Java Erweiterungen 2

|  |  |  |
| --- | --- | --- |
| OOA OOD OOP  OOA OOP OOD | | |
| OOD OOA | | OOP |
| OOD | OOP OOA | |
| OOP  OOP | OOA | OOD  OOA |
| OOD |

**Inhalt**

1. [Einleitung 8](#_bookmark0)
2. [Interfaces 12](#_bookmark1)
   1. [Grundlagen 13](#_bookmark2)
   2. [Beispiel: Yet another Comparator (YAC) 17](#_bookmark3)
   3. [Beispiel: Calculator 19](#_bookmark4)
   4. [Beispiel: DefaultWindowListener 21](#_bookmark5)
3. [Verschiedene Neuerung von Java 7 / 8 / 9 / 10 23](#_bookmark6)
   1. [Objects.requireNonNull 24](#_bookmark7)
   2. [Optional 26](#_bookmark8)
   3. [Optional-Beispiel: sum / product 29](#_bookmark9)
   4. [Initialisierung von Collections und Maps 31](#_bookmark10)
   5. [Type-Inference 34](#_bookmark11)
   6. [Wrapper 38](#_bookmark12)
   7. [newInstance 39](#_bookmark13)
4. [Von Top-Level-Klassen zu Lambdas 41](#_bookmark14)
   1. [Top-Level-Klassen 42](#_bookmark15)
   2. [Statische Member-Klassen 46](#_bookmark16)
   3. [Nicht statische Member-Klassen 48](#_bookmark17)
   4. [Local Classes 51](#_bookmark18)
   5. [Anonymous Classes 53](#_bookmark19)
   6. [Lambdas 55](#_bookmark20)
5. [Details zu Lambdas 58](#_bookmark21)
   1. [Target-Typing 59](#_bookmark22)
   2. [Der this- und der "Outer"-this-Zeiger 62](#_bookmark23)
   3. [Mit oder ohne "Outer".this? 66](#_bookmark24)
   4. [Lambda-Ausdrücke mit Return-Anweisungen 69](#_bookmark25)
   5. [Method-References 70](#_bookmark26)
   6. [Serialisierung von Lambdas 72](#_bookmark27)
6. [Closures 74](#_bookmark28)
   1. [Grundlagen 75](#_bookmark29)
   2. [Beispiel: Runnable 80](#_bookmark30)
   3. [Beispiel: Operators 83](#_bookmark31)
   4. [Das final-Problem 85](#_bookmark32)
   5. [andThen 87](#_bookmark33)
7. [Funktionale Programmierung 94](#_bookmark34)
   1. [Start 96](#_bookmark35)
   2. [forEach 99](#_bookmark36)
   3. [filter 101](#_bookmark37)

[7.4 map 102](#_bookmark38)

* 1. [reduce 103](#_bookmark39)
  2. [collect 105](#_bookmark40)
  3. [find 107](#_bookmark41)
  4. [match 108](#_bookmark42)
  5. [Combination von funktionalen Operationen 110](#_bookmark43)

1. [Streams 111](#_bookmark44)
   1. [Grundlagen 112](#_bookmark45)
   2. [Peek 115](#_bookmark46)
   3. [Eine SimpleStream-Implementierung 116](#_bookmark47)
2. [Generics - Erweiterungen 123](#_bookmark48)
   1. [Kovarianz und Kontravarianz 125](#_bookmark49)
   2. [Ein armer Dealer 130](#_bookmark50)
   3. [Ein reicher Dealer 132](#_bookmark51)
   4. [Eine statische deal-Methode 134](#_bookmark52)
   5. [Beispiel: andThen 135](#_bookmark53)
   6. [Die generische Nutzung der Klasse Class 136](#_bookmark54)
   7. [Vermeidung expliziter Casts 141](#_bookmark55)
3. [Enums - Erweiterungen 142](#_bookmark56)
   1. [Eine "alte" Implementierung 143](#_bookmark57)
   2. [Die "neue" Implementierung 146](#_bookmark58)
   3. [Die Basisklasse Enum 148](#_bookmark59)
   4. [EnumMap 150](#_bookmark60)
   5. [Implementierung von Interfaces 152](#_bookmark61)
   6. [Statische Elemente 155](#_bookmark62)
   7. [Serialisierung 159](#_bookmark63)
4. [Reflection - Erweiterungen 160](#_bookmark64)
   1. [Introspektion 161](#_bookmark65)
   2. [Set/Get, Invoke, NewInstance 167](#_bookmark66)
   3. [Zugriff auf private Member 172](#_bookmark67)
   4. [PropertyDescriptors 174](#_bookmark68)
5. [Reflection und Generics 178](#_bookmark69)
   1. [Parametrisierte Typen von Feldern und Methoden 180](#_bookmark70)
   2. [Generische Basisklassen 182](#_bookmark71)
   3. [Generische Interfaces 183](#_bookmark72)
   4. [Ein Beispiel: Iteration über Drinks 184](#_bookmark73)
   5. [Eine Utility-Klasse ActualTypeArguments 187](#_bookmark74)
   6. [Beispiel: Traversierung eines GUI-Baumes 190](#_bookmark75)
   7. [Beispiel: Restful 191](#_bookmark76)
6. [Dynamic Proxies 192](#_bookmark77)
   1. [Das Problem 194](#_bookmark78)
   2. [Ein einfaches Proxy 196](#_bookmark79)
   3. [Das Interface InvocationHandler 200](#_bookmark80)
   4. [Proxy.newProxyInstance 204](#_bookmark81)
   5. [Verfeinerungen 207](#_bookmark82)
   6. [Beispiel: Client-Server 209](#_bookmark83)
   7. [Das Konzept der AOPAlliance 215](#_bookmark84)
7. [Annotations 220](#_bookmark85)
   1. [Beispiel: OR-Mapping 221](#_bookmark86)
   2. [Mehr zu Annotationen 232](#_bookmark87)
   3. [Ein abschließendes Beispiel: Ein Dokumentations-Tool 239](#_bookmark88)
8. [Serialisierung - Erweiterungen 242](#_bookmark89)
   1. [Utilities 243](#_bookmark90)
   2. [Serialisierung von "Objektwolken" 248](#_bookmark91)
   3. [Vererbung 251](#_bookmark92)
   4. [Statische und transiente Attribute 254](#_bookmark93)
   5. [writeObject / readObject 256](#_bookmark94)
   6. [readResolve 258](#_bookmark95)
   7. [writeReplace 260](#_bookmark96)
   8. [Classes, Fields, Methods und Constructors 263](#_bookmark97)
9. [Multithreading - Erweiterungen 266](#_bookmark98)

|  |  |  |
| --- | --- | --- |
| [16.1](#_bookmark99) | [Uitlities](#_bookmark99) | [268](#_bookmark99) |
| [16.2](#_bookmark100) | [Synchronized](#_bookmark100) | [272](#_bookmark100) |
| [16.3](#_bookmark101) | [Synchronized – Lock Splitting](#_bookmark101) | [279](#_bookmark101) |
| [16.4](#_bookmark102) | [Synchronized – static Lock](#_bookmark102) | [283](#_bookmark102) |
| [16.5](#_bookmark103) | [Synchronized – Deadlock](#_bookmark103) | [285](#_bookmark103) |
| [16.6](#_bookmark104) | [Volatile](#_bookmark104) | [287](#_bookmark104) |
| [16.7](#_bookmark105) | [wait / notify](#_bookmark105) | [291](#_bookmark105) |
| [16.8](#_bookmark106) | [BlockingQueue](#_bookmark106) | [297](#_bookmark106) |
| [16.9](#_bookmark107) | [ThreadPool mit Runnables](#_bookmark107) | [301](#_bookmark107) |
| [16.10](#_bookmark108) | [Callables und Futures](#_bookmark108) | [304](#_bookmark108) |
| [16.11](#_bookmark109) | [ThreadPool mit Callables](#_bookmark109) | [308](#_bookmark109) |
| [16.12](#_bookmark110) | [ExceptionHandler](#_bookmark110) | [311](#_bookmark110) |
| [16.13](#_bookmark111) | [ThreadLocal](#_bookmark111) | [313](#_bookmark111) |
| [16.14](#_bookmark112) | [States](#_bookmark112) | [316](#_bookmark112) |
| [16.15](#_bookmark113) | [Enumerate](#_bookmark113) | [319](#_bookmark113) |
| [**17 Das**](#_bookmark114) | [**concurrent-Package**](#_bookmark114) | [**320**](#_bookmark114) |
| [17.1](#_bookmark115) | [Atomics](#_bookmark115) | [321](#_bookmark115) |
| [17.2](#_bookmark116) | [ConcurrentHashMap](#_bookmark116) | [323](#_bookmark116) |
| [17.3](#_bookmark117) | [CopyOnWriteArrayList](#_bookmark117) | [324](#_bookmark117) |
| [17.4](#_bookmark118) | [ReentrantLock](#_bookmark118) | [327](#_bookmark118) |
| [17.5](#_bookmark119) | [ReentrantReadWriteLock](#_bookmark119) | [330](#_bookmark119) |
| [17.6](#_bookmark120) | [Conditions](#_bookmark120) | [332](#_bookmark120) |

* 1. [ArrayBlockingQueue 336](#_bookmark121)
  2. [Executor mit Runnables 338](#_bookmark122)
  3. [Executor mit Callables 340](#_bookmark123)
  4. [ForkJoin 342](#_bookmark124)
  5. [CompletableFuture 345](#_bookmark125)
  6. [SimpleCompletableFuture 348](#_bookmark126)

#### [18 XML-Parser 352](#_bookmark127)

|  |  |  |
| --- | --- | --- |
| [18.1](#_bookmark128) | [SAX-Parser](#_bookmark128) | [355](#_bookmark128) |
| [18.2](#_bookmark129) | [DOM-Parser](#_bookmark129) | [358](#_bookmark129) |
| [18.3](#_bookmark130) | [JDOM-Parser](#_bookmark130) | [360](#_bookmark130) |
| [18.4](#_bookmark131) | [Pull-Parser](#_bookmark131) | [361](#_bookmark131) |
| [18.5](#_bookmark132) | [JAXB](#_bookmark132) | [365](#_bookmark132) |
| [**19 RMI**](#_bookmark133) | [**(Remote Method Invocation)**](#_bookmark133) | [**368**](#_bookmark133) |
| [19.1](#_bookmark134) | [Start](#_bookmark134) | [369](#_bookmark134) |
| [19.2](#_bookmark135) | [Factories](#_bookmark135) | [372](#_bookmark135) |
| [19.3](#_bookmark136) | [Listeners](#_bookmark136) | [376](#_bookmark136) |
| [19.4](#_bookmark137) | [Naming](#_bookmark137) | [380](#_bookmark137) |

#### [Class Loading 383](#_bookmark138)

* 1. [Grundlagen 384](#_bookmark139)
  2. [Eigene ClassLoader 391](#_bookmark140)
  3. [URLClassLoader 399](#_bookmark141)
  4. [Sichtbarkeit 402](#_bookmark142)
  5. [Plugins 406](#_bookmark143)
  6. [Hot Deployment 412](#_bookmark144)
  7. [ContextClassLoader 415](#_bookmark145)
  8. [Deserialisierung 418](#_bookmark146)
  9. [DynamicProxy 422](#_bookmark147)

#### [Literatur 425](#_bookmark148)

# Einleitung

Dieses Skript stellt "interessante" und wichtige fortgeschrittene Java-Themen vor.

Das Skript enthält ca. 450 Seiten - und der dem Skript zugrundeliegende Eclipse- Workspace umfasst etwa 150 Projekte. Das Material, was hier angeboten wird, kann natürlich unmöglich an fünf Seminartagen vollständig vorgestellt werden.

Das Skript und der Eclipse-Workspace sind vielmehr als Angebot gedacht – als Angebot, aus dem im Seminar eine Auswahl getroffen werden muss. Diese Auswahl wird sich an den Interessen der Teilnehmer / Teilnehmerinnen orientieren. Einige der Themen dieses Skripts werden dann ausführlich und in der Tiefe angesprochen werden

– viele andere Themen nur oberflächlich in Form eines groben Überblicks.

Themen, die aus Zeitgründen nur oberflächlich behandelt werden können, können dann aber Gegenstand eines späteren "Privat-Studiums" sein (während der Zugfahrt, im Wirtshaus, im Urlaub – wo auch immer…)

Dem Skript liegt der Gedanke zugrunde, dass es nicht darum gehen kann, einfach Klassen und deren Methoden "herunterzubeten". Der Focus liegt vielmehr auf generellen Konzepten und auf deren Bedeutung. Solche Konzepte versteht man am besten, wenn sie "entwickelt" werden. Dieser Gedanke der "Entwicklung" ist für das vorliegende Skript grundlegend.

Das Skript orientiert sich an einer Projekt-Sammlung, die in Form eines umfangreichen Eclipse-Workspaces vorliegt. Es enthält keine "Romane", sondern analysiert und erläutert den Quellcode dieser Projekte. Die Projekte sind natürlich unmittelbar übersetzbar und lauffähig und ermöglichen somit das Studium "am lebenden Objekt". Dieses Studium umfasst auch das Experimentieren, das "Spielen" mit dem Quellcode. Genau dieses Spielen mit dem Quellcode sollte u.a. der Gegenstand der Seminar- Übungen sein.

### "Truth can only be found in one place: the code." (Robert C. Martin)

Die Projekte des Eclipse-Workspaces sind genauso nummeriert und benamst wie die Kapitel und Abschnitte des Skripts – man kann also leicht von einem Projekt des Workspaces zum entsprechenden Kapitel und Abschnitt des Skripts wechseln und umgekehrt.

Für weitere Anregungen und / oder Hinweise auf Fehler ist der Autor natürlich dankbar (wenn's denn nicht einfache Rechschreibveler sind…)

### Übersicht

* Interfaces
* Verschiedene Neuerung von Java 7 / 8 / 9 / 10
* Von Top-Level-Klassen zu Lambdas
* Details zu Lambdas
* Closures
* Funktionale Programmierung
* Streams
* Generics
* Enums
* Reflection
* Reflection und Generics
* Dynamic Proxies
* Annotations
* Serialisierung
* Multithreading
* Das concurrent-Package
* XML-Parser
* RMI (Remote Method Invocation)
* Class Loading

### Diagramme

Wir benutzen den geläufigen UML-Klassendiagrammen eine eigene Form von Objektdiagrammen, welche sich von den UML-Objektdiagrammen unterscheiden.

Sei z.B. eine Klasse Foo gegeben:

public class **Foo** { private int **val**;

public void **alpha**() { ... }

public void **beta**() { ... }

}

Foo foo = new Foo();

Ein Objekt dieser Klasse stellen wir folgendermaßen dar:

foo

val

beta()

42

alpha()

: Foo

Anhand dieser Darstellung wird unmittelbar deutlich, welche Methoden auf dieses Objekt aufrufbar sind. Die Darstellung könnte allerdings den falschen(!) Eindruck vermitteln, die Methoden "klebten" an einem Objekt – tatsächlich sind es Methoden der Klasse (und existieren also nur ein einziges Mal), die auf alle möglichen Foo-Objekte aufgerufen werden können (den Methoden wird bekanntlich als this der Zeiger auf dasjenige Objekt übergeben, auf welches sie aufgerufen werden.

(Methoden sind in einem UML-Diagramm bekanntlich nicht vorgesehen.) Seien zwei weitere Typen gegeben - Bar und BarImpl:

public interface **Bar** { void **alpha**();

void **beta**();

}

public class **BarImpl** implements Bar { public void **alpha**(){ ... }

public void **beta**() { ... } public void **gamma** { ... }

}

Bar bar = new BarImpl()

bar ist eine Referenz vom Interface-Typ Bar – und sie zeigt auf ein Objekt vom Typ

BarImpl. Wir veranschaulichen diese Zusammenhänge wie folgt:

bar

*Bar*

: BarImpl

gamma()

beta()

alpha()

Die Darstellung verdeutlicht, dass wir via bar nur die im Interface Bar spezifizierten Methoden auf das BarImpl-Objekt aufrufen können – also alpha und beta. Die Methode gamma ist zwar public, aber über eine Bar-Referenz nicht aufrufbar. Der "Lolli" soll verdeutlichen, dass die Klasse des dargestellten Objekts das entsprechende Interface implementiert. Und die bar-Referenz zeigt nicht direkt auf das Objekt, sondern auf den Lolli.

(In UML sind solche Lollies bekanntlich in Objektdiagrammen unzulässig.)

Diese Diagramme haben sich aus Sicht des Autors gerade bei der Darstellung komplizierter Sachverhalte bewährt.

# Interfaces

### Voraussetzungen

* + Interfaces vor Java 8
  + Abstrakte Methoden
  + Statische finale Attribute
  + Implementierung von Interfaces

Seit Java 8 können in Interfaces nicht nur abstrakte Methoden und statische, öffentliche Konstanten definiert werden, sondern auch statische (öffentliche) Methoden und nicht- statische (öffentliche) Instanzmethoden. Letztere werden mit dem Schlüsselwort default vereinbart. Seit Java 10 sind auch private Methoden erlaubt. (protected Methoden sind allerdings nicht erlaubt!)

### Übersicht

* + Statische und nicht-statische Methoden (Technik)
  + Ein "Comparator"
  + Interface als Factory
  + Ein DefaultWindowListener

## Grundlagen

Im folgenden Foo-Interface werden eine statische Konstante, eine statische Methode, eine default-Methode, eine abstrakte Methode und eine private Methode definiert:

package **appl**;

public interface **Foo** {

public static final int **x** = 42; public static void **doThis**() {

System.out.println(x);

}

public default void **doThat**() { System.out.println("doThat"); this.doSomething();

}

public abstract void **doIt**(); private void **doSomething**() {

System.out.println("doSomething: " + x);

}

}

Eine Implementierungsklasse muss die abstrakte Methode doIt implementieren - sie könnte zudem auch die default-Methode doThat überschreiben:

package **appl**;

public class **FooImpl** implements Foo { @Override

public void **doIt**() {

System.out.println("doIt");

}

}

Die folgende Demo-Methode gibt Foo.x aus, ruft die statische Methode doThis auf, erzeugt ein FooImpl-Objekt und ruft auf dieses die doThat- und die doIt-Methoden auf:

static void **demoFoo**() { System.out.println(Foo.x); Foo.doThis();

final Foo foo = new FooImpl(); foo.doThat();

foo.doIt();

}

Die Ausgaben:

42

42

doThat doSomething: 42 doIt

In einer default-Methode eines Interfaces können auch deren abstrakte Methoden aufgerufen werden. Im folgenden Beispiel ruft die default-Methode doThisAndThat die abstrakten Methoden doThis und doThat auf:

package **appl**;

public interface **Bar** {

public abstract void **doThis**(); public abstract void **doThat**();

public default void **doThisAndThat**() { this.doThis();

this.doThat();

}

}

Eine Implementierungsklasse muss doThis und doThat implementieren – und kann die

default-Methode doThisAndThat überschreiben:

package **appl**;

public class **BarImpl** implements Bar { @Override

public void **doThis**() {

System.out.println("doThis");

}

@Override

public void **doThat**() { System.out.println("doThat");

}

@Override

public void **doThisAndThat**() { System.out.println("before this and that"); Bar.super.doThisAndThat(); System.out.println("after this and that");

}

}

In der folgenden Demo-Methode wird ein BarImpl-Objekt erzeugt und auf dieses die

doThisAndThat-Methode auf:

static void **demoBar**() {

final Bar bar = new BarImpl(); bar.doThisAndThat();

}

before this and that doThis

doThat

after this and that

Angenommen, sowohl das Interface Alpha als auch das Interface Beta enthalten eine default-Methode mit derselben Signatur:

package **appl**;

public interface **Alpha** {

public default void **doThis**() { System.out.println("\tAlpha.doThis");

}

}

package **appl**;

public interface **Beta** {

public default void **doThis**() { System.out.println("\tBeta.doThis");

}

}

Eine Klasse, die beide Interfaces implementieren will, muss(!) doThis überschreiben. In der überschreibenden Methode kann dann via Alpha.super resp. Beta.super die doThis-Methode von Alpha resp. die doThis-Methode von Beta aufgerufen werden:

package **appl**;

public class **AlphaBeta** implements Alpha, Beta { @Override

public void **doThis**() {

System.out.println(">> AlphaBeta.doThis"); Alpha.super.doThis(); Beta.super.doThis(); System.out.println("<< AlphaBeta.doThis");

}

}

In der folgenden Demo-Methode werden drei AlphaBeta-Objekte erzeugt. Das erste Objekt wird an eine Alpha-Variablen gebunden, das zweite an eine Beta-Variable und das dritte an eine AlphaBeta-Variable. Dann wird über jede dieser Variablen die

doThis-Methode aufgerufen. In allen drei Fällen wird die doThis-Methode von

AlphaBeta aufgerufen:

static void **demoConflict**() {

final Alpha alpha = new AlphaBeta(); alpha.doThis();

final Beta beta = new AlphaBeta(); beta.doThis();

final AlphaBeta alphaBeta = new AlphaBeta(); alphaBeta.doThis();

}

>> AlphaBeta.doThis Alpha.doThis Beta.doThis

<< AlphaBeta.doThis

>> AlphaBeta.doThis Alpha.doThis Beta.doThis

<< AlphaBeta.doThis

>> AlphaBeta.doThis Alpha.doThis Beta.doThis

<< AlphaBeta.doThis

## Beispiel: Yet another Comparator (YAC)

Das folgende Typ-parametrisierte Interface YAC spezifiziert zwei abstrakte Vergleichsmethoden: eq und gt (equals und greater-than) – und vier weitere default- Methoden (ge, lt, le, ne), die allesamt auf die abstrakten Methoden zurückgeführt werden:

package **appl**;

import java.util.Objects;

public interface **YAC**<T> { // Yet another Comparator... public abstract boolean **eq**(T v0, T v1);

public abstract boolean **gt**(T v0, T v1);

public default boolean **ge**(T v0, T v1) { checkParameters(v0, v1);

return this.gt(v0, v1) || this.eq(v0, v1);

}

public default boolean **lt**(T v0, T v1) { checkParameters(v0, v1);

return ! this.ge(v0, v1);

}

public default boolean **le**(T v0, T v1) { checkParameters(v0, v1);

return this.eq(v0, v1) || this.lt(v0, v1);

}

public default boolean **ne**(T v0, T v1) { checkParameters(v0, v1);

return ! this.eq(v0, v1);

}

public static <T> void **checkParameters**(T v0, T v1) { Objects.requireNonNull(v0); Objects.requireNonNull(v1);

}

}

Eine instanziierbare Klasse, die dieses Interface implementiert, muss nurmehr die Methode eq und gt implementieren:

package **appl**;

public class **IntegerYAC** implements YAC<Integer> { public boolean **eq**(Integer v0, Integer v1) {

YAC.checkParameters(v0, v1);

return v0.intValue() == v1.intValue();

}

public boolean **gt**(Integer v0, Integer v1) { YAC.checkParameters(v0, v1);

return v0.intValue() > v1.intValue();

}

}

Hier eine Demo-Methode:

static void **demoYAC**() {

final YAC<Integer> yac = new IntegerYAC(); System.out.println(yac.eq(10, 10));

System.out.println(yac.ne(10, 20));

System.out.println(yac.ge(20, 20));

System.out.println(yac.ge(20, 10));

System.out.println(yac.gt(20, 10));

System.out.println(yac.le(20, 20));

System.out.println(yac.le(10, 20));

System.out.println(yac.lt(10, 20));

}

## Beispiel: Calculator

Statische Methoden eines Interfaces können als Factory-Methoden fungieren. Sei folgendes Interface gegeben:

package **iface**;

import impl.CalculatorImpl; public interface **Calculator** {

public abstract void **add**(int value); public abstract void **subtract**(int value); public abstract int **getValue**();

public static Calculator **create**() { return new CalculatorImpl();

}

}

Und folgende Implementierung:

package **impl**;

import iface.Calculator;

public class **CalculatorImpl** implements Calculator { private int **value**;

public void **add**(int value) { this.value += value;

}

public void **subtract**(int value) { this.value -= value;

}

public int **getValue**() { return this.value;

}

}

Man beachte, dass das Interface im iface-Paket, die Implementierungsklasse im Paket

impl definiert ist.

