Fehler beim Kompilieren angezeigt.

```
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
```

Varargs (variables Argument) online lesen: https://riptutorial.com/de/java/topic/1948/varargs--variables-argument-

Kapitel 167: Verarbeiten

Bemerkungen

Beachten Sie, dass die API empfiehlt, dass ab Version 1.5 die bevorzugte Methode zum Erstellen eines Prozesses die Verwendung von ProcessBuilder.start() .

Eine weitere wichtige Anmerkung ist, dass der von waitFor erzeugte Exit-Wert von dem ausgeführten Programm / Skript abhängig ist. Beispielsweise unterscheiden sich die von **calc.exe** erzeugten **Exitcodes** von **notepad.exe** .

Examples

Einfaches Beispiel (Java-Version <1.5)

In diesem Beispiel wird der Windows-Rechner aufgerufen. Es ist wichtig zu wissen, dass der Exit-Code entsprechend dem aufgerufenen Programm / Skript variiert.

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

Verwendung der ProcessBuilder-Klasse

Die ProcessBuilder-Klasse vereinfacht das Senden eines Befehls über die Befehlszeile. Alles, was es braucht, ist eine Liste von Strings, aus denen sich die Befehle zusammensetzen. Sie rufen einfach die start () - Methode in Ihrer ProcessBuilder-Instanz auf, um den Befehl auszuführen.

Wenn Sie ein Programm namens Add.exe haben, das zwei Argumente verwendet und diese hinzufügt, würde der Code etwa so aussehen:

```
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();
}
```

Einige Dinge zu beachten:

- Das Array von Befehlen muss ein String-Array sein
- Die Befehle müssen in der Reihenfolge (im Array) sein, in der sie sich befinden, wenn Sie das Programm in der Befehlszeile selbst aufrufen (dh der Name der EXE-Datei kann nicht nach dem ersten Argument stehen
- Beim Einstellen des Arbeitsverzeichnisses müssen Sie ein File-Objekt übergeben und nicht nur den Dateinamen als String

Blockieren vs. Nicht-Blockieren von Anrufen

Wenn Sie die Befehlszeile aufrufen, sendet das Programm im Allgemeinen den Befehl und fährt dann mit der Ausführung fort.

Sie können jedoch warten, bis das aufgerufene Programm beendet ist, bevor Sie mit der Ausführung fortfahren (z. B. schreibt das aufgerufene Programm Daten in eine Datei, und Ihr Programm benötigt diese, um auf diese Daten zuzugreifen.)

Dies kann leicht durch Aufrufen der waitFor() Methode aus der zurückgegebenen Process Instanz erreicht werden.

Anwendungsbeispiel:

```
//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

Das direkte Starten externer Prozesse von Java mit der rohen java.lang.ProcessBuilder-API kann etwas umständlich sein. Die Apache Commons Exec-Bibliothek macht es etwas einfacher. Die ch.vorburger.exec-Bibliothek erweitert Commons Exec noch weiter, um es wirklich bequem zu machen:

```
 ManagedProcess proc = new ManagedProcessBuilder("path-to-your-executable-binary")
     .addArgument("arg1")
     .addArgument("arg2")
     .setWorkingDirectory(new File("/tmp"))
     .setDestroyOnShutdown(true)
```

```
     .setConsoleBufferMaxLines(7000)
     .build();

proc.start();
int status = proc.waitForExit();
int status = proc.waitForExitMaxMsOrDestroy(3000);
String output = proc.getConsole();

proc.startAndWaitForConsoleMessageMaxMs("started!", 7000);
// use service offered by external process...
proc.destroy();
```

Pitfall: Runtime.exec, Process und ProcessBuilder verstehen die Shell-Syntax nicht

Mit den Runtime.exec(String ...) und Runtime.exec(String) können Sie einen Befehl als externen Prozess ausführen. [1] In der ersten Version geben Sie den Befehlsnamen und die Befehlsargumente als separate Elemente des String-Arrays an. Die Java-Laufzeitumgebung fordert das Betriebssystemlaufzeitsystem auf, den externen Befehl zu starten. Die zweite Version ist täuschend einfach zu bedienen, weist jedoch einige Fallstricke auf.

Zunächst einmal ein Beispiel für die sichere Verwendung von exec(String) :

```
Process p = Runtime.exec("mkdir /tmp/testDir");
p.waitFor();
if (p.exitValue() == 0) {
    System.out.println("created the directory");
}
```

Leerzeichen in Pfadnamen

Angenommen, wir verallgemeinern das obige Beispiel, um ein beliebiges Verzeichnis erstellen zu können:

```
Process p = Runtime.exec("mkdir " + dirPath);
// ...
```

Dies funktioniert normalerweise, dirPath jedoch fehl, wenn dirPath beispielsweise "/ home / user / My Documents" ist. Das Problem ist, dass exec(String) die Zeichenfolge in einen Befehl und Argumente aufteilt, indem einfach nach Leerzeichen gesucht wird. Die Befehlszeichenfolge:

```
"mkdir /home/user/My Documents"
```

wird aufgeteilt in:

```
"mkdir", "/home/user/My", "Documents"
```

Dies führt dazu, dass der Befehl "mkdir" fehlschlägt, da er ein Argument erwartet, nicht zwei.

Angesichts dessen versuchen einige Programmierer, den Pfadnamen in Anführungszeichen zu setzen. Das funktioniert auch nicht:

```
"mkdir \"/home/user/My Documents\""
```

wird aufgeteilt in:

```
"mkdir", "\"/home/user/My", "Documents\""
```

Die zusätzlichen doppelten Anführungszeichen, die hinzugefügt wurden, um die Leerzeichen zu

"zitieren", werden wie alle anderen Zeichen ohne Leerzeichen behandelt. In der Tat wird alles, was wir zitieren oder aus den Räumen entkommen, versagen.

Um mit diesen Problemen exec(String ...) zu werden, verwenden Sie die exec(String ...) Überladung.

```
Process p = Runtime.exec("mkdir", dirPath);
// ...
```

Dies funktioniert, wenn dirpath Leerzeichen enthält, da diese Überladung von exec nicht versucht, die Argumente aufzuteilen. Die Zeichenfolgen werden unverändert an den Betriebssystem-exec Systemaufruf übergeben.

Umleitung, Pipelines und andere Shell-Syntax

Angenommen, wir möchten die Eingabe oder Ausgabe eines externen Befehls umleiten oder eine Pipeline ausführen. Zum Beispiel:

```
Process p = Runtime.exec("find / -name *.java -print 2>/dev/null");
```

oder

```
Process p = Runtime.exec("find source -name *.java | xargs grep package");
```

(Das erste Beispiel listet die Namen aller Java-Dateien im Dateisystem auf und das zweite druckt die package [2] in den Java-Dateien in der "Quell" -Struktur.)

Diese werden nicht wie erwartet funktionieren. Im ersten Fall wird der Befehl "find" mit "2> / dev / null" als Befehlsargument ausgeführt. Es wird nicht als Weiterleitung interpretiert. Im zweiten Beispiel werden das Pipe-Zeichen ("|") und die darauf folgenden Werke dem Befehl "find" übergeben.

Das Problem hierbei ist, dass die exec Methoden und ProcessBuilder keine ProcessBuilder verstehen. Dazu gehören Umleitungen, Pipelines, variable Erweiterungen, Globbing usw.

In einigen Fällen (z. B. einfache Weiterleitung) können Sie mit ProcessBuilder den gewünschten Effekt leicht erreichen. Dies trifft jedoch im Allgemeinen nicht zu. Ein alternativer Ansatz besteht darin, die Befehlszeile in einer Shell auszuführen. zum Beispiel:

```
Process p = Runtime.exec("bash", "-c",
                         "find / -name *.java -print 2>/dev/null");
```

oder

```
Process p = Runtime.exec("bash", "-c",
                         "find source -name \\*.java | xargs grep package");
```

Beachten Sie jedoch, dass wir im zweiten Beispiel dem Platzhalterzeichen ("*") entgehen mussten, da der Platzhalter von "find" und nicht von der Shell interpretiert werden soll.

Integrierte Shell-Befehle funktionieren nicht

Angenommen, die folgenden Beispiele funktionieren auf einem System mit einer UNIX-ähnlichen Shell nicht:

```
Process p = Runtime.exec("cd", "/tmp");     // Change java app's home directory
```

oder

```
Process p = Runtime.exec("export", "NAME=value");  // Export NAME to the java app's
```

```
environment
```

Es gibt einige Gründe, warum dies nicht funktioniert:

1. Bei den Befehlen "cd" und "export" handelt es sich um in die Shell eingebaute Befehle. Sie existieren nicht als ausführbare Dateien.
2. Damit die eingebauten Shell-Funktionen genau das tun können, was sie tun sollen (z. B. das Arbeitsverzeichnis ändern, die Umgebung aktualisieren), müssen sie den Ort ändern, an dem sich dieser Status befindet. Bei einer normalen Anwendung (einschließlich einer Java-Anwendung) ist der Status dem Anwendungsprozess zugeordnet. So konnte beispielsweise der untergeordnete Prozess, der den Befehl "cd" ausführen würde, das Arbeitsverzeichnis seines übergeordneten Prozesses "java" nicht ändern. Ebenso kann ein exec -Prozess nicht das Arbeitsverzeichnis für einen exec Prozess ändern.

Diese Argumentation gilt für alle integrierten Shell-Befehle.

---

1 - Sie können `ProcessBuilder` auch verwenden, dies ist jedoch für den Punkt dieses Beispiels nicht relevant.

2 - Dies ist etwas grob und bereit ... aber auch hier sind die Fehler dieses Ansatzes für das Beispiel nicht relevant.

Verarbeiten online lesen: https://riptutorial.com/de/java/topic/4682/verarbeiten

Kapitel 168: Vergleichbar und Vergleicher

Syntax

- public class MyClass implementiert Comparable <MyClass >
- public class MyComparator implementiert Comparator <SomeOtherClass >
- public int compareTo (Meine andere)
- public int compare (SomeOtherClass o1, SomeOtherClass o2)

Bemerkungen

Wenn Sie eine compareTo(..) -Methode implementieren, die von einem double abhängig ist, **gehen Sie nicht** wie folgt vor:

```
public int comareTo(MyClass other) {
    return (int)(doubleField - other.doubleField); //THIS IS BAD
}
```

Die durch die (int) -Umgabe verursachte (int) führt dazu, dass die Methode manchmal falsch anstatt einer positiven oder negativen Zahl 0 zurückgibt, was zu Vergleichen und Sortieren von Fehlern führen kann.

Stattdessen ist die einfachste korrekte Implementierung die Verwendung von Double.compare als solchen:

```
public int comareTo(MyClass other) {
    return Double.compare(doubleField,other.doubleField); //THIS IS GOOD
}
```

Eine nicht generische Version von Comparable<T> , einfach Comparable , existiert seit Java 1.2 . Abgesehen von der Anbindung an älteren Code ist es immer besser, die generische Version Comparable<T> zu implementieren, da kein Casting beim Vergleich erforderlich ist.

Es ist sehr Standard, dass eine Klasse mit sich selbst vergleichbar ist, wie in:

```
public class A implements Comparable<A>
```

Es ist zwar möglich, von diesem Paradigma abzubrechen, seien Sie jedoch vorsichtig.

Ein Comparator<T> kann weiterhin für Instanzen einer Klasse verwendet werden, wenn diese Klasse Comparable<T> implementiert. In diesem Fall wird die Logik des Comparator verwendet. Die von der Comparable Implementierung angegebene natürliche Reihenfolge wird ignoriert.

Examples

Sortieren einer Liste mit Comparable oder ein Komparator

Angenommen, wir arbeiten an einer Klasse, die eine Person mit ihrem Vor- und Nachnamen darstellt. Wir haben dafür eine grundlegende Klasse erstellt und richtige equals und hashCode Methoden implementiert.

```
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);
    }
}
```

Nun möchten wir eine Liste von Person nach ihrem Namen sortieren, wie im folgenden Szenario:

```
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.
}
```

Leider werden die oben genannten Punkte derzeit nicht kompiliert. Collections.sort(..) eine Liste nur sortieren, wenn die Elemente in dieser Liste vergleichbar sind oder eine benutzerdefinierte Vergleichsmethode angegeben wird.

Wenn Sie gebeten werden, die folgende Liste zu sortieren: 1,3,5,4,2 , haben Sie kein Problem mit der Antwort 1,2,3,4,5 . Dies liegt daran, dass Integer (sowohl in Java als auch mathematisch) eine *natürliche Reihenfolge haben* , eine Standard-Standardvergleichsbasisordnung. Um unserer Person-Klasse eine natürliche Reihenfolge zu geben, implementieren wir compareTo(Person p): Comparable<Person> , was die Implementierung der Methode compareTo(Person p): erfordert 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);
        }
    }
}
```

Die angegebene Hauptmethode funktioniert nun korrekt

```
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
}
```

Wenn Sie jedoch die Klasse Person nicht ändern möchten oder können, können Sie einen benutzerdefinierten Comparator<T> bereitstellen, der den Vergleich von zwei Person übernimmt. Wenn Sie aufgefordert wurden, die folgende Liste zu sortieren: circle, square, rectangle, triangle, hexagon , konnten Sie nicht. Wenn Sie jedoch aufgefordert wurden, diese Liste *anhand der Anzahl der Ecken* zu sortieren, können Sie dies tun. Wenn Sie einen Komparator bereitstellen, wird Java angewiesen, zwei normalerweise nicht vergleichbare Objekte zu vergleichen.

```
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
}
```

Komparatoren können auch als anonyme innere Klasse erstellt / verwendet werden

```
//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

Vergleicher auf Lambda-Basis

Ab Java 8 können Komparatoren auch als Lambda-Ausdrücke ausgedrückt werden

```
    //Lambda
    Collections.sort(people, (p1, p2) -> { //Legal
```

```
        //Method code....
    });
```

Comparator-Standardmethoden

Darüber hinaus gibt es auf der Comparator-Benutzeroberfläche interessante Standardmethoden zum lastName von Komparatoren: Im Folgenden wird ein Komparator erstellt, der nach lastName und dann nach firstName .

```
Collections.sort(people, Comparator.comparing(Person::getLastName)
                                .thenComparing(Person::getFirstName));
```

Umkehren der Reihenfolge eines Komparators

Jeder Komparator kann auch leicht mit der reversedMethod wodurch die aufsteigende Reihenfolge in absteigend geändert wird.

Die compareTo und Compare-Methoden

Die Comparable<T> -Schnittstelle erfordert eine Methode:

```
public interface Comparable<T> {

    public int compareTo(T other);

}
```

Und die Comparator<T> -Schnittstelle erfordert eine Methode:

```
public interface Comparator<T> {

    public int compare(T t1, T t2);

}
```

Diese beiden Methoden tun im Wesentlichen die gleiche Sache, mit einem kleinen Unterschied: compareTo vergleicht this auf other , während compare vergleicht t1 bis t2 , nicht darum kümmern überhaupt über this .

Abgesehen von diesem Unterschied haben die beiden Methoden ähnliche Anforderungen. Spezifisch (für compareTo) Vergleicht dieses Objekt mit dem angegebenen Objekt für die Bestellung. Gibt eine negative Ganzzahl, Null oder eine positive Ganzzahl zurück, da dieses Objekt kleiner als, gleich oder größer als das angegebene Objekt ist. Für den Vergleich von a und b :

- Wenn a < b , a.compareTo(b) und compare(a,b) eine negative Ganzzahl und b.compareTo(a) und compare(b,a) eine positive Ganzzahl zurückgeben sollen
- Wenn a > b , sollten a.compareTo(b) und compare(a,b) eine positive Ganzzahl und b.compareTo(a) und compare(b,a) eine negative Ganzzahl zurückgeben
- Wenn a gleich b zum Vergleich, sollten alle Vergleiche zurückkehren 0 .

Natürliche (vergleichbare) vs. explizite (Vergleicher) Sortierung

Es gibt zwei Collections.sort() -Methoden:

- Eine, die eine List<T> als Parameter verwendet, in der T compareTo() implementieren muss, und die compareTo() -Methode überschreiben, die die compareTo() bestimmt.
- Eine, die eine Liste und einen Vergleicher als Argument verwendet, wobei der Vergleicher die Sortierreihenfolge festlegt.

Hier ist zunächst eine Person-Klasse, die Comparable implementiert:

```
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();
    }

}
```

So würden Sie die obige Klasse verwenden, um eine Liste in der natürlichen Reihenfolge ihrer Elemente zu sortieren, die durch die Methode compareTo() Methode überschrieben wird:

```
//-- 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);
```

So würden Sie einen anonymen Inline Comparator verwenden, um eine Liste zu sortieren, die Comparable nicht implementiert, oder in diesem Fall eine Liste in einer anderen Reihenfolge als der natürlichen Reihenfolge zu sortieren:

```
            //-- 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);
```

Karteneinträge sortieren

Ab Java 8 gibt es in der Map.Entry Schnittstelle Standardmethoden, um das Sortieren von Map.Entry zu ermöglichen.

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
```

Natürlich können diese auch außerhalb der Stream-API verwendet werden:

Java SE 8

```
List<Map.Entry<String, Integer>> entries = new ArrayList<>(numberOfEmployees.entrySet());
Collections.sort(entries, Map.Entry.comparingByValue());
```

Erstellen eines Komparators mit der Vergleichsmethode

```
Comparator.comparing(Person::getName)
```

Dadurch wird ein Vergleicher für die Klasse Person , die diesen Personennamen als Vergleichsquelle verwendet. Es ist auch möglich, die Methodenversion zu verwenden, um long, int und double zu vergleichen. Zum Beispiel:

```
Comparator.comparingInt(Person::getAge)
```

Umgekehrte Reihenfolge

Um einen Komparator zu erstellen, der die umgekehrte Reihenfolge erzwingt, verwenden Sie die reversed() Methode:

```
Comparator.comparing(Person::getName).reversed()
```

Kette von Komparatoren

```
Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstName)
```

Dadurch wird ein Vergleicher erstellt, der mit dem Nachnamen und dann mit dem Vornamen verglichen wird. Sie können beliebig viele Komparatoren verketten.

Vergleichbar und Vergleicher online lesen:
https://riptutorial.com/de/java/topic/3137/vergleichbar-und-vergleicher

Kapitel 169: Verkapselung

Einführung

Stellen Sie sich vor, Sie hätten eine Klasse mit ziemlich wichtigen Variablen, und diese wurden (von anderen Programmierern aus ihrem Code) auf inakzeptable Werte gesetzt. Als Lösung lassen Sie in OOP zu, dass der Status eines Objekts (in seinen Variablen gespeichert) nur durch Methoden geändert wird. Das Ausblenden des Status eines Objekts und das Bereitstellen der gesamten Interaktion durch Objektmethoden wird als Data Encapsulation bezeichnet.

Bemerkungen

Es ist viel einfacher, eine Variable als private zu markieren und bei Bedarf verfügbar zu machen, als eine bereits public Variable auszublenden.

Es gibt eine Ausnahme, bei der die Kapselung möglicherweise nicht vorteilhaft ist: "dumme" Datenstrukturen (Klassen, deren einziger Zweck darin besteht, Variablen zu enthalten).

```
public class DumbData {
    public String name;
    public int timeStamp;
    public int value;
}
```

In diesem Fall *sind* die Daten der Klasse die Schnittstelle der Klasse.

Beachten Sie, dass als final gekennzeichnete Variablen als public markiert werden können, ohne die Kapselung zu verletzen, da sie nach dem Setzen nicht mehr geändert werden können.

Examples

Kapselung zur Erhaltung von Invarianten

Eine Klasse besteht aus zwei Teilen: der Schnittstelle und der Implementierung.

Die Schnittstelle ist die offengelegte Funktionalität der Klasse. Seine öffentlichen Methoden und Variablen sind Teil der Schnittstelle.

Die Implementierung ist das interne Arbeiten einer Klasse. Andere Klassen sollten nicht über die Implementierung einer Klasse Bescheid wissen müssen.

Encapsulation bezieht sich auf die Praxis, die Implementierung einer Klasse vor allen Benutzern dieser Klasse zu verbergen. Dies erlaubt der Klasse, Annahmen über ihren internen Zustand zu treffen.