Dann können wir eine Anwendung schreiben, die (unmittelbar) nur vom Interface abhängig ist (nur vom iface-Paket):

static void **demoCalculator**() {

Calculator c1 = Calculator.create(); c1.add(50);

c1.subtract(8);

Calculator c2 = Calculator.create();

c2.add(80); c2.subtract(3);

System.out.println(c1.getValue()); System.out.println(c2.getValue());

}

Die Ausgaben:

42

77

(Die Bedeutung von 42 ist allgemein bekannt. Was hat es mit der 77 auf sich? – Denken Sie an COBOL…)

## Beispiel: DefaultWindowListener

Das AWT-Interface WindowListener hat sieben(!) Methoden. Typischerweise interessiert man sich aber nur für die Methode windowClosing (diese wird von einem JFrame dann aufgerufen, wenn das Fester über das Systemmenu oder über den X- Knopf geschlossen werden soll (und somit die Anwendung terminiert werden soll).

Wollte man das Interface implementieren, müssten natürlich auch alle anderen Methoden (leer) implementiert werden.

Das AWT definiert eine Klasse WindowAdapter, welches alle Methoden bereits leer implementiert. Man könnte eine Klasse schreiben, die von eben von WindowAdapter abgeleitet ist – und diese Klasse müsste dann nurmehr die windowClosing-Methode überschreiben.

Dann aber könnte ein Fenster, dass ja bereits von JFrame abgeleitet ist, nicht zusätzlich auch von dieser Listener-Klasse abgeleitet werden.

Das Problem hätte vermieden werden können, wenn Interfaces immer schon default- Methoden hätten implementieren können. Dann nämlich hätten alle Methoden des WindowListener-Interface eben als leere default-Methoden implementiert werden können.

Im nachhinein können wir das Problem aber dadurch lösen, dass wir von WindowListener ein Interface DefaultWindowListener ableiten – und eben in diesem Interface alle von WindowListener geerbten Methode als leere default-Methoden implementieren:

package **util**;

import java.awt.event.WindowEvent; import java.awt.event.WindowListener;

public interface **DefaultWindowListener** extends WindowListener { @Override

public default void **windowOpened**(WindowEvent e) {

}

@Override

public default void **windowClosing**(WindowEvent e) {

}

@Override

public default void **windowClosed**(WindowEvent e) {

}

@Override

public default void **windowIconified**(WindowEvent e) {

}

@Override

public default void **windowDeiconified**(WindowEvent e) {

}

@Override

public default void **windowActivated**(WindowEvent e) {

}

@Override

public default void **windowDeactivated**(WindowEvent e) {

}

}

Dann können wir einen MyFrame schreiben, in welcher der JFrame selbst als WindowListener fungiert. Wir leiten MyFrame von JFrame ab – und zusätzlich von dem Interface(!) DefaultWindowListener. Wobei wir in der MyFrame-Klasse dann die windowClosing-Methode überschreiben:

package **appl**;

import java.awt.event.WindowEvent; import javax.swing.JFrame;

import util.DefaultWindowListener;

public class **MyFrame** extends JFrame implements DefaultWindowListener { public **MyFrame**() {

this.addWindowListener(this);

this.setBounds(100, 100, 500, 100); this.setVisible(true);

}

@Override

public void **windowClosing**(WindowEvent e) { System.out.println("do some terminating work..."); this.dispose();

}

}

# Verschiedene Neuerung von Java 7 / 8 / 9 / 10

Dieses Kapitel enthält einige wesentliche Neuerungen in den Java-Versions von Java 8 an aufwärts. Diese Neuerungen werden in den weiteren Kapiteln häufiger genutzt werden.

### Übersicht

* + Objects.requireNonNull
  + Das Optional-Konzept
  + Initialisierung von Collections.
  + Type-Inference und var-Variablen
  + Parametriesierte Konstruktoren von Wrapper-Klassen deprecated
  + Class.newInstance: deprecated.

## Objects.requireNonNull

Der folgenden Methode isPalindrom muss ein gültiger String übergeben werden. Diese Precondition kann zu Beginn der Methode mittels einer if-Anweisung geprüft werden, deren then-Zweig eine NullPointerException wirft:

static boolean **isPalindrom**(String s) { if (s == null)

throw new NullPointerException(); int left = 0;

int right = s.length() - 1; while(left < right) {

if (s.charAt(left) != s.charAt(right)) return false;

left++; right--;

}

return true;

}

Der if-Anweisung sieht man auf den ersten Blick nicht an, dass es sich um die Prüfung einer Precondition handelt. Dies sollte aber explizit deutlich werden. Hier kann die Objects-Methode requireNonNull verwendet werden:

static boolean **isPalindrom**(String s) { Objects.requireNonNull(s);

// ...

}

Ist s eine null-Referenz, wird requireNonNull eine NullPointerException werden. Ansonsten kehrt requireNonNull lautlos zurück.

Einer überladenen requireNonNull-Methode kann noch ein Text übergeben werden:

Objects.requireNonNull(s, "s mustn't be null");

Im folgenden Beispiel prüft der Konstruktor sein Argument auf null und wirft ggf. eine Exception (denn die print-Methode der Klasse verlangt eine gültige Referenz):

class **Foo** {

private String **s**; public **Foo**(String s) {

if (s == null)

throw new NullPointerException(); this.s = s;

}

public void **print**() { System.out.println(s.toUpperCase());

}

}

Die folgende Variante ist besser:

class **Foo** {

private String **s**; public **Foo**(String s) {

this.s = Objects.requireNonNull(s);

}

public void **print**() { System.out.println(s.toUpperCase());

}

}

Im Fall, dass der an requireNonNull übergebene Zeiger gültig ist, kehrt die Methode zurück und liefert dabei genau diejenige Referenz zurück, die ihr übergeben wurde (und die im obigen Beispiel dann dazu verwendet werden, die Instanzvariable this.s direkt an Ort und Stelle zu initialisieren.

Die Klasse Objects enthält noch eine weitere Methode requireNonNullElse. Diese kann wie folgt verwendet werden:

class **Bar** {

private String **s**; public **Bar**(String s) {

this.s = Objects.requireNonNullElse(s, "");

}

public void **print**() { System.out.println(s.toUpperCase());

}

}

Diese Methode wirft keine(!) Exception. Sie kehrt stets zurück – entweder mit der an sie übergebenen Referenz (sofern diese null ist) oder mit dem an die Methode übergebenen Default-Wert.

## Optional

Ein Optional-Objekt kann eine Referenz auf ein anderes Objekt enthalten – oder auch nicht. Überall dort, wo traditionell eine Referenz genutzt wird, die eventuell null sein kann, kann ein Optional genutzt werden.

Die konsequente Verwendung von Optional-Objekten kann NullPointerExceptions nahezu ausschließen.

empty ist eine statische Factory-Methode, welches ein Optional ohne "Inhalt" erzeugt. Mittels der Instanzmethode get kann der Inhalt ermittelt werden – ist keiner vorhanden, wird eine NoSuchElementException geworfen:

static void **demoEmtpyGet**() { Optional<Integer> o = Optional.empty(); System.out.println(o);

try { o.get();

}

catch (NoSuchElementException e) { System.out.println("Expected: " + e.getMessage());

}

}

Die Ausgaben:

Optional.empty

Expected: No value present

Mittels der statischen Factory-Methode of kann ein Optional mit Inhalt erzeugt werden. Die get-Methode liefert dann diesen Inhalt zurück:

static void **demoOfGet**() { Optional<Integer> o = Optional.of(42); System.out.println(o);

Integer v = o.get(); System.out.println(v);

}

Die Ausgaben:

Optional[42] 42

Der Versuch, an of eine null-Referenz zu übergeben (die of-Methode also "auszutricksen"), wird mit einer NullPointerException beantwortet:

static void **demoOfException**() { Integer i = null;

try {

Optional<Integer> o = Optional.of(i);

}

catch (NullPointerException e) { System.out.println("Expected");

}

}

Mittels isPresent kann ein Optional gefragt werden, ob es einen Inhalt hat:

static void **demoIsPresent1**() { Optional<Integer> o = Optional.empty(); if (o.isPresent())

System.out.println(o.get()); else

System.out.println("not present");

}

Die Ausgaben:

not present

static void **demoIsPresent2**() { Optional<Integer> o = Optional.of(42); if (o.isPresent())

System.out.println(o.get()); else

System.out.println("not present");

}

Die Ausgaben:

42

orElse liefet den Inhalt eines Optional zurück – oder, falls kein Inhalt vorhanden, einen Default-Wert, welcher der Methode als Parameter übergeben wird:

static void **demoOrElse1**() { Optional<Integer> o = Optional.of(42); Integer v = o.orElse(77); System.out.println(v);

}

Die Ausgaben:

42

static void **demoOrElse2**() { Optional<Integer> o = Optional.empty();

Integer v = o.orElse(77); System.out.println(v);

}

Die Ausgaben:

77

ofNullable ist eine Factory-Methode, der entweder eine null oder eine gültige Referenz übergeben werden kann.

static void **demoNullable1**() { Integer i = null;

Optional<Integer> o = Optional.ofNullable(i); Integer v = o.orElse(77); System.out.println(v);

}

Die Ausgaben:

77

static void **demoNullable2**() { Integer i = 42;

Optional<Integer> o = Optional.ofNullable(i); Integer v = o.orElse(77); System.out.println(v);

}

Die Ausgaben:

42

Neben der generischen Optional<T>-Klasse existieren die Klassen OptionalInt, OptionalLong und OptionalDouble.

## Optional-Beispiel: sum / product

Sei folgende sum-Methode gegeben:

static int **sum**(int... values) { int sum = 0;

for (int value : values) sum += value;

return sum;

}

Die Anwendung zeigt, dass diese Methode völlig korrekt funktioniert (u.a. ist die Summe der Elemente eines leeren Arrays 0):

static void **demoSum**() { System.out.println(sum(1, 2, 3, 4)); System.out.println(sum());

}

Die Ausgaben:

10

0

Betrachten wir nun aber die folgende product1-Methode (die nach demselben Muster gestrickt ist wie die sum-Methode – nur dass hier als "Startwert" 1 verwendet wird):

static int **product1**(int... values) { int product = 1;

for (int value : values) product \*= value;

return product;

}

Eine Anwendung:

static void **demoProduct1**() { System.out.println(product1(1, 2, 3, 4)); System.out.println(product1());

}

Die zweite der Ausgaben ist sehr merkwürdig: Das Produkt aller Elemente eines leeren Arrays ist 1 ???):

24

1

Die zweite Variante (product2) ist besser:

static Optional<Integer> **product2**(int... values) { if (values.length == 0)

return Optional.empty(); int product = 1;

for (int value : values) product \*= value;

return Optional.of(product);

}

static void **demoProduct2**() { System.out.println(product2(1, 2, 3, 4)); System.out.println(product2());

}

Der erste Aufruf liefert ein gefülltes, der zweite Aufruf ein leeres Optional:

Optional[24] Optional.empty

Die dritte Variante (product3) benutzt anstelle der generischen Optional-Klasse die Klasse OptionalInt:

static OptionalInt **product3**(int... values) { if (values.length == 0)

return OptionalInt.empty(); int product = 1;

for (int value : values) product \*= value;

return OptionalInt.of(product);

}

static void **demoProduct3**() { System.out.println(product3(1, 2, 3, 4)); System.out.println(product3());

}

}

Die Ausgaben:

OptionalInt[24] OptionalInt.empty

## Initialisierung von Collections und Maps

Die Collection-Interfaces sind erweitert worden um Factory-Methoden, mittels derer immutable Collections erzeugt werden können.

Das List-Interface enthält nun 12 überladene of-Methoden:

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 e10) static <E> List<E> **of**(E ... elements)

Die jeweils zurückgelieferte List ist immutable; sie ist Serializable:

static void **demoListOf**() {

final List<Integer> list = List.of(1, 2, 3); list.forEach(System.out::println); System.out.println(list.getClass()); System.out.println(list instanceof Serializable); try {

list.add(4);

}

catch (final Exception e) { System.out.println(e);

}

}

Die Ausgaben:

1

2

3

class java.util.ImmutableCollections$ListN true java.lang.UnsupportedOperationException

Der Versuch, ein neues Element zur Liste hinzuzufügen, wird mit einer Exception quittiert.

Das Interface Set ist auf ähnliche Weise erweitert worden:

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 e10) static <E> Set<E> **of**(E... elements)

Eine beispielhafte Anwendung:

static void **demoSetOf**() {

final Set<String> set = Set.of("red", "green", "blue"); set.forEach(System.out::println); System.out.println(set.getClass()); System.out.println(set instanceof Serializable);

try { set.remove("red");

}

catch (final Exception e) { System.out.println(e);

}

}

Auch das von der of-Methode des Set-Interfaces zurückgelieferte Set ist immutable. Die Ausgaben:

green blue red

class java.util.ImmutableCollections$SetN true java.lang.UnsupportedOperationException

Das Interface Map ist wie folgt erweitert worden:

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 k10, V v10)

Eine Demo-Anwendung:

static void **demoMapOf**() {

final Map<Integer, String> map =

Map.of(42, "red", 43, "green", 44, "blue"); map.forEach((k, v) -> System.out.println(k + " => " + v)); System.out.println(map.getClass());

System.out.println(map instanceof Serializable); try {

map.put(45, "yellow");

}

catch (final Exception e) { System.out.println(e);

}

}

Die Ausgaben:

44 => blue

43 => green

42 => red

class java.util.ImmutableCollections$MapN true java.lang.UnsupportedOperationException

Map enthält zusätzlich eine Varargs-basierte ofEntries-Methode:

static <K,V> Map<K,V> **ofEntries**(

Map.Entry<? extends K,? extends V>... entries)

Eine Anwendung:

static void **demoMapOfEntries**() {

final Map<Integer, String> map = Map.ofEntries( entry(77, "RED"),

entry(78, "GREEN"),

entry(79, "BLUE"));

map.forEach((k, v) -> System.out.println(k + " => " + v)); System.out.println(map.getClass());

System.out.println(map instanceof Serializable); try {

map.put(45, "yellow");

}

catch (final Exception e) { System.out.println(e);

}

}

Die obige Anwendung setzt folgenden statischen Import voraus:

import static java.util.Map.entry;

Die Ausgaben:

77 => RED

79 => BLUE

78 => GREEN

class java.util.ImmutableCollections$MapN true java.lang.UnsupportedOperationException

## Type-Inference

Der Typ einer lokalen Variablen muss nicht mehr explizit definiert werden, sofern der Compiler diesen Typ aufgrund der Initialisierung der Variablen schlussfolgern kann. Das funktioniert allerdings nur dann, wenn die Variable unmittelbar bei ihrer Definition auch initialisiert wird.

Eine Demo-Anwendung:

static void **demoVar**() {

//

//

var i = 42; i = 77;

i = 42L;

// illegal

i = "Hello"; // illegal int var = 77;

System.out.println(var);

//

//

//

illegal: class var {

}

//

var j; // illegal

}

Wir können nun eine lokale Variable mit dem "Schlüsselwort" var definieren:

var i = 42;

Auf den ersten Blick sieht es so aus, als hätten wir eine "typlose" Variable definiert. Aber Java ist (und bleibt) eine streng typisierte Sprache. Also muss auch i einen bestimmten Typ besitzen.

Da i mit dem Wert 42 initialisiert wird und 42 ein int-Literal ist, definiert der Compiler i

automatisch als int – und i wird auch in der Folge immer diesen Typ besitzen. An i kann natürlich ein neuer int zugewiesen werden:

i = 77;

Aber folgende Zuweisungen sind illegal:

i = 42L;

// illegal

i = "Hello"; // illegal

Da der Compiler bei einer var-Definition den Typ der Variablen bestimmen können muss, muss eine solche Variablen-Definition eine unmittelbare Initialisierung einhalten –

der Typ wird ja gerade aus dem Initialisierungs-Ausdruck berechnet. Folgende Definition also ist illegal:

var i; // illegal

Das Wort var ist KEIN Schlüsselwort. Wir können weiterhin var z.B. als Bezeichner für Variablen verwenden:

int var = 77;

Oder, um den Leser so richtig zu verwirren:

var var = 77;

Nur dann, wenn var anstelle eines Typs verwendet wird, besitzt dieses Wort den Status eines "Schlüsselworts" – es handelt sich also um ein "Kontext-Schlüsselwort".

Folgende Definition ist also unzulässig:

//

illegal: class **var**

}

Wir können mittels var auch finale Variablen definieren:

static void **demoVarAndFinal**() { final var i = 42;

// i = 77; // illegal

}

(Es wäre schön, wenn wir final var mit val abkürzen könnten – leider ist diese Abkürzung aber nicht vorgesehen.)

Eine Anwendung mit Listen:

static void **demoListsAndLoops**() { var list = List.of(10, 20, 30); List<Integer> l = list;

for (var value : list) { Integer v = value; System.out.println(v);

}

for (var i = 0; i < list.size(); i++) { Integer v = list.get(i); System.out.println(v);

}

// List<Double> doubleList = list; // illegal

}

Wir definieren und initialisieren eine List-Variable. Und zeigen, dass diese Variable vom Typ List<Integer> ist.

Mittels var können wir nun auch die Variablen der alten for-Schleife und der "for-each"- Schleife definieren.

Die Variable i ist implizit vom Typ int; die Variable value ist implizit vom Typ Integer.

Variablen, die einen expliziten Typ haben, können auch dann als final definiert werden, wenn sie erst in beiden Zweigen einer if-else-Anweisung initialisiert werden:

static void **demoImmediateAssign**() { final int foo;

if ("1".equals("1")) foo = 42;

else

foo = 77;

// illegal:

// final var bar;

// if ("1".equals("1"))

// bar = 42;

// else

// bar = 77;

}

Eine solche ("aufgeschobene") Initialisierung ist bei var-Variablen nicht(!) erlaubt (diese müssen also unmittelbar bei ihrer Definition initialisiert werden):

Wird das Resultat einer Instanziierung einer anonymen Klasse an eine var-Variable gebunden, so ist der Typ dieser Variablen exakt gleich dem Typ der anonymen Klasse:

static void **demoAnonymousClass**() { var runnable = new Runnable() {

@Override

public void run() {

}

};

// illegal:

// runnable = new Runnable() {

// @Override

// public void run() {

// }

// };

}

runnable hat nun exakt den Typ der anonymen Klasse (etwa: …$0). An dieselbe Variable kann also kein(!) zweites Runnable zugewiesen werden. Denn der Ausdruck auf der rechten Seite hat nun einen anderen(!) Typ (etwa ...$1) – und ist daher nicht kompatibel zum Typ von runnable.

Der Umstand, dass eine var-Variable, die mit einer Instanz einer anonymen Klasse initialisiert wird, eben vom exakten Typ dieser anonymen Klasse ist, kann z.B. wie folgt genutzt werden:

static void **demoAnonymousClassMethods**() { var foo = new **Object**() {

public void **alpha**() { System.out.println("alpha");

}

public void **beta**() { System.out.println("beta");

}

public int **bar** = 77;

};

foo.alpha();

foo.beta(); System.out.println(foo.bar);

}

Natürlich kann diese "Object-Definition" nur im Kontext der umschließenden Methode vernünftig genutzt werden…

Nur lokale Variablen können mit var definiert werden. Attribute einer Klasse müssen weiterhin mit einem expliziten Typ ausgestattet werden. Der folgende Code ist illegal:

static void **demoAttribute**() { class **Foo** {

// var **i** = 42; // not allowed

}

}

Und ebenso wenig können natürlich auch Methoden-Parameter oder Return-Typen mir

var definiert werden…

## Wrapper

Die direkte Erzeugung von Wrapper-Objekten mittels des parametrisierten Konstruktors ist mittlerweile deprecated. Integer-Objekte z.B. sollten nicht via new Integer(n), sondern mittels Integer.valueOf(n) nachgefragt werden.

Die Klasse Integer hat einen statischen Cache, in welchem bereits alle Integer- Objekte mit den Werten von -128 bis +127 gespeichert sind. new Integer(n) erzeugt aber immer(!) ein neues Objekt. Auch die Klasse Character hat einen entsprechenden Cache. Mittels der valueOf-Methoden wird nun ggf. eines der Objekte aus dem jeweiligen Cache geliefert. Und auch Autoboxing benutzt die valueOf-Methoden:

static void **demoIntegerCache**() {

System.out.println(Integer.valueOf(-129) == Integer.valueOf(-129)); System.out.println(Integer.valueOf(-128) == Integer.valueOf(-128)); System.out.println(Integer.valueOf(0) == Integer.valueOf(0)); System.out.println(Integer.valueOf(127) == Integer.valueOf(127)); System.out.println(Integer.valueOf(128) == Integer.valueOf(128));

}

Die Ausgaben:

false true true true false

Deshalb ist new Integer(n), new Character(n) etc. seit Java 9 deprecated:

@SuppressWarnings("deprecation") static void **demoDeprecated**() {

new Integer(42); new Character('A');

}

## newInstance

Die newInstance-Methode der Klasse Class ist seit Java 9 deprecated.

Bekanntlich wurde diese Methode bislang benutzt, um ein Objekt derjenigen Klasse zu erzeugen, auf deren Class-Objekt sie aufgerufen wurde. Bei der Erzeugung des Objekts wurde der parameterlose Konstruktor der Klasse genutzt.

Statt der newInstance-Methode der Klasse Class können wir die newInstance- Methode der Klasse Constructor nutzen.

Warum ist Class.newInstance deprecated?

Sei eine Klasse gegeben, deren parameterloser Konstruktor eine IOException wirft (warum auch immer…):

package **appl**;

import java.io.IOException; public class **Foo** {

public **Foo**() throws IOException { throw new IOException();

}

}

Sei weiterhin eine kleine Hilfsmethode gegeben, welche zu einem gegebenen Klassen- Namen das Class-Objekt liefert:

static Class<?> **getClass**(String className) { try {

return Class.forName(className);

}

catch(Exception e) {

throw new RuntimeException(e);

}

}

Wir ermitteln das Class-Objekt der Klasse appl.Foo und rufen auf dieses Class-Objekt newInstance auf:

@SuppressWarnings("deprecation") static void **demoClassNewInstance**() {

try {

final Class<?> cls = getClass("appl.Foo"); final Object obj = cls.newInstance(); System.out.println(obj);

}

//

//

//

catch (InstantiationException | IllegalAccessException e) { System.out.println(e);

}

catch (IOException e) { System.out.println(e);

}

}

Das Resultat ist etwas merkwürdig. newInstance kann "offiziell" (gemäß der Spezifikation) nur eine InstantiationException oder eine IllegalAccessException werfen. Nur diese beiden checked Exceptions können also auch gefangen werden – wir können die vom Foo-Konstruktor geworfene IOException überhaupt nicht als solche auffangen (der Compiler erlaubt dies nicht!). Wir könnten im letzten catch zwar eine Exception fangen, aber keine IOException.

Wir benutzen nun die Constructor-eigene newInstance-Methode. Wir ermitteln dasjenige Constructor-Objekt, welches den parameterlosen Konstruktor der Foo- Klasse repräsentiert und rufen auf dieses dann newInstance auf:

static void **demoConstructorNewInstance**() { try {

final Class<?> cls = getClass("appl.Foo");

final Constructor<?> ctor = cls.getConstructor(); final Object obj = ctor.newInstance(); System.out.println(obj);

}

catch (InstantiationException | IllegalAccessException | IllegalArgumentException | NoSuchMethodException | SecurityException e) {

System.out.println(e);

}

catch (InvocationTargetException e) { System.out.println(e.getTargetException());

}

}

Die vom Foo-Konstruktor geworfene IOException wird nun in eine Exception des Typs InvocationTargetException eingewickelt – und wir können diese Exception fangen. Und auf die geworfene InvocationTargetException deren Methode getTargetException aufrufen – welche uns dann die vom Foo-Konstruktor geworfene IOException zurückliefert.

By the way: Genauso funktioniert auch die invoke-Methode der Klasse Method…

# Von Top-Level-Klassen zu Lambdas

Lambdas können in erster Näherung aufgefasst werden als "Vereinfachungen" von anonymen Klassen.

Um Lambdas zu verstehen, ist es daher sinnvoll, alle bisherigen Typen von Klassen noch einmal zu rekapitulieren.

Zur Demonstration verwenden wir ein kleines GUI-Programm, welches auf Swing basiert:



(Der Kalkulator ist nicht schön, aber er funktioniert.)

### Übersicht

* + Top-Level-Klassen
  + Statische Member-Klassen
  + Nicht-statische Member-Klassen (Outer.this)
  + Lokale Klassen
  + Anonyme Klassen
  + Lambda-Ausdrücke

## Top-Level-Klassen

Die Anwendung soll zunächst anhand eines kleinen Objekt-Diagramms erläutert werden:



: MathFrame

: ButtonPlusAdapter

: ButtonMinusAdapter

mathFrame

mathFrame

*ActionListener*

*ActionListener*

: JButton

"Plus"

: JButton

"Minus"

dActionListener

ad

dActionListener

ad

ormed

actionPerf

OR

CT

ormed

actionPerf

OR

CT

Minus

on

Plus

on

:JTextField

:JTextField

:JTextField

Eine kurze Erläuterung der Anwendung:

Ein MathFrame (die Klasse ist abgeleitet von JFrame) besitzt Referenzen auf drei

JTextField- und auf zwei JButton-Objekte.

Ein JButton hat eine Registratur, in welcher mittels des Aufrufs von addActionListener Objekte registriert werden können, deren Klassen das Interface ActionListener implementieren. Dieses Interface spezifiziert genau eine Methode: actionPerformed. Wird ein Button angeklickt, ruft dieser die actionPerformed- Methode auf alle bei ihm registrierten ActionListener auf.

Die folgende Anwendung definiert zwei globale Klassen, welche das ActionListener- Interface implementieren: ButtonPlusAdapter und ButtonMinusAdapter. Jede dieser beiden Klassen wird genau einmal instanziiert – und die so erzeugten Adapter-Objekte bei den beiden JButtons registriert. Bei der Erzeugung der Adapter wird der

Konstruktor der entsprechenden Klasse aufgerufen (im Bild als CTOR bezeichnet). Diesem wird die Referenz auf den MathFrame als Parameter übergeben – welche dann in der Instanzvariablen mathFrame gespeichert wird.

Die beiden Adapter-Objekte werden natürlich – wie auch die JButtons und JTextFields – vom MathFrame erzeugt (dieser übergibt this bei der Erzeugung der beiden Adapter).

Der MathFrame besitzt zwei öffentliche Methoden: onPlus und onMinus. Die actionPerformed-Methode der ButtonPlusAdapter-Klasse kann dann über die mathFrame-Referenz die onPlus-Methode aufrufen, die actionPerformed-Methode der Klasse ButtonMinusAdapter kann onMinus aufrufen.

Wird als nun z.B. der "Plus"-Button betätigt, so wird dieser die actionPerformed- Methode auf den ButtonPlusAdapter aufrufen – und diese wird nichts weiter tun, als die Verarbeitung an die onPlus-Methode des MathFrames zu delegieren. Letztere wird dann die Werte der beiden Eingabefelder ermitteln, die entsprechende Berechnung ausführen und das Ergebnis im Ausgabefeld abstellen.

Hier der komplette Quellcode:

public class **Application** {

public static void **main**(String[] args) { new MathFrame();

}

}

// ...

public class **MathFrame** extends JFrame {

private final JTextField **textFieldX** = new JTextField(10); private final JTextField **textFieldY** = new JTextField(10); private final JButton **buttonPlus** = new JButton("Plus"); private final JButton **buttonMinus** = new JButton("Minus"); private final JTextField textF**i**eldResult = new JTextField(10);

public **MathFrame**() { this.setLayout(new FlowLayout()); this.add(this.textFieldX); this.add(this.textFieldY); this.add(this.buttonPlus); this.add(this.buttonMinus); this.add(this.textFieldResult); this.registerListeners();

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE); this.pack();

this.setVisible(true);

}

private void **registerListeners**() {

this.buttonPlus.addActionListener(new ButtonPlusAdapter(this)); this.buttonMinus.addActionListener(new ButtonMinusAdapter(this));

}

public void **onPlus**() { try {

int x = Integer.parseInt(this.textFieldX.getText()); int y = Integer.parseInt(this.textFieldY.getText()); int result = x + y; this.textFieldResult.setText(String.valueOf(result));

}

catch(NumberFormatException e) { this.textFieldResult.setText("Illegal input");

}

}

public void **onMinus**() { try {

int x = Integer.parseInt(this.textFieldX.getText()); int y = Integer.parseInt(this.textFieldY.getText()); int result = x - y; this.textFieldResult.setText(String.valueOf(result));

}

catch(NumberFormatException e) { this.textFieldResult.setText("Illegal input");

}

}

}

// ...

public class **ButtonPlusAdapter** implements ActionListener { final MathFrame **mathFrame**;

public **ButtonPlusAdapter**(MathFrame mathFrame) { this.mathFrame = mathFrame;

}

public void **actionPerformed**(ActionEvent e) { this.mathFrame.onPlus();

}

}

// ...

public class **ButtonMinusAdapter** implements ActionListener { final MathFrame **mathFrame**;

public **ButtonMinusAdapter**(MathFrame mathFrame) { this.mathFrame = mathFrame;

}

public void **actionPerformed**(ActionEvent e) { this.mathFrame.onMinus();

}

}

Die Anwendung hat zwei offensichtliche Schwächen: Erstens enthalten die onPlus- und onMinus-Methoden von MathFrame fast denselben Code (aus onPlus ist per Copy&Paste onMinus gemacht worden...). Zweitens unterscheiden sich die beiden

Adapter-Klassen eigentlich nur durch die Implementierung der jeweiligen

actionPerformed-Methode.

Das erste Problem werden wir später angehen. Hier geht's zunächst einmal um das zweite. Um gerade einmal die Clicks zweier Buttons an eine entsprechende Verarbeitungsmethode weiterzuleiten, werden zwei öffentliche Adapter-Klassen definiert

- mit jeweils etwa 10 Zeilen (die sich jeweils nur in einer einzigen Zeile unterscheiden). Trotzdem ist diese Lösung, was die Verwendung von Adaptern angeht, state of the art.

In den folgenden Abschnitten werden die inneren Klassen vorgestellt. Die Verwendung der inneren Klassen wird dazu führen, dass der Schreibaufwand radikal reduziert werden kann. Wobei aber die Technik – die Weiterleitung eines Events via Adapter – unverändert bleibt. Das Objektdiagramm bleibt also im Prinzip immer dasselbe.

## Statische Member-Klassen

Statt globaler Adapter-Klassen werden nun statische Member-Klassen verwendet.

Die neuen Klassen werden deshalb als Member-Klassen bezeichnet, weil sie auf derselben Ebene definiert sind wie die Member (Felder und Methoden) der umschließenden Klasse.

Für die beiden Adapter-Klassen werden keine eigenen java-Dateien mehr angelegt – stattdessen werden sie mit dem Schlüsselwort static direkt im Kontext der MathFrame- Klasse definiert (nur dort werden sie benötigt). Was static dabei bedeutet, wird im nächsten Abschnitt deutlich werden.

Das Objektdiagramm sieht fast genauso aus wie dasjenige der letzten Anwendung:



: MathFrame

: MathFrame$ ButtonPlusAdapter

: MathFrame$ ButtonMinusAdapter

mathFrame

mathFrame

*ActionListener*

*ActionListener*

: JButton

"Plus"

: JButton

"Minus"

dActionListener

ad

dActionListener

ad

ormed

actionPerf

OR

CT

ormed

actionPerf

OR

CT

Minus

on

Plus

on

:JTextField

:JTextField

:JTextField

Nur die Namen der Adapter-Klassen haben sich geändert. Die Klassen werden nun über den Namen der "äußeren" Klasse angesprochen (MathFrame.ButtonPlusAdapter – der Name im obigen Bild ist der "interne" Name). Aber technisch hat sich an dem Bild nichts verändert.

Der Quellcode:

// ...

public class **MathFrame** extends JFrame {

static class **ButtonPlusAdapter** implements ActionListener { final MathFrame **mathFrame**;

public **ButtonPlusAdapter**(MathFrame mathFrame) { this.mathFrame = mathFrame;

}

public void **actionPerformed**(ActionEvent e) { this.mathFrame.onPlus();

}

}

static class **ButtonMinusAdapter** implements ActionListener { final MathFrame **mathFrame**;

public **ButtonMinusAdapter**(MathFrame mathFrame) { this.mathFrame = mathFrame;

}

public void **actionPerformed**(ActionEvent e) { this.mathFrame.onMinus();

}

}

private final JTextField **textFieldX** = new JTextField(10); private final JTextField **textFieldY** = new JTextField(10); private final JButton **buttonPlus** = new JButton("Plus"); private final JButton **buttonMinus** = new JButton("Minus"); private final JTextField **textFieldResult** = new JTextField(10);

public **MathFrame**() { ... }

private void **registerListeners**() { this.buttonPlus.addActionListener(new ButtonPlusAdapter(this)); this.buttonMinus.addActionListener(new ButtonMinusAdapter(this));

}

private void **onPlus**() { ... } private void **onMinus**() { ... }

}

Die statischen Adapter-Klassen hätten wir auch als private definieren können – außerhalb der umschließenden Klassen ist ihre Verwendung wenig sinnvoll... Die actionPerformed-Methode der Adapter-Klassen rufen onPlus und onMinus auf. Diese Methoden können nun private sein (anders als bei der Verwendung globaler Adapter- Klassen).

Der Compiler hat folgende class-Dateien erzeugt:

MathFrame$ButtonPlusAdapter MathFrame$ButtonDiffAdapter

## Nicht statische Member-Klassen

Im folgenden wird auf der letzten Lösung aufgebaut. Statt aber die Adapter-Klassen als static zu deklarieren, werden sie als nicht-static definiert – und das hat Konsequenzen.

Wird eine nicht statische Klasse in Kontext einer anderen Klasse (einer "äußeren", "umschließenden") Klasse definiert, generiert der Compiler automatisch eine Instanzvariable, die auf das "äußere" Objekt zeigen wird – und einen Konstruktor, der diese Instanzvariable initialisiert. Der Entwickler muss also nurmehr die actionPerformed-Methode implementierten. Warum generiert der Compiler diesen Code? Weil alle Adapter-Klassen eine entsprechende Infrastruktur haben – die müssen ihren "Erzeuger" kennen, um später auf diesen Erzeuger Methoden aufrufen zu können.

Und bei der Erzeugung der Adapter muss nicht mehr explizit this mehr übergeben werden – dies behält sich der Compiler vor (der automatisch das this ergänzt...).

Das Objektdiagramm hat sich kaum verändert:



: MathFrame

: MathFrame$ ButtonPlusAdapter

: MathFrame$ ButtonMinusAdapter

MathFrame.this

MathFrame.this

*ActionListener*

*ActionListener*

: JButton

"Plus"

: JButton

"Minus"

dActionListener

ad

dActionListener

ad

ormed

actionPerf

OR

CT

ormed

actionPerf

OR

CT

Minus

on

Plus

on

:JTextField

:JTextField

:JTextField

Der Compiler muss der von ihm generierten MathFrame-Referenz natürlich einen Namen geben. Diese Referenz wird nun über den Namen MathFrame.this angesprochen (also: Name der umschließenden Klasse plus .this).

Der Compiler "schenkt" uns also die Referenz auf das MathFrame-Objekt und den Konstruktor, der diese Referenz initialisiert. Diese "Geschenke" sind im obigen Diagramm gestrichelt dargestellt.

Hier der Quellcode (dessen Umfang deutlich geschrumpft ist):

// ...

public class **MathFrame** extends JFrame {

private class **ButtonPlusAdapter** implements ActionListener { public void **actionPerformed**(ActionEvent e) {

MathFrame.this.onPlus();

}

}

private class **ButtonMinusAdapter** implements ActionListener { public void **actionPerformed**(ActionEvent e) {

MathFrame.this.onMinus();

}

}

private final JTextField **textFieldX** = new JTextField(10); private final JTextField **textFieldY** = new JTextField(10); private final JButton **buttonPlus** = new JButton("Plus"); private final JButton **buttonMinus** = new JButton("Minus"); private final JTextField **textFieldResult** = new JTextField(10);

public **MathFrame**() { ... }

private void **registerListeners**() { this.buttonPlus.addActionListener(new ButtonPlusAdapter()); this.buttonMinus.addActionListener(new ButtonMinusAdapter());

}

private void **onPlus**() { ... } private void **onMinus**() { ... }

}

Im Unterschied zu statischen Member-Klassen besitzen nicht-statische Memberklassen also automatisch eine vom Compiler bereitgestellte "Infrastruktur" (Referenzvariable plus Konstruktor).

Man beachte, dass zwar die Klassen "geschachtelt" sind, nicht aber die Objekte!

Der Compiler generiert aus den obigen Memberklassen die folgenden class-Dateien: MathFrame$ButtonPlusAdapter

MathFrame$ButtonMinusAdapter

Wir könnten diese beiden Klassen per Reflection analysieren – und würden genau diejenigen Features wiederfinden, die wir auch bei den globalen Adapter-Klassen finden würden (technisch ist also alles beim alten geblieben).

## Local Classes

Die im letzten Abschnitt vorgestellten Member-Klassen können natürlich in allen Methoden der umschließenden Klasse genutzt werden. Wir könnten also z.B. auch (völlig sinnloserweise) in der onPlus-Methode Instanzen dieser Klassen erstellen. Das ist natürlich nicht wünschenswert.

Es wäre also schön, wenn die Adapter-Klassen direkt im Kontext derjenigen Methode definiert werden könnten, in welcher sie ausschließlich gebraucht werden. Wir müssten sie also dort definieren können, wo auch lokale Variablen definiert werden können. Ebenso wie die Sichtbarkeit von lokalen Variablen auf denjenigen Block beschränkt ist, in welchem sie definiert sind, wäre dann auch die Sichtbarkeit solcher "lokale Klassen" auf den jeweiligen Block beschränkt.

Das Objektdiagramm ändert sich nur unwesentlich:



: MathFrame

: MathFrame$1 ButtonPlusAdapter

: MathFrame$1 ButtonMinusAdapter

MathFrame.this

MathFrame.this

*ActionListener*

*ActionListener*

: JButton

"Plus"

: JButton

"Minus"

dActionListener

ad

dActionListener

ad

ormed

actionPerf

OR

CT

ormed

actionPerf

OR

CT

Minus

on

Plus

on

:JTextField

:JTextField

:JTextField

Der Compiler generiert nun folgende Klassen:

MathFrame$1ButtonPlusAdapter MathFrame$1ButtonMinusAdapter

Hier der Quellcode:

// ...

public class **MathFrame** extends JFrame {

private final JTextField **textFieldX** = new JTextField(10); private final JTextField **textFieldY** = new JTextField(10); private final JButton **buttonPlus** = new JButton("Plus"); private final JButton **buttonMinus** = new JButton("Minus"); private final JTextField **textFieldResult** = new JTextField(10);

public **MathFrame**() { ... }

private void **registerListeners**() {

class **ButtonPlusAdapter** implements ActionListener { public void **actionPerformed**(ActionEvent e) {

MathFrame.this.onPlus();

}

}

this.buttonPlus.addActionListener(new ButtonPlusAdapter());

class **ButtonMinusAdapter** implements ActionListener { public void **actionPerformed**(ActionEvent e) {

MathFrame.this.onMinus();

}

}

this.buttonMinus.addActionListener(new ButtonMinusAdapter());

}

private void **onPlus**() { ... } private void **onMinus**() { ... }

}

Natürlich kann eine lokale Klasse keinen Sichtbarkeits-Modifizierer besitzen – sie ist einfach nur lokal.

Unmittelbar nach der Definition der Adapter-Klassen wird nun jeweils ein Objekt dieser Klassen erzeugt und bei den Buttons registriert. Der Text ist wesentlich besser lesbar geworden – und je lokaler eine Definition, desto besser.

## Anonymous Classes

Die letzte Überlegung: Im letzten Abschnitt wurde eine Klasse ButtonPlusAdapter definiert – die aber nur an einer einzigen Textstelle instanziiert wurde. (Dasselbe gilt für die Klasse ButtonMinusAdapter). Der Name, der bei der Klassendefinition eingeführt wurde, wird also nur genau an einer einzigen Textstelle benutzt. Dann kann aber auf den Namen komplett verzichtet werden – indem der Name, also der Bezeichner, durch das von ihm Bezeichnete ersetzt wird. So sind wir schließlich bei anonymen Klassen angelangt.

Das Klassendiagramm unterscheidet sich vom letzten Diagramm wiederum nur durch die Klassennamen, die der Compiler vergibt:



: MathFrame

: MathFrame$1

: MathFrame$2

MathFrame.this

MathFrame.this

*ActionListener*

*ActionListener*

: JButton

"Plus"

: JButton

"Minus"

dActionListener

ad

dActionListener

ad

ormed

actionPerf

OR

CT

ormed

actionPerf

OR

CT

Minus

on

Plus

on

:JTextField

:JTextField

:JTextField

Der Quellcode:

// ...

public class **MathFrame** extends JFrame {

// wie gehabt ...

private void **registerListeners**() { this.buttonPlus.addActionListener(new ActionListener {

public void **actionPerformed**(ActionEvent e) { MathFrame.this.onPlus();

}

});

this.buttonMinus.addActionListener(new ActionListener { public void **actionPerformed**(ActionEvent e) {

MathFrame.this.onMinus();

}

});

}

// wie gehabt ...

}

In registerListeners werden zwei anonyme Klassen definiert, welche das Interface ActionListener implementieren – also die Methode actionPerformed bereitstellen. Der Compiler spendiert automatisch den MathFrame.this-Verweis und einen Konstruktor, der diesen Verweis initialisiert. Beide Klassen werden dann jeweils instanziiert. Und schließlich wird das Ergebnis der Instanziierung (die jeweilige Adapter) an die addActionListener-Methode übergeben.

In den jeweils 5 Zeilen passiert also recht viel – sowohl zur Compilezeit als auch zur Laufzeit. Man sieht es allerdings nicht mehr so deutlich. Aber noch einmal: technisch hat sich im Vergleich zur Lösung mit globalen Klassen nichts verändert.

## Lambdas

Und der Compiler kann uns noch mehr schenken.