Nehmen Sie zum Beispiel diese Klasse, die einen Winkel darstellt:

```
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(){}
}
```

Diese Klasse **basiert** auf einer Grundannahme (oder *Invariante*): **angleInDegrees und angleInRadians sind immer synchron** . Wenn die Teilnehmer öffentlich sind, gibt es keine Garantie dafür, dass die beiden Darstellungen von Winkeln korreliert sind.

Kapselung zur Reduzierung der Kopplung

Mit der Kapselung können Sie interne Änderungen an einer Klasse vornehmen, ohne den Code zu beeinflussen, der die Klasse aufruft. Dies reduziert die *Kopplung* oder wie sehr eine bestimmte Klasse von der Implementierung einer anderen Klasse abhängt.

Ändern wir beispielsweise die Implementierung der Angle-Klasse aus dem vorherigen Beispiel:

```
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(){}
}
```

Die Implementierung dieser Klasse wurde geändert, sodass nur eine Darstellung des Winkels gespeichert und der andere Winkel bei Bedarf berechnet wird.

Die **Implementierung wurde jedoch geändert, die Schnittstelle jedoch nicht** . Wenn eine aufrufende Klasse auf den Zugriff auf die angleInRadians-Methode angewiesen ist, muss sie geändert werden, um die neue Version von Angle . Beim Aufruf von Klassen sollte die interne Repräsentation einer Klasse nicht berücksichtigt werden.

Verkapselung online lesen: https://riptutorial.com/de/java/topic/1295/verkapselung

Kapitel 170: Vernetzung

Syntax

- neuer Socket ("localhost", 1234); // Stellt eine Verbindung zu einem Server unter der Adresse "localhost" und Port 1234 her
- neuer SocketServer ("localhost", 1234); // Erstellt einen Socket-Server, der an der Adresse localhost und Port 1234 auf neue Sockets warten kann
- socketServer.accept (); // Akzeptiert ein neues Socket-Objekt, das zur Kommunikation mit dem Client verwendet werden kann

Examples

Grundlegende Client- und Server-Kommunikation über einen Socket

Server: Starten Sie und warten Sie auf eingehende Verbindungen

```
//Open a listening "ServerSocket" on port 1234.
ServerSocket serverSocket = new ServerSocket(1234);

while (true) {
    // Wait for a client connection.
    // Once a client connected, we get a "Socket" object
    // that can be used to send and receive messages to/from the newly
    // connected client
    Socket clientSocket = serverSocket.accept();

    // Here we'll add the code to handle one specific client.
}
```

Server: Umgang mit Clients

Wir behandeln jeden Client in einem separaten Thread, sodass mehrere Clients gleichzeitig mit dem Server interagieren können. Diese Technik funktioniert gut, solange die Anzahl der Clients gering ist (<< 1000 Clients, abhängig von der Betriebssystemarchitektur und der erwarteten Last jedes Threads).

```
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: Verbinden Sie sich mit dem Server und senden Sie eine Nachricht

```
// 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();
```

Schließen von Sockets und Behandeln von Ausnahmen

Die obigen Beispiele ließen einige Dinge aus, um sie leichter lesbar zu machen.

1. Genau wie Dateien und andere externe Ressourcen ist es wichtig, dass wir dem Betriebssystem mitteilen, wenn wir damit fertig sind. Wenn wir mit einem Socket fertig sind, rufen Sie socket.close() auf, um ihn richtig zu schließen.

2. Sockets behandeln E / A-Vorgänge (Eingabe / Ausgabe), die von verschiedenen externen Faktoren abhängen. Was passiert zum Beispiel, wenn die andere Seite plötzlich getrennt wird? Was ist, wenn ein Netzwerkfehler vorliegt? Diese Dinge liegen außerhalb unserer Kontrolle. Aus diesem Grund können viele IOException Ausnahmen IOException , insbesondere IOException .

Ein vollständigerer Code für den Client wäre daher etwa so:

```
// "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
}
```

Basic Server und Client – vollständige Beispiele

Server:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Server {
    public static void main(String args[]) {
        try (ServerSocket serverSocket = new ServerSocket(1234)) {
            while (true) {
                // Wait for a client connection.
                Socket clientSocket = serverSocket.accept();

                // Create and start a thread to handle the new client
                new Thread(() -> {
                    try {
                        // Get the socket's InputStream, to read bytes
                        // from the socket
                        InputStream in = clientSocket.getInputStream();
                        // wrap the InputStream in a reader so you can
                        // read a String instead of bytes
```

```
                        BufferedReader reader = new BufferedReader(
                             new InputStreamReader(in, StandardCharsets.UTF_8));
                        // Read from the socket and print line by line
                        String line;
                        while ((line = reader.readLine()) != null) {
                            System.out.println(line);
                        }
                    }
                    catch (IOException e) {
                        e.printStackTrace();
                    } finally {
                        // This finally block ensures the socket is closed.
                        // A try-with-resources block cannot be used because
                        // the socket is passed into a thread, so it isn't
                        // created and closed in the same block
                        try {
                            clientSocket.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                }).start();
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

Klient:

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

TrustStore und KeyStore werden aus InputStream geladen

```
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();
            }

    }
}
```

Die Initialisierung eines KeyStores funktioniert genauso, mit der Ausnahme, dass ein beliebiges Wort Trust in einem Objektnamen durch Key . Darüber hinaus muss das KeyManager[] Array an das erste Argument von SSLContext.init . Das ist SSLContext.init(keyMangers, trustMangers, null)

Socket-Beispiel - Lesen einer Webseite mit einem einfachen Socket

```
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);
        }
    }
}
```

Sie sollten eine Antwort erhalten, die mit HTTP/1.1 200 OK beginnt HTTP/1.1 200 OK ist eine normale HTTP-Antwort, gefolgt vom restlichen HTTP-Header, gefolgt von der rohen Webseite im HTML-Format.

Beachten Sie, dass die readFully() -Methode wichtig ist, um eine vorzeitige EOF-Ausnahme zu verhindern. Möglicherweise fehlt in der letzten Zeile der Webseite ein Return, um das Zeilenende zu signalisieren. Anschließend wird sich readLine() beschweren. readLine() muss man es readLine() lesen oder Dienstprogramme von Apache commons-io IOUtils verwenden

Dieses Beispiel dient als einfache Demonstration der Verbindung mit einer vorhandenen Ressource über einen Socket. Es ist keine praktische Möglichkeit, auf Webseiten zuzugreifen. Wenn Sie mit Java auf eine Webseite zugreifen müssen, verwenden Sie am besten eine vorhandene HTTP-Client-Bibliothek wie den HTTP-Client von Apache oder den HTTP-Client von Google

Grundlegende Client / Server-Kommunikation über UDP (Datagramm)

Client.java

```
import java.io.*;
import java.net.*;

public class Client{
    public static void main(String [] args) throws IOException{
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress address = InetAddress.getByName(args[0]);

        String ex = "Hello, World!";
        byte[] buf = ex.getBytes();

        DatagramPacket packet = new DatagramPacket(buf,buf.length, address, 4160);
        clientSocket.send(packet);
    }
}
```

In diesem Fall übergeben wir die Adresse des Servers über ein Argument (args[0]). Der verwendete Port ist 4160.

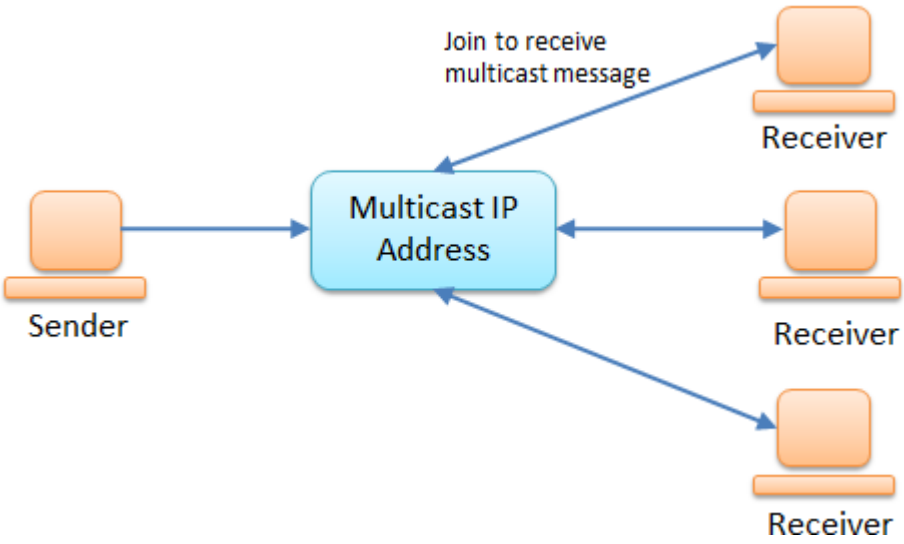
Server.java

```
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);
    }
}
```

Deklarieren Sie serverseitig ein DatagramSocket an demselben Port, an den wir unsere Nachricht gesendet haben (4160), und warten Sie auf eine Antwort.

Multicasting

Multicasting ist eine Art Datagram-Socket. Im Gegensatz zu regulären Datagrammen behandelt Multicasting nicht jeden Client einzeln, sondern sendet ihn an eine IP-Adresse und alle abonnierten Clients erhalten die Nachricht.



Beispielcode für eine Serverseite:

```
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();
    }
}
```

Beispielcode für eine Clientseite:

```
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 zum Ausführen des Servers:

```
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 zum Ausführen eines Clients:

```
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();
    }
}
```

Führen Sie den Client zuerst aus: Der Client muss die IP-Adresse abonnieren, bevor er Pakete empfangen kann. Wenn Sie den Server starten und die send() -Methode aufrufen, printMessage() einen Client (& rufen Sie printMessage()). Nichts wird passieren, weil der Client nach dem Senden der Nachricht eine Verbindung hergestellt hat.

Deaktivieren Sie die SSL-Überprüfung vorübergehend (zu Testzwecken).

In einer Entwicklungs- oder Testumgebung wurde die SSL-Zertifikatskette manchmal (noch) nicht vollständig eingerichtet.

Um die Entwicklung und das Testen fortzusetzen, können Sie die SSL-Überprüfung programmgesteuert deaktivieren, indem Sie einen "all-trusting" Trust Manager installieren:

```
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();
}
```

Eine Datei mit Channel herunterladen

Wenn die Datei bereits existiert, wird sie überschrieben!

```
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 */ }
```

Anmerkungen

- Lassen Sie die Fangblöcke nicht leer!
- Überprüfen Sie im Fehlerfall, ob die Remote-Datei vorhanden ist
- Dies ist ein Blockierungsvorgang, der bei großen Dateien lange dauern kann

Vernetzung online lesen: https://riptutorial.com/de/java/topic/149/vernetzung

Kapitel 171: Verschachtelte und innere Klassen

Einführung

Mit Java können Entwickler eine Klasse in einer anderen Klasse definieren. Eine solche Klasse wird als verschachtelte Klasse bezeichnet . Verschachtelte Klassen werden als Innere Klassen bezeichnet, wenn sie als nicht statisch deklariert wurden. Andernfalls werden sie einfach als statische verschachtelte Klassen bezeichnet. Diese Seite dient zum Dokumentieren und Bereitstellen von Details mit Beispielen zur Verwendung von geschachtelten und inneren Klassen von Java.

Syntax

- public class OuterClass {public class InnerClass {}} // Innere Klassen können auch privat sein
- public class OuterClass {public static class StaticNestedClass {}} // Statisch verschachtelte Klassen können auch privat sein
- public void method () {private class LocalClass {}} // Lokale Klassen sind immer privat
- SomeClass anonymousClassInstance = new SomeClass () {}; // Anonyme innere Klassen können nicht benannt werden, daher ist der Zugriff moot. Wenn 'SomeClass ()' abstrakt ist, muss der Rumpf alle abstrakten Methoden implementieren.
- SomeInterface anonymousClassInstance = new SomeInterface () {}; // Der Body muss alle Schnittstellenmethoden implementieren.

Bemerkungen

Terminologie und Klassifizierung

Die Java Language Specification (JLS) klassifiziert die verschiedenen Arten von Java-Klassen wie folgt:

> Eine *Klasse der obersten Ebene* ist eine Klasse, die keine verschachtelte Klasse ist.
>
> Eine *verschachtelte Klasse* ist eine Klasse, deren Deklaration im Hauptteil einer anderen Klasse oder Schnittstelle auftritt.
>
> Eine *innere Klasse* ist eine verschachtelte Klasse, die nicht explizit oder implizit als statisch deklariert ist.
>
> Eine innere Klasse kann eine *nicht statische Member-Klasse* , eine *lokale Klasse* oder eine *anonyme Klasse sein* . Eine Member-Klasse eines Interfaces ist implizit statisch und wird daher niemals als innere Klasse betrachtet.

In der Praxis beziehen sich Programmierer auf eine Klasse der obersten Ebene, die eine innere Klasse als "äußere Klasse" enthält. Es besteht auch die Tendenz, "verschachtelte Klasse" zu verwenden, um nur auf (explizit oder implizit) statische verschachtelte Klassen zu verweisen.

Beachten Sie, dass zwischen anonymen inneren Klassen und den Lambdas eine enge Beziehung besteht, aber Lambdas sind Klassen.

Semantische Unterschiede

- Top-Level-Klassen sind der "Basisfall". Sie sind für andere Teile eines Programms sichtbar, die normalen Sichtbarkeitsregeln unterliegen, die auf der Zugriffsmodifikationssemantik basieren. Wenn sie nicht abstrakt sind, können sie durch jeden Code instanziiert werden, bei dem die relevanten Konstruktoren basierend auf den Zugriffsmodifizierern sichtbar sind.
- Statische verschachtelte Klassen folgen mit zwei Ausnahmen denselben Zugriffs- und Instantiierungsregeln wie Klassen der obersten Ebene:
  - Eine verschachtelte Klasse kann als private deklariert werden, sodass sie außerhalb der sie umgebenden Top-Level-Klasse nicht zugänglich ist.
  - Eine verschachtelte Klasse hat Zugriff auf die private Mitglieder der umgebenden

Klasse der obersten Ebene und auf alle getesteten Klassen.

Dies macht statische verschachtelte Klassen nützlich, wenn Sie mehrere "Entitätstypen" innerhalb einer engen Abstraktionsgrenze darstellen müssen. ZB wenn die verschachtelten Klassen zum Ausblenden von "Implementierungsdetails" verwendet werden.

- Innere Klassen bieten die Möglichkeit, auf nicht statische Variablen zuzugreifen, die in umschließenden Bereichen deklariert sind:
    - Eine nicht statische Member-Klasse kann sich auf Instanzvariablen beziehen.
    - Eine lokale Klasse (in einer Methode deklariert) kann auch auf die lokalen Variablen der Methode verweisen, vorausgesetzt, sie sind final . (Für Java 8 und höher können sie *effektiv final sein* .)
    - Eine anonyme innere Klasse kann entweder innerhalb einer Klasse oder einer Methode deklariert werden und auf Variablen nach denselben Regeln zugreifen.

  Die Tatsache, dass eine Instanz der inneren Klasse auf Variablen in einer umgebenden Klasseninstanz verweisen kann, hat Auswirkungen auf die Instantiierung. Insbesondere muss eine einschließende Instanz implizit oder explizit bereitgestellt werden, wenn eine Instanz einer inneren Klasse erstellt wird.

Examples

Ein einfacher Stapel mit einer verschachtelten Klasse

```
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;
    }
}
```

Und deren Verwendung, die (insbesondere) die Existenz der verschachtelten Klasse überhaupt nicht anerkennt.

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

Statische vs. nicht statische verschachtelte Klassen

Wenn Sie eine verschachtelte Klasse erstellen, haben Sie die Wahl, diese verschachtelte Klasse statisch zu haben:

```
public class OuterClass1 {

    private static class StaticNestedClass {

    }

}
```

Oder nicht statisch:

```
public class OuterClass2 {

    private class NestedClass {

    }

}
```

Im Grunde haben statische verschachtelte Klassen *keine umgebende* ***Instanz*** der äußeren Klasse, nicht statische verschachtelte Klassen. Dies betrifft sowohl, wo / wann man eine verschachtelte Klasse instanziieren darf, als auch, auf welche Instanzen dieser verschachtelten Klassen zugegriffen werden darf. Hinzufügen zum obigen Beispiel:

```
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
    }
}
```

Daher hängt Ihre Entscheidung zwischen statisch und nicht statisch hauptsächlich davon ab, ob Sie direkt auf Felder und Methoden der äußeren Klasse zugreifen müssen. Dies hat jedoch auch Auswirkungen darauf, wann und wo Sie die verschachtelte Klasse erstellen können.

Als Faustregel sollten Sie Ihre verschachtelten Klassen statisch machen, sofern Sie nicht auf Felder und Methoden der äußeren Klasse zugreifen müssen. Ähnlich wie bei der Privatisierung Ihrer Felder, es sei denn, Sie müssen sie öffentlich machen, verringert dies die Sichtbarkeit der verschachtelten Klasse (indem sie keinen Zugriff auf eine äußere Instanz zulässt), wodurch die Fehlerwahrscheinlichkeit verringert wird.

Zugriffsmodifizierer für innere Klassen

Eine vollständige Erklärung der Zugriffsmodifizierer in Java finden Sie hier . Aber wie interagieren sie mit den inneren Klassen?

public gibt wie üblich uneingeschränkten Zugriff auf alle Bereiche frei, die auf den Typ zugreifen können.

```
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
    }
}
```

Sowohl protected als auch der Standardmodifikator (von nothing) verhalten sich erwartungsgemäß genauso wie bei nicht verschachtelten Klassen.

private sich interessanterweise nicht auf die Klasse, zu der er gehört. Sie beschränkt sich vielmehr auf die Kompilierungseinheit - die .java-Datei. Dies bedeutet, dass äußere Klassen uneingeschränkten Zugriff auf Felder und Methoden der inneren Klasse haben, selbst wenn sie als private gekennzeichnet sind.

```
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;
    }
}
```

Die innere Klasse selbst kann eine andere Sichtbarkeit als public . Durch das Markieren als private oder einen anderen Zugriffsmodifikator mit eingeschränktem Zugriff können andere (externe) Klassen den Typ nicht importieren und zuweisen. Sie können jedoch immer noch Verweise auf Objekte dieses Typs erhalten.