Betrachten wir noch einmal die Definition der ersten anonymen Adapter-Klasse und die Instanziierung dieser Klasse:

this.buttonPlus.addActionListener(new ActionListener() { @Override

public void **actionPerformed**(ActionEvent event) { MathFrame.this.onPlus();

}

});

Die Voraussetzung der folgenden Überlegungen besteht darin, dass wir ein Interface implementieren müssen, welches nur genau eine einzige abstrakte Methode hat. Genau diese Voraussetzung aber ist beim Interface ActionListener gegeben (seine einzige Methode ist actionPerformed).

Solche Interfaces, die nur genau eine einzige abstrakte Methode haben, heißen seit Java 8 "Funktionale Interfaces". (Ein funktionales Interface kann neben dieser einen abstrakten Methode aber durchaus noch default- und static-Methoden besitzen.)

Nun zu den zusätzlichen Geschenken des Compilers.

Der Compiler weiß, dass der actionPerformed-Methode ein ActionListener übergeben werden muss. Weshalb reiben wir dies dann dem Compiler noch unter seine Nase? Der Compiler weiß weiterhin, dass dieses Interface nur eine einzige Methode hat

* eine Methode namens actionPerformed (dass es sich also um ein funktionales Interface handelt). Weshalb reiben wir auch dies dann dem Compiler wieder unter seine Nase? Genau diese beiden Dinge müsste der Compiler automatisch hinzufügen können, wenn wir sie einfach unerwähnt ließen. Übrig bliebe dann nur die Parameterliste der actionPerformed-Methode und deren Implementierung:

this.buttonPlus.addActionListener( (ActionEvent event) {

MathFrame.this.onPlus();

}

);

Damit der Compiler nun aber weiß, dass wir bequem sein wollen und es ihm überlassen, wieder zu ergänzen, was wir weggelassen haben, müssen wir zwischen der Parameterliste und der Implementierung der Methode ein -> setzen:

this.buttonPlus.addActionListener( (ActionEvent event) -> {

MathFrame.this.onPlus();

}

);

Der obige Code wird anstandslos übersetzt.

Wir können noch mehr weglassen. Der Compiler kennt natürlich auch den Typ des Parameters der handle-Methode (ActionEvent). Auch auf die Nennung diesen Namen können wir verzichten:

this.buttonPlus.addActionListener( (event) -> {

MathFrame.this.onPlus();

}

);

Da nun die Parameterliste nurmehr den Namen genau eines einzigen Parameters beseht, können wir auch auf die runden Klammern verzichten:

this.buttonPlus.addActionListener( event -> {

MathFrame.this.onPlus();

}

);

Und da die Implementierung nurmehr aus einer einzigen Anweisung besteht, können wir auch auf die geschweiften Klammern verzichten:

this.buttonPlus.addActionListener( event ->

MathFrame.this.onPlus();

);

Schließlich können wir dann der Kürze halber auch alles in einer einzigen Zeile formulieren:

this.buttonPlus.addActionListener(event -> MathFrame.this.onPlus());

Und wie auch bei anderen Klassen, die eine "automatische" Referenz auf das Objekt der äußeren Klassen enthalten, können wir auch hier auf MathFrame.this verzichten:

this.buttonPlus.addActionListener(event -> onPlus());

Hier die komplette neue registerHandlers-Methode:

package **appl**;

// ...

public class **MathFrame** extends JFrame {

// ...

private void **registerListeners**() {

this.buttonPlus.addActionListener( event -> MathFrame.this.onPlus());

this.buttonMinus.addActionListener( event -> MathFrame.this.onMinus());

Members.print(this.buttonPlus.getOnAction().getClass());

}

// ...

}

Der Aufruf von Members.print zeigt, dass auch hier (prinzipiell) alles beim Alten geblieben ist:

class **MathFrame$$Lambda$94/1756944838** extends Object Constructors

private **MathFrame$$Lambda$94/1756944838**(MathFrame arg0)

Fields

private final MathFrame **arg$1**

Methods

private static ActionListener **get$Lambda**(MathFrame arg0) public void **actionPerforme**(ActionEvent arg0)

Auch die "Lambda-Klasse" hat eine Referenz auf den MathFrame (die intern aber nicht mehr mit this$0, sondern mit arg$1 bezeichnet wird) und einen Konstruktor, der diese Referenz initialisiert. Neben der (von uns selbst definierten) actionPerformed-Methode besitzt diese Klasse aber noch eine weitere private Methode, die für unsere Zwecke aber irrelevant ist (get$Lambda).

Der Name der Klasse (und daher auch der Name des Konstruktors) ist aber recht merkwürdig. Und ein weiterer wichtiger Unterschied zu allen bislang diskutierten Klassen: für diese Klasse wird keine(!) eigene .class-Datei erzeugt. Der Code dieser Klasse befindet sich also in derjenigen .class-Datei, die für die äußere Klasse erzeugt wurde (hier: in der MathFrame.class-Datei).

# Details zu Lambdas

Lambdas, so hatten wir bislang argumentiert, sind eine "Vereinfachung" von anonymen Klassen – eine Vereinfachung, die aber nur dann verwendet werden kann, wenn es sich um Implementierungen funktionaler Interfaces handelt (solche Interfaces werden auch als Single-Method-Interfaces – SMI – bezeichnet).

Dabei haben wir ausgeblendet, dass anonyme Klasse und Lambdas aber auch in mancher Hinsicht unterscheiden.

### Eine Übersicht

* + Target Typing
  + this und "Outer".this
  + Lambda-Ausdrücke mit returns
  + "Method-References"
  + Serialisierung von Lambdas

## Target-Typing

Ein Lambda-Ausdruck hat "als solcher" keinen Typ – der Typ eines solchen Ausdrucks kann nur aufgrund des Ziels einer Zuweisung ermittelt werden (oder aufgrund eines Downcasts).

Das Interface Runnable der Standardbibliothek ist ein funktionales Interface:

package **java.util**;

@FunctionalInterface

public interface **Runnable** { public void **run**();

}

By the way: die Java 8- Annotation @FunctionalInterface kann verwendet werden, um eben ein funktionales Interface auszuzeichnen. Dies dient der Dokumentation und ermöglicht dem Compiler die Ausgabe einer Fehlermeldung, wenn das Interface mehr als eine abstrakte Methode spezifiziert (oder gar keine!).

Wir bauen ein eigenes funktionales Interface:

package **appl**;

@FunctionalInterface public interface **Speaker** {

public void **speak**();

}

Sowohl die run-Methode von Runnable als auch die speak-Methode von Speaker sind parameterlos und liefern void.

Bislang konnte an eine Object-Variable alles mögliche zugewiesen werden:

Object o1 = "Hello"; Object o2 = 42;

Bei der letzten Zuweisung findet bekanntlich Autoboxing statt (die automatische Umwandlung von int nach Integer).

Die folgende Zeile wird allerdings nicht(!) übersetzt:

Object o3 = () -> System.out.println("Hello"); // illegal

Warum nicht?

Bei jeder Zuweisung muss der Compiler den Typ der rechten Seite berechnen können – um entscheiden zu können, ob die Zuweisung erlaubt ist.

Der Lambda-Ausdruck auf der rechten Seite der Zuweisung kann aber eben nicht berechnet werden: es könnte sich um sowohl um den Typ Runnable als auch um den Typ Speaker handeln (und um alle weiteren Typen funktionaler Interfaces, die eine parameterlose void-Methode spezifizieren).

Folgende Zeilen aber werden übersetzt:

Runnable r = () -> System.out.println("Hello"); Speaker s = () -> System.out.println("World");

Die Typen der Lambda-Ausdrücke können nun vom Typ der jeweiligen Zielvariablen abgeleitet werden: die erste Lambda-Klasse implementiert Runnable, die zweite Speaker.

Natürlich kann der Typ eines Lambdas auch dann berechnet werden, wenn dem Lambda ein Downcast vorangestellt wird (und das Resultat kann dann wieder einer Object-Variablen zugewiesen werden):

Object o3 = (Runnable) () -> System.out.println("Hello"); Object o4 = (Speaker) () -> System.out.println("World");

Angenommen, wir definieren folgende Methoden:

static void **workWithRunnable**(Runnable r) { r.run();

}

static void **workWithSpeaker**(Speaker s) { s.speak();

}

Dann sind folgende Aufrufe korrekt:

workWithRunnable(() -> System.out.println("Hello")); workWithSpeaker(() -> System.out.println("World"));

Die Typ eines jeden der beiden Lambda-Ausdrücke ist der Type des Parameters der jeweiligen workWith-Methode.

Eine Methode, die Object liefert, kann keine return-Anweisung mit einem Lambda enthalten:

static Object **createRunnable**() {

return () -> System.out.println("Hello"); // illegal

}

Die folgende Methode aber wird übersetzt:

static Speaker **createSpeaker**() {

return () -> System.out.println("Hello");

}

Sie kann wie folgt aufgerufen werden:

Speaker speaker = createSpeaker(); speaker.speak();

Der Umstand, dass ein Lambda-Ausdruck "als solcher" keinen Typ hat, sondern dieser Typ nur aus dem "Ziel" heraus berechnet werden kann, wird als "Target Typing" bezeichnet.

## Der this- und der "Outer"-this-Zeiger

Eine anonyme Klasse, die ein funktionales Interface implementiert, kann neben der Methode, die im Interface spezifiziert ist, noch weitere Methoden implementieren – und Attribute definieren, auf denen diese Methoden operieren.

Eine Lambda-Klasse dagegen kann nur die Methode des Interfaces implementieren – ein Objekt einer solchen Klasse ist also im Grunde nichts anderes als eine objektorientierte Hülle um eben diese Methode (deshalb könnte man solche Objekte auch als "Funktions-Objekte" bezeichnen).

Ein Schaubild:

Ein Objekt einer anonymen Klasse Ein Objekt einer Lambda-Klasse



: Foo

x

creates

*Runnable*

Foo.this

: ???

x

this

run

lper

he

mo

de

lper

he



: Bar

x

creates

*Runnable*

Bar.this

: ???

this

run

mo

de

lper

he

Ein Foo-Objekt hat ein Attribut x, eine Methode namens helper und eine demo- Methode, innerhalb derer ein Objekt einer anonymen Klasse erzeugt wird – eine Klasse, die das Interface Runnable implementiert (also die run-Methode).

Die anonyme Klasse hat ihrerseits ebenfalls ein Attribut namens x und eine Methode helper. Wenn wir nun innerhalb der run-Methode auf das x-Attribut und die helper- Methode des Foo-Objekts zugreifen wollen, können wir dies via Foo.this.x resp. Foo.this.helper() tun. Wollen wir dagegen auf das x-Attribut und die helper-

Methode des Runnable-Objekts zugreifen, erledigen wir dies via this.x resp.

this.helper().

Und in der run-Methode des Runnables könnten wir schließlich auch eben diese run- Methode wieder rekursiv aufrufen.

Die beiden Ausdrücke Foo.this und this zeigen also auf unterschiedliche Objekte: der erste zeigt auf das Foo-Objekt, in dessen run-Methode das Runnable erzeugt wurde; der zweite zeigt auf das Runnable-Objekt selbst. Den Versuch, beide Referenzen miteinander zu vergleichen, beantwortet der Compiler mit einer Fehlermeldung – da sie unterschiedlichen Typs sind, können sie natürlich nie und nimmer auf dasselbe Objekt zeigen (der Vergleich ist als unsinnig).

Hier die Foo-Klasse mit ihrer anonymen Runnable-Klasse:

package **appl**; public class **Foo** {

public int **x** = 77;

public void **demo**() {

final Runnable runnable = new **Runnable**() { private int **x** = 42;

private void **helper**() { System.out.println("inner helper");

}

@Override

public void **run**() { System.out.println(this); System.out.println(Foo.this);

// System.out.println(this == Foo.this); // error...

System.out.println(x); System.out.println(this.x); System.out.println(Foo.this.x);

helper(); this.helper(); Foo.this.helper();

}

};

runnable.run();

}

public void **helper**() { System.out.println("outer helper");

}

}

Die Ausgaben:

appl.Foo$1@5f205aa appl.Foo@57fa26b7 42

42

77

inner helper inner helper outer helper

Die demo-Methode der Bar-Klasse erzeugt ein Runnable, dessen Klasse als Lambda- Klasse implementiert ist. Auch hier zeigt Bar.this auf dasjenige Bar-Objekt, in deren demo-Methode das Runnable erzeugt wird. Aber hier zeigt nun auch der this- Parameter der run-Methode des Runnables nicht auf das Runnable, sondern auf das Bar-Objekt.

Die Ausdrücke Bar.this und this meinen hier also dasselbe – und somit liefert der der Referenzvergleich der beiden Pointer true. Der Compiler gibt bei einem Vergleich eine Warnung aus: "Comparing identical expressions".

Anders ausgedrückt: in der run-Methode des Lambda-Runnables können wir überhaupt nicht mehr auf eben dieses Runnable zugreifen. Würde this auf dieses Runnable zeigen, so könnte dieser Pointer ja ohnehin nur zu einem einzigen Zweck genutzt werden: zum rekursiven Aufruf der run-Methode des Runnables (und dies wäre in fast allen Fällen ziemlicher Unsinn…).

Hier die Bar-Klasse mit ihrer mittels eines Lambda-Ausdrucks definierten Runnable- Klasse:

package **appl**; public class **Bar** {

public int **x** = 77;

public void **demo**() {

final Runnable runnable = () -> { System.out.println(this); System.out.println(Bar.this);

// System.out.println(this == Bar.this); // Warning...

System.out.println(x); System.out.println(this.x); System.out.println(Bar.this.x);

helper(); this.helper(); Bar.this.helper();

};

runnable.run();

}

public void **helper**() { System.out.println("outer helper");

}

}

Die Ausgaben:

appl.Bar@2471cca7 appl.Bar@2471cca7 77

77

77

outer helper outer helper outer helper

## Mit oder ohne "Outer".this?

Bei anonymen Klassen wird uns die "Outer".this-Referenz (und der Konstruktor, der diese initialisiert) vom Compiler auch dann zur Verfügung gestellt, wenn diese in der Implementierung der anonymen Klasse gar nicht genutzt wird.

In der demo1-Methode der folgenden Foo-Klasse wird innerhalb des Runnables die Foo.this-Referenz genutzt; in der demo2-Methode wird diese Referenz nicht genutzt. Dennoch wird sie in den Klassen beider Runnables definiert:

package **appl**;

import jn.util.Members; public class **Foo** {

public int **x** = 42;

public void **demo1**() {

final Runnable runnable = new **Runnable**() { @Override

public void **run**() { System.out.println(Foo.this.x);

}

};

runnable.run(); Members.print(runnable.getClass());

}

public void **demo2**() {

final Runnable runnable = new **Runnable**() { @Override

public void **run**() {

}

};

runnable.run(); Members.print(runnable.getClass());

}

}

Rufen wir die demo1- und die demo2-Methoden auf ein Foo-Objekt auf, so ergeben sich die folgenden Ausgaben:

42

class **Foo$1** extends Object Constructors

**Foo$1**(Foo this$0)

Fields

final Foo **this$0**

Methods

public void **run**()

class **Foo$2** extends Object Constructors

**Foo$2**(Foo this$0)

Fields

final Foo **this$0**

Methods

public void **run**()

Bei Lambda-Klassen wird diese "Outer".this-Referenz (und der Konstruktor, der diese initialisiert) vom Compiler nur dann generiert, wenn sie in der Lambda-Methode tatsächlich genutzt wird.

In der demo1-Methode der folgenden Bar-Klasse wird diese Referenz genutzt – in der

demo2-Methode wird sie nicht genutzt:

package **appl**;

import jn.util.Members; public class **Bar** {

public int **x** = 42;

public void **demo1**() {

final Runnable runnable = () -> { System.out.println(Bar.this.x);

};

runnable.run(); Members.print(runnable.getClass());

}

public void **demo2**() {

final Runnable runnable = () -> { }; runnable.run(); Members.print(runnable.getClass());

}

}

Beim Aufruf der demo1- und die demo2-Methoden auf ein Bar-Objekt ergeben sich die folgenden Ausgaben:

Die Ausgaben:

42

class **Bar$$Lambda$3/475266352** extends Object Constructors

private **Bar$$Lambda$3/475266352**(Bar arg0)

Fields

private final Bar **arg$1**

Methods

private static Runnable **get$Lambda**(Bar arg0) public void **run**()

class **Bar$$Lambda$4/1355531311** extends Object Constructors

private **Bar$$Lambda$4/1355531311**()

Fields Methods

public void **run**()

## Lambda-Ausdrücke mit Return-Anweisungen

Sofern der Implementierungs-Teil eines Lambda-Ausdrucks nur eine einzige Anweisung enthält, muss diese Anweisung bekanntlich nicht mehr geschweift geklammert werden. Wenn nun diese Anweisung eine (nicht geklammerte) return-Anweisung ist, schenkt uns der Compiler zusätzlich auch das Schlüsselwort return.

Sei folgendes Interface gegeben:

package **appl**;

@FunctionalInterface

public interface **BinaryOperator** {

public abstract int **apply**(int x, int y);

}

(Ein ähnliches Interface wurde auch von Java 8 eingeführt – angesiedelt im Paket java.util.functions. Wir verwenden aber zunächst selbst-definierte Interfaces. Die Interfaces von Java 8 werden später eingehend behandelt werden.)

Wir könnten ein Plus-Operator-Objekt einer anonymen Klasse erzeugen und anwenden:

final BinaryOperator plus = new **BinaryOperator**() { @Override

public int **apply**(int x, int y) {

return x + y;

}

};

System.out.println(plus.apply(40, 2));

Wir könnten einen Lambda-Ausdruck mit einer geschweift geklammerten return- Anweisung definieren (hier ist return verpflichtend):

final BinaryOperator minus = (int x, int y) -> { return x - y;

};

System.out.println(minus.apply(40, 2));

Verzichten wir nun aber auch auf die geschweiften Klammern, so müssen wir auch auf das Schlüsselwort return verzichten:

final BinaryOperator times = (int x, int y) -> x \* y; System.out.println(times.apply(40, 2));

(Die Anwendung des Operators mittels apply liefert übrigens in allen drei Fällen den Wert 42.)

## Method-References

Rufen wir im Implementierungs-Teil eines Lambda-Ausdrucks eine Methode auf, an welche genau diejenigen Parameter weitergereicht werden, die als Parameter an die Lambda-Methode übergeben werden, so erübrigt sich auch die Definition und das Weiterreichen dieser Parameter.

Das, was von einem Lambda-Ausdruck dann übrig bleibt, wird als "Method-Reference" bezeichnet (ein durchaus irreführender Name – in Java können Referenzen bekanntlich nur auf Objekte zeigen (und eben nicht auf Methoden).

Gegeben sei folgendes funktionale Interface (ein ähnliches Interface existiert auch bereits in java.util.functions):

package **appl**;

@FunctionalInterface

public interface **Consumer**<T> {

public abstract void **accept**(T value);

}

Wir bauen ein Lambda-Objekt, dessen Klasse Consumer<String> implementiert – und rufen auf dieses Objekt die accept-Methode auf:

Consumer<String> consumer1 = s -> System.out.println(s); consumer1.accept("Spring");

Der Parameter s des Lambda-Ausdrucks wird unverändert an die Methode

System.out.println weitergereicht.

Deshalb kann der obige Lambda-Ausdruck auch kürzer formuliert werden – mittels einer Method-Reference (wobei der Punkt vor dem Methodennamen ersetzt wird durch den doppelten Doppelpunkt):

Consumer<String> consumer2 = System.out::println; consumer2.accept("Summer");

Sei print eine Methode der umschließenden Klasse eines Lambda-Ausdruckt, die einen String-Argument erwartet:

void **print**(String s) { System.out.println(s);

}

Dann kann diese Methode im Implementierungs-Teil des Lambda-Ausdrucks wie folgt aufgerufen werden:

Consumer<String> consumer3 = s -> this.print(s);; consumer3.accept("Autumn");

Auch hier kann der komplette Lambda-Ausdruck ersetzt werden durch eine Method- Referenz (man beachte den doppelten Doppelpunkt vor print):

Consumer<String> consumer4 = this::print; consumer4.accept("Winter");

Sei schließlich sprint eine statische Methode der Klasse Application:

class **Application** {

// ...

static void **sprint**(String s) { System.out.println(s);

}

}

Dann könnte folgender Consumer<String> erzeugt und verwendet werden:

Consumer<String> consumer5 = Application::print; consumer5.accept("Karneval");

Sei schließlich folgende Klasse gegeben:

class **Foo** {

public **Foo**(String s) {

// ...

}

}

Dann kann ein Consumer<String>, der in seiner accept-Methode aufgrund des ihr übergebenen String-Parameters ein Foo-Objekt erzeugen will, wie folgt definiert werden:

Consumer<String> consumer6 = Foo::new; consumer6.accept("Fasching");

Der an accept übergebene String-Parameter wird eins zu eins an den Foo-Konstruktor weitergereicht.

## Serialisierung von Lambdas

Sei wiederum folgendes Consumer-Interface gegeben:

package **appl**;

@FunctionalInterface

public interface **Consumer**<T> {

public abstract void **accept**(T value);

}

Eine Lambda-Implementierung dieses Interfaces ist von Haus aus nicht(!) serialisierbar. Der Aufruf von writeObject wirft eine NotSerializableException:

static void **demo1**() {

final Consumer<String> c = s -> System.out.println(s); try(ObjectOutputStream out = new ObjectOutputStream(

new FileOutputStream("zzz.dat"))) { out.writeObject(c);

}

catch(Exception e) { System.out.println(e);

}

}

Wir können aber mittels eines "Tricks" für die Serialisierbarkeit sorgen. Wir casten den Lambda-Ausdruck auf (Consumer<String> & Serializable):

static void **demo2**() {

final Consumer<String> c = (Consumer<String> & Serializable) s -> System.out.println(s);

try(ObjectOutputStream out = new ObjectOutputStream( new FileOutputStream("zzz.dat"))) {

out.writeObject(c);

}

catch(Exception e) { System.out.println(e);

}

try(ObjectInputStream in = new ObjectInputStream( new FileInputStream("zzz.dat"))) {

Consumer<String> cc = (Consumer<String>)in.readObject(); cc.accept("Hello");

}

catch(Exception e) { System.out.println(e);

}

}

Dann kann das Lambda-Objekt serialisiert werden; und auch die Deserialisierung verläuft problemlos.

Natürlich könnten wir alternativ ein weiteres Interface definieren, das sowohl von

Consumer als auch von Serializable abgeleitet ist:

package **appl**;

import java.io.Serializable; @FunctionalInterface

public interface **SerializableConsumer**<T> extends Consumer<T>, Serializable {

}

Bauen wir nun einen Lambda-Ausdruck, der einer SerializableConsumer-Variablen zugewiesen wird, verläuft auch hier die Serialisierung und Deserialisierung ohne Probleme:

static void **demo3**() {

final SerializableConsumer<String> c = s -> System.out.println(s); try(ObjectOutputStream out = new ObjectOutputStream(

new FileOutputStream("zzz.dat"))) { out.writeObject(c);

}

catch(Exception e) { System.out.println(e);

}

try(ObjectInputStream in = new ObjectInputStream( new FileInputStream("zzz.dat"))) {

Consumer<String> cc = (Consumer<String>)in.readObject(); cc.accept("World");

}

catch(Exception e) { System.out.println(e);

}

}

Hinweis: In Java 8 ist eine Vielzahl funktionaler Interfaces eingeführt worden (diese Interfaces werden später vorgestellt werden) – Lambda-Klassen, welche diese Interfaces implementieren, sind von Haus aus ebenfalls nicht serialisierbar.

# Closures

Eine Definition aus Wikipedia:

*Eine Closure (oder Funktionsabschluss) ist ein Konzept aus der funktionalen Programmierung. Es beschreibt eine Funktion, die Zugriffe auf ihren Erstellungskontext enthält. Beim Aufruf greift die Funktion dann auf diesen Erstellungskontext zu.*

Ziemlich abstrakt…

Die Implementierung lokaler Klassen, anonymer Klassen oder Lambda-Klassen kann auf lokale Variablen (resp. Parameter) ihrer umschließenden Methode zugreifen – sofern diese als final definiert sind.

Insbesondere bei Lambda-Ausdrücken wird diese Möglichkeit extensiv genutzt.

### Übersicht

* Technische Aspekte von Closures.
* Beispiel: Implementierung von Runnable
* Ein BinaryOperator
* final und "effictively final"

## Grundlagen

Zunächst ein Objektdiagramm:

: Application

calls

copy

*Runnable*

: MyRunnable

calls

run

point

nnable

createRu

>

<init

: Point

Der Konstruktor der Klasse Application ruft die Methode createRunable auf. Diese Methode definiert eine lokale Variable point, die mit einem Point initialisiert wird.

Dann wird ein Runnable gebaut – in Form einer lokalen Klasse namens MyRunnable.

Die run-Methode dieser Klasse kann nun neben dem Application.this-Pointer auch den Namen der lokalen point-Variablen der createRunnable-Methode verwenden.

Die run-Methode wird schließlich vom Application-Konstruktor aufgerufen. Hier der Quellcode:

package **appl**;

import java.awt.Point; import jn.util.Members;

public class **ApplicationWithLocalClass** { public static void **main**(String[] args) {

new ApplicationWithLocalClass();

}

public **ApplicationWithLocalClass**() {

final Runnable runnable = this.createRunnable(); runnable.run(); Members.print(runnable.getClass());

}

public Runnable **createRunnable**() { final Point point = new Point(1, 2);

class MyRunnable implements **Runnable** {

@Override

public void **run**() { System.out.println(point);

}

}

return new MyRunnable();

}

}

Die Ausgaben:

java.awt.Point[x=1,y=2]

Worin liegt das Problem? Und worin besteht die Lösung?

Die Variable point ist eine lokale Variable der Methode createRunnable. Sie liegt somit auf dem Stack. Wird createRunnable verlassen, so wird diese Variable (die Referenz auf den Point) wieder vom Stack heruntergenommen.

Die Methode run des MyRunnables aber wird erst später(!) aufgerufen – wenn die Methode createRunnable längst verlassen wurde und die point-Variable vom Stack entfernt wurde.

Zunächst scheint hier etwas nicht mit rechten Dingen zuzugehen.

Die Lösung besteht auch wieder darin, dass der Compiler uns wieder einige Dinge schenkt.

Er sieht, dass point in run referenziert wird. Er sieht weiterhin, dass point eine lokale Variable der umschließenden Methode (createRunnable) ist. Er stattet daher das MyRunnable automatisch mit einem Point-Attribut aus – und generiert einen Konstruktor, der eben dieses Attribut initialisiert. Und bei der Erzeugung des MyRunnables wird die point-Referenz eben an diesen Konstruktor übergeben.

Hier die Ausgaben von Members.print:

class **ApplicationWithLocalClass$1MyRunnable** extends Object Constructors

**ApplicationWithLocalClass$1MyRunnable**(

ApplicationWithLocalClass this$0, Point val$point)

Fields

final ApplicationWithLocalClass **this$0**

private final Point **val$point**

Methods

public void **run**()

Das MyRunnable hat zwei Attribute, die automatisch vom Compiler generiert werden – und einen Konstruktor, der zwei Argumente verlangt und aufgrund dieser Argumente die Attribute initialisiert.

Die point-Referenz der createRunnable-Methode wird also in das erzeugte

MyRunnable hineinkopiert – und somit "gerettet".

Dies funktioniert aber nur dann, wenn die lokale Variable, deren Inhalt gerettet werden soll, final oder "effektiv" final ist. "Effektiv" final heißt, dass der Compiler den Quellcode auch dann übersetzen würde, wenn sie explizit als final gekennzeichnet wäre.

Nach der Initialisierung der Variablen darf also keine Zuweisung mehr an diese Variable erfolgen – weder an die lokale Variable noch an das generierte Attribut.

Sofern die Variable eine Referenzvariable ist, ist es natürlich ohne weiteres möglich, den Zustand des durch diese Variable referenzierten Objekts zu ändern.

Aufgrund folgenden Zeilen:

Point point = new Point(1, 2); point = new Point(1, 2)

ist point nicht mehr effektiv final.

Und auch aufgrund der folgenden Zeilen:

public void **run**() {

point = new Point(1, 2) System.out.println(point);

}

ist point nicht mehr effektiv final. Folgende Zeilen aber sind völlig okay:

final Point point = new Point(1, 2); point.x = 3;

point.y = 4;

Das ganze funktioniert auch bei anonymen Klassen:

public class **ApplicationWithAnonymousClass** {

public static void **main**(String[] args) { new ApplicationWithAnonymousClass();

}

public **ApplicationWithAnonymousClass**() {

final Runnable runnable = createRunnable(); runnable.run(); Members.print(runnable.getClass());

}

public Runnable **createRunnable**() { final Point point = new Point(1, 2); return new Runnable() {

@Override

public void run() { System.out.println(point);

}

};

}

}

Die Ausgaben:

java.awt.Point[x=1,y=2]

class **ApplicationWithAnonymousClass$1** extends Object Constructors

**ApplicationWithAnonymousClass$1**( ApplicationWithAnonymousClass this$0, Point val$point)

Fields

final ApplicationWithAnonymousClass **this$0**

private final Point **val$point**

Methods

public void **run**()

Und das ganze funktioniert schließlich auch bei Lambda-Klassen:

package **appl**;

import java.awt.Point; import jn.util.Members;

public class **ApplicationWithLambda** {

public static void **main**(String[] args) { new ApplicationWithLambda();

}

public **ApplicationWithLambda**() {

final Runnable runnable = createRunnableLambda(); runnable.run(); Members.print(runnable.getClass());

}

public Runnable **createRunnableLambda**() {

final Point point = new Point(1, 2); return () -> System.out.println(point);

}

}

Die Ausgaben:

java.awt.Point[x=1,y=2]

class **ApplicationWithLambda$$Lambda$1/26117480** extends Object Constructors

private **ApplicationWithLambda$$Lambda$1/26117480**(Point arg0)

Fields

private final Point **arg$1**

Methods

private static Runnable **get$Lambda**(Point arg0) public void **run**()

Man beachte, dass hier dem generierten Konstruktor nur ein einziges Argument übergeben wird (denn Application.this resp. this wird in Lambda-Implementierung nicht angesprochen).

Man beachte schließlich folgende Zeile:

return () -> System.out.println(point);

Hier wird ein "ausgewachsenes" Objekt erzeugt – ein Objekt einer Klasse, die das Interface Runnable implementiert. Dass es sich bei diesem Lambda-Ausdruck um ein Runnable handeln muss, erkennt der Compiler am Return-Typ der createRunnableLambda-Methode.

## Beispiel: Runnable

Im folgenden werden zwei simple, funktional gleichwertige Multithreading-Anwendungen vorgestellt. Die erste ist traditionell gestrickt, die zweite benutzt Closures.

Wir definieren eine statische Memberklasse MyRunnable, die das Interface Runnable implementiert. Die Klasse hat drei Attribute, die über den Konstruktor initialisiert werden. Diese Attribute steuern das Verhalten der run-Methode dieser Klasse.

In der main-Methode werden zwei Threads erzeugt und gestartet, welche zwei verschieden konfigurierte MyRunnables verwenden:

package appl;

import jn.util.Members;

public class **ApplicationWithoutClosure** {

static class **MyRunnable** implements Runnable { private final String **indent**;

private final int **loopCount**; private final int **sleepTime**;

public **MyRunnable**(String indent, int loopCount, int sleepTime) { this.indent = indent;

this.loopCount = loopCount; this.sleepTime = sleepTime;

}

@Override

public void **run**() {

for(int i = 0; i < this.loopCount; i++) { try {

Thread.sleep(this.sleepTime); System.out.println(this.indent + i);

}

catch(InterruptedException e) { throw new RuntimeException(e);

}

}

}

}

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

final Runnable runnable0 = new MyRunnable("", 5, 400); final Runnable runnable1 = new MyRunnable("\t", 3, 1000); final Thread t0 = new Thread(runnable0);

final Thread t1 = new Thread(runnable1); t0.start();

t1.start();

try { t0.join();

t1.join();

}

catch (InterruptedException e) { throw new RuntimeException(e);

}

System.out.println(Members.toString(runnable1.getClass()));

}

}

Die Ausgaben von Members.print:

class **ApplicationWithoutClosure$MyRunnable** extends Object Constructors

public **ApplicationWithoutClosure$MyRunnable**( String indent, int loopCount, int sleepTime)

Fields

private final String **indent** private final int **loopCount** private final int **sleepTime**

Methods

public void **run**()

Die zweite Applikation lässt die beiden Runnables von einer Methode createRunable erzeugen – in Form von Lambda-Klassen. Die Methode hat drei Parameter – Parameter, die vom Compiler automatisch in das jeweils erzeugte Runnable hineinkopiert werden. Die run-Methode dieser Lambda-Klasse kann somit direkt die Parameter der createRunnable-Methode nutzen.

Die main-Methode erzeugt auch hier wieder zwei Threads, von denen jeder durch ein eigenes Runnable gesteuert werden:

package **appl**;

import jn.util.Members;

public class **ApplicationWithClosure** {

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

final Runnable runnable0 = createRunnable("", 5, 400); final Runnable runnable1 = createRunnable("\t", 3, 1000); final Thread t0 = new Thread(runnable0);

final Thread t1 = new Thread(runnable1); t0.start();

t1.start(); try {

t0.join();

t1.join();

}

catch (InterruptedException e) { throw new RuntimeException(e);

}

System.out.println(Members.toString(runnable0.getClass()));

}

static Runnable **createRunnable**(

String indent, int loopCount, int sleepTime) { return () -> {

for(int i = 0; i < loopCount; i++) { try {

Thread.sleep(sleepTime); System.out.println(indent + i);

}

catch(InterruptedException e) { throw new RuntimeException(e);

}

}

};

}

}

Die Augaben von Members.print zeigen, dass die Werte der drei Parameter der createRunnable-Methode in Attributen des jeweils erzeugten Runnables gerettet werden:

class **ApplicationWithClosure$$Lambda$1/654582261** extends Object Constructors

private **ApplicationWithClosure$$Lambda$1/654582261**(

int arg0, int arg1, String arg2)

Fields

private final int **arg$1** private final int **arg$2** private final String **arg$3**

Methods

private static Runnable **get$Lambda**( int arg0, int arg1, String arg2)

public void **run**()

## Beispiel: Operators

Wir kommen hier auf unsere GUI-Anwendung zurück – auf das Problem der Code- Duplikation. Die bisherige Lösung enthielt bekanntlich zwei fast gleiche Methoden onPlus und onMinus. Wir refaktorieren nun ein wenig…

Wir benutzen folgendes funktionales Interface:

package **appl**;

@FunctionalInterface

public interface **BinaryOperator** {

public abstract int **apply**(int x, int y);

}

In der registerListeners-Methode der MathFrame-Klasse erzeugen wird zwei BinaryOperator-Objekte – in Form von Lambdas. Aus den beiden Methoden onPlus und onMinus-Methoden wird eine einzige: onCalc. Diese ist mit einem BinaryOperator parametrisiert und ruft zur Berechnung des Resultats einfach die apply-Methode des jeweils übergebenen Operators auf. Beim Plus-Button wird ein ActionListener registriert, dessen actionPerformed-Methode die onCalc-Methode mit dem Plus- Operator aufruft – beim Minus-Butten ein ActionListener, dessen actionPerformed- Methode die onCalc-Methode mit einem Minus-Operator aufruft:

package **appl**;

// ...

public class **MathFrame** extends GridPane {

// ...

private void **registerListeners**() {

final BinaryOperator plus = (x, y) -> x + y; final BinaryOperator minus = (x, y) -> x - y;

this.buttonPlus.addActionListener(

event -> MathFrame.this.onCalc(plus)); this.buttonMinus.addActionListener(

event -> MathFrame.this.onCalc(minus)); Members.print(this.buttonPlus.getOnAction().getClass());

}

private void **onCalc**(BinaryOperator operator) { try {

final int x = Integer.parseInt(this.textFieldX.getText()); final int y = Integer.parseInt(this.textFieldY.getText()); final int result = operator.apply(x, y); this.textFieldResult.setText(String.valueOf(result));

}

catch (NumberFormatException e) { this.textFieldResult.setText("Error");

}

}

// ...

}

Was umfasst die Closure der beiden erzeugten ActionListener-Objekte? Die Antwort liefert die Ausgabe von Members.print (für den beim Plus-Button registrierten ActionListener):

class **MathFrame$$Lambda$96/2043192447** extends Object Constructors

private **MathFrame$$Lambda$96/2043192447**(

MathFrame arg0, BinaryOperator arg1)

Fields

private final MathFrame **arg$1**

private final BinaryOperator **arg$2**

Methods

private static ActionListener **get$Lambda**( MathFrame arg0, BinaryOperator arg1)

public void **actionPerformed**(Event arg0)

Neben dem "Outer".this-Zeiger auf den MathFrame ist uns die Referenz auf den Operator geschenkt worden (die lokale plus- resp. minus-Referenz ist also in ein Attribut der ActionListener-Klasse hineinkopiert worden).

## Das final-Problem

In Methoden von lokalen, anonymen und Lambda-Klassen können wir bekanntlich auf lokale Variablen oder Parameter der umschließenden Methode zugreifen – aber nur dann, wenn diese Elemente final oder effektiv final sind.

Das bereitet zuweilen Probleme.

Zur Demonstration solcher Probleme verwenden wir wieder folgendes Interface:

@FunctionalInterface

public interface **Consumer**<T> {

public abstract void **accept**(T value);

}

Wir verwenden zusätzlich folgende Klasse:

public class **Sequence**<T> { private final T[] **elements**;

public **Sequence**(T... elements) { this.elements = elements.clone();

}

public void **forEach**(Consumer<T> consumer) { for (T element : this.elements)

consumer.accept(element);

}

}

Der forEach-Methode dieser Klasse wird ein Consumer übergeben. Sie iteriert über alle Elemente der Sequenz und ruft für jedes Element die accept-Methode des Consumers auf, wobei das aktuelle Element als Parameter übergeben wird.

Wir wollen nun alle Elemente einer Sequence ausgeben:

static void **demo1**() {

final Sequence<String> colors =

new Sequence<>("red", "green", "blue"); colors.forEach(color -> {

System.out.println(color);

});

}

Kein Problem.

Nun soll die Ausgabe aber etwas erweitert werden – zusätzlich zu der Farbe soll auch noch der Index der Farbe ausgegeben werden:

1: red

2: green

3: blue

Der erste Versuch scheitert:

static void **demo2**() {

final Sequence<String> colors =

new Sequence<>("red", "green", "blue"); int n = 0;

colors.forEach(color -> { n++; // illegal

System.out.println(n + ": " + color);

});

}

Die Variable n, auf die in der accept-Methode des Consumers zugegriffen wird, ist nicht effektiv final.

Wir können aber eine kleine Utility-Klasse bauen:

package **util**;

public class **Var**<T> { public T **value**; public **Var**(T value) {

this.value = value;

}

}

Und statt einer int-Variablen eine Var<Integer>-Variablen nutzen:

static void **demo3**() {

final Sequence<String> colors =

new Sequence<>("red", "green", "blue"); Var<Integer> n = new Var<>(0); colors.forEach(color -> {

n.value++;

System.out.println(n.value + ": " + color);

});

}

Die Variable n ist nun effektiv final (sie könnte auch explizit mit final deklariert sein).

## andThen

Im Java 8-Paket java.util.function sind eine Reihe von funktionale Interfaces definiert, die allesamt eine Methode namens andThen besitzen. Die Implementierungen dieser Methoden sind auf den ersten Blick recht unverständlich.

Hier z.B. die andThen-Methode des Interfaces Function:

default <V> Function<T, V> **andThen**(

Function<? super R, ? extends V> after) { Objects.requireNonNull(after);

return (T t) -> after.apply(apply(t));

}

Das folgende kleine Beispiel soll den Sinn und die Implementierung dieser Methoden verdeutlichen – und ein wichtige Konstruktions-Prinzip funktionaler Programmierung erläutern.

Wir wollen folgende Zeilen schreiben können:

Worker w1 = () -> System.out.println("w1 works..."); Worker w2 = () -> System.out.println("w2 works..."); Worker w3 = w1.andThen(w2);

w3.work();

Die Ausführung der ersten drei Zeilen soll noch keinerlei Ausgaben bewirken. Erst die Ausführung der vierten Zeile soll folgende zwei Zeilen ausgeben:

w1 works… w2 works…

Zunächst ein Objektdiagramm:

*Worker*

: Combiner

first

second

*Worker*

*Worker*

work

dThen

an

andThen

rk

wo

<init>

andThen

rk

wo

Betrachten wir die ersten beiden Zeilen des obigen Programmfragments:

Worker w1 = () -> System.out.println("w1 works..."); Worker w2 = () -> System.out.println("w2 works...");

Worker muss offensichtlich ein funktionales Interface sein (sonst könnten wir w1 und w2 keine Lambda-Ausdrücke zuweisen). Das Interface muss eine abstrakte Methode besitzen, die parameterlos ist und nichts liefert. Ihr Name kann schallUndRauch sein – besser aber wahrscheinlich: work (ein Arbeiter arbeitet).

@FunctionalInterface interface **Worker** {

public abstract void **work**();

}

Betrachten wir dann die dritte Zeile des obigen Fragments:

Worker w3 = w1.andThen(w2);

Auf einen Worker muss die andThen-Methode aufgerufen werden können. Dieser wird ein Worker übergeben. Und sie muss einen Worker zurückliefern.

Die andThen-Methode muss also als default-Methode im Worker-Interface definiert sein:

@FunctionalInterface interface **Worker** {

public abstract void **work**();

public default Worker **andThen**(Worker other) {

// ???

}

}

Was könnte andThen liefern? Einen Combiner, der mit zwei Referenzen ausgestattet wird: die eine Referenz (first) zeigt auf denjenigen Worker, auf den andThen aufgerufen wird; die zweite Referenz (second) zeigt auf den Worker, der an andThen übergeben wird. Diese beiden Referenzen werden über den Konstruktor initialisiert.

Ein Combiner muss natürlich selbst wiederum ein Worker sein (denn andThen muss einen Worker zurückliefern).

Die work-Methode der Combiner-Klasse wird dann einfach die work-Methode auf den ersten Worker und dann die work-Methode auf den zweiten Worker aufrufen.

class **Combiner** implements Worker {

final Worker **first**; final Worker **second**;

public **Combiner**(Worker first, Worker second) { this.first = first;

this.second = second;

}

@Override

public void **work**() { this.first.work(); this.second.work();

}

}

Hier nun das vollständige Worker-Interface:

@FunctionalInterface interface **Worker** {

public abstract void **work**();

public default Worker **andThen**(Worker other) { return new Combiner(this, other)

}

}

Wir haben zwei einfache Funktionalitäten, die in Form von Objekten implementiert sind, zu einer "höherwertigen" Funktionalität verbunden (die "höhere" Funktionalität wird durch den Combiner repräsentiert). Natürlich könnten wir die Sache weitertreiben: da der Combiner seinerseits ebenfalls ein Worker ist, könnte er mit einem weiteren Worker zu einem noch "höherwertigen" Combiner verbunden werden etc.

Die ist ein grundlegendes Konzept funktionaler Programmierung: elementare Funktionalitäten zu höherwertigen Funktionalitäten "zusammenzustecken".

Im folgenden werden wir die obige Implementierung schrittweise "vereinfachen" – wobei die Anwendung (siehe das einführende Programmfragment) von diesen Änderungen unberührt bleiben wird).

#### Variante 2

Die Klasse Combiner wurde in der ersten Lösung als Top-Level-Klasse definiert. Da sie aber nur in der work-Methode des Worker-Interfaces benutzt wird, könnten wir sie auch also lokale Klasse eben dieser work-Methode implementieren:

@FunctionalInterface interface **Worker** {

public abstract void **work**();

public default **Worker** andThen(Worker other) {

class **Combiner** implements Worker { final Worker **second**;

public **Combiner**(Worker second) { this.second = second;

}

@Override

public void **work**() { Worker.this.work(); this.second.work();

}

}

return new **Combiner**(other);

}

}

Warum kommen wir hier mit einem einzigen Attribut (second) aus? Und warum hat der Konstruktor nur einen einzigen Parameter?

Jedes Objekt einer lokalen Klasse hat bekanntlich automatisch eine Referenz auf dasjenige Objekt, dessen Methode das Objekt der lokalen Klasse erzeugt hat – den "Outer".this-Zeiger. Im vorliegenden Falle wurde die andThen-Methode auf w1 aufgerufen – also hat das Combiner-Objekt automatisch eine Referenz eben auf diesen Erzeuger (w1). Diese Referenz kann als Worker.this angesprochen werden (siehe die work-Methode der lokalen Klasse).