```
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
    }
}
```

Anonyme innere Klassen

Eine anonyme innere Klasse ist eine Form der inneren Klasse, die mit einer einzigen Anweisung deklariert und instanziiert wird. Infolgedessen gibt es keinen Namen für die Klasse, der an anderer Stelle im Programm verwendet werden kann. dh es ist anonym.

Anonyme Klassen werden normalerweise in Situationen verwendet, in denen Sie eine leichte Klasse erstellen müssen, die als Parameter übergeben werden soll. Dies erfolgt normalerweise mit einer Schnittstelle. Zum Beispiel:

```
public static Comparator<String> CASE_INSENSITIVE =
        new Comparator<String>() {
            @Override
            public int compare(String string1, String string2) {
                return string1.toUpperCase().compareTo(string2.toUpperCase());
            }
        };
```

Diese anonyme Klasse definiert ein Comparator<String> -Objekt (CASE_INSENSITIVE), das zwei Zeichenfolgen vergleicht und Unterschiede in diesem Fall ignoriert.

Andere Schnittstellen, die häufig mithilfe anonymer Klassen implementiert und instanziiert werden, sind Runnable und Callable . Zum Beispiel:

```
// 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"
```

Anonyme innere Klassen können auch auf Klassen basieren. In diesem Fall wird die anonyme Klasse implizit extends die bestehende Klasse. Wenn die Klasse, die erweitert wird, abstrakt ist, muss die anonyme Klasse alle abstrakten Methoden implementieren. Es können auch nicht abstrakte Methoden überschrieben werden.

Konstrukteure

Eine anonyme Klasse kann keinen expliziten Konstruktor haben. Stattdessen wird ein impliziter Konstruktor definiert, der super(...) , um Parameter an einen Konstruktor in der Klasse zu übergeben, die erweitert wird. Zum Beispiel:

```
SomeClass anon = new SomeClass(1, "happiness") {
            @Override
            public int someMethod(int arg) {
                // do something
            }
        };
```

Der implizite Konstruktor für unsere anonyme Unterklasse SomeClass ruft einen Konstruktor von SomeClass , der mit der SomeClass(int, String) . Wenn kein Konstruktor verfügbar ist, wird ein Kompilierungsfehler angezeigt. Alle Ausnahmen, die vom übereinstimmenden Konstruktor ausgelöst werden, werden auch vom impliziten Konstruktor ausgelöst.

Das funktioniert natürlich nicht, wenn Sie eine Schnittstelle erweitern. Wenn Sie eine anonyme Klasse aus einer Schnittstelle erstellen, lautet die Superklasse der Klassen java.lang.Object die nur über einen No-Args-Konstruktor verfügt.

Methode Lokale innere Klassen

Eine Klasse, die innerhalb einer Methode geschrieben wurde und als **lokale Klasse der Klasse bezeichnet wird** . In diesem Fall ist der Umfang der inneren Klasse innerhalb der Methode eingeschränkt.

Eine methodenlokale innere Klasse kann nur innerhalb der Methode instanziiert werden, in der die innere Klasse definiert ist.

Das Beispiel für die Verwendung der Methode local inner class:

```
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();
    }
}
```

Beim Ausführen wird eine Ausgabe ausgegeben: Method local inner class 1 .

Zugriff auf die äußere Klasse von einer nicht statischen inneren Klasse

Der Verweis auf die äußere Klasse verwendet den Klassennamen und this

```
public class OuterClass {
    public class InnerClass {
        public void method() {
            System.out.println("I can access my enclosing class: " + OuterClass.this);
        }
    }
}
```

Sie können direkt auf Felder und Methoden der äußeren Klasse zugreifen.

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        public void method() {
            System.out.println("I can access " + counter);
        }
    }
}
```

Bei Namenskollisionen können Sie jedoch die äußere Klassenreferenz verwenden.

```
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;
        }
    }
}
```

Instanz einer nicht statischen inneren Klasse von außen erstellen

Eine innere Klasse, die für alle externen Klassen sichtbar ist, kann auch aus dieser Klasse erstellt werden.

Die innere Klasse hängt von der äußeren Klasse ab und erfordert einen Verweis auf eine Instanz davon. Um eine Instanz der inneren Klasse zu erstellen, muss der new Operator nur für eine Instanz der äußeren Klasse aufgerufen werden.

```
class OuterClass {

    class InnerClass {
    }
}

class OutsideClass {

    OuterClass outer = new OuterClass();

    OuterClass.InnerClass createInner() {
        return outer.new InnerClass();
    }
}
```

Beachten Sie die Verwendung als outer.new .

Verschachtelte und innere Klassen online lesen:
https://riptutorial.com/de/java/topic/3317/verschachtelte-und-innere-klassen

Kapitel 172: Verwenden anderer Skriptsprachen in Java

Einführung

Java an sich ist eine extrem mächtige Sprache, die jedoch dank JSR223 (Java Specification Request 223), einer Skript-Engine, noch erweitert werden kann

Bemerkungen

Die Java Scripting API ermöglicht die Interaktion externer Skripts mit Java

Die Skript-API kann die Interaktion zwischen dem Skript und Java ermöglichen. Die Skriptsprachen müssen über eine Implementierung von Script Engine im Klassenpfad verfügen.

Standardmäßig wird JavaScript (auch als ECMAScript bezeichnet) standardmäßig von nashorn bereitgestellt. Jede Script Engine verfügt über einen Skriptkontext, in dem alle Variablen, Funktionen und Methoden in Bindungen gespeichert werden. Manchmal möchten Sie möglicherweise mehrere Kontexte verwenden, da diese die Umleitung der Ausgabe auf einen gepufferten Writer und einen anderen Fehler unterstützen.

Es gibt viele andere Skript-Engine-Bibliotheken wie Jython und JRuby. Solange sie sich auf dem Klassenpfad befinden, können Sie Code überprüfen.

Wir können Bindungen verwenden, um Variablen im Skript anzuzeigen. Wir benötigen in einigen Fällen mehrere Bindungen, da das Aussetzen von Variablen an die Engine im Wesentlichen nur das Aussetzen von Variablen für diese Engine ermöglicht. In manchen Fällen müssen bestimmte Variablen wie Systemumgebung und Pfad für alle Engines desselben Typs verfügbar gemacht werden. In diesem Fall benötigen wir eine globale Bindung. Durch das Aussetzen von Variablen werden alle Skriptmodule von derselben EngineFactory erstellt

Examples

Evaluierung Eine Javascript-Datei im -scripting-Modus von 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;
    }

}
```

Nun die Hauptmethode

```
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);");
}
```

Ihre Ausgabe sollte ähnlich sein
hello world

Wie Sie sehen, wurde die exponierte Variable x gedruckt. Testen Sie jetzt mit einer Datei.

Hier haben wir test.js

```
print(x);
function test(){
```

```
    print("hello test.js:test");
}
test();
```

Und die aktualisierte Hauptmethode

```
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");
}
```

Angenommen, test.js befindet sich in demselben Verzeichnis wie Ihre Anwendung. Sie sollten eine ähnliche Ausgabe haben

```
hello world
hello test.js:test
```

Verwenden anderer Skriptsprachen in Java online lesen:
https://riptutorial.com/de/java/topic/9926/verwenden-anderer-skriptsprachen-in-java

Kapitel 173: Verwenden Sie das statische Schlüsselwort

Syntax

- public static int myVariable; // Eine statische Variable deklarieren
- public static myMethod () {} // Eine statische Methode deklarieren
- public static final double MY_CONSTANT; // Deklarieren einer konstanten Variablen, die von allen Instanzen der Klasse gemeinsam genutzt wird
- öffentliches Finale Doppel MY_CONSTANT; // Deklarieren einer für diese Instanz der Klasse spezifischen Konstantenvariablen (am besten in einem Konstruktor verwendet, der für jede Instanz eine andere Konstante generiert)

Examples

Statik verwenden, um Konstanten zu deklarieren

Da das static Schlüsselwort für den Zugriff auf Felder und Methoden ohne instanziierte Klasse verwendet wird, kann es zum Deklarieren von Konstanten für die Verwendung in anderen Klassen verwendet werden. Diese Variablen bleiben für jede Instanziierung der Klasse konstant. Konventionell sind static Variablen immer ALL_CAPS und verwenden Unterstriche anstelle von Kamel. Ex:

```
 static E STATIC_VARIABLE_NAME
```

Da sich Konstanten nicht ändern können, kann static auch mit dem final Modifikator verwendet werden:

Um zum Beispiel die mathematische Konstante von pi zu definieren:

```
public class MathUtilities {

    static final double PI = 3.14159265358

}
```

Welche kann in jeder Klasse als Konstante verwendet werden, zum Beispiel:

```
public class MathCalculations {

    //Calculates the circumference of a circle
    public double calculateCircumference(double radius) {
        return (2 * radius * MathUtilities.PI);
    }

}
```

Verwenden Sie statisch mit diesem

Static gibt einen Methoden- oder Variablenspeicher an, der *nicht* für jede Instanz der Klasse zugeordnet ist. Die statische Variable wird vielmehr von allen Klassenmitgliedern gemeinsam genutzt. Der Versuch, die statische Variable wie ein Member der Klasseninstanz zu behandeln, führt zu einer Warnung:

```
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
```

Statisch deklarierte Methoden verhalten sich auf die gleiche Weise, jedoch mit einer zusätzlichen Einschränkung:

Sie können this Schlüsselwort nicht in ihnen verwenden!

```
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
    }

}
```

Im Allgemeinen empfiehlt es sich, generische Methoden zu deklarieren, die für verschiedene Instanzen einer Klasse (z. B. Klonmethoden) static , während Methoden wie equals() als nicht statisch gelten. Die main eines Java-Programms ist immer statisch. this bedeutet, dass das Schlüsselwort this nicht in main() .

Verweis auf nicht statisches Member aus statischem Kontext

Statische Variablen und Methoden sind nicht Teil einer Instanz. Es gibt immer eine einzige Kopie dieser Variablen, unabhängig davon, wie viele Objekte Sie von einer bestimmten Klasse erstellen.

Beispielsweise möchten Sie möglicherweise eine unveränderliche Liste von Konstanten haben. Es ist ratsam, sie statisch zu halten und nur einmal in einer statischen Methode zu initialisieren. Dies würde zu einer erheblichen Leistungssteigerung führen, wenn Sie regelmäßig mehrere Instanzen einer bestimmten Klasse erstellen.

Außerdem können Sie auch einen statischen Block in einer Klasse haben. Sie können es verwenden, um einer statischen Variablen einen Standardwert zuzuweisen. Sie werden nur einmal ausgeführt, wenn die Klasse in den Speicher geladen wird.

Instanzvariable, wie der Name vermuten lässt, hängt von einer Instanz eines bestimmten Objekts ab, sie lebt, um ihren Launen zu dienen. Sie können während eines bestimmten Lebenszyklus eines Objekts mit ihnen herumspielen.

Alle Felder und Methoden einer Klasse, die in einer statischen Methode dieser Klasse verwendet werden, müssen statisch oder lokal sein. Wenn Sie versuchen, instabile (nicht statische) Variablen oder Methoden zu verwenden, wird Ihr Code nicht kompiliert.

```
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
    }

}
```

Verwenden Sie das statische Schlüsselwort online lesen:
https://riptutorial.com/de/java/topic/2253/verwenden-sie-das-statische-schlusselwort

Kapitel 174: Wahlweise

Einführung

Optional ist ein Containerobjekt, das einen Wert ungleich Null enthalten kann oder nicht. Wenn ein Wert vorhanden ist, gibt isPresent() true und get() gibt den Wert zurück.

Es werden zusätzliche Methoden bereitgestellt, die vom Vorhandensein des enthaltenen Werts abhängen, z. B. orElse() , das einen Standardwert zurückgibt, wenn kein Wert vorhanden ist, und ifPresent() das einen Codeblock ausführt, wenn der Wert vorhanden ist.

Syntax

- Optional.empty () // Erstellt eine leere optionale Instanz.
- Optional.of (value) // Gibt ein Optional mit dem angegebenen Wert ungleich Null zurück. Eine NullPointerException wird ausgelöst, wenn der übergebene Wert null ist.
- Optional.ofNullable (value) // Gibt ein Optional mit dem angegebenen Wert zurück, der null sein kann.

Examples

Gibt den Standardwert zurück, wenn optional leer ist

Verwenden Sie nicht nur Optional.get() da dies NoSuchElementException . Die Methoden Optional.orElse(T) und Optional.orElseGet(Supplier<? extends T>) bieten eine Möglichkeit, einen Standardwert anzugeben, falls das Optional leer ist.

```
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
```

Der entscheidende Unterschied zwischen orElse und orElseGet besteht darin, dass letztere nur ausgewertet wird, wenn das Optional leer ist, während das an den vorherigen gelieferte Argument ausgewertet wird, selbst wenn das Optional nicht leer ist. Die orElse sollte daher nur für Konstanten verwendet werden und niemals für die Bereitstellung von Werten, die auf irgendeiner Art von Berechnung basieren.

Karte

Verwenden Sie die map() Methode von Optional , um mit Werten zu arbeiten, die möglicherweise null ohne explizite null durchzuführen:

(Beachten Sie, dass die Operationen map() und filter() Gegensatz zu ihren Stream-Gegenstücken, die nur bei einer *Terminal-Operation* ausgewertet werden, sofort ausgewertet werden.)

Syntax:

```
public <U> Optional<U> map(Function<? super T,? extends U> mapper)
```

Code-Beispiele:

```
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"
```

Da Optional.map () ein leeres optionales Element zurückgibt, wenn seine Zuordnungsfunktion NULL zurückgibt, können Sie mehrere map () - Vorgänge als eine Form der nullsicheren Dereferenzierung verketten. Dies wird auch als **null-sichere Verkettung bezeichnet** .

Betrachten Sie das folgende Beispiel:

```
String value = foo.getBar().getBaz().toString();
```

getBar , getBaz und toString können möglicherweise eine NullPointerException .

Eine alternative Methode zum Abrufen des Werts von toString() mithilfe von Optional :

```
String value = Optional.ofNullable(foo)
                       .map(Foo::getBar)
                       .map(Bar::getBaz)
                       .map(Baz::toString)
                       .orElse("");
```

Wenn eine der Zuordnungsfunktionen null zurückgibt, wird eine leere Zeichenfolge zurückgegeben.

Unten ist ein weiteres Beispiel, aber etwas anders. Der Wert wird nur gedruckt, wenn keine der Zuordnungsfunktionen null zurückgegeben hat.

```
Optional.ofNullable(foo)
        .map(Foo::getBar)
        .map(Bar::getBaz)
        .map(Baz::toString)
        .ifPresent(System.out::println);
```

Eine Ausnahme auslösen, wenn kein Wert vorhanden ist

Verwenden Sie die orElseThrow() -Methode von Optional , um den enthaltenen Wert orElseThrow() oder eine Ausnahme orElseThrow() , falls dies nicht festgelegt wurde. Dies ist ähnlich wie beim Aufruf von get() , nur dass beliebige Ausnahmetypen zulässig sind. Die Methode nimmt einen Lieferanten, der die Ausnahme zurückgeben muss, um ausgelöst zu werden.

Im ersten Beispiel gibt die Methode einfach den enthaltenen Wert zurück:

```
Optional optional = Optional.of("something");

return optional.orElseThrow(IllegalArgumentException::new);
// returns "something" string
```

Im zweiten Beispiel löst die Methode eine Ausnahme aus, da kein Wert festgelegt wurde:

```
Optional optional = Optional.empty();

return optional.orElseThrow(IllegalArgumentException::new);
// throws IllegalArgumentException
```

Sie können auch die Lambda-Syntax verwenden, wenn eine Ausnahme mit Nachricht ausgelöst werden soll:

```
optional.orElseThrow(() -> new IllegalArgumentException("Illegal"));
```

Filter

filter() wird verwendet, um anzuzeigen, dass Sie den Wert *nur* möchten, *wenn* er Ihrem Prädikat entspricht.

if (!somePredicate(x)) { x = null; } Sie sich das so vor, if (!somePredicate(x)) { x = null; } .

Code-Beispiele:

```
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
```

Verwendung von optionalen Containern für primitive Nummerntypen

OptionalDouble , OptionalInt und OptionalLong funktionieren wie Optional , sind jedoch speziell für das Umschließen von primitiven Typen konzipiert:

```
OptionalInt presentInt = OptionalInt.of(value);
OptionalInt absentInt = OptionalInt.empty();
```

Da numerische Typen einen Wert haben, gibt es keine spezielle Behandlung für Null. Leere Behälter können überprüft werden mit:

```
presentInt.isPresent(); // Is true.
absentInt.isPresent(); // Is false.
```

In ähnlicher Weise gibt es Kurzwörter, um das Wertemanagement zu unterstützen:

```
// 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();
```

Führen Sie Code nur aus, wenn ein Wert vorhanden ist

```
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.
```

Geben Sie mit einem Lieferanten faul einen Standardwert an

Bei der *normalen* orElse Methode wird ein Object . Sie können sich also fragen, warum es hier eine Option gibt, einen Supplier orElseGet (die orElseGet Methode).

Erwägen:

```
String value = "something";
return Optional.ofNullable(value)
               .orElse(getValueThatIsHardToCalculate()); // returns "something"
```

Es würde immer noch getValueThatIsHardToCalculate() aufrufen, obwohl das Ergebnis nicht verwendet wird, da das optionale Element nicht leer ist.

Um diese Strafe zu vermeiden, liefern Sie einen Lieferanten:

```
String value = "something";
return Optional.ofNullable(value)
               .orElseGet(() -> getValueThatIsHardToCalculate()); // returns "something"
```

Auf diese Weise wird getValueThatIsHardToCalculate() nur aufgerufen, wenn die Optional leer ist.

FlatMap

flatMap ähnelt map . Der Unterschied wird vom Javadoc folgendermaßen beschrieben:

> Diese Methode ähnelt der map(Function) , der bereitgestellte Mapper ist jedoch einer, dessen Ergebnis bereits Optional ist. Wenn dies der flatMap ist, wird es von flatMap nicht mit einem zusätzlichen Optional umbrochen.

Mit anderen Worten: Wenn Sie einen Methodenaufruf verketten, der ein Optional zurückgibt, können Sie mithilfe von Optional.flatMap das Erstellen verschachtelter Optionals vermeiden.

Zum Beispiel die folgenden Klassen gegeben:

```
public class Foo {
    Optional<Bar> getBar(){
        return Optional.of(new Bar());
    }
}

public class Bar {
}
```

Wenn Sie Optional.map , erhalten Sie ein verschachteltes Optional . dh Optional<Optional<Bar>> .

```
Optional<Optional<Bar>> nestedOptionalBar =
    Optional.of(new Foo())
        .map(Foo::getBar);
```

Wenn Sie jedoch Optional.flatMap , erhalten Sie ein einfaches Optional . dh Optional<Bar> .

```
Optional<Bar> optionalBar =
    Optional.of(new Foo())
        .flatMap(Foo::getBar);
```

Wahlweise online lesen: https://riptutorial.com/de/java/topic/152/wahlweise

Kapitel 175: Währung und Geld

Examples

Fügen Sie benutzerdefinierte Währung hinzu

Erforderliche JARs für Klassenpfad:

- javax.money:money-api:1.0 (JSR354 Geld und Währungs-API)
- org.javamoney: moneta: 1.0 (Referenzimplementierung)
- Javax: Annotation-api: 1.2. (Allgemeine Anmerkungen, die von der Referenzimplementierung verwendet werden)

```
// 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
```

Währung und Geld online lesen: https://riptutorial.com/de/java/topic/8359/wahrung-und-geld

Kapitel 176: Warteschlangen und Deques

Examples

Die Verwendung der PriorityQueue

PriorityQueue ist eine Datenstruktur. Wie SortedSet sortiert PriorityQueue auch seine Elemente nach ihren Prioritäten. Die Elemente, die eine höhere Priorität haben, stehen an erster Stelle. Der Typ von PriorityQueue sollte eine comparable oder comparator Schnittstelle implementieren, deren Methoden die Prioritäten der Elemente der Datenstruktur bestimmen.

```
//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 als FIFO-Warteschlange

Die java.util.LinkedList Klasse ist bei der Implementierung von java.util.List eine universelle Implementierung der java.util.Queue Schnittstelle, die ebenfalls nach einem FIFO-Prinzip (First In, First Out) arbeitet.

In dem folgenden Beispiel werden mit der Methode offer() die Elemente in die LinkedList eingefügt. Dieser enqueue wird als enqueue . In der while Schleife werden die Elemente basierend auf dem FIFO aus der Queue entfernt. Diese Operation wird als 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() );
}
```

Die Ausgabe dieses Codes ist

```
first element
```

```
second element
third element
fourth element
fifth element
```

Wie in der Ausgabe zu sehen ist, wird das erste eingefügte Element "erstes Element" zuerst entfernt, "zweites Element" wird an zweiter Stelle usw. entfernt.

Stacks

Was ist ein Stack?

In Java sind Stacks eine LIFO-Datenstruktur (Last In, First Out) für Objekte.

Stack-API

Java enthält eine Stack-API mit den folgenden Methoden

```
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
```

Beispiel

```
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");
      }
   }
}
```

Dies kehrt zurück:

```
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

Eine BlockingQueue ist eine Schnittstelle, die eine Warteschlange ist, die blockiert, wenn Sie versuchen, die Warteschlange von der Warteschlange abzunehmen und die Warteschlange leer ist oder wenn Sie versuchen, Elemente in die Warteschlange aufzunehmen, und die Warteschlange bereits voll ist. Ein Thread, der versucht, sich aus einer leeren Warteschlange zu entschlüsseln, wird blockiert, bis ein anderer Thread ein Element in die Warteschlange einfügt. Ein Thread, der versucht, ein Element in eine vollständige Warteschlange einreihen zu lassen, wird blockiert, bis ein anderer Thread Platz in der Warteschlange bereitstellt, indem entweder ein oder mehrere Elemente aus der Warteschlange entfernt oder die Warteschlange vollständig gelöscht wird.