Tatsächlich also (man prüfe dies mittels Members.print!) hat die lokale Combiner- Klasse zwei Attribute – und einen Konstruktor, dem zwei Parameter übergeben werden.

#### Variante 3

Was passiert mit dem an die andThen-Methode des Worker-Interfaces übergebenen Parameter (other)? Er wird an den Konstruktor vom Combiner übergeben und in einer Instanzvariablen (second) gespeichert. Auf diese Instanzvariable wird dann in der andThen-Methode des Combiners zugegriffen: this.second.work().

Eine lokale Klasse kann bekanntlich aber auch auf lokale Variablen oder Parameter der umschließenden Methode zugreifen. Die Combiner-Klasse hat also auch direkten Zugriff auf den an die andThen-Methode des Worker-Interfaces übergebenen Parameter (Stichwort Closure). Also können wir den Quellcode des Worker-Interfaces vereinfachen:

@FunctionalInterface interface **Worker** {

public abstract void **work**();

public default Worker **andThen**(Worker other) { class **Combiner** implements Worker {

@Override

public void **work**() { Worker.this.work(); other.work();

}

}

return new Combiner();

}

}

In der work-Methode heißt es nun: other.work().

Tatsächlich hat die Combiner-Klasse weiterhin zwei Worker-Attribute und einen Konstruktor, der beide Attribute initialisiert (man prüfe dies mittels Members.print!).

#### Variante 4

Da die Klasse Combiner nur an einer einzigen Stelle instanziiert wird, können wir sie auch durch eine anonyme Klasse ersetzten:

@FunctionalInterface interface **Worker** {

public abstract void **work**();

public default Worker **andThen**(Worker other) { return new **Worker**() {

@Override

public void **work**() { Worker.this.work(); other.work();

}

};

}

}

Die anonyme Klasse hat weiterhin zwei Worker-Attribute und einen Konstruktor, der beide Attribute initialisiert. Technisch ist also wiederum alles beim Alten geblieben.

#### Variante 5

Da die anonyme Klasse ein funktionales Interface (Worker) implementiert, können wir sie auch durch eine Lambda-Implementierung ersetzen:

@FunctionalInterface

interface **Worker** {

public abstract void **work**();

public default **Worker** andThen(Worker other) { return () -> {

Worker.this.work(); other.work();

};

}

}

Hinweis: Statt Worker.this.work() können wird auch schreiben: this.work() oder noch kürzer: work(). (Zur Erinnerung: bei Lambdas sind "Outer".this und this identisch.):

interface **Worker** {

public abstract void **work**();

public default **Worker** andThen(Worker other) { return () -> { work(); other.work(); };

}

}

Man vergleiche diese Zeilen mit der anfangs vorgestellten andThen des Function- Interfaces!

Und man zeige mittels Members.print, dass alles beim Alten geblieben ist…

**Consumer.andThen und Function.andThen** Abschließend zwei weitere Beispiele von andThen. Wir benötigen folgende Importe:

import java.util.function.Consumer; import java.util.function.Function;

Ein Consumer-Beispiel:

Consumer<String> c1 =

s -> System.out.println("c1 consumes(" + s + ")"); Consumer<String> c2 =

s -> System.out.println("c2 consumes(" + s + ")"); Consumer<String> c3 = c1.andThen(c2); c3.accept("Wine");

Die Ausgaben:

c1 consumes(Wine) c2 consumes(Wine)

Man analysiere auch die Ausgaben von Members.print(c3.getClass()). Ein Function-Beispiel:

Function<String,Integer> f1 = s -> s.length(); Function<Integer, Double> f2 = i -> Math.sqrt(i); Function<String, Double> f3 = f1.andThen(f2); System.out.println(f3.apply("blue"));

Die Ausgaben:

2.0

Man analysiere auch die Ausgaben von Members.print(f3.getClass()).

# Funktionale Programmierung

In diesem Kapitel führen wir wesentliche funktionale Operationen ein.

Wir benutzen dabei eine kleine selbst gebaute Klasse Sequence<T>. Es handelt sich dabei um eine Klasse, deren Objekte immutable sind.

Wir werden sehen, dass sich die funktionalen Operationen auch verbinden lassen zu einer Folge solcher Operationen, die dann ihrerseits eine "höherwertige" Funktionalität repräsentiert.

Bei der Entwicklung des Beispiels werden wir sukzessive auch die Interfaces aus

java.util.function vorstellen: Consumer, Function, Predicate etc.

Dabei aber werden wird sehen, dass unserer Implementierungs-Ansatz alles andere als performant sein wird – Grund, uns einen alternativen Ansatz auszudenken.

Einen Ansatz nämlich, der pfeilgrad zu dem in Java 8 eingeführten sog. Stream-Konzept führt…

### Übersicht

* Eine Sequence-Klasse
* forEach, filter, map, reduce, collect, find und match.
* Verknüpfung dieser Operationen
* Eine einfache Implementierung des Stream-Konzepts
* Ein einfaches "real world"-Beispiel: Kunden und Konten.

### Funktionale Interfaces der Standardbibliothek

Hier bereits ein kleiner Überblick über die funktionalen Interfaces von Java 8:

**Runnable**

void run()

R

**Supplier**

R get()

T R

**Function**

R apply(T)

T0 T1

**BiFunction**

R apply(T0, T1)

R

T

**Consumer**

void accept(T)

T T

**UnaryOperator**

T apply(T)

T T

**BinaryOperator**

T apply(T, T)

T

T boolean

**Predicate**

boolean test(T)

(Das Interface Runnable gab's natürlich schon immer…)

## Start

Sequence ist eine Klasse, deren Objekte immutable sind.

Klassen, deren Objekte immutable sind, spielen in der funktionalen Programmierung eine wichtige Rolle. Sie haben einige nicht zu unterschätzende Vorteile: wir müssen uns über den Zustand solcher Objekte keinen Kopf zerbrechen – er ist ja unveränderlich; wir können solche Objekte ohne jeden Synchronisations-Aufwand in Multithreading- Systemen verwenden etc.

Hier die erste Version der Sequence<T>-Klasse:

package **appl**;

import java.util.Arrays; import java.util.Objects;

public class **Sequence**<T> implements Iterable<T> { private final T[] **elements**;

@SafeVarargs

public **Sequence**(T... elements) { Objects.requireNonNull(elements); for (T element : elements)

Objects.requireNonNull(element);

this.elements = Arrays.copyOf(elements, elements.length);

}

public int **size**() {

return this.elements.length;

}

public T **get**(int index) {

if (index < 0 || index >= this.size()) throw new IndexOutOfBoundsException();

return this.elements[index];

}

@Override

public String **toString**() {

return this.getClass().getName() + " " + Arrays.toString(elements);

}

@Override

public int **hashCode**() {

return 31 + Arrays.hashCode(elements);

}

@Override

public boolean **equals**(Object obj) {

if (this == obj) return true;

if (obj == null) return false;

if (getClass() != obj.getClass()) return false;

final Sequence<T> other = (Sequence<T>) obj; return Arrays.equals(elements, other.elements);

}

public int **indexOf**(T element) { Objects.requireNonNull(element);

for (int i = 0; i < this.size(); i++) if (this.get(i).equals(element))

return i; return -1;

}

public boolean **contains**(T element) { return this.indexOf(element) >= 0;

}

@Override

public Iterator<T> **iterator**() { return new **Iterator**<T>() {

int **currentIndex** = 0; @Override

public boolean **hasNext**() {

return this.currentIndex < Sequence.this.size();

}

@Override

public T **next**() {

if (!this.hasNext())

throw new NoSuchElementException();

final T element = Sequence.this.get(this.currentIndex); this.currentIndex++;

return element;

}

};

}

}

* + - Die Klasse implementiert Iterable (und implementiert daher eine iterator- Methode, die einen Iterator liefert)
    - Dem Konstruktor wird ein Java-Array von T-Objekten übergeben (in Form von Varargs).
    - Die size-Methode liefert die Anzahl der Elemente zurück.
    - Die get-Methode ermöglicht den indizierten Zugriff auf die Elemente einer

Sequence.

* + - Die toString-Methode liefert eine String-förmige Repräsentation des Sequence- Zustands.
    - Die equals / hashcode-Methoden erlauben den Vergleich von Sequence- Objekten und die Benutzung solcher Objekte als Schlüssel von Maps.
    - Die indexOf-Methode liefert die Position eines Elements (oder -1).
    - Via contains lässt sich ermitteln, ob ein bestimmtes Element in einer Sequence

enthalten ist.

* + - Die iterator-Methode schließlich liefert einen Iterator, mittels dessen wir über die Elemente einer Sequence iterieren können (wir können daher dann auch die for-each-Schleife nutzen).

Eine drei beispielhafte Anwendungen:

final Sequence<String> colors =

new Sequence<>("red", "green", "blue");

for (int i = 0; i < colors.size(); i++) { final String color = colors.get(i); System.out.println(color.toUpperCase());

}

final Iterator<String> iter = colors.iterator(); while (iter.hasNext()) {

final String color = iter.next(); System.out.println(color.toUpperCase());

}

for (String color : colors) { System.out.println(color.toUpperCase());

}

Die drei Schleifen erzeugen allesamt dieselben Ausgaben:

RED GREEN BLUE

## forEach

Wir möchten eine weitere Iterations-Variante bereitstellen. Die Elemente einer Sequence

sollten wie folgt ausgegeben werden können:

final Sequence<String> colors =

new Sequence<>("red", "green", "blue");

colors.forEachElement(color -> System.out.println(color.toUpperCase()));

Der Methode forEachElement wird ein java.util.function.Consumer übergeben. Hier ein Ausschnitt aus dem Consumer-Interface

package **java.util.function**;

@FunctionalInterface

public interface **Consumer**<T> { void **accept**(T t);

// ...

}

Hier die Erweiterung der Sequence-Klasse:

// ...

public class **Sequence**<T> implements Iterable<T>{

// ...

public void **forEachElement**(Consumer<T> consumer) { for(int i = 0; i < this.size(); i++) {

final T element = this.get(i); consumer.accept(element);

}

}

}

Statt forEachElement könnten wir allerdings auch forEach aufrufen:

colors.forEach(color -> System.out.println(color.toUpperCase()));

forEach nämlich ist eine default-Methode des Iterable-Interfaces, die wie folgt implementiert ist:

package **java.lang**;

public interface **Iterable**<T> { Iterator<T> **iterator**();

default void **forEach**(Consumer<? super T> action) {

Objects.requireNonNull(action); for (T t : this) {

action.accept(t);

}

}

// ...

}

Diese default-Implementierung ist allerdings nicht sehr performant – es wird ein

Iterator genutzt.

Aber wir können diese default-Implementierung durch eine eigene, performantere überschreiben:

// ...

public class **Sequence**<T> implements Iterable<T>{

// ...

@Override

public void **forEach**(Consumer<? super T> consumer) { for (int i = 0; i < this.size(); i++) {

final T element = this.get(i); consumer.accept(element);

}

}

}

By the way: Auch die Klasse ArrayList z.B. überschreibt forEach…

## filter

Mittels einer filter-Methode soll eine neue Sequence erzeugt werden, welches nur diejenigen Elemente der Ursprungs-Sequence enthält, die einer bestimmten Bedingung genügen.

Ein beispielhafter Aufruf der filter-Methode:

final Sequence<String> colors =

new Sequence<>("red", "green", "blue");

final Sequence<String> shortColors = colors.filter(color -> color.length() <= 4);

shortColors soll nur "red" und "blue" enthalten.

An filter wird ein Predicate<T> übergeben. Das Interface ist wie folgt definiert:

package **java.util.function**;

@FunctionalInterface

public interface **Predicate**<T> { boolean **test**(T t);

// ...

}

Hier die Erweiterung der Sequence-Klasse:

// ...

public class **Sequence**<T> implements Iterable<T>{

// ...

public Sequence<T> **filter**(Predicate<T> predicate) { final List<T> list = new ArrayList<>(); this.forEach(element -> {

if(predicate.test(element)) list.add(element);

});

return new Sequence<T>((T[]) list.toArray());

}

}

## map

Mittels des Aufrufs von map soll ebenfalls eine neue Sequence erzeugt werden. Jedes Element der Ursprungs-Sequence soll abgebildet werden auf einen neuen Wert (der möglicherweise anderen Typs ist als der Typ der Ursprungs-Elemente).

Ein beispielhafter Aufruf der map-Methode:

final Sequence<String> colors =

new Sequence<>("red", "green", "blue");

final Sequence<Integer> colorLengths = colors.map(color -> color.length());

colorLengths soll folgende Werte enthalten: 3 5 4

Elements von Typ String werden hier abgebildet auf Elemente des Typs Integer. An map wird eine Function übergeben. Das Interface ist wie folgt definiert:

package **java.util.function**;

@FunctionalInterface

public interface **Function**<T, R> { R **apply**(T t);

// ...

}

Die Sequence-Klasse wird wie folgt erweitert:

// ...

public class **Sequence**<T> implements Iterable<T>{

// ...

public <R> Sequence<R> **map**(Function<T, R> function) { Sequence<R> sequence = new Sequence<>(this.size()); for (int i = 0; i < this.size(); i++) {

T element = this.get(i);

R result = function.apply(element); sequence.elements[i] = result;

}

return sequence;

}

}

## reduce

reduce soll die Elements einer Sequence auf ein einziges Element "reduzieren". Das Resultat muss vom selben Typ wie die Elemente der Sequence. So könnte man die Elemente eines Sequence<Integer> z.B. auf die Summe aller Elemente reduzieren (welche ebenfalls vom Typ Integer ist).

Im folgenden Beispiel sollen die Elemente eines Sequence<String> reduziert werden auf die Konkatenation dieser Elemente:

final Sequence<String> colors =

new Sequence<>("red", "green", "blue");

String allColors = colors.reduce("", (c1, c2) -> c1 + c2);

An allColors soll der Wert "redgreenblue" zugewiesen werden. An reduce wird ein BinaryOperator übergeben.

package **java.util.function**;

@FunctionalInterface

public interface BinaryOperator<T> extends BiFunction<T,T,T> {

// ...

}

BinaryOperator ist offensichtlich eine Spezial-Variante von BiFunction. Dieses Interface ist wie folgt definiert:

package **java.util.function**;

@FunctionalInterface

public interface **BiFunction**<T, U, V> { R apply(T t, U u);

// ...

}

Bei einem BinaryOperator sind im Unterschied zu einer BiFuntion alle Inputs und der Output vom selben Typ.

Die Erweiterung der Sequence-Klasse:

// ...

public class **Sequence**<T> implements Iterable<T>{

// ...

public T **reduce**(T initialValue, BinaryOperator<T> operator) { T result = initialValue;

for(T element : this.elements)

result = operator.apply(result, element); return result;

}

}

Man beachte, dass im Unterschied zu der filter- und der map-Methode die reduce- Methode nicht mehr die forEach-Methode nutzt, sondern die "alte" for-Schleife. Der Grund sollte klar sein…

Wir könnten auch eine reduce-Variante implementieren, die ein Optional<T> liefert:

// ...

public class **Sequence**<T> implements Iterable<T>{

// ...

public Optional<T> **reduce**(BinaryOperator<T> operator) { if (this.size() == 0)

return Optional.empty(); T result = this.get(0);

for (int i = 1; i < this.size(); i++)

result = operator.apply(result, this.get(i)); return Optional.of(result);

}

}

Diese Variante könnte wie folgt genutzt werden:

final Sequence<Integer> numbers1 = new Sequence<>(10, 20, 30); Optional<Integer> sum1 = numbers1.reduce((n1, n2) -> n1 + n2); System.out.println(sum1); System.out.println(sum1.isPresent()); System.out.println(sum1.get());

Hier würde ein Optional<Integer> mit dem Wert 60 geliefert. Wäre die Sequence leer, so würde ein leeres Optional geliefert:

final Sequence<Integer> numbers2 = new Sequence<>(); Optional<Integer> sum2 = numbers2.reduce((n1, n2) -> n1 + n2); System.out.println(sum2); System.out.println(sum2.isPresent());

try { System.out.println(sum2.get());

}

catch(NoSuchElementException e) { System.out.println(e);

}

## collect

Mittels des Aufrufs von collect sollen alle Elemente einer Sequence "eingesammelt" werden können. Dabei soll die Anwendung bestimmen, in welcher Kollektion diese Elemente gesammelt werden sollen. Dann muss die Anwendung natürlich auch bestimmen, wie die Elemente zu der Kollektion hinzugefügt werden sollen.

Die Elemente einer Sequence sollen z.B. in einer ArrayList gesammelt werden:

final Sequence<String> colors =

new Sequence<>("red", "green", "blue");

final List<String> colorList = colors.collect( () -> new ArrayList<String>(),

(list, element) -> list.add(element));

Der collect-Methode werden zwei Lambdas übergeben. Das erste Lambda implementiert das Supplier-Interface. Die get-Methode dieses Interfaces wird aufgerufen, um die Kollektion zu erzeugen (hier: eine ArrayList). Das zweite Lambda implementiert einen BiConsumer: der accept-Methode dieses Interfaces wird die Kollektion übergeben, die der Supplier geliefert hat, und das jeweils hinzuzufügende Element. Das erste Lambda wird nur einmal aufgerufen; das zweite Lambda für jedes Element der Sequence.

Die collect-Methode liefert schließlich diejenige Kollektion zurück, die der Supplier

erzeugt hat.

Hier die beiden relevanten Interfaces:

package **java.util.function**;

@FunctionalInterface

public interface **Supplier**<T> { T **get**();

// ...

}

package **java.util.function**;

@FunctionalInterface

public interface **BiConsumer**<T, U> { void **accept**(T t, U u);

// ...

}

Die Erweiterung der Sequence-Klasse:

ackage **appl**;

// ...

public class **Sequence**<T> implements Iterable<T> {

// ...

public <R> R **collect**(

Supplier<R> creator, BiConsumer<R, T> accumulator) {

R collection = creator.get(); for (T element : this.elements)

accumulator.accept(collection, element); return collection;

}

}

Häufig möchte man die Elemente einer Sequence in einer List-förmigen Struktur sammeln – für solche Zwecke sollte es eine Convenience-Methode geben (eine parameterlose collect-Methode):

final Sequence<String> colors =

new Sequence<>("red", "green", "blue"); final List<String> colorList = colors.collect();

Die hier genutzte parameterlose collect-Methode kann einfach auf die parametrisierte zurückgeführt werden:

// ...

public class **Sequence**<T> implements Iterable<T> {

// ...

public List<T> **collect**() { return this.collect(

() -> new ArrayList<T>(),

(list, element) -> list.add(element));

}

}

## find

Wir möchten ermitteln, ob in einer Sequence ein Element enthalten ist, welches einer bestimmten Bedingung genügt. Die Bedingung wird in Form eines Predicates übergeben. Wenn ja, dann soll das erste gefundene Element, das der Bedingung genügt, in Form eines Optionals zurückgeliefert werden; ansonsten soll ein leeres Optional geliefert werden.

Der folgende Code sollte ein Optional mit dem Wert 20 liefern:

final Sequence<Integer> numbers = new Sequence<>(10, 20, 30); Optional<Integer> optional1 = numbers.find(n -> n >= 15);

Hier sollte ein leeres Optional geliefert werden:

Optional<Integer> optional2 = numbers.find(n -> n >= 200); System.out.println(optional2);

Die Erweiterung der Sequence-Klasse:

// ...

public class **Sequence**<T> implements Iterable<T> {

// ...

public Optional<T> **find**(Predicate<T> predicate) { for(T element : this)

if (predicate.test(element)) return Optional.of(element);

return Optional.empty();

}

}

## match

match gibt's in drei Varianten: anyMatch, allMatch und noneMath. Auch an diese

match –Varianten wird ein Predicate übergeben. Sei folgende Sequence gegeben:

final Sequence<Integer> numbers = new Sequence<>(10, 20, 30);

Ein Beispiel zu anyMatch:

System.out.println(numbers.anyMatch(n -> n > 20)); System.out.println(numbers.anyMatch(n -> n > 200));

Der erste Aufruf sollte true, der zweite false liefern. Ein Beispiel zu allMatch:

System.out.println(numbers.allMatch(n -> n >= 10)); System.out.println(numbers.allMatch(n -> n >= 20));

Der erste Aufruf sollte true, der zweite false liefern. Ein Beispiel zu noneMatch:

System.out.println(numbers.noneMatch(n -> n >= 100)); System.out.println(numbers.noneMatch(n -> n >= 30));

Und auch hier sollte der erste Aufruf true, der zweite false liefern. Die Erweiterung der Sequence-Klasse:

// ...

public class **Sequence**<T> implements Iterable<T> {

// ...

public boolean **anyMatch**(Predicate<T> predicate) { for (T element : this)

if (predicate.test(element)) return true;

return false;

}

public boolean **allMatch**(Predicate<T> predicate) { for (T element : this)

if (!predicate.test(element)) return false;

return true;

}

public boolean **noneMatch**(Predicate<T> predicate) { for (T element : this)

if (predicate.test(element)) return false;

return true;

}

}

## Combination von funktionalen Operationen

Wir können nun auch eine Reihe von Transformationen implementieren:

final Sequence<String> s1 = new Sequence<>("red", "green", "blue"); final Sequence<String> s2 = s1.filter(str -> str.length() >= 4); final Sequence<Integer> s3 = s2.map(str -> str.length()); s3.forEach(len -> System.out.println(len));

* + - s1 enthält die Werte "red", "green" und "blue".
    - s2 enthält die Werte "green" und "blue".
    - s3 enthält die Werte 5 und 4.
    - Und diese Werte (5 und 4) werden schließlich ausgegeben.

Wir können auch auf die Definition der Variablen s1, s2 etc. ganz verzichten und die Operationen kaskadierend aufrufen:

new Sequence<>("red", "green", "blue")

.filter(str -> str.length() >= 4)

.map(str -> str.length())

.forEach(len -> System.out.println(len));

Diese Schreibweise wird auch als "fluent programming" bezeichnet.

forEach, filter, map, reduce, collect, find und match sind grundlegende funktionale Operationen. In unserem Sequence-Beispiel lieferten einige dieser Operationen (map, filter) wieder eine Sequence zurück – eine Sequence, auf die wiederum eine der Operationen aufgerufen werden kann. Solche Operationen werden als "intermediate operations" bezeichnet. Andere Operationen liefern entweder nichts (forEach) zurück, oder aber einen Resultat eines anderen Typs (reduce, collect, find, match). Solche Operationen werden als "terminal operations" bezeichnet – weil an sie kein weiterer "Anschluss" mehr möglich ist.

Die oben entwickelten Methoden filter und map (die intermediate operations) sind aber offensichtlich alles andere als performant und vor allem alles andere als speicherschonend: denn jede dieser Operationen erzeugt wieder eine neue Sequence (die im Falle von map sogar genauso groß ist wie die ursprüngliche Sequence).

Wir benötigen offensichtlich ein alternatives Konzept. Dieses Konzept wird im folgenden Kapitel vorgestellt. Es handelt sich um ein Konzept, welches in Java 8 unter dem Namen "Streams" eingeführt wurde (das Konzept selbst ist natürlich nicht erst seit Java 8 bekannt).

# Streams

Streams ermöglichen die Kombination "einfacher" Funktionalitäten zu "höheren" Funktionalitäten.

Streams können genutzt werden, um Pipelines zu bauen. Eine Pipeline bezieht die zu verarbeitenden Daten aus irgendeiner Quelle und schickt diese Daten dann an verschiedenen Stationen entlang, an denen sie verarbeitet werden können. Die Verarbeitung wird beschrieben wird durch Functions, Predicates etc. Sie kann sequentiell oder möglicherweise auch parallel erfolgen. Am Ende der Pipeline steht ein Resultat, welches nicht mehr als Eingabe für eine weitere Station genutzt werden kann.

Anstatt eine komplette Verarbeitung prozedural mittels Schleifen und Abfragen zu programmieren, müssen bei der Verwendung von Streams nur noch einzelne Funktionalitäten "ausprogrammiert" werden – die eigentliche Kontrolle über die Verarbeitung liegt bei den Streams. Somit wird der "prozedurale" Stil von einem "deklarativen" Stil abgelöst.

Die zentrale Klasse des neuen API ist Stream – sorry: nicht die zentrale Klasse, sondern das zentrale Interface. Der Entwickler muss die eigentlichen Implementierungsklassen überhaupt nicht kennen.

### Übersicht

* + Einfache Stream-basierte Anwendungen
  + peek-Operation
  + Eine einfache Implementierung des Stream-Konzepts

## Grundlagen

Ein erster Stream:

static void **demo1**() {

final Stream<String> s1 = Stream.of("red", "green", "blue"); final Stream<String> s2 = s1.filter(str -> str.length() >= 4); final Stream<Integer> s3 = s2.map(str -> str.length()); s3.forEach(len -> System.out.println(len));

}

Die Ausgben:

5

4

Das Schaubild soll den logischen Ablauf demonstrieren (der tatsächliche Ablauf sieht anders aus – hierzu später…):

**Stream<String>**

"red", "green", "blue"

filter s -> s.length() >= 4

**Stream<String>**

"green", "blue"

map

s -> s.length()

**Stream<Integer>**

5, 4

Mittels der statischen Stream.of-Methode wird ein Stream von Strings erzeugt. s1

repräsentiert einen Stream mit drei Werten: "red", "green" und "blue".

Auf diesen Stream (s1) wird die filter-Methode aufgerufen. Diese erzeugt einen neuen Stream (s2). Der filter-Methode wird ein Predicate<String> übergeben. Dieses Predicate ist nur mit solchen Strings zufrieden, die mindestens vier Zeichen enthalten. An dieses Predicate werden alle Werte von s1 übergeben. Die filter- Station lässt somit nur die Werte "green" und "blue" durch – s2 repräsentiert also die Werte "green" und "blue".

Auf s2 wird nun map aufgerufen. An map wird eine Function<String, Integer> übergeben. Alle Werte, welche die s2-Station verlassen, werden der apply-Methode dieser Function übergeben. map erzeugt dann einen weiteren Stream (einen Stream<Integer>), welcher die Resultate dieser apply-Aufrufe repräsentiert.

Und die Elemente dieser letzten Station (s3) werden dann an die accept-Methode des an forEach übergebenen Consumers übergeben. Die accept-Methode dieses Consumers gibt jeden an ihr übergebenen Wert auf der Konsole aus.

Der folgende Stream verhält sich genauso wie der erste:

static void **demo2**() { Stream.of("red", "green", "blue")

.filter(str -> str.length() >= 4)

.map(str -> str.length())

.forEach(len -> System.out.println(len));

}

Anstatt den jeweils neuen Stream an eine Variablen (s1, s2, …) zu binden, können die Stream-Methoden aus kaskadierend aufgerufen werden. Wir benötigen dann überhaupt keine Variablen mehr…

Ein weiterer Stream:

static void **demo3**() {

final Stream<Integer> s1 = Stream.of(10, 11, 12, 13); final Stream<Integer> s2 = s1.map(n -> n \* 3);

final Stream<Integer> s3 = s2.filter(n -> n % 2 == 0); final Integer sum = s3.reduce(0, (x, y) -> x + y); System.out.println(sum);

}

Die Ausgaben:

66

Der reduce-Methode wird ein Anfangswert (hier: 0) und ein "Akkumulator"-Operator übergeben. Durch die fortgesetzte Anwendung des binären Plus-Operators wir hier die Summe aller von s3 repräsentierten Werte berechnet.

Die reduce-Methode operiert wie folgt (Pseudocode):

**reduce** (start, binaryOperator) { result = start;

foreach value in Stream

result = binaryOperator.apply(result, value) return result;

}

Der Ablauf:

**Stream<Integer>**

10, 11, 12, 13

map n -> n \* 3

**Stream<Integer>**

30, 33, 36, 39

filter n -> n % 2 == 0

**Stream<Integer>**

30, 36

reduce 0, (x, y) -> x + y

**Integer**

66

Derselbe Stream noch einmal – ohne Variablen:

static void **demo4**() {

final int sum = Stream.of(10, 11, 12, 13)

.map(n -> n \* 3)

.filter(n -> n % 2 == 0)

.reduce(0, (x, y) -> x + y); System.out.println(sum);

}

Hinweis: Aufgrund der Schaubilder könnte man meinen, dass die Stationen des Streams als Output jeweils eine neue Collection erzeugen, welche dann als Input in die nächste Station eingeht. Dies ist nicht(!) der Fall – wäre es der Fall, dann wären Streams absolut inperformant. (Die tatsächliche Funktionsweise wird später verdeutlicht werden…)

## Peek

Mittels der peek-Methode kann die Verarbeitung an den verschiedenen Stationen beobachtet werden – wobei peek natürlich selbst wiederum eine Station darstellt.

Dieser Methode wird ein Consumer übergeben, dessen accept-Methode dann das jeweils von der Station verarbeitete Element übergeben wird.

Hier eine Anwendung:

Stream.of("red", "green", "blue")

.peek(str -> System.out.println(str))

.filter(str -> str.length() >= 4)

.peek(str -> System.out.println("\t" + str))

.map(str -> str.length())

.forEach(len -> System.out.println("\t\t" + len));

Die Ausgaben:

red green

green

5

blue

blue

4

Wir erkennen, dass das zweite (das dritte) Element des anfänglichen Streams offenbar erst dann verarbeitet wird, wenn des erste (das zweite) Elements bereits komplett verarbeitet wurde. Für die Zwischenergebnisse werden also keinerlei Collections angelegt.

## Eine SimpleStream-Implementierung

In diesem Abschnitt werden die zentralen Konzepte des Stream-APIs von Java 8 auf einfache Weise nachgebaut: wir entwickeln einen SimpleStream. Dabei werden die Nachteile, die der Sequence-Implementierung anhaften, vermieden werden.

SimpleStream soll wie folgt genutzt werden können:

import util.SimpleStream;

SimpleStream.of("red", "green", "blue")

.filter(str -> str.length() >= 4)

.map(str -> str.length())

.forEach(len -> System.out.println(len));

Hier sollen 5 und 4 ausgegeben werden. Ein anderes Beispiel:

final int sum = SimpleStream.of(10, 11, 12, 13)

.map(n -> n \* 3)

.filter(n -> n % 2 == 0)

.reduce(0, (x, y) -> x + y); System.out.println(sum);

Hier sollte 66 ausgegeben werden.

SimpleStream ist ein Interface, von welchem eine abstrakte Klasse AbstractStream abgeleitet wird. Von letzterer werden dann die instanziierbaren Klassen StartStream, FilterStream und MapStream abgeleitet:

*AbstractStream<P,T>*

*SimpleStream<T>*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | |  | | | |
|  |  | | |  | | |  |
| StartStream<T> | |  | FilterStream<T> | |  | MapStream<P, T> | |

Die Zeilen:

SimpleStream.of("red", "green", "blue")

.filter(str -> str.length() >= 4)

.map(str -> str.length())

.forEach(len -> System.out.println(len));

resultieren in dem Aufbau der folgenden Objekte:



*SimpleStream*

SimpleStream.of

: StartStream

*SimpleStream*

: FilterStream

*Predicate*

*SimpleStream*

: MapStream

*Function*

ply

ap

t

ge

uce

red

Each

for

ap

m

r

filte

st

te

t

ge

uce

red

Each

for

ap

m

r

filte

t

ge

uce

red

Each

for

ap

m

r

filte

blue

green

red

Der Aufruf von SimpleStream.of("red", "green", "blue") erzeugt ein Objekt vom Typ StartStream. Diesem Objekt werden die Parameter der of-Methode übergeben. Ein StartStream kennte also die Start-Elemente.

Wird auf diesen StartStream die filter-Methode aufgerufen, so erzeugt diese filter-Methode ein Objekt vom Typ FilterStream. Diesem wird das an filter übergebene Predicate weitergereicht. Ein FilterStream besitzt also ein Predicate.

Wird auf den FilterStream die Methode map aufgerufen, erzeugt die map-Methode ein Objekt vom Typ MapStream. Die an map übergebene Function wird an den MapStream weitergereicht, so dass der MapStream also eine Function kennt.

Jedes erzeugte Stream-Objekt enthält eine Referenz auf seinen Vorgänger. Die erzeugten Stream-Knoten werden also von "unten" nach "oben" miteinander verkettet.

Noch ist nichts passiert (genauer: noch ist kein einziges der an die of-Methode übergebenen Elemente verarbeitet worden). Insbesondere sollte deutlich geworden sein, dass die filter-Methode eben nicht(!) filtert – und die map-Methode nicht(!) mappt (und eben darin liegt der wesentliche Unterschied zu der in den letzten Abschnitten entwickelten Sequence).

Erst der Aufruf einer terminalen Operation – hier: forEach – setzt die ganze Maschinerie in Bewegung:

forEach wird eine Schleife implementieren. In jedem Durchlauf wird die get-Methode desjenigen Stream-Objekts aufgerufen, auf welches auch forEach aufgerufen wurde. Diese get-Methode wird die get-Methode des Vorgängers (des FilterStreams) aufrufen, diese die get-Methode des StartStreams. Die get-Methode des StartStreams liefert jeweils das nächste Element seiner Daten zurück. Dieses wird von der get-Methode des FilterStreams an das zum FilterStream gehörige Predicate übergeben. Ist das Predicate mit dem Element unzufrieden, wird erneut die get- Methode der Vorgängers aufgerufen. Ansonsten wird das gefilterte Elemente von get zurückgeliefert. Es erreicht die get-Methode des MapStreams. Diese übergibt das Element der zum MapStream gehörigen Function. Der Wert, der dann von dieser Function geliefert wird, wird von forEach an den an forEach übergebenen Consumer weitergereicht. Dann ruft forEach erneut die get-Methode auf den MapStream auf – etc.

Bei diesem Verfahren rauscht zu einem gegebenen Zeitpunkt offensichtlich stets nur ein einziges Element durch die "Pipeline" – es werden keine zusätzlichen Datenstrukturen benötigt. Das Verfahren ist performant und benötigt keine zusätzlichen Speicherflächen.

**Das Interface SimpleStream**

SimpleStream ist mit einem Typ parametrisiert – mit dem Typ der Elemente, die von der

get-Methode eines Stream-Knotens geliefert werden.

Es enthält eine statische of-Methode, welche einen StartStream erzeugt, Und es spezifiziert die Methoden filter, map, forEach und reduce:

package **util**;

import java.util.function.Consumer; import java.util.function.Function; import java.util.function.Predicate;

public interface **SimpleStream**<T> {

public static <T> **SimpleStream**<T> of(T... elements) { return new StartStream<T>(elements);

}

public abstract SimpleStream<T> **filter**(Predicate<T> predicate); public abstract <R> SimpleStream<R> **map**(Function<T, R> function); public abstract void **forEach**(Consumer<T> consumer);

public abstract T **reduce**(T initialValue, BinaryOperator<T> accumulator);

}

**Die abstrakte Klasse AbstractStream**

Die Klasse AbstractStream ist mit zwei Typ-Parametern ausgestattet. T ist der Resultat-Typ (der Typ derjenigen Elemente, die die get-Methode liefert); P ist der Typ derjenigen Elemente, die der Vorgänger (der (P)revious-Knoten) liefert.

AbstractNode definiert eine previous-Referenz, die auf den Vorgänger-Knoten zeigt – und einen Konstruktor, der diese Referenz initialisiert.

Die get-Methode ist abstrakt spezifiziert – sie wird in den instanziierbaren Klassen jeweils unterschiedlich implementiert sein.

Die Intermediate-Methoden filter und map erzeugen (und liefern) jeweils einen neuen

FilterStream resp. MapStream.

Die terminalen Methoden forEach und reduce besorgen sich vom Vorgänger jeweils den nächsten Wert und verarbeiten diesen.

Die Schleifen, die sie implementieren, terminieren jeweils dann, wenn die in ihr aufgerufen get-Methode den Wert null liefert (d.h.: wir gehen davon aus, das in den an die of-Methode übergebenen Werte keine null-Referenzen enthalten sind!).

package **util**;

import java.util.function.Consumer; import java.util.function.Function; import java.util.function.Predicate;

abstract class **AbstractStream**<P, T> implements SimpleStream<T> { final AbstractStream<?, P> **previous**; **AbstractStream**(AbstractStream<?, P> previous) {

this.previous = previous;

}

abstract T **get**(); @Override

public SimpleStream<T> **filter**(Predicate<T> predicate) {

return new FilterStream<T>(this, predicate);

}

@Override

public <R> SimpleStream<R> **map**(Function<T, R> function) { return new MapStream<T, R>(this, function);

}

@Override

public void **forEach**(Consumer<T> consumer) { T element = this.get();

while(element != null) { consumer.accept(element); element = this.get();

}

}

@Override

public T **reduce**(T initalValue, BinaryOperator<T> accumulator) { T result = initalValue;

T element = this.get(); while(element != null) {

result = accumulator.apply(result, element); element = this.get();

}

return result;

}

}

**Die Klasse StartStream**

Ein StartStream speichert die seinem Konstruktor übergebenen Elemente. Er besitzt ein Attribut currentIndex, um in der get-Methode jeweils das nächste Element ermitteln und zurückliefern zu können (currentIndex wird dann in get natürlich inkrementiert). get liefert null, wenn alle Elemente geliefert wurden:

package **util**;

class **StartStream**<T> extends AbstractStream<T, T> { T[] **elements**;

int **currentIndex** = 0;

**StartStream**(T[] elements) { super(null); this.elements = elements;

}

@Override T **get**() {

if (this.currentIndex == this.elements.length) return null;

T element = this.elements[this.currentIndex]; this.currentIndex++;

return element;

}

}

**Die Klasse FilterStream**

Ein FilterStream kennt ein Predicate.

In der get-Methode wird die get-Methode des Vorgängers aufgerufen. Das Element, welches diese get-Methode liefert, wird an die test-Methode des Predicates übergeben. Ist diese mit dem Element zufrieden, wird eben dieses Element zurückgeliefert. Ansonsten wird erneut die get-Methode des Vorgängers aufgerufen – solange, bis das Predicate mit dem Element zufrieden ist resp. die get-Methode des Vorgängers null liefert. Von get wird dann entweder null zurückgeliefert oder aber ein Element, welches den test bestanden hat.

package **util**;

import java.util.function.Predicate;

class **FilterStream**<T> extends AbstractStream<T, T> { final Predicate<T> **predicate**;

**FilterStream**(AbstractStream<?, T> previous, Predicate<T> predicate) { super(previous);

this.predicate = predicate;

}

@Override T **get**() {

T element = this.previous.get();

while(element != null && ! this.predicate.test(element))

element = this.previous.get(); return element;

}

}

**Die Klasse MapStream**

Ein MapStream kennt eine Function.

Die get-Methode ruft die get-Methode des Vorgänger-Knotens auf. Liefert diese null, so wird null zurückgeliefert. Ansonsten wird das vom Vorgänger gelieferte Element an die apply-Methode der Function übergeben und das Resultat eben dieses apply- Aufrufs zurückgeliefert:

package **util**;

import java.util.function.Function;

class **MapStream**<P, T> extends AbstractStream<P, T> { final Function<P, T> **function**;

**MapStream**(AbstractStream<?, P> previous, Function<P, T> function) { super(previous);

this.function = function;

}

@Override T **get**() {

P element = this.previous.get(); if (element == null)

return null;

return this.function.apply(element);

}

}

Man beachte die Verwendung der Typ-Parameter P und T!

Eine schöne Erweiterung besteht in der Einführung einer peek-Operation und einer entsprechenden PeekStream-Klasse, die es erlauben, einem Stream bei seiner Arbeit "zuzuschauen". Hier ein Verwendungs-Beispiel:

SimpleStream.of("red", "green", "blue")

.peek(str -> System.out.println(str))

.filter(str -> str.length() >= 4)

.peek(str -> System.out.println("\t" + str))

.map(str -> str.length())

.forEach(len -> System.out.println("\t\t" + len));

# Generics - Erweiterungen

### Voraussetzungen

* + Kenntnis der generischen ArrayList und HashMap-Klassen

Dieses Kapitel beleuchtet insbesondere die Themen Ko- und Kontravarianz.

### Übersicht

* + Ko- und Kontravarianz: technische Aspekte
  + Beispiel: ein Getränke-Dealer
  + Beispiel: eine statische deal-Methode
  + andThen-Methoden der Interfaces Consumer und Function
  + Class<T>
  + casten, ohne zu casten…

In den ersten der folgenden Abschnitten wird eine einfache Vererbungshierarchie verwendet:

Wine

Beer

Drink

Für unsere Zwecke reicht eine triviale Implementierung dieser Klassen:

RedWine

WhiteWine

public class **Drink** {

}

public class **Beer** extends Drink {

}

public class **Wine** extends Drink {

}

public class **WhiteWine** extends Wine {

}

public class **RedWine** extends Wine {

}

## Kovarianz und Kontravarianz

Bekanntlich kann eine Referenzvariable des Typs Wine auf Objekte des Klasse Wine und auf all solche Objekte zeigen, deren Klassen (direkt oder indirekt) von Wine abgeleitet sind – auch Objekte der Klassen WhiteWine und RedWine z.B. können als Winess betrachtet werden – nicht aber Drink-Objekte:

Wine w;

w = new Drink(); // illegal w = new Wine();

w = new RedWine();

(Natürlich können über eine Variable vom Typ Wine nur diejenigen Elemente (Attribute oder Methoden) der Objekte angesprochen werden, die eben in dieser Klasse definiert sind.)

Ganz anders sieht die Sache aus, wenn wir Listen von Wines betrachten. Wir definieren eine Variable vom Typ ArrayList<Wine> - und probieren einige Zuweisungen an diese Variable:

ArrayList<Wine> list;

list = new ArrayList<Drink>(); list = new ArrayList<Wine>(); list = new ArrayList<RedWine>();

// illegal

// Illegal

list.add(new RedWine());

Dass die erste Zuweisung vom Compiler zurückgewiesen wird, ist klar (wir könnten ja einer Variablen vom Typ Wine auch kein Objekt vom Typ Drink zuweisen).

Dass aber auch die dritte Zuweisung als illegal zurückgewiesen wird, ist auf den ersten Blick unverständlich. Einer Variable vom Typ Wine könnte doch ein RedWine zugewiesen werden – warum kann dann einer ArrayList<Wine> keine ArrayList<RedWine> zugewiesen werden? Das Konzept der Polymorphie scheint hier nicht mehr zu greifen…

Warum also wir die dritte Zuweisung vom Compiler zurückgewiesen?

Angenommen einmal, sie würde nicht zurückgewiesen. Dann könnte anschließend die add-Methode auf die ArrayList<RedWine> aufgerufen werden. Wir könnten bei diesem Aufruf ein RedWine übergeben. Aber: wir könnten auch ein Wine übergeben – weil die Variable, über welche die add-Methode aufgerufen wird, vom Typ ArrayList<Wine> ist. Dann wären wir mit folgender katastrophalen Situation konfrontiert: eine ArrayList, die für RedWine-Objekte erzeugt wurde, könnte auch Wine-Objekte enthalten.

Betrachten wir eine weitere Situation:

ArrayList<Drink> list;

list = new ArrayList<Beer>(); list.add(new RedWine());

// illegal

Wäre es möglich, einer Variable des Typs ArrayList<Drink> eine für Beer bestimmte ArrayList zuzuweisen, könnte die add-Methode (die ja Drink verlangt!) auch ein RedWine übergeben werden. Das Resultat wäre eine ArrayList<Beer>, die RedWine enthält – auch dies eine Katastrophe.

Wir können nun aber die list-Variable ein wenig anders definieren:

ArrayList<? extends Wine> list;

list = new ArrayList<Drink>(); // illegal list = new ArrayList<Wine>();

list = new ArrayList<RedWine>();

list.add(new Wine()); // illegal list.size();

list.get(0);

Statt die list-Variable als ArrayList<Wine> zu definieren, definieren wir sie als ArrayList<? extends Wine>. Dann ist natürlich weiterhin die erste der obigen Zuweisungen illegal. Aber die dritte Zuweisung funktioniert nun.

Dann aber erleben wir eine neue Überraschung: der Aufruf der add-Methode wird als illegal zurückgewiesen – und zwar unabhängig davon, ob beim Aufruf dieser Methode Drink, Wine oder RedWine übergeben wird. Der Aufruf der size- und der get-Methode ist dagegen problemlos.