BlockingQueue-Methoden gibt es in vier Formen. Es gibt verschiedene Arten, Operationen zu behandeln, die nicht sofort erfüllt werden können, aber zu einem späteren Zeitpunkt erfüllt werden können: Eine gibt eine Ausnahme aus, die zweite gibt einen speziellen Wert zurück (entweder null oder falsch, abhängig von der Methode) operation), der dritte blockiert den aktuellen Thread auf unbestimmte Zeit, bis die Operation erfolgreich ausgeführt werden kann, und der vierte blockiert nur eine bestimmte Zeitdauer, bevor er aufgibt.

Operation	Löst eine Ausnahme aus	Sonderwert	Blöcke	Die Zeit ist abgelaufen
Einfügen	hinzufügen()	Angebot (e)	Put (e)	Angebot (e, Zeit, Einheit)
Löschen	Löschen()	Umfrage()	nehmen()	Umfrage (Zeit, Einheit)
Untersuchen	Element()	spähen()	N / A	N / A

Eine BlockingQueue kann **gebunden** oder **unbegrenzt sein** . Eine begrenzte BlockingQueue ist eine, die mit der ursprünglichen Kapazität initialisiert wird.

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);
```

Alle Aufrufe einer put () - Methode werden blockiert, wenn die Größe der Warteschlange der ursprünglich festgelegten Kapazität entspricht.

Eine unbegrenzte Warteschlange ist eine Warteschlange, die ohne Kapazität initialisiert wird. Standardmäßig wird sie mit Integer.MAX_VALUE initialisiert.

Einige gängige Implementierungen von BlockingQueue sind:

1. ArrayBlockingQueue
2. LinkedBlockingQueue
3. PriorityBlockingQueue

Betrachten wir nun ein Beispiel für 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");
```

Dies wird drucken:

```
Entry one done
Entry two done
```

Und der Thread wird nach der zweiten Ausgabe blockiert.

Warteschlangenschnittstelle

Grundlagen

Eine Queue ist eine Sammlung zum Halten von Elementen vor der Verarbeitung. Warteschlangen ordnen Elemente, jedoch nicht notwendigerweise, in einer FIFO-Art (first-in-first-out).

Kopf der Warteschlange ist das Element, das durch einen Aufruf zum Entfernen oder Abfragen entfernt werden würde. In einer FIFO-Warteschlange werden alle neuen Elemente am Ende der Warteschlange eingefügt.

Die Warteschlangenschnittstelle

```
public interface Queue<E> extends Collection<E> {
    boolean add(E e);

    boolean offer(E e);

    E remove();

    E poll();

    E element();

    E peek();
}
```

Jede Queue existiert in zwei Formen:

- man gibt eine Ausnahme aus, wenn die Operation fehlschlägt;
- other gibt einen speziellen Wert zurück, wenn der Vorgang fehlschlägt (je nach Vorgang entweder null oder false .

Art des Vorgangs	Wirft eine Ausnahme	Gibt einen speziellen Wert zurück
Einfügen	add(e)	offer(e)
Löschen	remove()	poll()
Untersuchen	element()	peek()

Deque

Ein Deque ist eine "doppelseitige Warteschlange". Deque bedeutet, dass ein Element am Deque oder am Ende der Warteschlange hinzugefügt werden kann. Die Warteschlange kann nur Elemente am Ende einer Warteschlange hinzufügen.

Deque erbt die Queue , was bedeutet, dass die regulären Methoden erhalten bleiben. Die Deque-Schnittstelle bietet jedoch zusätzliche Methoden, um mit einer Warteschlange flexibler zu sein. Die zusätzlichen Methoden sprechen für sich, wenn Sie wissen, wie eine Warteschlange funktioniert, da diese Methoden mehr Flexibilität bieten sollen:

Methode	Kurze Beschreibung
getFirst()	Ruft das erste Element des **Kopfes** der Warteschlange ab, ohne es zu entfernen.
getLast()	Ruft das erste Element des **Endes** der Warteschlange ab, ohne es zu entfernen.
addFirst(E e)	Fügt dem **Kopf** der Warteschlange ein Element hinzu
addLast(E e)	Fügt ein Element zu dem **Ende** der Warteschlange
removeFirst()	Entfernt das erste Element an der **Spitze** der Warteschlange
removeLast()	Entfernt das erste Element am **Ende** der Warteschlange

Natürlich stehen die gleichen Optionen für offer , poll und peek zur Verfügung, sie funktionieren jedoch nicht mit Ausnahmen, sondern mit speziellen Werten. Es macht keinen Sinn zu zeigen, was sie hier tun.

Elemente hinzufügen und darauf zugreifen

Um dem Ende eines Deque Elemente hinzuzufügen, rufen Sie die add() Methode auf. Sie können auch die addFirst() und addLast() verwenden, die dem Kopf und dem Schwanz des Deque Elemente hinzufügen.

```
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
```

Sie können auf das Element am Anfang der Warteschlange zugreifen, ohne das Element aus der

Warteschlange zu entfernen. Dies geschieht über die Methode element() . Sie können auch die getFirst() und getLast() verwenden, die das erste und letzte Element im Deque . So sieht das aus:

```
String firstElement0 = dequeA.element();
String firstElement1 = dequeA.getFirst();
String lastElement = dequeA.getLast();
```

Elemente entfernen

Um Elemente aus einem Deque zu entfernen, rufen Sie die Methoden remove() , removeFirst() und removeLast() . Hier einige Beispiele:

```
String firstElement = dequeA.remove();
String firstElement = dequeA.removeFirst();
String lastElement  = dequeA.removeLast();
```

Warteschlangen und Deques online lesen:
https://riptutorial.com/de/java/topic/7196/warteschlangen-und-deques

Kapitel 177: WeakHashMap

Einführung

Konzepte der schwachen Hashmap

Examples

Konzepte von WeakHashmap

Schlüsselpunkte: -

- Implementierung der Karte.
- speichert nur schwache Verweise auf seine Schlüssel.

Schwache Referenzen : Die Objekte, auf die nur durch schwache Referenzen verwiesen wird, sind eifrig gesammelte Abfälle. Der GC wartet nicht, bis er in diesem Fall Speicherplatz benötigt.

Unterschied zwischen Hashmap und WeakHashMap: -

Wenn der Java-Speichermanager keinen starken Verweis mehr auf das als Schlüssel angegebene Objekt hat, wird der Eintrag in der Karte in WeakHashMap entfernt.

Beispiel: -

```
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"));
    }
```

Größenunterschiede (HashMap vs WeakHashMap):

Beim Aufruf der size () -Methode für das HashMap-Objekt wird dieselbe Anzahl von Schlüsselwertpaaren zurückgegeben. size nimmt nur ab, wenn die remove () - Methode explizit für das HashMap-Objekt aufgerufen wird.

Da der Garbage Collector jederzeit Schlüssel verwerfen kann, verhält sich eine WeakHashMap möglicherweise so, als würde ein unbekannter Thread Einträge unbemerkt entfernen. So ist es möglich, dass die Größenmethode im **Laufe der Zeit** kleinere Werte zurückgibt. Die Größenreduzierung erfolgt in **WeakHashMap automatisch** .

WeakHashMap online lesen: https://riptutorial.com/de/java/topic/10749/weakhashmap

Kapitel 178: XJC

Einführung

XJC ist ein Java SE-Tool, das eine XML-Schemadatei in vollständig kommentierte Java-Klassen kompiliert.

Es ist innerhalb des JDK-Pakets verteilt und befindet sich im Pfad /bin/xjc .

Syntax

- xjc [Optionen] Schemadatei / URL / dir / jar ... [-b bindinfo] ...

Parameter

Parameter	Einzelheiten
Schemadatei	Die xsd-Schemadatei, die in Java konvertiert werden soll

Bemerkungen

Das XJC-Tool ist als Teil des JDK verfügbar. Es ermöglicht das Erstellen von Java-Code, der mit JAXB- Annotationen versehen ist, die sich für (das) Rangieren eignen.

Examples

Java-Code aus einfachen XSD-Dateien generieren

XSD-Schema (schema.xsd)

Das folgende XML - Schema (XSD) definiert eine Liste der Benutzer mit Attributen name und reputation .

```
<?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>
```

Xjc verwenden

Dazu muss der Pfad zum xjc-Tool (JDK-Binärdateien) in der OS-Pfadvariablen enthalten sein.

Die Codegenerierung kann mit gestartet werden

```
xjc schema.xsd
```

Dadurch werden Java-Dateien im Arbeitsverzeichnis generiert.

Ergebnisdateien

Es wird einige zusätzliche Kommentare geben, aber im Grunde sehen die generierten Java-Dateien so aus:

```
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;
```

XJC online lesen: https://riptutorial.com/de/java/topic/4538/xjc

Kapitel 179: XML XPath-Bewertung

Bemerkungen

XPath-Ausdrücke werden verwendet, um einen oder mehrere Knoten in einem XML-Strukturdokument zu navigieren und auszuwählen, z.

In dieser W3C-Empfehlung finden Sie eine Referenz zu dieser Sprache.

Examples

Auswerten einer NodeList in einem XML-Dokument

Gegeben das folgende XML-Dokument:

```
<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>
```

Im Folgenden werden alle example für das Java-Tag abgerufen (Verwenden Sie diese Methode, wenn Sie XPath nur einmal in XML auswerten. Weitere Beispiele finden Sie, wenn mehrere XPath-Aufrufe in derselben XML-Datei ausgewertet werden.):

```
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
...
```

Analysieren mehrerer XPath-Ausdrücke in einem einzigen XML

Wenn Sie dasselbe Beispiel wie **" Auswerten einer NodeList" in einem XML-Dokument verwenden** , können Sie mehrere XPath-Aufrufe effizient **ausführen** :

Gegeben das folgende XML-Dokument:

```
<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>
```

So würden Sie XPath verwenden, um mehrere Ausdrücke in einem Dokument auszuwerten:

```
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

...
```

Einzelnen XPath-Ausdruck mehrmals in XML analysieren

In diesem Fall möchten Sie, dass der Ausdruck vor den Auswertungen kompiliert wird, damit nicht jeder Aufruf zum evaluate denselben Ausdruck compile . Die einfache Syntax wäre:

```
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
```

Insgesamt sind zwei Aufrufe von XPathExpression.evaluate() wesentlich effizienter als zwei

Aufrufe von XPath.evaluate() .

XML XPath-Bewertung online lesen: https://riptutorial.com/de/java/topic/4148/xml-xpath-bewertung

Kapitel 180: XML-Analyse mit den JAXP-APIs

Bemerkungen

XML-Parsing ist die Interpretation von XML-Dokumenten, um ihren Inhalt mithilfe sinnvoller Konstrukte zu manipulieren, entweder "Knoten", "Attribute", "Dokumente", "Namespaces" oder Ereignisse, die sich auf diese Konstrukte beziehen.

Java verfügt über eine native API für die XML-Dokumentenverarbeitung, die als JAXP oder Java-API für die XML-Verarbeitung bezeichnet wird . JAXP und eine Referenzimplementierung sind seit Java 1.4 (JAXP v1.1) in jeder Java-Version enthalten und haben sich seitdem weiterentwickelt. Java 8 wird mit JAXP Version 1.6 ausgeliefert.

Die API bietet verschiedene Möglichkeiten für die Interaktion mit XML-Dokumenten.

- Die DOM-Schnittstelle (Document Object Model)
- Die SAX-Schnittstelle (Simple API for XML)
- Die StAX-Schnittstelle (Streaming-API für XML)

Prinzipien der DOM-Schnittstelle

Die DOM-Schnittstelle soll eine W3C-DOM- kompatible Methode zur Interpretation von XML bieten. Verschiedene Versionen von JAXP haben verschiedene DOM-Spezifikationsebenen (bis zu Stufe 3) unterstützt.

Unter der Document Object Model-Schnittstelle wird ein XML-Dokument beginnend mit dem "Dokumentelement" als Baum dargestellt. Der Basistyp des API ist die Node - Typ, es von einem navigieren können Node zu seinem übergeordneten, seine Kinder oder seine Geschwister (obwohl nicht alle Node s Kinder haben kann, beispielsweise Text - Knoten endgültig in dem Baum sind, und niemals Kinder haben). XML-Tags werden als Element s dargestellt, wodurch der Node durch attributbezogene Methoden erweitert wird.

Die DOM-Schnittstelle ist sehr nützlich, da sie ein "einzeiliges" Parsing von XML-Dokumenten als Baumstrukturen ermöglicht und die einfache Modifizierung des erstellten Baums (Knotenzusatz, Unterdrückung, Kopieren, ...) und schließlich dessen Serialisierung (Back to Disk) ermöglicht) nach Änderungen. Dies hat jedoch einen Preis: Der Baum befindet sich im Arbeitsspeicher. Daher sind DOM-Bäume für große XML-Dokumente nicht immer praktisch. Außerdem ist die Konstruktion des Baums nicht immer die schnellste Methode, um mit XML-Inhalten umzugehen, insbesondere wenn man nicht an allen Teilen des XML-Dokuments interessiert ist.

Prinzipien der SAX-Schnittstelle

Die SAX-API ist eine ereignisorientierte API für den Umgang mit XML-Dokumenten. Bei diesem Modell werden die Komponenten eines XML-Dokuments als Ereignisse interpretiert (z. B. "Ein Tag wurde geöffnet", "Ein Tag wurde geschlossen", "Ein Textknoten wurde gefunden", "Ein Kommentar wurde gefunden"). ..

Die SAX-API verwendet einen "Push-Parsing" -Ansatz, bei dem ein SAX- Parser für die Interpretation des XML-Dokuments verantwortlich ist und Methoden auf einem Delegaten (einem ContentHandler) ContentHandler , um mit allen Ereignissen im XML-Dokument umzugehen. Normalerweise schreibt man nie einen Parser, aber man stellt einen Handler bereit, um alle benötigten Informationen aus dem XML-Dokument zu erhalten.

Die SAX-Schnittstelle überwindet die Einschränkungen der DOM-Schnittstelle, indem nur die minimal erforderlichen Daten auf Parser-Ebene (z. B. Namespaces-Kontexte, Validierungsstatus) beibehalten werden. Daher sind nur Informationen, die vom ContentHandler aufbewahrt werden ContentHandler für die Sie als Entwickler verantwortlich sind, verantwortlich in Erinnerung gehalten. Der Kompromiss besteht darin, dass es mit einem solchen Ansatz keine Möglichkeit gibt, "in der Zeit zurück zu gehen / das XML-Dokument": Während DOM einem Node erlaubt, zu seinem übergeordneten Element zurückzukehren, gibt es keine solche Möglichkeit in SAX.

Prinzipien der StAX-Schnittstelle

Die StAX-API verfolgt einen ähnlichen Ansatz für die Verarbeitung von XML wie die SAX-API (dh ereignisgesteuert). Der einzige sehr wichtige Unterschied besteht darin, dass StAX ein Pull-Parser ist (bei dem SAX ein Push-Parser war). In SAX hat der Parser Kontrolle und verwendet Callbacks für den ContentHandler . In Stax rufen Sie den Parser auf und steuern, wann das nächste XML-Ereignis abgerufen werden soll.

Die API beginnt mit XMLStreamReader (oder XMLEventReader). Hierbei handelt es sich um die Gateways, über die der Entwickler nextEvent() in einer Iterator-Art fragen kann.

Examples

Analysieren und Navigieren eines Dokuments mithilfe der DOM-API

Betrachten Sie das folgende Dokument:

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

Man kann den folgenden Code verwenden, um aus einem String eine DOM-Struktur zu erstellen:

```
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");
}
}
```

Der Code ergibt Folgendes:

```
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
```

Analysieren eines Dokuments mit der StAX-API

Betrachten Sie das folgende Dokument:

```
<?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>
```

Sie können den folgenden Code verwenden, um ihn zu analysieren und eine Zuordnung der Buchtitel anhand der Buch-ID zu erstellen.

```
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);
}
```

Dies gibt aus:

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

In diesem Beispiel muss man einige Dinge beachten:

1. Die Verwendung von xmlStreamReader.getAttributeValue funktioniert, da wir zuerst überprüft haben, ob sich der Parser im START_ELEMENT . In allen anderen Zuständen (außer ATTRIBUTES) muss der Parser IllegalStateException , da Attribute nur am Anfang von Elementen IllegalStateException werden können.

2. Dasselbe gilt für xmlStreamReader.getTextContent() . Es funktioniert, weil wir uns bei START_ELEMENT und in diesem Dokument wissen, dass das <book> -Element keine START_ELEMENT Nicht-Text-Knoten enthält.

Für die Analyse komplexerer Dokumente (tiefere, verschachtelte Elemente, ...) BookParser es sich, den Parser an BookParser oder andere BookParser zu "delegieren", z vom START_ELEMENT bis zum END_ELEMENT des Buch-XML-Tags.

Man kann auch ein Stack Objekt verwenden, um wichtige Daten in der Struktur nach oben und unten zu halten.

XML-Analyse mit den JAXP-APIs online lesen: https://riptutorial.com/de/java/topic/3943/xml-analyse-mit-den-jaxp-apis

Kapitel 181: XOM - XML-Objektmodell

Examples

Eine XML-Datei lesen

Um die XML-Daten mit XOM zu laden, müssen Sie einen Builder erstellen, aus dem Sie sie in ein Document erstellen können.

```
Builder builder = new Builder();
Document doc = builder.build(file);
```

Um das getRootElement() , das höchste übergeordnete Element in der XML-Datei, zu erhalten, müssen Sie das getRootElement() für die Document Instanz verwenden.

```
Element root = doc.getRootElement();
```

Die Element-Klasse verfügt jetzt über viele praktische Methoden, die das Lesen von XML sehr einfach machen. Einige der nützlichsten sind unten aufgeführt:

- getChildElements(String name) - gibt eine Elements Instanz zurück, die als Array von Elementen fungiert
- getFirstChildElement(String name) - gibt das erste getFirstChildElement(String name) Element mit diesem Tag zurück.
- getValue() - gibt den Wert innerhalb des Elements zurück.
- getAttributeValue(String name) - gibt den Wert eines Attributs mit dem angegebenen Namen zurück.

Wenn Sie getChildElements() aufrufen, getChildElements() Sie eine Elements Instanz. Von hier aus können Sie die Methode get(int index) durchlaufen und aufrufen, get(int index) alle darin enthaltenen Elemente abzurufen.

```
Elements colors = root.getChildElements("color");
for (int q = 0; q < colors.size(); q++){
    Element color = colors.get(q);
}
```

Beispiel: Hier ist ein Beispiel zum Lesen einer XML-Datei:

XML-Datei:

```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 zum Lesen und Ausdrucken:

```
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();
            }
        }
    }

}
```

Dies wird in der Konsole ausgedruckt:

```
Name: Smith, Dan
Age: 23 (years)
Favorite Color: green
----------------
Name: Autry, Bob
Age: 3 (months)
Favorite Color: N/A
----------------
```

In eine XML-Datei schreiben

Das Schreiben in eine XML-Datei mit XOM ist dem Lesen sehr ähnlich. In diesem Fall erstellen wir die Instanzen, anstatt sie vom Stammverzeichnis abzurufen.

Um ein neues Element zu erstellen, verwenden Sie den Konstruktor Element(String name) . Sie möchten ein Wurzelelement erstellen, damit Sie es einfach zu einem Document hinzufügen können.

```
Element root = new Element("root");
```

Die Element Klasse verfügt über einige praktische Methoden zum Bearbeiten von Elementen. Sie sind unten aufgeführt:

- appendChild(String name) - Dies setzt den Wert des Elements grundsätzlich auf name.
- appendChild(Node node) - Dies macht node zum übergeordneten Element. (Elemente sind Knoten, damit Sie Elemente analysieren können).
- addAttribute(Attribute attribute) - fügt dem Element ein Attribut hinzu.

Die Attribute Klasse verfügt über verschiedene Konstruktoren. Das einfachste ist Attribute(String name, String value) .

Wenn Sie alle Elemente zu Ihrem Stammelement hinzugefügt haben, können Sie es in ein Document . Document ein Element als Argument in seinem Konstruktor.

Sie können einen Serializer , um Ihr XML in eine Datei zu schreiben. Sie müssen einen neuen Ausgabestrom erstellen, um im Konstruktor von Serializer zu analysieren.

```
FileOutputStream fileOutputStream = new FileOutputStream(file);
Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
serializer.setIndent(4);
serializer.write(doc);
```

Beispiel

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; }
    }

}
```

Dies wird der Inhalt von "out.xml" sein:

```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>

```

XOM - XML-Objektmodell online lesen: https://riptutorial.com/de/java/topic/5091/xom---xml-objektmodell

Kapitel 182: Zahlenformat

Examples

Zahlenformat

Verschiedene Länder haben unterschiedliche Zahlenformate und wenn man dies berücksichtigt, können wir mit Locale of Java unterschiedliche Formate haben. Die Verwendung des Gebietsschemas kann bei der Formatierung helfen

```
Locale locale = new Locale("en", "IN");
NumberFormat numberFormat = NumberFormat.getInstance(locale);
```

Mit dem obigen Format können Sie verschiedene Aufgaben ausführen

1. Formatnummer

   ```
   numberFormat.format(10000000.99);
   ```

2. Währung formatieren

   ```
   NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
   currencyFormat.format(10340.999);
   ```

3. Format Prozentsatz

   ```
   NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
   percentageFormat.format(10929.999);
   ```

4. Kontrollieren Sie die Anzahl der Ziffern

```
numberFormat.setMinimumIntegerDigits(int digits)
numberFormat.setMaximumIntegerDigits(int digits)
numberFormat.setMinimumFractionDigits(int digits)
numberFormat.setMaximumFractionDigits(int digits)
```

Zahlenformat online lesen: https://riptutorial.com/de/java/topic/7399/zahlenformat

Kapitel 183: Zeichenketten

Einführung

Zeichenfolgen (java.lang.String) sind Textstücke, die in Ihrem Programm gespeichert sind. Strings sind in Java **kein** primitiver Datentyp , in Java-Programmen jedoch sehr häufig.

In Java sind Strings unveränderlich, was bedeutet, dass sie nicht geändert werden können. (Klicken Sie hier, um eine genauere Erklärung der Unveränderlichkeit zu erhalten.)

Bemerkungen

Da Java-Zeichenfolgen unveränderlich sind , geben alle Methoden, die einen String bearbeiten , **ein neues String Objekt zurück** . Sie ändern den ursprünglichen String . Dazu gehören untergeordnete Zeichenfolgen und Ersetzungsmethoden, von denen C- und C ++ - Programmierer erwarten würden, dass sie das Ziel- String Objekt mutieren.

Verwenden Sie einen StringBuilder anstelle von String wenn Sie mehr als zwei String Objekte verketten möchten, deren Werte nicht zur Kompilierzeit bestimmt werden können. Diese Technik ist performanter als das Erstellen neuer String Objekte und deren Verkettung, da StringBuilder veränderbar ist.

StringBuffer kann auch verwendet werden, um String Objekte zu verketten. Diese Klasse ist jedoch weniger leistungsfähig, da sie Thread-sicher ist und vor jeder Operation einen Mutex erhält. Da Sie bei der Verkettung von Strings fast nie Thread-Sicherheit benötigen, sollten Sie am besten StringBuilder .

Wenn Sie eine Zeichenfolgenverkettung als einen einzelnen Ausdruck ausdrücken können, ist es besser, den Operator + zu verwenden. Der Java-Compiler konvertiert einen Ausdruck, der + Verkettungen enthält, in eine effiziente Folge von Vorgängen, wobei entweder String.concat(...) oder StringBuilder . Der Hinweis zur Verwendung von StringBuilder explizit nur, wenn die Verkettung mehrere Ausdrücke enthält.

Speichern Sie vertrauliche Informationen nicht in Zeichenfolgen. Wenn jemand in der Lage ist, einen Speicherauszug Ihrer laufenden Anwendung abzurufen, kann er alle vorhandenen String Objekte finden und deren Inhalt lesen. Dies umfasst String Objekte, die nicht erreichbar sind und auf die Garbage Collection warten. Wenn dies ein Problem ist, müssen Sie die sensiblen String-Daten löschen, sobald Sie damit fertig sind. Mit String Objekten ist dies nicht möglich, da sie nicht veränderbar sind. Daher ist es ratsam, char[] -Objekte zum Speichern sensibler Zeichendaten zu verwenden und sie zu löschen (z. B. mit '\000' Zeichen überschreiben), wenn Sie fertig sind.

Alle String Instanzen werden auf dem Heap erstellt, sogar Instanzen, die String-Literalen entsprechen. Das Besondere an String-Literalen ist, dass die JVM dafür sorgt, dass alle Literale, die gleich sind (dh aus den gleichen Zeichen bestehen), durch ein einzelnes String Objekt dargestellt werden (dieses Verhalten wird in JLS angegeben). Dies wird von JVM-Klassenladern implementiert. Wenn ein Klassenladeprogramm eine Klasse lädt, sucht es nach String-Literalen, die in der Klassendefinition verwendet werden, und prüft jedes Mal, wenn es einen Eintrag im String-Pool für dieses Literal gibt (das Literal als Schlüssel). . Wenn bereits ein Eintrag für das Literal vorhanden ist, wird der Verweis auf eine String Instanz verwendet, die als Paar für dieses Literal gespeichert ist. Andernfalls wird eine neue String Instanz erstellt und ein Verweis auf die Instanz für das Literal (als Schlüssel verwendet) im String-Pool gespeichert. (Siehe auch String Interning).

Der String-Pool wird im Java-Heap gehalten und unterliegt der normalen Speicherbereinigung.

Java SE 7

In Java-Versionen vor Java 7 wurde der String-Pool in einem speziellen Teil des Heap-Speichers gehalten, der als "PermGen" bezeichnet wird. Dieser Teil wurde nur gelegentlich gesammelt.

Java SE 7

In Java 7 wurde der String-Pool von "PermGen" verschoben.

Beachten Sie, dass String-Literale implizit von jeder Methode, die sie verwendet, erreichbar sind. Dies bedeutet, dass die entsprechenden String Objekte nur dann gesammelt werden können, wenn der Code selbst gesammelt wird.

Bis Java 8 werden String Objekte als ein UTF-16-Char-Array implementiert (2 Byte pro Char). In Java 9 gibt es einen Vorschlag, String als ein Byte-Array mit einem Codierungsflag-Feld zu implementieren, um zu beachten, ob die Zeichenfolge als Byte (LATIN-1) oder Zeichen (UTF-16) codiert ist.

Examples

Zeichenfolgen vergleichen

Um Strings zu vergleichen für Gleichheit, sollten Sie das String - Objekt verwenden equals oder equalsIgnoreCase Methoden.

Das folgende Snippet bestimmt beispielsweise, ob die beiden Instanzen von String für alle Zeichen gleich sind:

```
String firstString = "Test123";
String secondString = "Test" + 123;

if (firstString.equals(secondString)) {
   // Both Strings have the same content.
}
```

Live-Demo

In diesem Beispiel werden sie unabhängig von ihrem Fall verglichen:

```
String firstString = "Test123";
String secondString = "TEST123";

if (firstString.equalsIgnoreCase(secondString)) {
    // Both Strings are equal, ignoring the case of the individual characters.
}
```

Live-Demo

Beachten Sie, **dass** equalsIgnoreCase nicht Sie lassen eine angeben Locale . Wenn Sie beispielsweise die beiden Wörter "Taki" und "TAKI" auf Englisch vergleichen, sind sie gleich; In Türkisch sind sie jedoch unterschiedlich (in Türkisch ist das Kleinbuchstabe I ı). Für Fälle wie diese, die beiden Strings Umwandlung mit in Kleinbuchstaben (oder Großbuchstaben) Locale und dann mit den Vergleich equals ist die Lösung.

```
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
```

Live-Demo

Verwenden Sie nicht den Operator ==, um Strings zu vergleichen

Wenn Sie nicht garantieren können, dass alle Zeichenfolgen intern sind (siehe unten), **sollten** Sie die Operatoren == oder != **Nicht** zum Vergleichen von Zeichenfolgen verwenden. Diese Operatoren testen tatsächlich Referenzen, und da mehrere String Objekte denselben String darstellen können, kann dies zu einer falschen Antwort führen.

Verwenden String.equals(Object) stattdessen die String.equals(Object) -Methode, die die String-Objekte basierend auf ihren Werten vergleicht. Eine ausführliche Erklärung finden Sie unter Pitfall: Verwenden Sie ==, um Zeichenfolgen zu vergleichen .

Strings in einer switch-Anweisung vergleichen

Java SE 7

Ab Java 1.7 ist es möglich, eine String-Variable mit Literalen in einer switch Anweisung zu vergleichen. NullPointerException Sie sicher, dass der String nicht NullPointerException ist, andernfalls wird immer eine NullPointerException . Die Werte werden mit String.equals verglichen, dh die Groß- und Kleinschreibung wird String.equals .

```
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;
}
```

Live-Demo

Strings mit konstanten Werten vergleichen

Wenn Sie einen String mit einem konstanten Wert vergleichen, können Sie den konstanten Wert auf die linke Seite von equals um sicherzustellen, dass Sie keine NullPointerException wenn der andere String null .

```
"baz".equals(foo)
```

Während foo.equals("baz") eine NullPointerException wenn foo null , wird "baz".equals(foo) zu false ausgewertet.

Java SE 7

Eine besser lesbare Alternative ist die Verwendung von Objects.equals() , die beide Parameter auf Null überprüft: Objects.equals(foo, "baz") .

(**Anmerkung**: Es ist fraglich, ob es besser ist, NullPointerExceptions generell zu vermeiden oder sie passieren zu lassen und dann die eigentliche Ursache zu beheben; siehe hier und hier . Es ist nicht zu rechtfertigen, die Vermeidungsstrategie als "Best Practice" zu bezeichnen.)

Stringbestellungen

Die String Klasse implementiert String.compareTo Comparable<String> mit der String.compareTo Methode (wie zu Beginn dieses Beispiels beschrieben). Dadurch wird die natürliche Reihenfolge von String Objekten zwischen Groß- und Kleinschreibung unterschieden. Die String Klasse stellt eine Comparator<String> Konstante Comparator<String> Namen CASE_INSENSITIVE_ORDER die für die Sortierung nach Groß- und Kleinschreibung geeignet ist.

Vergleich mit internen Strings

Die Java-Sprachspezifikation (JLS 3.10.6) besagt Folgendes:

> „Darüber hinaus ist ein Stringliteral bezieht sich immer auf die gleiche Instanz der Klasse String Dies liegt daran , Stringliterale -. Oder, allgemeiner, Strings , die die Werte der konstanten Ausdrücke sind - *interniert* sind , um eindeutige Instanzen zu teilen, wobei das Verfahren unter Verwendung von String.intern . "

Das heißt, es ist sicher, Referenzen auf zwei String- *Literale* mithilfe von == zu vergleichen. Dasselbe gilt für Verweise auf String Objekte, die mit der Methode String.intern() .

Zum Beispiel:

```
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");
}
```

Hinter den Kulissen führt der Internierungsmechanismus eine Hashtabelle, die alle intern erreichbaren Zeichenfolgen enthält, die noch *erreichbar sind* . Wenn Sie intern() für einen String aufrufen, sucht die Methode das Objekt in der Hash-Tabelle:

- Wenn der String gefunden wird, wird dieser Wert als interner String zurückgegeben.
- Andernfalls wird der Hashtabelle eine Kopie der Zeichenfolge hinzugefügt, und diese Zeichenfolge wird als interne Zeichenfolge zurückgegeben.

Es ist möglich, Interning zu verwenden, um den Vergleich von Strings mit == . Dabei gibt es jedoch erhebliche Probleme. Siehe Pitfall - Interning Strings, sodass Sie == verwenden können, ist eine schlechte Idee für Details. In den meisten Fällen wird dies nicht empfohlen.

Ändern der Groß- / Kleinschreibung von Zeichen in einem String

Der String Typ bietet zwei Methoden zum Konvertieren von Zeichenfolgen zwischen Groß- und Kleinschreibung:

- toUpperCase , um alle Zeichen in toUpperCase zu konvertieren
- toLowerCase , um alle Zeichen in toLowerCase zu konvertieren

Diese Methoden geben beide die konvertierten Zeichenfolgen als neue String Instanzen zurück. Die ursprünglichen String Objekte werden nicht geändert, da String in Java unveränderlich ist. Weitere Informationen zur Unveränderlichkeit finden Sie hier: Unveränderlichkeit von Strings in Java

```
String string = "This is a Random String";
String upper = string.toUpperCase();
String lower = string.toLowerCase();

System.out.println(string);   // prints "This is a Random String"
System.out.println(lower);    // prints "this is a random string"
System.out.println(upper);    // prints "THIS IS A RANDOM STRING"
```

Nicht alphabetische Zeichen wie Ziffern und Satzzeichen bleiben von diesen Methoden unberührt. Beachten Sie, dass diese Methoden unter bestimmten Bedingungen auch falsch mit bestimmten Unicode-Zeichen umgehen können.

Hinweis : Diese Methoden sind *abhängig vom Gebietsschema* und können zu unerwarteten Ergebnissen führen, wenn sie für Zeichenfolgen verwendet werden, die unabhängig vom Gebietsschema interpretiert werden sollen. Beispiele sind Programmiersprachen-IDs, Protokollschlüssel und HTML Tags.

Beispielsweise gibt "TITLE".toLowerCase() in einem türkischen Gebietsschema " tıtle " zurück, wobei ı (\u0131) der Buchstabe LATIN SMALL LETTER DOTLESS I ist . Locale.ROOT als Parameter an die entsprechende Methode zur toLowerCase(Locale.ROOT) z. B. toLowerCase(Locale.ROOT) oder toUpperCase(Locale.ROOT)), um korrekte Ergebnisse für toUpperCase(Locale.ROOT) .

Obwohl die Verwendung von Locale.ENGLISH für die meisten Fälle auch richtig ist, ist die **Sprache , unveränderliche** Art und Weise Locale.ROOT .

Eine detaillierte Liste der Unicode-Zeichen, für die ein spezielles Gehäuse erforderlich ist, finden Sie auf der Website des Unicode Consortium .

Groß- und Kleinschreibung eines bestimmten Zeichens innerhalb einer ASCII-Zeichenfolge ändern:

Um den Fall eines bestimmten Zeichens einer ASCII-Zeichenfolge zu ändern, kann folgender Algorithmus verwendet werden:

Schritte:

1. Deklarieren Sie einen String.
2. Geben Sie die Zeichenfolge ein.
3. Konvertieren Sie die Zeichenfolge in ein Zeichenfeld.
4. Geben Sie das Zeichen ein, nach dem gesucht werden soll.
5. Suchen Sie nach dem Zeichen im Zeichenfeld.
6. Wenn gefunden, überprüfen Sie, ob das Zeichen Klein- oder Großbuchstaben ist.
    - Wenn Sie Großbuchstaben verwenden, fügen Sie dem ASCII-Code des Zeichens 32 hinzu.
    - Bei Kleinbuchstaben subtrahieren Sie 32 vom ASCII-Code des Zeichens.
7. Ändern Sie das ursprüngliche Zeichen aus dem Zeichen-Array.
8. Konvertieren Sie das Zeichenarray wieder in die Zeichenfolge.

Voila, der Fall des Charakters wurde geändert.

Ein Beispiel für den Code für den Algorithmus ist:

```
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);
```

Suchen einer Zeichenfolge in einer anderen Zeichenfolge

Um zu überprüfen, ob ein bestimmter String a in einem String b oder nicht, können wir die Methode String.contains() mit folgender Syntax verwenden:

```
b.contains(a); // Return true if a is contained in b, false otherwise
```

Die Methode String.contains() kann verwendet werden, um zu überprüfen, ob eine CharSequence in der CharSequence kann. Die Methode sucht nach der Zeichenfolge a in der Zeichenfolge b .

```
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
```

Live Demo auf Ideone

Um die genaue Position zu finden, an der ein String in einem anderen String beginnt, verwenden Sie 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
```

Live Demo auf Ideone

Die String.indexOf() Methode gibt den ersten Index einen char oder String in einem anderen String . Die Methode gibt -1 wenn sie nicht gefunden wird.

Hinweis : Die Methode String.indexOf() Groß- und Kleinschreibung.

Beispiel für eine Suche, bei der der Fall ignoriert wird:

```
String str1 = "Hello World";
String str2 = "wOr";
str1.indexOf(str2);                                 // -1
str1.toLowerCase().contains(str2.toLowerCase());  // true
str1.toLowerCase().indexOf(str2.toLowerCase());   // 6
```

Live Demo auf Ideone

Länge eines Strings ermitteln

Um die Länge eines String Objekts String , rufen Sie die length() -Methode auf. Die Länge entspricht der Anzahl der UTF-16-Codeeinheiten (Zeichen) in der Zeichenfolge.

```
String str = "Hello, World!";
System.out.println(str.length()); // Prints out 13
```

Live Demo auf Ideone

Ein char in einem String ist der UTF-16-Wert. Unicode-Codepunkte mit Werten ≥ 0x1000 (zum Beispiel die meisten Emojis) verwenden zwei Zeichenpositionen. Um die Anzahl der Unicode - Codepoints in einem String zu zählen, unabhängig davon , ob die einzelnen Codepoint passen in einem UTF-16 char Wert, können Sie die Verwendung codePointCount Methode:

```
int length = str.codePointCount(0, str.length());
```

Sie können ab Java 8 auch einen Stream mit Codepoints verwenden:

```
int length = str.codePoints().count();
```

Substrings

```
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")
```

Teilstrings können auch auf das Slice angewendet und Zeichen in den ursprünglichen String eingefügt / ersetzt werden. Beispielsweise haben Sie ein chinesisches Datum gefunden, das chinesische Zeichen enthält, Sie möchten es jedoch als Datumszeichenfolge speichern.

```
String datestring = "2015□11□17□"
datestring = datestring.substring(0, 4) + "-" + datestring.substring(5,7) + "-" +
datestring.substring(8,10);
//Result will be 2015-11-17
```

Die Teilzeichenfolge- Methode extrahiert ein Stück einer String . Wenn ein Parameter angegeben wird, ist der Parameter der Anfang und das Teil erstreckt sich bis zum Ende des String . Wenn zwei Parameter angegeben werden, ist der erste Parameter das Startzeichen und der zweite Parameter der Index des Zeichens direkt nach dem Ende (das Zeichen am Index ist nicht enthalten). Eine einfache Möglichkeit zur Überprüfung besteht darin, dass die Subtraktion des ersten Parameters vom zweiten Parameter die erwartete Länge der Zeichenfolge ergibt.

Java SE 7

In JDK <7u6 Versionen der substring Methode instanziiert ein String , der die gleiche Unterstützung teilt char[] wie das Original String und hat die interne offset und count Felder auf dem Ergebnis Start und Länge. Eine solche gemeinsame Nutzung kann zu Speicherverlusten führen, die verhindert werden können, indem der new String(s.substring(...)) , um die Erstellung einer Kopie zu erzwingen. new String(s.substring(...)) kann char[] Speicherbereinigung erhalten.

Java SE 7

Von JDK 7u6 der substring Methode immer kopiert die gesamte zugrunde liegende char[] Array, so

dass die Komplexität linear im Vergleich zum vorherigen konstant einem , sondern das Fehlen von Speicherlecks in der gleichen Zeit zu gewährleisten.

Das n-te Zeichen in einem String abrufen

```
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"
```

Um die n - te Zeichen in einer Zeichenfolge zu erhalten, rufen Sie einfach charAt(n) auf einem String , wobei n der Index des Zeichens möchten Sie abrufen

HINWEIS: Der Index n beginnt bei 0 , das erste Element befindet sich also bei n = 0.

Plattformunabhängiges neues Trennzeichen

Da das neue Zeilentrennzeichen von Plattform zu Plattform variiert (z. B. \n bei Unix-ähnlichen Systemen oder \r\n bei Windows), ist häufig ein plattformunabhängiger Zugriff erforderlich. In Java kann es von einer Systemeigenschaft abgerufen werden:

```
System.getProperty("line.separator")
```

Java SE 7

Da das neue Zeilentrennzeichen so häufig benötigt wird, steht in Java 7 eine Verknüpfungsmethode zur Verfügung, die genau das gleiche Ergebnis wie der obige Code zurückgibt:

```
System.lineSeparator()
```

Hinweis : Da es sehr unwahrscheinlich ist, dass sich das neue Zeilentrennzeichen während der Programmausführung ändert, ist es eine gute Idee, es in einer statischen Endvariablen zu speichern, anstatt es jedes Mal, wenn es benötigt wird, aus der Systemeigenschaft abzurufen.

Wenn Sie String.format , verwenden Sie %n anstelle von \n oder '\ r \ n', um ein plattformunabhängiges neues Trennzeichen für Zeilen auszugeben.

```
System.out.println(String.format('line 1: %s.%nline 2: %s%n', lines[0],lines[1]));
```

Hinzufügen der toString () - Methode für benutzerdefinierte Objekte

Angenommen, Sie haben die folgende Person definiert:

```
public class Person {

    String name;
    int age;

    public Person (int age, String name) {
        this.age = age;
        this.name = name;
    }
}
```

Wenn Sie ein neues Person instanziieren:

```
Person person = new Person(25, "John");
```

und später in Ihrem Code verwenden Sie die folgende Anweisung, um das Objekt zu drucken:

```
System.out.println(person.toString());
```

Live Demo auf Ideone

Sie erhalten eine Ausgabe ähnlich der folgenden:

```
Person@7ab89d
```

Dies ist das Ergebnis der Implementierung der toString() Methode, die in der Object Klasse, einer Superklasse von Person . Die Dokumentation von Object.toString() besagt:

> Die toString-Methode für die Klasse Object gibt eine Zeichenfolge zurück, die aus dem Namen der Klasse, deren Objekt das Objekt ist, dem Zeichen @, und der vorzeichenlosen hexadezimalen Darstellung des Hashcodes des Objekts besteht. Mit anderen Worten, diese Methode gibt einen String zurück, der dem Wert von entspricht:
>
> ```
> getClass().getName() + '@' + Integer.toHexString(hashCode())
> ```

Für eine sinnvolle Ausgabe müssen Sie daher die toString() -Methode **überschreiben** :

```
@Override
public String toString() {
    return "My name is " + this.name + " and my age is " + this.age;
}
```

Jetzt wird die Ausgabe sein:

```
My name is John and my age is 25
```

Du kannst auch schreiben

```
System.out.println(person);
```

Live Demo auf Ideone

Tatsächlich ruft println() die toString Methode implizit für das Objekt auf.

Saiten teilen

Sie können einen String für ein bestimmtes Begrenzungszeichen oder einen regulären Ausdruck String.split() Sie können die String.split() -Methode verwenden, die die folgende Signatur aufweist:

```
public String[] split(String regex)
```

Beachten Sie, dass Begrenzungszeichen oder reguläre Ausdrücke aus dem resultierenden String-Array entfernt werden.

Beispiel mit Begrenzungszeichen:

```
String lineFromCsvFile = "Mickey;Bolton;12345;121216";
String[] dataCells = lineFromCsvFile.split(";");
// Result is dataCells = { "Mickey", "Bolton", "12345", "121216"};
```

Beispiel mit regulärem Ausdruck:

```
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?"};
```

Sie können sogar ein String Literal direkt aufteilen:

```
String[] firstNames = "Mickey, Frank, Alicia, Tom".split(", ");
// Result is firstNames = {"Mickey", "Frank", "Alicia", "Tom"};
```

Warnung : Vergessen Sie nicht, dass der Parameter immer als regulärer Ausdruck behandelt wird.

```
"aaa.bbb".split("."); // This returns an empty array
```

Im vorigen Beispiel . wird als Platzhalter für reguläre Ausdrücke behandelt, der mit einem beliebigen Zeichen übereinstimmt. Da jedes Zeichen ein Trennzeichen ist, ist das Ergebnis ein leeres Array.

Aufteilung basierend auf einem Trennzeichen, das ein Regex-Meta-Zeichen ist

Die folgenden Zeichen werden in Regex als Sonderzeichen (auch als Metazeichen bezeichnet) bezeichnet

```
  < > - = ! ( ) [ ] { } \ ^ $ | ? * + .
```

Um eine Zeichenfolge basierend auf einem der oben genannten Trennzeichen zu Pattern.quote() , müssen Sie sie entweder mit \\ *escape* oder mit Pattern.quote() :

- Pattern.quote() :

  ```
  String s = "a|b|c";
  String regex = Pattern.quote("|");
  String[] arr = s.split(regex);
  ```

- Die Sonderzeichen umgehen:

  ```
  String s = "a|b|c";
  String[] arr = s.split("\\|");
  ```

Split entfernt leere Werte

split(delimiter) entfernt standardmäßig leere Zeichenfolgen aus dem Ergebnis-Array. Um diesen Mechanismus zu deaktivieren, müssen Sie eine überladene Version von split(delimiter, limit) wobei der Grenzwert auf einen negativen Wert gesetzt ist, wie

```
String[] split = data.split("\\|", -1);
```

split(regex) intern das Ergebnis der split(regex, 0) .

Der Parameter limit steuert, wie oft das Muster angewendet wird, und wirkt sich daher auf die Länge des resultierenden Arrays aus.
Wenn die Grenze *n* größer als Null ist, wird das Muster höchstens *n - 1* mal angewendet, die Länge des Arrays ist nicht größer als *n* , und der letzte Eintrag des Arrays enthält alle Eingaben, die über den letzten übereinstimmenden Begrenzer liegen.
Wenn *n* negativ ist, wird das Muster so oft wie möglich angewendet und das Array kann eine beliebige Länge haben.
Wenn *n gleich* Null ist, wird das Muster so oft wie möglich angewendet, das Array kann beliebig

lang sein und nachfolgende leere Zeichenfolgen werden verworfen.

StringTokenizer mit einem StringTokenizer

Neben der split() Methode können Strings auch mit einem StringTokenizer .

StringTokenizer ist noch restriktiver als String.split() und auch etwas schwieriger zu verwenden. Es ist im Wesentlichen für das Herausziehen von Token gedacht, die durch einen festen Zeichensatz (als String) begrenzt sind. Jedes Zeichen dient als Trennzeichen. Aufgrund dieser Einschränkung ist es etwa doppelt so schnell wie String.split() .

Der standardmäßige Zeichensatz ist ein Leerzeichen (\t\n\r\f). Das folgende Beispiel wird jedes Wort separat ausdrucken.

```
String str = "the lazy fox jumped over the brown fence";
StringTokenizer tokenizer = new StringTokenizer(str);
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

Dies wird ausgedruckt:

```
the
lazy
fox
jumped
over
the
brown
fence
```

Sie können verschiedene Zeichensätze zur Trennung verwenden.

```
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());
}
```

Dies wird ausgedruckt:

```
j
mp
d ov
r
```

Strings mit einem Trennzeichen verbinden

Java SE 8

Ein String-Array kann mit der statischen Methode String.join() :

```
String[] elements = { "foo", "bar", "foobar" };
String singleString = String.join(" + ", elements);

System.out.println(singleString); // Prints "foo + bar + foobar"
```

Ebenso gibt es eine überladene String.join() -Methode für Iterable s.

Um eine genaue Kontrolle über das Verbinden zu erhalten, können Sie die Klasse StringJoiner verwenden :

```
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]"
```

Um einem Stream von Strings beizutreten, können Sie den Verbindungs-Collector verwenden :

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", "));
System.out.println(joined); // Prints "foo, bar, foobar"
```

Es gibt auch eine Option, um Präfix und Suffix zu definieren:

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", ", "{", "}"));
System.out.println(joined); // Prints "{foo, bar, foobar}"
```

Strings umkehren

Es gibt mehrere Möglichkeiten, wie Sie eine Zeichenfolge umkehren, um sie rückwärts zu machen.

1. StringBuilder / StringBuffer:

```
 String code = "code";
 System.out.println(code);

 StringBuilder sb = new StringBuilder(code);
 code = sb.reverse().toString();

 System.out.println(code);
```

2. Char Array:

```
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));
```

Anzahl der Vorkommen eines Teilstrings oder Zeichens in einer Zeichenfolge

countMatches Methode von org.apache.commons.lang3.StringUtils wird normalerweise verwendet, um die Häufigkeit eines Teilstrings oder eines Zeichens in einem 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
```

Andernfalls können Sie für reguläre Java-APIs reguläre Ausdrücke verwenden:

```
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;
}
```

Stringverkettung und StringBuilders

String-Verkettung kann mit dem Operator + werden. Zum Beispiel:

```
String s1 = "a";
String s2 = "b";
String s3 = "c";
String s = s1 + s2 + s3; // abc
```

Normalerweise führt eine Compilerimplementierung die obige Verkettung mit Methoden durch, die einen StringBuilder unter der Haube verwenden. Beim Kompilieren würde der Code wie folgt aussehen:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append("c").toString();
```

StringBuilder verfügt über mehrere überladene Methoden zum Anhängen verschiedener Typen, z. B. zum Anhängen eines int anstelle eines String . Beispielsweise kann eine Implementierung Folgendes konvertieren:

```
String s1 = "a";
String s2 = "b";
```

```
String s = s1 + s2 + 2; // ab2
```

Zu dem Folgendem:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append(2).toString();
```

Die obigen Beispiele veranschaulichen eine einfache Verkettung, die effektiv an einer Stelle im Code ausgeführt wird. Die Verkettung umfasst eine einzelne Instanz des StringBuilder . In einigen Fällen wird eine Verkettung kumulativ durchgeführt, z. B. in einer Schleife:

```
String result = "";
for(int i = 0; i < array.length; i++) {
    result += extractElement(array[i]);
}
return result;
```

In solchen Fällen wird die Compileroptimierung normalerweise nicht angewendet, und bei jeder Wiederholung wird ein neues StringBuilder Objekt erstellt. Dies kann optimiert werden, indem der Code explizit in einen einzelnen StringBuilder :

```
StringBuilder result = new StringBuilder();
for(int i = 0; i < array.length; i++) {
    result.append(extractElement(array[i]));
}
return result.toString();
```

Ein StringBuilder wird mit einem Leerzeichen von nur 16 Zeichen initialisiert. Wenn Sie im Voraus wissen, dass Sie größere Zeichenfolgen erstellen, kann es vorteilhaft sein, sie mit ausreichender Größe im Voraus zu initialisieren, damit der interne Puffer nicht in der Größe geändert werden muss:

```
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
```

Wenn Sie viele Strings produzieren, ist es ratsam, StringBuilder wiederzuverwenden:

```
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());
}
```

Wenn (und nur wenn) mehrere Threads in *denselben* Puffer schreiben, verwenden Sie StringBuffer , eine synchronized Version von StringBuilder . Da jedoch normalerweise nur ein einzelner Thread in einen Puffer schreibt, ist es normalerweise schneller, StringBuilder ohne Synchronisation zu verwenden.

Verwenden der concat () -Methode:

```
String string1 = "Hello ";
String string2 = "world";
String string3 = string1.concat(string2);  // "Hello world"
```

Dies gibt eine neue Zeichenfolge zurück, die string1 ist und am Ende mit string2 versehen wird. Sie können die concat () -Methode auch mit String-Literalen verwenden, wie in:

```
"My name is ".concat("Buyya");
```

Teile von Strings ersetzen

Zwei Möglichkeiten zum Ersetzen: durch Regex oder durch exakte Übereinstimmung.

Hinweis: Das ursprüngliche String-Objekt bleibt unverändert, der Rückgabewert enthält den geänderten String.

Genaue Übereinstimmung

Ersetzen Sie ein einzelnes Zeichen durch ein anderes einzelnes Zeichen:

```
String replace(char oldChar, char newChar)
```

> Gibt eine neue Zeichenfolge zurück, die sich aus dem Ersetzen aller Vorkommen von oldChar in dieser Zeichenfolge durch newChar ergibt.

```
String s = "popcorn";
System.out.println(s.replace('p','W'));
```

Ergebnis:

```
WoWcorn
```

Ersetzen Sie die Zeichenfolge durch eine andere Zeichenfolge:

```
String replace(CharSequence target, CharSequence replacement)
```

> Ersetzt jeden Teilstring dieser Zeichenfolge, der der Literal-Zielsequenz entspricht, durch die angegebene Literal-Ersetzungssequenz.

```
String s = "metal petal et al.";
System.out.println(s.replace("etal","etallica"));
```

Ergebnis:

```
metallica petallica et al.
```

Regex

Hinweis : Die Gruppierung verwendet das Zeichen $ um auf die Gruppen zu verweisen, wie beispielsweise $1 .

Alle Spiele ersetzen:

```
String replaceAll(String regex, String replacement)
```

> Ersetzt jede Teilzeichenfolge dieser Zeichenfolge, die dem angegebenen regulären Ausdruck entspricht, durch die angegebene Ersetzung.

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Ergebnis:

```
spiral metallica petallica et al.
```

Nur das erste Spiel ersetzen:

```
String replaceFirst(String regex, String replacement)
```

> Ersetzt den ersten Teilstring dieser Zeichenfolge, der dem angegebenen regulären Ausdruck entspricht, durch die angegebene Ersetzung

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)","$1lica"));
```

Ergebnis:

```
spiral metallica petal et al.
```

Entfernen Sie Whitespace vom Anfang und Ende einer Zeichenfolge

Die trim() -Methode gibt einen neuen String zurück, wobei der führende und der nachfolgende Leerraum entfernt werden.

```
String s = new String("   Hello World!!  ");
String t = s.trim();  // t = "Hello World!!"
```

Wenn Sie trim einen String, der keine Leerzeichen hat zu entfernen, müssen Sie die gleiche String - Instanz zurückgegeben werden.

Beachten Sie, dass die trim() -Methode eine eigene Vorstellung von Whitespace hat , die sich von der von der Character.isWhitespace() -Methode verwendeten Vorstellung unterscheidet:

- Alle ASCII-Steuerzeichen mit den Codes U+0000 bis U+0020 werden als Leerzeichen betrachtet und mit trim() . Dies beinhaltet U+0020 'SPACE' , U+0009 'CHARACTER TABULATION' , U+000A 'LINE FEED' und U+000D 'CARRIAGE RETURN' , aber auch die Zeichen wie U+0007 'BELL' .
- Unicode-Whitespace wie U+00A0 'NO-BREAK SPACE' oder U+2003 'EM SPACE' werden von trim() *nicht* erkannt.

String-Pool und Heapspeicher

Wie viele Java-Objekte werden **alle** String Instanzen auf dem Heap erstellt, sogar Literale. Wenn die JVM ein String Literal findet, das keine entsprechende Referenz im Heap hat, erstellt die JVM eine entsprechende String Instanz auf dem Heap **und** speichert auch einen Verweis auf die neu erstellte String Instanz im String-Pool. Alle anderen Verweise auf dasselbe String Literal werden durch die zuvor erstellte String Instanz im Heap ersetzt.

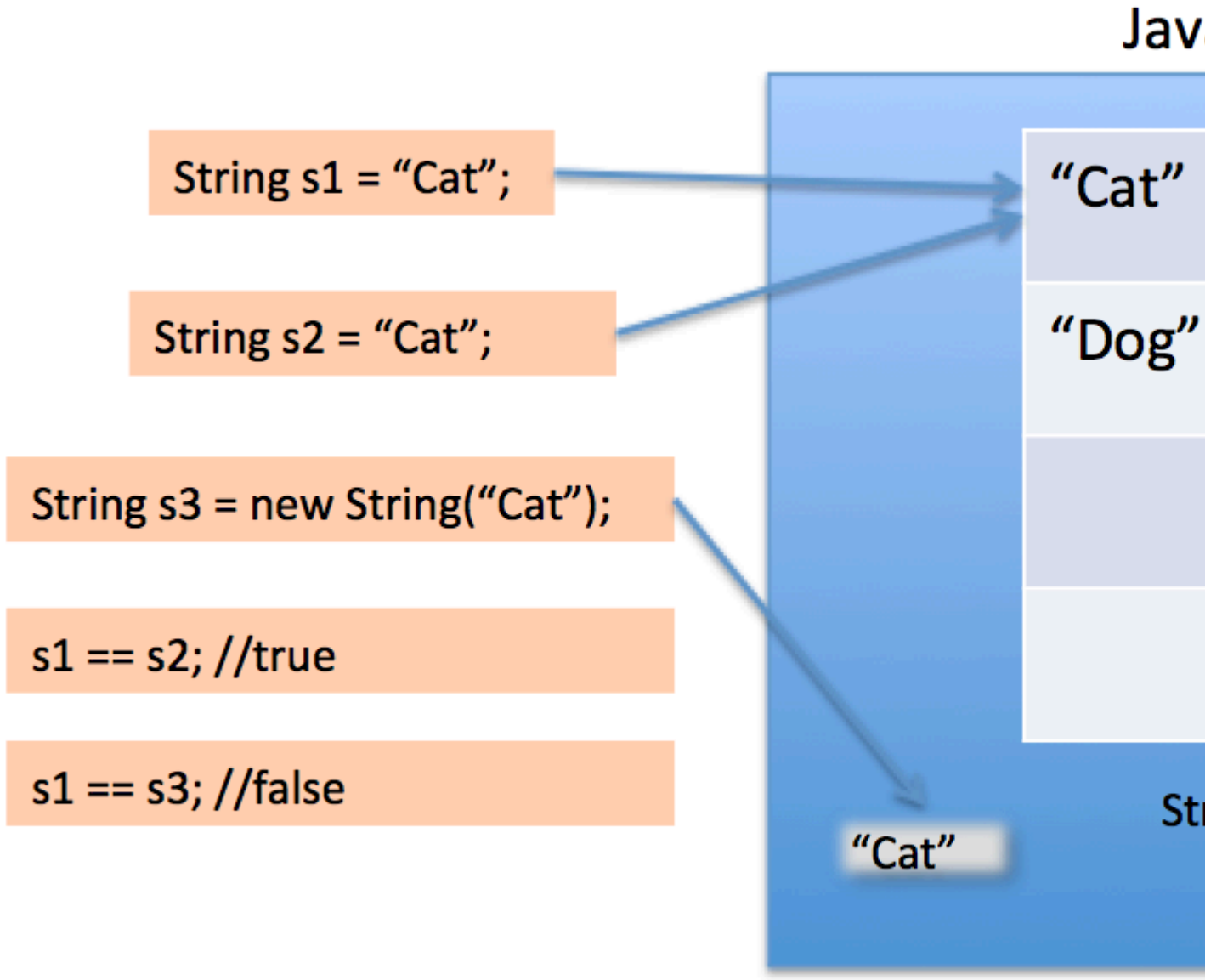
Schauen wir uns das folgende Beispiel an:

```
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);
    }
}
```

Die Ausgabe des obigen ist:

```
true
true
true
true
```



Wenn Sie einen String in doppelte Anführungszeichen setzen, wird zuerst nach String mit demselben Wert im String-Pool gesucht. Wenn er gefunden wird, wird nur der Verweis zurückgegeben. Andernfalls wird ein neuer String im Pool erstellt, und der Verweis wird zurückgegeben.

Mit dem neuen Operator erzwingen wir jedoch die String-Klasse, um ein neues String-Objekt im Heap-Bereich zu erstellen. Wir können die intern () -Methode verwenden, um sie in den Pool

einzufügen, oder auf ein anderes String-Objekt aus einem String-Pool mit demselben Wert verweisen.

Der String-Pool selbst wird auch auf dem Heap erstellt.

Java SE 7

Vor Java 7 wurden String **Literale** im Laufzeitkonstantenpool im Methodenbereich von PermGen , der eine feste Größe hatte.

Der String-Pool befand sich auch in PermGen .

Java SE 7

RFC: 6962931

> In JDK 7 werden internierte Zeichenfolgen nicht mehr bei der permanenten Generierung des Java-Heaps zugewiesen, sondern im Hauptteil des Java-Heaps (als junge und alte Generationen bezeichnet), zusammen mit den anderen von der Anwendung erstellten Objekten . Diese Änderung führt dazu, dass sich mehr Daten im Haupt-Java-Heap und weniger Daten bei der permanenten Generierung befinden. Daher müssen möglicherweise die Heap-Größen angepasst werden. Die meisten Anwendungen werden aufgrund dieser Änderung nur relativ geringe Unterschiede in der Heap-Nutzung String.intern() größeren Anwendungen, die viele Klassen laden oder die String.intern() -Methode String.intern() verwenden, werden jedoch größere Unterschiede auftreten.

Groß- und Kleinschreibung

Java SE 7

switch selbst kann nicht so konfiguriert werden, dass er nicht zwischen Groß- und Kleinschreibung unterscheidet. Wenn er jedoch unbedingt erforderlich ist, kann er sich mit toLowerCase() oder toUpperCase unempfindlich gegenüber der Eingabezeichenfolge toUpperCase :

```
switch (myString.toLowerCase()) {
     case "case1" :
        ...
     break;
     case "case2" :
        ...
     break;
}
```

In acht nehmen

- Locale kann beeinflussen, wie sich die Fälle ändern.
- Es muss darauf geachtet werden, dass in den Beschriftungen keine Großbuchstaben enthalten sind - diese werden niemals ausgeführt!

Zeichenketten online lesen: https://riptutorial.com/de/java/topic/109/zeichenketten

Kapitel 184: Zeichenkodierung

Examples

Lesen von Text aus einer in UTF-8 codierten Datei

```
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);
    }
}
```

Schreiben von Text in eine Datei in UTF-8

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class WritingUTF8TextFile {
    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for ealier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test2.txt"),
StandardCharsets.UTF_8)) {
            wr.write("Cyrillic symbol Ы");
        }
    }
```

```
}
```

Byte-Darstellung einer Zeichenfolge in UTF-8 abrufen

```
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));
    }
}
```

Zeichenkodierung online lesen: https://riptutorial.com/de/java/topic/2735/zeichenkodierung

Kapitel 185: Zufallszahlengenerierung

Bemerkungen

Nichts ist wirklich zufällig und daher nennt der Javadoc diese Zahlen pseudozufällig. Diese Nummern werden mit einem Pseudozufallszahlengenerator erstellt .

Examples

Pseudo-Zufallszahlen

Java bietet als Teil des utils Pakets einen grundlegenden Pseudo-Zufallszahlengenerator, der entsprechend als Random . Dieses Objekt kann verwendet werden, um einen Pseudozufallswert als einen der eingebauten numerischen Datentypen (int , float usw.) zu generieren. Sie können es auch verwenden, um einen zufälligen booleschen Wert oder ein zufälliges Byte-Array zu generieren. Ein Beispiel ist wie folgt:

```
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.
```

HINWEIS: Diese Klasse produziert nur ziemlich minderwertige Pseudozufallszahlen, und sollte nie Zufallszahlen für Verschlüsselungsoperationen oder andere Situationen , in denen hochwertigere Zufälligkeit kritisch zu erzeugen , verwendet werden (für das, würden Sie wollen , die verwenden SecureRandom Klasse, wie unten angegeben). Eine Erklärung für die Unterscheidung zwischen "sicherer" und "unsicherer" Zufälligkeit liegt außerhalb des Rahmens dieses Beispiels.

Pseudo-Zufallszahlen in einem bestimmten Bereich

Die Methode nextInt(int bound) von Random akzeptiert eine obere Exklusivgrenze, dh eine Zahl, deren zurückgegebener Zufallswert kleiner sein muss. Allerdings akzeptiert nur die nextInt Methode eine Bindung. nextLong , nextDouble usw. nicht.

```
Random random = new Random();
random.nextInt(1000); // 0 - 999

int number = 10 + random.nextInt(100); // number is in the range of 10 to 109
```

Ab Java 1.7 können Sie auch ThreadLocalRandom (source) verwenden. Diese Klasse stellt ein threadsicheres PRNG (Pseudo-Random Number Generator) bereit. Beachten Sie, dass die nextInt Methode dieser Klasse sowohl eine obere als auch eine untere Grenze akzeptiert.

```
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);
```

Beachten Sie, dass die offizielle Dokumentation besagt, dass nextInt(int bound) seltsame Dinge nextInt(int bound) kann, wenn die bound in der Nähe von $2^{30}+1$ liegt (Hervorhebung hinzugefügt):

> Der Algorithmus ist etwas schwierig. **Werte, die zu einer ungleichmäßigen Verteilung führen würden, werden zurückgewiesen** (da 2 ^ 31 nicht durch n teilbar ist). Die Wahrscheinlichkeit, dass ein Wert abgelehnt wird, hängt von n ab. **Der ungünstigste Fall ist n = 2 ^ 30 + 1, für den die Wahrscheinlichkeit einer Zurückweisung 1/2 ist und die erwartete Anzahl von Iterationen vor Abschluß der Schleife 2 beträgt.**

nextInt eine Grenze nextInt , wird die Leistung der nextInt Methode (geringfügig) verringert, und diese Leistungsabnahme wird deutlicher, wenn sich die bound dem halben max int-Wert nähert.

Generierung kryptographisch sicherer Pseudozufallszahlen

Random und ThreadLocalRandom sind gut genug für den täglichen Gebrauch, sie haben jedoch ein großes Problem: Sie basieren auf einem linearen Kongruenzgenerator , einem Algorithmus, dessen Ausgabe ziemlich leicht vorhergesagt werden kann. Daher sind diese beiden Klassen **nicht** für kryptographische Zwecke (z. B. zur Schlüsselgenerierung) geeignet.

java.security.SecureRandom in Situationen verwendet java.security.SecureRandom in denen ein PRNG mit einer Ausgabe erforderlich ist, die sehr schwer vorherzusagen ist. Das Vorhersagen der Zufallszahlen, die durch Instanzen dieser Klasse erstellt werden, ist schwer genug, um die Klasse als **kryptografisch sicher zu kennzeichnen** .

```
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));
    }
}
```

SecureRandom ist nicht nur kryptografisch sicher, SecureRandom verfügt auch über eine gigantische Periode von 2^{160} im Vergleich zu Random s von 2^{48} . Sie hat jedoch den Nachteil, dass sie wesentlich langsamer ist als Random und andere lineare PRNGs wie Mersenne Twister und Xorshift .

Beachten Sie, dass die SecureRandom-Implementierung sowohl plattform- als auch anbieterabhängig ist. Der SecureRandom (gegeben durch SUN - Anbieter in sun.security.provider.SecureRandom):

- auf Unix-ähnlichen Systemen mit Daten aus /dev/random und / oder /dev/urandom .
- unter Windows mit Aufrufe an CryptGenRandom() in CryptoAPI .

Wählen Sie Zufallszahlen ohne Duplikate

```
/**
 * 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;
}
```

Die Methode arbeitet, indem ein Array durchlaufen wird, das die angeforderte Länge hat und die verbleibende Länge möglicher Zahlen ermittelt. Es legt eine zufällige Anzahl dieser möglichen Zahlen newRandSpot und ermittelt diese Zahl innerhalb der nicht newRandSpot Anzahl. Dies geschieht durch Durchlaufen des Bereichs und Überprüfen, ob diese Nummer bereits vergeben ist.

Zum Beispiel, wenn der Bereich 5 ist und die Länge 3 ist und wir bereits die Zahl 2 gewählt haben. Dann haben wir 4 verbleibende Zahlen, so dass wir eine Zufallszahl zwischen 1 und 4 erhalten und wir durchlaufen den Bereich (5) und überspringen alle Zahlen das wir bereits verwendet haben (2).

Nehmen wir an, die nächste Zahl zwischen 1 und 4 ist 3. Die erste Schleife ergibt 1, die noch nicht genommen wurde, so dass wir 1 von 3 entfernen können, um 2 zu werden. Jetzt in der zweiten Schleife bekommen wir 2, die genommen wurde also machen wir nichts. Wir folgen diesem Muster, bis wir zu 4 gelangen, wenn wir 1 entfernen, wird es 0, und wir setzen die neue randomNumber auf 4.

Zufallszahlen mit einem angegebenen Startwert erzeugen

```
//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);
```

Wenn Sie den gleichen Startwert für die Generierung von Zufallszahlen verwenden, werden jedes Mal dieselben Zahlen zurückgegeben. Wenn Sie also für jede Random einen anderen Startwert Random empfiehlt es sich, wenn Sie nicht mit doppelten Zahlen enden möchten.

Eine gute Methode, um einen Long , der bei jedem Aufruf anders ist, ist `System.currentTimeMillis()` :

```
Random random = new Random(System.currentTimeMillis());
ThreadLocalRandom.current().setSeed(System.currentTimeMillis());
```

Zufallszahlen mit apache-common lang3 generieren

Wir können org.apache.commons.lang3.RandomUtils , um Zufallszahlen mithilfe einer einzelnen Zeile zu generieren.

```
int x = RandomUtils.nextInt(1, 1000);
```

Die Methode nextInt(int startInclusive, int endExclusive) nimmt einen Bereich an.

Abgesehen von int können wir mit dieser Klasse zufällige long , double , float und bytes generieren.

RandomUtils Klasse enthält die folgenden Methoden:

```
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.
```

Zufallszahlengenerierung online lesen:
https://riptutorial.com/de/java/topic/890/zufallszahlengenerierung

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Java Language	aa_oo, Aaqib Akhtar, abhinav, Abhishek Jain, Abob, acdcjunior, Adeel Ansari, adsalpha, AER, akhilsk, Akshit Soota, Alex A, alphaloop, altomnr, Amani Kilumanga, AndroidMechanic, Ani Menon, ankit dassor, Ankur Anand, antonio, Arkadiy, Ashish Ahuja, Ben Page, Blachshma, bpoiss , Burkhard, Carlton, Charlie H, Coffeehouse Coder, c□ LDS□ EED , Community, Confiqure, CraftedCart, dabansal, Daksh Gupta, Dan Hulme, Dan Morenus, DarkV1, David G., David Grinberg, David Newcomb, DeepCoder, Do Nhu Vy, Draken, Durgpal Singh, Dushko Jovanovski, E_net4, Edvin Tenovimas, Emil Sierżęga, Emre Bolat, enrico.bacis, Eran, explv, fgb, Francesco Menzani, Functino, garg10may, Gautam Jose, GingerHead, Grzegorz Górkiewicz, iliketocode, ıɯɐɓoɥ 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, Snađoш fa□, 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	2D-Grafiken in Java	17slim, ABDUL KHALIQ
3	Alternative Sammlungen	mnoronha, ppeterka, Viacheslav Vedenin
4	Anmerkungen	Ad Infinitum, Alon .G., Andrei Maieras, Andrii Abramov, bruno, Conrad.Dean, Dariusz, Demon Coldmist, Drizzt321, Dushko Jovanovski, fabian, faraa, GhostCat, hd84335, Hendrik Ebbers, J Atkin, Jorn Vernee, Kapep, Malt, MasterBlaster, matt freake, Nolequen, Ortomala Lokni, Ram, shmosel, Stephen C, Umberto Raimondi, Vogel612, ΦXocę □ Π epeúpa ヅ
5	Apache Commons Lang	Jonathan Barbero
6	AppDynamics und TIBCO BusinessWorks Instrumentation für einfache Integration	Alexandre Grimaud

7	Applets	ArcticLord, Enigo, MadProgrammer, ppeterka
8	Arrays	3442, 416E64726577, A Boschman, A.M.K, A_Arnold, Abhishek Jain, Abubakkar, acdcjunior, Ad Infinitum, Addis, Adrian Krebs, AER, afzalex, agilob, Alan, Alex Shesterov, Alexandru, altomnr, Amani Kilumanga, Andrew Tobilko, Andrii Abramov, AndroidMechanic, Anil, ankidaemon, ankit dassor, anotherGatsby, antonio, Ares, Arthur, Ashish Ahuja, assylias, AstroCB, baao, Beggs, Berzerk, Big Fan, BitNinja, bjb568, Blubberguy22, Bob Rivers, bpoiss, Bryan, BudsNanKis , Burkhard, bwegs, c1phr, Cache Staheli, Cerbrus, Charitha, Charlie H, Chris Midgley, Christophe Weis, Christopher Schneider, Codebender, coder-croc, Cold Fire, Colin Pickard , Community, Confiqure, CptEric, Daniel Käfer, Daniel Stradowski, Dariusz, DarkV1, David G., DeepCoder, Devid Farinelli, Dhrubajyoti Gogoi, Dmitry Ginzburg, dorukayhan, Duh-Wayne-101, Durgpal Singh, DVarga, Ed Cottrell, Edvin Tenovimas, Eilit, eisbehr, Elad, Emil Sierżęga, Emre Bolat, Eng.Fouad, enrico.bacis, Eran, Erik Minarini, Etki, explv, fabian, fedorqui, Filip Haglund, Forest White, fracz, Franck Dernoncourt, Functino, futureelite7, Gal Dreiman, gar, Gene Marin, GingerHead, granmirupa, Grexis, Grzegorz Sancewicz, Gubbel, Guilherme Torres Castro, Gustavo Coelho, hhj8i, Hiren, Idos, ihatecsv, iliketocode, Ilya, Ilyas Mimouni, intboolstring, Irfan, J Atkin, jabbathehutt1234, JakeD, James Taylor, Jamie, Jamie Rees, Janez Kuhar, Jared Rummler, Jargonius, Jason Sturges, JavaHopper, Javant, Jeeter, Jeffrey Bosboom, Jens Schauder, Jérémie Bolduc, Jeutnarg, jhnance, Jim Garrison, jitendra varshney, jmattheis, Joffrey, Johannes, johannes_preiser, John Slegers, JohnB, Jojodmo, Jonathan, Jordi Castilla, Jorn, Jorn Vernee, Josh, JStef, JudgingNotJudging, Justin, Kapep, KartikKannapur, Kayathiri, Kaz Wolfe, Kenster, Kevin Thorne , Lambda Ninja, Liju Thomas, llamositopia, Loris Securo, Luan Nico, Lucas Paolillo, maciek, Magisch, Makoto, Makyen, Malt, Marc, Markus, Marvin, MasterBlaster, Matas Vaitkevicius, matsve, Matt, Matt, Matthias Braun, Maxim Kreschishin, Maxim Plevako, Maximillian Laumeister, MC Emperor, Menasheh, Michael Piefel, michaelbahr, Miljen Mikic, Minhas Kamal, Mitch Talmadge, Mohamed Fadhl, Muhammed Refaat, Muntasir, Mureinik, Mzzzzzz, NageN, Nathaniel Ford, Nayuki, nicael, Nigel Nop, niyasc, noɥʎd ʎzɐɹƆ, 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, ΦXocę □ Π epeúpa ツ
9	Atomtypen	Daniel Nugent, Stephen C, Suminda Sirinath S. Dharmasena,

		xTrollxDudex
10	Audio	Dac Saunders, Petter Friberg, RamenChef, TNT, tonirush, Tot Zam, Vogel612
11	Aufteilen einer Schnur in Teile mit fester Länge	Bohemian
12	Aufzählung beginnend mit der Nummer	Sugan
13	Aufzählungen	1d0m3n30, A Boschman, aioobe, Amani Kilumanga, Andreas Fester, Andrew Sklyarevsky, Andrew Tobilko, Andrii Abramov, Anony-Mousse, bcosynot, Bob Rivers, coder-croc, Community, Constantine, Daniel Käfer, Daniel M., Danilo Guimaraes, DVarga, Emil Sierżęga, enrico.bacis, f_puras, fabian, Gal Dreiman, Gene Marin, Grexis, Grzegorz Oledzki, ipsi, J Atkin, Jared Hooper, javac, Jérémie Bolduc, Johannes, Jon Ericson, k3b, Kenster, Lahiru Ashan, Maarten Bodewes, madx, Mark, Michael Myers, Mick Mnemonic, NageN, Nef10, Nolequen, OldCurmudgeon, OliPro007, OverCoder, P.J.Meisch, Panther, Paweł Albecki, Petter Friberg, Punika, Radouane ROUFID, RamenChef, rd22, Ronon Dex, Ryan Hilbert, S.K. Venkat, Samk , shmosel, Spina, Stephen Leppik, Tarun Maganti, Tim, Torsten, VGR, Victor G., Vinay , Wolf, Yury Fedorov, Zefick , ΦXocę Перeúpa ツ
14	Ausdrücke	1d0m3n30, Andreas, EJP, Li357, RamenChef, shmosel, Stephen C, Stephen Leppik
15	Ausnahmen und Ausnahmebehandlung	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
16	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
17	Befehlszeile Argumentverarbeitung	Burkhard, Michael von Wenckstern, Stephen C
18	Benchmarks	esin88
19	BigDecimal	alain.janinm, Christian, Dth, Enigo, ggolding, Harish Gyanani, John Nash, Loris Securo, Łukasz Piaszczyk, Manish Kothari, mszymborski, RamenChef, sudo, xwoker
20	BigInteger	Alek Mieczkowski, Alex Shesterov, Amani Kilumanga, Andrii

		Abramov, azurefrog, Byte1518, dimo414, dorukayhan, Emil Sierżęga, fabian, GPI, Ha., hd84335, janos, Kaushal28, Maarten Bodewes, Makoto, matt freake, Md. Nasir Uddin Bhuiyan, Nufail, Pritam Banerjee, Ruslan Bes, ShivBuyya, Stendika, Vogel612
21	Bilder programmgesteuert erstellen	alain.janinm, Dariusz, kajacx, Kenster, mnoronha
22	Bit-Manipulation	Aimee Borda, Blubberguy22, dosdebug, esin88, Gerald Mücke, Jorn Vernee, Kineolyan, mnoronha, Nayuki, Rednivrug, Ryan Hilbert, Stephen C, thatguy
23	BufferedWriter	Andrii Abramov, fabian, Jorn Vernee, Robin, VatsalSura
24	ByteBuffer	Community, Jon Ericson, Jorn Vernee, Tarık Yılmaz, Tomasz Bawor, victorantunes, Vogel612
25	Bytecode-Änderung	bloo, Display Name, rakwaht, Squidward
26	C ++ - Vergleich	John DiFini
27	CompletableFuture	Adowrath, Kishore Tulsiani, WillShackleford
28	Datei 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
29	Datum und Uhrzeit (java.time. *)	Bilbo Baggins, bowmore, Michael Piefel, Miles, mnoronha, Simon, Squidward, Tarun Maganti, Vogel612, ΦXocę Пepeúpa ツ
30	Datumsklasse	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
31	Demontieren und Dekompilieren	ipsi, mnoronha
32	Dequeue-Schnittstelle	Suketu Patel
33	Der Java-Befehl - 'Java' und 'Javaw'	4444, Ben, mnoronha, Stephen C, Vogel612
34	Der Klassenpfad	Aaron Digulla, GPI, K'', Kenster, Ruslan Ulanov, Stephen C, trashgod
35	Die java.util.Objects-Klasse	mnoronha, RamenChef, Stephen C
36	Durchsetzung	Jonathan, Makoto, rajah9, RamenChef, The Guy with The Hat, Uri Agassi
37	Dynamischer Methodenversand	Jeet

38	Eigenschaften Klasse	17slim, Arthur, J Atkin, Jabir, KIRAN KUMAR MATAM, Marvin, peterh, Stephen C, VGR, vorburger
39	Einstellungen	RAnders00
40	Enum Map	KIRAN KUMAR MATAM
41	EnumSet-Klasse	KIRAN KUMAR MATAM
42	Erbe	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
43	Executor-, ExecutorService- und Thread-Pools	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
44	FileUpload in AWS	Amit Gujarathi
45	Fließende Schnittstelle	bn., noscreenname, P.J.Meisch, RamenChef, TuringTux
46	FTP (File Transfer Protocol)	Kelvin Kellner
47	Funktionale Schnittstellen	Andreas
48	Generics	1d0m3n30, 4444, Aaron Digulla, Abhishek Jain, Alex Meiburg, alex s, Andrei Maieras, Andrii Abramov, Anony-Mousse, Bart Enkelaar, bitek, Blubberguy22, Bob Brinks, Burkhard, Cache Staheli, Cannon, Ce7, Chriss, code11, Codebender, Daniel Figueroa, daphshez, DVarga, Emil Sierżęga, enrico.bacis, Eran, faraa, hd84335, hexafraction, Jan Vladimir Mostert, Jens Schauder, Jorn Vernee, Jude Niroshan, kcoppock, Kevin Montrose, Lahiru Ashan, Lii, manfcas, Mani Muthusamy, Marc, Matt, Mistalis, Mshnik, mvd, Mzzzzzz, NatNgs, nishizawa23, Oleg Sklyar, Onur, Ortomala Lokni, paisanco, Paul Bellora, Paweł Albecki, PcAF, Petter Friberg, phant0m, philnate, Radouane ROUFID, RamenChef, rap-2-h, rd22, Rogério, rolve, RutledgePaulV, S.K. Venkat, Siguza, Stephen C, Stephen Leppik, suj1th, tainy, ThePhantomGamer, Thomas, TNT, ʇolɐəz əɥʇ qoq, Unihedron, Vlad-HC, Wesley, Wilson, yiwei, Yury Fedorov
49	Getter und Setter	Fildor, Ironcache, Krös, martin, Petter Friberg, Stephen C, Sujith Niraikulathan, Thisaru Guruge, uzaif
50	Gleichzeitige Programmierung (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
51	Gleichzeitige Sammlungen	GPI, Kenster, Powerlord, user2296600
52	Grundlegende Kontrollstrukturen	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
53	Hash-tabelle	KIRAN KUMAR MATAM
54	Häufige Java-Fallstricke	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
55	HTTP-Verbindung	Community, Datagrammar, EJP, Inzimam Tariq IT, JonasCz, kiedysktos, Mureinik, NageN, Stephen C, still_learning
56	InputStreams und OutputStreams	akgren_soar, EJP, Gubbel, J Atkin, Jens Schauder, John Nash , Kip, KIRAN KUMAR MATAM, Matt Clark, Michael, RamenChef, Stephen C, Vogel612
57	Iterator und Iterable	Abubakkar, Comic Sans, Dariusz, Hulk, Lukas Knuth, RamenChef, Stephen C, user1121883, WillShackleford
58	JAR-Dateien mit mehreren Versionen	manouti
59	Java Compiler - "Javac"	CraftedCart, Jatin Balodhi, Mark Stewart, nishizawa23, Stephen C, Snađoшfa□, Tom Gijselinck
60	Java installieren (Standard Edition)	4444, Adeel Ansari, ajablonski, akhilsk, Alex A, altomnr, Ani Menon, Anthony Raymond, anuvab1911, Confiqure, CraftedCart, Emil Sierżęga, Gautam Jose, hd84335, ipsi, Jeffrey Brett Coleman, Lambda Ninja, Nithanim, Radouane ROUFID, Rakitić, ronnyfm, Sanandrea, Sandeep Chatterjee, sohnryang, Stephen C, Snađoшfa□, tonirush, Walery Strauch, Ze Rubeus
61	Java Native Access	Ezekiel Baniaga, Stephan, Stephen C
62	Java SE 7-Funktionen	compuhosny, RamenChef
63	Java SE 8-Funktionen	compuhosny, RamenChef, sun-solar-arrow
64	Java Virtual Machine (JVM)	Dushman, RamenChef, Rory McCrossan, Stephen Leppik

65	Java-Agenten	Display Name, mnoronha
66	JavaBean	foxt7ot, J. Pichardo, James Fry, SaWo, Stephen C
67	Java-Bereitstellung	garg10may, nishizawa23, Pseudonym Patel, RamenChef, Smit, Stephen C
68	Java-Code dokumentieren	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
69	Java-Code generieren	Tony
70	Java-Druckdienst	Danilo Guimaraes, Leonardo Pina
71	Java-Editionen, Versionen, Releases und Distributionen	Gal Dreiman, screab, Stephen C
72	Java-Fallstricke - Leistungsprobleme	Dorian, GPI, John Starich, Jorn Vernee, Michał Rybak, mnoronha, ppeterka, Sharon Rozinsky, steffen, Stephen C, xTrollxDudex
73	Java-Fallstricke - Nulls und NullPointerException	17slim, Andrii Abramov, Daniel Nugent, dorukayhan, fabian, François Cassin, Miles, Stephen C, Zircon
74	Java-Fallstricke - Sprachsyntax	Alex T., Cody Gray, Enwired, Friederike, Gal Dreiman, hd84335, Hiren, Peter Rader, piyush_baderia, RamenChef, Ravindra HV, RudolphEst, Stephen C, Todd Sewell, user3105453
75	Java-Fallstricke - Threads und Parallelität	dorukayhan, james large, Stephen C
76	Java-Fallstricke - Verwendung von Ausnahmen	Bhoomika, bruno, dimo414, Gal Dreiman, hd84335, SachinSarawgi, scorpp, Stephen C, Stephen Leppik, user3105453
77	Java-Gleitkommaoperationen	Dariusz, hd84335, HTNW, Ilya, Mr. P, Petter Friberg, ravthiru, Stephen C, Stephen Leppik, Vogel612
78	Java-Leistungsoptimierung	Gene Marin, jatanp, Stephen C, Vogel612
79	Java-Plugin-Systemimplementierungen	Alexiy
80	Java-Sockets	Nikhil R
81	Java-Speichermodell	Shree, Stephen C, Suminda Sirinath S. Dharmasena
82	Java-Speicherverwaltung	Daniel M., engineercoding, fgb, John Nash, jwd630, mnoronha , OverCoder, padippist, RamenChef, Squidward, Stephen C
83	JAXB	Dariusz, Drunix, fabian, hd84335, Jabir, ppeterka, Ram, Stephan, Thomas Fritsch, vallismortis, Walery Strauch
84	JAX-WS	ext1812, Jonathan Barbero, Stephen Leppik

85	JMX	esin88
86	JNDI	EJP, neohope, RamenChef
87	JShell	ostrichofevil, Sudip Bhandari
88	JSON in Java	Asaph, Bogdan Korinnyi, Burkhard, Cache Staheli, hd84335, ipsi, Jared Hooper, Kurzalead, MikaelF, Mrunal Pagnis, Nicholas J Panella, Nikita Kurtin, ppeterka, Prem Singh Bist, RamenChef, Ray Kiddy, SirKometa, still_learning, Stoyan Dekov, systemfreund, Tim, Vikas Gupta, vsminkov, Yury Fedorov
89	Just in Time (JIT) - Compiler	Liju Thomas, Stephen C
90	JVM-Flags	Configure, RamenChef
91	JVM-Tool-Schnittstelle	desilijic
92	Kalender und seine Unterklassen	Bob Rivers, cdm, kann, Makoto, mnoronha, ppeterka, Ram, VGR
93	Karten	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
94	Klasse - Java Reflection	gobes, KIRAN KUMAR MATAM
95	Klassen und Objekte	Community, Configure, Daniel LIn, Dave Ranjan, EJP, eveysky , fabian, Jens Schauder, Kevin Johnson, KIRAN KUMAR MATAM, MasterBlaster, Mureinik, Rakitić, Ram, RamenChef, Ryan Cocuzzo, Salman Kazmi, Tyler Zika
96	Klassenlader	FFY00, Flow, Holger, Makoto, Stephen C
97	Konsolen-E / A	Aaron Franke, Ani Menon, Erkan Haspulat, Francesco Menzani, jayantS, Lankymart, Loris Securo, manetsus, Olivier Grégoire, Petter Friberg, rolve, Saagar Jha, Stephen C
98	Konstrukteure	Andrii Abramov, Asiat, BrunoDM, ced-b, Codebender, Dylan, George Bailey, Jeremy, Ralf Kleberhoff, RamenChef, Thomas Gerot, tynn, Vogel612
99	Konvertieren in und aus Strings	Chirag Parmar, DarkV1, Gihan Chathuranga, Jabir, JonasCz, Kaushal28, Lachlan Dowding, Laurel, Maarten Bodewes, Matt Clark, PSo, RamenChef, Shaan, Stephen C, still_learning
100	Lambda-Ausdrücke	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
101	Laufzeitbefehle	RamenChef
102	Leser und Schriftsteller	JD9999, KIRAN KUMAR MATAM, Mureinik, Stephen C, VatsalSura
103	LinkedHashMap	Amit Gujarathi, KIRAN KUMAR MATAM
104	Liste vs SET	KIRAN KUMAR MATAM
105	Listen	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
106	Literale	1d0m3n30, EJP, ParkerHalo, Stephen C, ThePhantomGamer
107	log4j / log4j2	Daniel Wild, Fildor, HCarrasko, hd84335, Mrunal Pagnis, Rens van der Heijden
108	Lokale innere Klasse	KIRAN KUMAR MATAM
109	Lokalisierung und Internationalisierung	Code.IT, dimo414, Eduard Wirch, emotionlessbananas, Squidward, sun-solar-arrow
110	Methoden der Sammlungsfabrik	Jacob G.
111	Module	Jonathan, user140547
112	Nashorn-JavaScript-Engine	ben75, ekaerovets, Francesco Menzani, hd84335, Ilya, InitializeSahib, kasperjj, VatsalSura
113	Native Java-Schnittstelle	Coffee Ninja, Fjoni Yzeiri, Jorn Vernee, RamenChef, Stephen C, user1803551
114	Neue Datei-E / A	dorukayhan, niheno, TuringTux
115	Nichtzugriffsmodifizierer	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
116	NIO - Vernetzung	Matthieu, mnoronha
117	Objektklassenmethoden und Konstruktor	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
118	Objektklonen	Ayush Bansal, Christophe Weis, Jonathan
119	Objektreferenzen	Andrii Abramov, arcy, Vasiliy Vlasov
120	Operatoren	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
121	Oracle Official Code Standard	Ahmed Ashour, aioobe, akhilsk, alex s, Andrii Abramov, Cassio Mazzochi Molin, Dan Whitehouse, Enigo, erickson, f_puras, fabian, giucal, hd84335, J.D. Sandifer, Lahiru Ashan, Mac70, NamshubWriter, Nicktar, Petter Friberg, Pradatta, Pritam Banerjee, RamenChef, sanjaykumar81, Santa Claus, Santhosh Ramanan, VGR
122	Ortszeit	100rabh, A_Arnold, Alex, Andrii Abramov, Bob Rivers, Cache Staheli, DimaSan, Jasper, Kakarot, Kuroda, Manuel Vieda, Michael Piefel, phatfingers, RamenChef, Skylar Sutton, Vivek Anoop
123	Pakete	JamesENL, KIRAN KUMAR MATAM
124	Parallele Programmierung mit dem Fork / Join-Framework	Community, Joe C
125	Polymorphismus	Adrian Krebs, Amani Kilumanga, Daniel LIn, Dushman, Kakarot, Lernkurve, Markus L, NageN, Pawan, Ravindra babu, Saiful Azad, Stephen C
126	Primitive Datentypen	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
127	Protokollierung (java.util.logging)	bn., Christophe Weis, Emil Sierżęga, P.J.Meisch, vallismortis
128	Referenzdatentypen	Do Nhu Vy, giucal, Jorn Vernee, Lord Farquaad, Yohanes Khosiawan □□□

129	Referenztypen	EJP, NageN, Thisaru Guruge
130	Reflexions-API	Ali786, ArcticLord, Aurasphere, Blubberguy22, Bohemian, Christophe Weis, Drizzt321, fabian, hd84335, Joeri Hendrickx, Luan Nico, madx, Michael Myers, Onur, Petter Friberg, RamenChef, Ravindra babu, Squidward, Stephen C, Tony BenBrahim, Universal Electricity, ΦXocę Пepeúpa ツ
131	Reguläre Ausdrücke	Amani Kilumanga, Andy Thomas, Asaph, ced-b, Daniel M., fabian, hd84335, intboolstring, kaotikmynd, Laurel, Makoto, nhahtdh, ppeterka, Ram, RamenChef, Saif, Tot Zam, Unihedron , Vogel612
132	Rekursion	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
133	Remote Method Invocation (RMI)	RamenChef, smichel, Stephen C, user1803551, Vasiliy Vlasov
134	Ressourcen (auf Klassenpfad)	Androbin, Christian, Emily Mabrey, Enwired, fabian, Gerald Mücke, Jesse van Bekkum, Kenster, Stephen C, timbooo, VGR, vorburger
135	RSA-Verschlüsselung	Dennis Kriechel, Drunix, iqbal_cs, Maarten Bodewes, Nicktar , Shog9
136	Sammlungen	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
137	Sammlungen auswählen	John DiFini
138	Scanner	Alek Mieczkowski, Chirag Parmar, Community, Jon Ericson, JonasCz, Ram, RamenChef, Redterd, Stephen C, sun-solar-arrow, ΦXocę Пepeúpa ツ
139	Schnittstellen	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

140	Serialisierung	akhilsk, Batty, Bilesh Ganguly, Burkhard, EJP, emotionlessbananas, faraa, GradAsso, KIRAN KUMAR MATAM, noscreenname, Onur, rokonoid, Siva Sainath Reddy Bandi, Vasilis Vasilatos, Vasiliy Vlasov
141	ServiceLoader	fabian, Florian Genser, Gerald Mücke
142	Sets	A_Arnold, atom, ced-b, Chirag Parmar, Daniel Stradowski, demongolem, DimaSan, fabian, Kaushal28, Kenster
143	Sichere Objekte	Ankit Katiyar
144	Sicherheit & Kryptographie	John Nash, shibli049
145	Sicherheitsmanager	alphaloop, hexafraction, Uux
146	Sichtbarkeit (Kontrolle des Zugriffs auf Mitglieder einer Klasse)	Aasmund Eldhuset, Abhishek Balaji R, Catalina Island, Daniel M., intboolstring, Jonathan, Mark Yisri, Mureinik, NageN, ParkerHalo, Stephen C, Vogel612
147	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
148	SortedMap	Amit Gujarathi
149	Stack-Walking-API	manouti
150	Standardmethoden	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
151	Steckdosen	Ordiel
152	Streams	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, sujlth, thiagogcm, tpunt, Tunaki, Unihedron, user1133275, user1803551, Valentino, vincentvanjoe, vsnyc, Wilson, Ze Rubeus, zwl
153	String Tokenizer	M M
154	StringBuffer	Amit Gujarathi
155	StringBuilder	Andrii Abramov, Cache Staheli, David Soroko, Enigo, fabian, fgb, JudgingNotJudging, KIRAN KUMAR MATAM, Nicktar, P.J.Meisch, Stephen C
156	sun.misc.Unsafe	4444, Daniel Nugent, Grexis, Stephen C, Suminda Sirinath S. Dharmasena
157	Super Keyword	Abhijeet
158	ThreadLocal	Dariusz, Liju Thomas, Manish Kothari, Nithanim, taer
159	ThreadPoolExecutor in MultiThreaded-Anwendungen verwenden.	Brendon Dugan
160	TreeMap und TreeSet	Malt, Stephen C
161	Typumwandlung	4castle, Filip Smola, Joshua Carmody, Nick Donnelly, RamenChef, Squidward
162	Unit Testing	Ironcache
163	Unveränderliche Klasse	Mykola Yashchenko
164	Unveränderliche Objekte	1d0m3n30, Bohemian, Holger, Idcmp, Jon Ericson, kristyna, Michael Piefel, Stephen C, Vogel612
165	Varargs (variables Argument)	Daniel Nugent, Dushman, Omar Ayala, Rafael Pacheco, RamenChef, VGR, xsami
166	Verarbeiten	Andy Thomas, Bob Rivers, ppeterka, vorburger, yitzih
167	Vergleichbar und Vergleicher	Andrii Abramov, Conrad.Dean, Daniel Nugent, fabian, GPI, Hazem Farahat, JAVAC, Mshnik, Nolequen, Petter Friberg, Prateek Agarwal, sebkur, Stephen C
168	Verkapselung	Adam Ratzman, Adil, Daniel M., Drayke, VISHWANATH N P
169	Vernetzung	Arthur, Burkhard, devnull69, DonyorM, glee8e, Grayson Croom , Ilya, Malt, Matej Kormuth, Matthieu, Mine_Stone, ppeterka , RamenChef, Stephen C, Tot Zam, vsav
170	Verschachtelte und innere Klassen	ChemicalFlash, DimaSan, fgb, hd84335, Mshnik, RamenChef, Sandesh, sargue, Slava Babin, Stephen C, tynn
171	Verwenden anderer Skriptsprachen in Java	Nikhil R
172	Verwenden Sie das	17slim, Amir Rachum, Andrew Brooke, Arthur, ben75,

	statische Schlüsselwort	CarManuel, Daniel Nugent, EJP, Hi I'm Frogatto, Mark Yisri, Sadiq Ali, Skepter, Squidward
173	Wahlweise	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
174	Währung und Geld	Alexey Lagunov
175	Warteschlangen und Deques	Ad Infinitum, Alek Mieczkowski, Androbin, DimaSan, engineercoding, ppeterka, RamenChef, rd22, Samk, Stephen C
176	WeakHashMap	Amit Gujarathi, KIRAN KUMAR MATAM
177	XJC	Danilo Guimaraes, fabian
178	XML XPath-Bewertung	17slim, manouti
179	XML-Analyse mit den JAXP-APIs	GPI
180	XOM - XML-Objektmodell	Arthur, Makoto
181	Zahlenformat	arpit pandey, John Nash, RamenChef, ΦΧοςę □ Περεύρα ツ
182	Zeichenketten	17slim, A.J. Brown, A_Arnold, Abhishek Jain, Abubakkar, Adam Ratzman, Adrian Krebs, agilob, Aiden Deom, Alex Meiburg, Alex Shesterov, altomnr, Amani Kilumanga, Andrew Tobilko, Andrii Abramov, Andy Thomas, Anony-Mousse, Asaph, Ataeraxia, Austin, Austin Day, ben75, bfd, Bob Brinks, bpoiss, Burkhard, Cache Staheli, Caner Balım, Chris Midgley , Christian, Christophe Weis, coder-croc, Community, cyberscientist, Daniel Käfer, Daniel Stradowski, DarkV1, dedmass, DeepCoder, dnup1092, dorukayhan, drov, DVarga, ekeith, Emil Sierżęga, emotionlessbananas, enrico.bacis, Enwired, fabian, FlyingPiMonster, Gabriele Mariotti, Gal Dreiman, Gergely Toth, Gihan Chathuranga, GingerHead, giucal, Gray, GreenGiant, hamena314, Harish Gyanani, HON95, iliketocode, Ilya, Infuzed guy, intboolstring, J Atkin, Jabir, javac, JavaHopper, Jeffrey Lin, Jens Schauder, Jérémie Bolduc, John Slegers, Jojodmo, Jon Ericson, JonasCz , Jordi Castilla, Jorn Vernee, JSON C11, Jude Niroshan, Kamil Akhuseyinoglu, Kapep, Kaushal28, Kaushik NP, Kehinde Adedamola Shittu, Kenster, kstandell, Lachlan Dowding, Lahiru Ashan, Laurel, Leo Aso, Liju Thomas, LisaMM, M.Sianaki, Maarten Bodewes, Makoto, Malav, Malt, Manoj, Manuel Spigolon, Mark Stewart, Marvin, Matej Kormuth, Matt Clark, Matthias Braun, maxdev, Maxim Plevako, mayha, Michael, MikeW, Miles, Miljen Mikic, Misa Lazovic, mr5, Myridium, NikolaB, Nufail, Nuri Tasdemir, OldMcDonald, OliPro007, Onur, Optimiser, ozOli, P.J.Meisch, Paolo Forgia , Paweł Albecki, Petter Friberg, phant0m, piyush_baderia, ppeterka, Přemysl Šťastný, PSo, QoP, Radouane ROUFID, Raj, RamenChef, RAnders00, Rocherlee, Ronnie Wang, Ryan Hilbert, ryanyuyu, Sayakiss, SeeuD1, sevenforce, Shaan, ShivBuyya, Shoe, Sky, SmS, solidcell, Squidward, Stefan Isele - prefabware.com, stefanobaghino, Stephen C, Stephen Leppik,

		Steven Benitez, still_learning, Sudhir Singh, Swanand Pathak, Sнађошfa, TDG, TheLostMind, ThePhantomGamer, Tony BenBrahim, Unihedron, VGR, Vishal Biyani, Vogel612, vsminkov, vvtx, Wilson, winseybash, xwoker, yuku, Yury Fedorov, Zachary David Saunders, Zack Teater, Ze Rubeus, Ф Хосę Переúра ツ
183	Zeichenkodierung	Ilya
184	Zufallszahlengenerierung	Arthur, David Grant, David Soroko, dorukayhan, F. Stephen Q , Kichiin, MasterBlaster, michaelbahr, rokonoid, Stephen C, Thodgnir