Nach welchem Kriterium unterscheidet der Compiler legale und illegale Aufrufe? (Der Compiler kennt sich natürlich nicht in der Semantik von Getränken aus…)

Der add-Methode wird ein Parameter des in Frage stehenden generischen Typs übergeben; die get-Methode liefert eine Referenz dieses Typs zurück; und die size- Methode hat mit diesem Typ überhaupt nichts zu schaffen. Und was für add, get und size gilt, gilt für alle Methoden: Methoden, denen ein Parameter des generischen Typs übergeben wird, können nun nicht mehr aufgerufen werden – der Aufruf aller anderen Methoden ist problemlos.

Angenommen wir definieren folgende Methode:

static void **print**(ArrayList<Drink> drinks) { for (Drink d : drinks)

System.out.println(d);

}

Dieser Methode könnten nur Objekte der Klasse ArrayList<Drink> übergeben; Argumente des Typs ArrayList<Wine> oder ArrayList<Beer> würde der Compiler zurückweisen:

print(new ArrayList<Drink>());

print(new ArrayList<Wine>()); // illegal print(new ArrayList<Beer>()); // illegal

Definieren wir die print-Methode aber ein klein wenig anders:

static void **print**(ArrayList<? extends Drink> drinks) { for (Drink d : drinks)

System.out.println(d);

}

Dann wäre alle der drei obigen Aufrufe legal.

In der ersten Version der print-Methode könnten wir auf die übergebene ArrayList die add-Methode aufrufen – wir könnten "schreiben"; in der zweiten Version aber wäre der Aufruf von add illegal – wir könnten nur "lesen".

Man sagt auch: eine Variable des Typs ArrayList<? extends Drink> erlaubt die kovariante Nutzung (schreiben: nein; lesen: ja).

Variablen können auch derart definiert werden, dass sie kontravariant genutzt werden können:

ArrayList<? super Wine> list;

list = new ArrayList<Object>(); list = new ArrayList<Drink>(); list = new ArrayList<Wine>();

list = new ArrayList<RedWine>(); // illegal

list.add(new Wine()); list.add(new RedWine()) list.add(new Drink()); list.size();

Drink d = list.get(0);

// illegal

// illegal

Wir definieren eine Variable vom Typ ArrayList<? super Wine>. Dieser Variablen können all solche ArrayLists zugewiesen werden, deren aktueller Parameter vom Typ Wine oder von einem Typ ist, von welchem Wine direkt oder indirekt abgeleitet ist. Eine ArrayList, deren Typ-Parameter aber von Wine abgeleitet ist, kann an diese Variable nicht zugewiesen werden.

Nun können wir zwar die add-Methode aufrufen (und dieser Wine oder RedWine übergeben – nicht aber Drink!) – die Methode get liefert nun aber nicht mehr Drink zurück, sondern nurmehr Object. Wir können nun zwar "schreiben", aber nicht mehr (vernünftig) "lesen".

Wir könnten aber nun eine fill-Methode schreiben:

static void **fill**(ArrayList<? super Wine> drinks) { drinks.add(new Drink()); // illegal drinks.add(new Beer()); // illegal drinks.add(new Wine());

drinks.add(new WhiteWine()); drinks.add(new RedWine());

}

Diese könnte wie folgt aufgerufen werden:

fill(new ArrayList<RedWine>()); // illegal fill(new ArrayList<Wine>());

fill(new ArrayList<Drink>()); fill(new ArrayList<Object>());

Polymorphie funktioniert also weiterhin – aber nicht mehr auf "naive" Weise.

Wie verhalten sich einfache Java-Arrays bezüglich des Themas Ko- und Kontravarianz? Betrachten wir folgendes – legale! - Programmfragment:

Beer[] beers = new Beer[10]; beers[0] = new Beer(); Drink[] drinks = beers; drinks[1] = new Drink();

In einem Array, der für Beer bestimmt ist, wird nun mittels der letzten Zuweisung ein Drink eingefügt. Mit der Konsequenz, dass erst zur Laufzeit das Problem gemeldet wird: in Form einer ArrayStoreException.

Ein Java-Array kennt zur Laufzeit seinen Komponenten-Typ: hier Beer. Eine ArrayList<T> kennt diesen Typ zur Laufzeit bekanntlich nicht. Anders als eine ArrayList<T> kann ein Array aber eben nicht typsicher (zur Compilationszeit!) verwendet werden. Generische Klassen verhalten sich also komplett anders als Java- Arrays. Beide Konzepte beißen sich.

Dass sich beide Konzepte beißen, erkennt man auch daran, dass kein Array eines generischen Typs erzeugt werden kann:

class **Foo**<T> {

T[] **array** = new T[10]; // illegal

}

Wir können zwar eine Variable des Typs T[] definieren, nicht aber ein Array-Objekt mit dem Komponententyp T erzeugen. Wir müssen also etwas tricksen:

class **Foo**<T> {

T[] **array** = (T[])new Object[10];

}

Weitere Hinweise:

Häufig sieht man Variablen, die wie folgt definiert sind:

ArrayList<?> list;

Dies ist nichts anderes als eine Kurzform von

ArrayList<? extends Object> list;

Einer Variablen vom Typ ArrayList<?> können wir also alle möglichen ArrayLists zuweisen; wir können über eine solche Variable aber nur solche Methoden aufrufen, die keinen Parameter des generischen Typs besitzen.

Das Schlüsselwort extends kann nicht nur zur Definition eines kovarianten Typs verwendet werden, sondern auch zur Einschränkung eines generischen Typs:

class **DrinkContainer**<T extends Drink> extends ArrayList <T > {

}

Ein DrinkContainer kann dann nur mit einem solchen aktuellen Typ-Parameter erzeugt werden, der Drink ist oder von Drink abgeleitet ist:

DrinkContainer<Wine> wines = new DrinkContainer<>(); DrinkContainer<Petrol> petrols = new DrinkContainer<>(); // illegal

Hier hat extends also eine ganz andere Bedeutung…

## Ein armer Dealer

Ein Getränke-Dealer bringt Getränke-Produzenten und -Konsumenten zusammen.

Die Dealer-Klasse enthält also eine deal-Methode, der ein Supplier und ein Consumer übergeben wird. Sie besorgt sich vom Supplier ein Getränk und reicht dieses dann an den Consumer weiter:

public class **Dealer**<T extends Drink> { public void **deal**(

Supplier<T> producer, Consumer<T> consumer) {

final T drink = producer.get(); consumer.accept(drink);

}

}

Dealer ist mit einem T parametrisiert (wobei T auf Drink eingeschränkt ist – einen

Dealer<Petrol> wird's also nicht geben können).

Daher ist der erste Parameter vom Typ Supplier<T> und der zweite vom Typ

Consumer<T>.

Wir erzeugen drei Supplier: der erste produziert Objekte der Klasse Drink, der zweite Objekte der Klasse Wine und der dritte Objekte der Klasse RedWine:

final Supplier<Drink> drinkProducer = () -> new Drink(); final Supplier<Wine> wineProducer = () -> new Wine();

final Supplier<RedWine> redWineProducer = () -> new RedWine();

Dann erzeugen wir drei Consumer: der erste Consumer verlangt Drink-Objekte, der zweite Wine-Objekte und der dritte RedWine-Objekte:

final Consumer<Drink> drinkConsumer = d -> System.out.println(d); final Consumer<Wine> wineConsumer = d -> System.out.println(d); final Consumer<RedWine> redWineConsumer = d -> System.out.println(d);

Schließlich besorgen wir uns einen Dealer vom Typ Dealer<Wine>:

final Dealer<Wine> wineDealer = new Dealer<>();

Mit welchen Suppliern und mit welchen Consumern kann dieser Dealer<Wine>

zusammenarbeiten?

Die folgende Zeile ist unproblematisch:

wineDealer.deal(wineProducer, wineConsumer);

Die beiden folgenden Zeilen werden vom Compiler zurecht als illegal zurückgewiesen:

wineDealer.deal(drinkProducer, wineConsumer); // illegal wineDealer.deal(wineProducer, redWineConsumer); // illegal

Der drinkProducer garantiert nur, dass er irgendeinen Drink produziert (dabei könnte es sich um Wine handeln, muss aber nicht). Also kann ein Dealer<Wine> nicht mit einem Supplier<Drink> zusammenarbeiten.

Ein redWineConsumer ist anspruchsvoll: er verlangt RedWine. Er wird also sich also nicht mit Wine zufriedengeben. Ein Dealer<Wine> allerdings garantiert nur Wine.

Der Compiler hat mit seiner Zurückweisung dieser Zeilen also vollkommen Recht. Aber auch folgende Zeilen werden vom Compiler zurückgewiesen:

wineDealer.deal(redWineProducer, wineConsumer); // should be ok! wineDealer.deal(wineProducer, drinkConsumer); // should be ok!

Ein Dealer<Wine> sollte aber doch mit einem Supplier<RedWine> benutzen können – er würde dann von einem solchen Produzenten sogar "mehr" bekommen, als er verlangt.

Und ein Dealer<Wine> sollte auch mit einem Supplier<Drink> klarkommen – verlangt dieser doch "weniger" als der Dealer auf garantierte Weise liefert.

Und auch folgende Zeile sollte möglich sein (auch diese wird aber als illegal zurückgewiesen):

wineDealer.deal(redWineProducer, drinkConsumer); // should be ok!

Der hier vorgestellte Dealer<Wine> kann also nur mit genau einem Supplier-Typ (Supplier<Wine>) und genau einem Consumer-Typ (Consumer<Wine>) klarkommen. Er lässt Möglichkeiten ungenutzt und wird also nicht allzu reich werden...

## Ein reicher Dealer

Ein Dealer<Wine> sollte nicht nur mit einem Supplier<Wine>, sondern auch mit einem Supplier<RedWine> zusammenarbeiten können; und nicht nur mit einem Consumer<Wine>, sondern auch mit einem Consumer<Drink>:

deal(s, c)

: Dealer<Wine>

: Supplier<RedWine>

: Consumer<Wine>

: Supplier<Wine>

: Consumer<Drink>

Wir ändern die Parameter-Typen der deal-Methode: statt Supplier<T> verwenden wir den Typ Supplier<? extends T>; und statt Consumer<T> verwenden wir Consumer<? super T>:

public class **Dealer**<T extends Drink> {

public void **deal**(

Supplier<? extends T> producer, Consumer<? super T> consumer) {

final T drink = producer.get(); consumer.accept(drink);

}

}

Auf producer können nur "Lese"-Methoden aufgerufen werden – und get ist eine solche Lese-Methode; auf consumer können nur "Schreib"-Methoden aufgerufen werden; accept ist eine solche Schreib-Methode. Die Klasse Dealer wird also übersetzt.

Angenommen, wir bauen wieder einen Dealer<Wine>:

final Dealer<Wine> wineDealer = new Dealer<>();

Dann kann deal natürlich weiterhin mit einem Supplier<Wine> und einem

Consumer<Wine> aufgerufen werden:

wineDealer.deal(wineProducer, wineConsumer);

Zusätzlich sind nun aber auch folgende Aufrufe legal:

wineDealer.deal(redWineProducer, wineConsumer); wineDealer.deal(wineProducer, drinkConsumer);

Der Dealer wird nun also reicher werden können… Die folgenden Aufrufe sind natürlich weiter illegal:

wineDealer.deal(drinkProducer, wineConsumer); // illegal wineDealer.deal(wineProducer, redWineConsumer); // illegal

## Eine statische deal-Methode

Im letzten Beispiel musste zunächst ein Dealer erzeugt werden, auf den dann die deal- Methode aufgerufen werden konnte. Wir können natürlich auch eine Dealer-Klasse mit einer statischen deal-Methoden bauen:

package **appl**;

// ...

public class **Dealer** {

public static <T extends Drink> void **deal**( Supplier<? extends T> producer, Consumer<? super T> consumer) {

final T drink = producer.get(); consumer.accept(drink);

}

}

Die statische Methode ist nun mit einem T extends Drink parametrisiert. Der

Supplier- und der Consumer-Parameter sind genauso definiert wie im letzten Beispiel. Seien wieder einige Supplier und Consumer definiert:

final Supplier<Drink> drinkProducer = () -> new Drink(); final Supplier<Wine> wineProducer = () -> new Wine();

final Supplier<RedWine> redWineProducer = () -> new RedWine();

final Consumer<Drink> drinkConsumer = d -> System.out.println(d); final Consumer<Wine> wineConsumer = d -> System.out.println(d);

final Consumer<RedWine> redWineConsumer = d -> System.out.println(d);

Wir können die neue Dealer-Klasse dann wie folgt nutzen (man beachte, dass der aktuelle Typ-Parameter der deal-Methode vor dem Namen der Methode angegeben wird!):

Dealer.<Wine>deal(wineProducer, wineConsumer); Dealer.<RedWine>deal(redWineProducer, wineConsumer); Dealer.<Wine>deal(wineProducer, drinkConsumer); Dealer.<RedWine>deal(redWineProducer, drinkConsumer);

Die aktuellen Typ-Parameter der deal-Methode können aber auch vom Compiler berechnet werden. Wir können also auf die explizite Definition dieser Typ-Parameter verzichten:

Dealer.deal(wineProducer, wineConsumer); Dealer.deal(redWineProducer, wineConsumer); Dealer.deal(wineProducer, drinkConsumer); Dealer.deal(redWineProducer, drinkConsumer);

## Beispiel: andThen

Das Standard-Interface Consumer definiert eine andThen-Methode mit einem Parameter vom Typ Consumer<? super T>:

default Consumer<T> **andThen**(Consumer<? super T> after) { Objects.requireNonNull(after);

return (T t) -> { accept(t); after.accept(t); };

}

Wir erzeugen zwei Consumer:

Consumer<Wine> c1 = w -> System.out.println(w); Consumer<RedWine> c2 = w -> System.out.println(w);

Dann können wir einen dritten Consumer bauen:

Consumer<RedWine> c3 = c2.andThen(c1); c3.accept(new RedWine());

Wäre der Parameter von andThen vom Typ Consumer<T>, so wäre c2.andThen(c1)

illegal.

Das Standard-Interface Function definiert eine andThen-Methode mit einem Parameter des Typs Function<? super R, ? extends V>:

default <V> Function<T, V> **andThen**(

Function<? super R, ? extends V> after) { Objects.requireNonNull(after);

return (T t) -> after.apply(apply(t));

}

Wir erzeugen zwei Function-Objekte:

Function<String,Integer> f1 = s -> s.length(); Function<Number,Double> f2 = n -> Math.sqrt(n.doubleValue());

Diese beiden Functions können nun kombiniert werden zu einer neuen Function:

Function<String,Number> f3 = f1.andThen(f2); System.out.println(f3.apply("hello"));

Wäre der Parameter von andThen vom Typ Function<R, V>, so wäre die Kombination illegal.

## Die generische Nutzung der Klasse Class

Die Klasse Class hat einen Typ-Parameter:

public class **Class**<T> ...

Auch die Klasse Constructor hat einen Type-Parameter:

public class **Constructor**<T> ...

Die Klasse Class hat eine Methode getConstructor:

public class **Class**<T> ... {

// ...

public Constructor<T> **getConstructor**(Class<?>... parameterTypes) { ... }

}

Eine Variable vom Typ Class<?> kann auf alle möglichen Class-Objekte zeigen:

Class<?> cls1 = Wine.class; Class<?> cls2 = Beer.class;

Eine Variable des Typs Class<Wine> kann aber nur auf dasjenige Class-Objekt zeigen, welches die Klasse Wine repräsentiert:

Class<Wine> cls1 = Wine.class;

Class<Wine> cls2 = RedWine.class; // illegal

Eine Variable des Typs Class<? extends Wine> kann sowohl auf das Class-Objekt für Wine als auch auf das Class-Objekt für RedWine zeigen:

Class<? extends Wine> cls1 = Wine.class; Class<? extends Wine> cls2 = RedWine.class;

Wie kann nun der Umstand, dass Class mit einem Typ - mit dem Typ der von dem

Class-Objekt repräsentierten Klasse! - parametrisiert ist, genutzt werden?

Wir bauen eine Mapper-Klasse, deren map-Methode den Inhalt einer Map<String, String> auf ein Objekt beliebigen Typs abbildet. Dabei wird natürlich Reflection benutzt werden. Statt aber den Property-basierten Zugriff zu implementieren (Aufruf von setter- Methoden), greift die map-Methode einfach schreibend auf öffentliche Felder der Objekt- Klasse zu.

Hier die Beispiel-Klasse:

public class **Book** {

public String **isbn**; public String **title**; public Integer **year**; public Double **price**;

@Override

public String **toString**() { ... }

}

Hier zunächst eine Anwendung des Mappers:

public class **Application** {

public static void **main**(String[] args) { Map<String, String> map = new HashMap<>(); map.put("isbn", "1111");

map.put("title", "Pascal");

map.put("year", "1970");

map.put("price", "11.11");

Book book = Mapper.map(Book.class, map); System.out.println(book);

}

}

Die in der Map enthaltenen Werte sind alle vom Typ String. Dem year-Feld eines Book-Objekts aber muss ein int zugewiesen werden; dem price-Feld muss ein double zugewiesen werden.

Wir benötigen also Parser, die Strings in andere Typen konvertieren. Solche Parser werden folgendes Interface implementieren:

package **util**;

@FunctionalInterface

public interface **Parser**<T> {

public abstract T **parse**(String s);

}

Parser ist ein funktionales Interface, welches mit einem Typ T parametrisiert ist. Die parse-Methode bekommt einen String und liefert ein T zurück.

Hier zunächst die vollständige Mapper-Klasse:

package **util**;

import java.lang.reflect.Field; import java.util.HashMap; import java.util.Map;

public class **Mapper** {

private final static Map<Class<?>, Parser<?>> **parsers** = new HashMap<>();

public static <T> void **register**(Class<T> type, Parser<T> parser) { parsers.put(type, parser);

}

**static** {

register(String.class, s -> s); register(Integer.class, s -> Integer.parseInt(s)); register(Double.class, s -> Double.parseDouble(s));

// ...

}

public static <B> B **map**(Class<B> cls, Map<String, String> map) { try {

B obj = cls.getConstructor().newInstance(); map.forEach((k, v) -> assignField(obj, k, v)); return obj;

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

private static void **assignField**(

Object obj, String name, String stringValue) { try {

Field field = obj.getClass().getField(name); Parser<?> parser = parsers.get(field.getType()); if (parser == null)

throw new Exception(

"Parser for " + field.getType() + " not found"); Object value = parser.parse(stringValue); field.set(obj, value);

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

}

Mapper enthält eine statische Map, welche Typen (Class<?>-Objekte) auf Parser<?>

abbildet:

private final static Map<Class<?>, Parser<?>> **parsers** = new HashMap<>();

Der Zugriff auf diese Map ist natürlich nicht typsicher – wir könnten folgenden Eintrag hinzufügen:

register(Double.class, s -> Integer.parseInt(s));

Um eine typsichere Registrierung von Parsern zu ermöglichen, definieren wird folgende Methode:

public static <T> void **register**(Class<T> type, Parser<T> parser) { parsers.put(type, parser);

}

register ist eine generische Methode mit zwei Parametern: Class<T> und Parser<T>. Wird dieser Methode nun z.B. ein Class<Integer> übergeben, so erzwingt der Compiler, dass auch der zweite Parameter vom Typ Parser<Integer> ist.

Wir können nun z.B. folgende Parser registrieren:

register(String.class, s -> s); register(Integer.class, s -> Integer.parseInt(s)); register(Double.class, s -> Double.parseDouble(s));

Da die register-Methode public ist, könnten nun natürlich auch weitere Applikations- spezifische Parser registriert werden.

Auch die map-Methode ist eine generische Methode:

public static <B> B **map**(Class<B> cls, Map<String, String> map) { ... }

B ("Bean") steht für den Typ des zu erzeugenden und zu initialisierenden Objekts (also

z.B. für den Typ Book.class). Wenn dieser Methode eine Class<B>-Referenz übergeben wird, so ist der Rückgabe-Typ von map vom Typ B (z.B. Book).

Die map-Methode erzeugt dann ein Objekt derjenigen Klasse, die vom Class<B>-Objekt repräsentiert wird und liefert dieses zurück:

B obj = cls.getConstructor().newInstance(); map.forEach((k, v) -> assignField(obj, k, v)); return obj;

Der map.forEach-Aufruf sorgt dafür, dass für alle Einträge der an map übergebenen Map die private assignField-Methode aufgerufen wird – wobei dieser das zu initialisierende Objekt, der Schlüssel und der Wert des jeweiligen Map-Eintrags übergeben wird.

private static void **assignField**(

Object obj, String name, String stringValue) {

Field field = obj.getClass().getField(name); Parser<?> parser = parsers.get(field.getType()); Object value = parser.parse(stringValue); field.set(obj, value);

}

Dies assignField-Methode ermittelt aufgrund des Schlüssels das Field-Objekt und aufgrund dieses Field-Objekts dann den Typ des zu initialisierenden Feldes. Für diesen Typ wird der passende Parser ermittelt und dieser dann mit dem Wert des jeweiligen Map-Eintrags aufgerufen. Die parse-Methode liefert dann ein Object, mittels dessen dann via field.set das entsprechende Feld initialisiert wird.

## Vermeidung expliziter Casts

Angenommen, wir füllen eine HashMap<String, Object>:

Map<String, Object> map = new HashMap<>();

map.put("isbn", "1111");

map.put("title", "Pascal");

map.put("year", 1970);

map.put("price", 11.11);

Die Werte der ersten beiden Einträge sind vom Typ String, der Wert des dritten Eintrage ist vom Typ Integer und der Wert des vierten vom Typ Double.

Bekanntlich können wir diese Werte wie folgt ermitteln:

String isbn = (String) map.get("isbn"); String title = (String) map.get("title"); int year = (Integer) map.get("year"); double price = (Double) map.get("price");

Alle Resultate der get-Aufrufe müssen explizit gecastet werden (natürlich könnten wir auch ohne Cast auskommen – wenn alle Zielvariablen vom Typ Object wären).

Wir können aber eine get-Methode schreiben, deren Benutzung die oben benötigten Downcasts vermeidet:

static <T> T get(Map<String, Object> map, String name) { return (T) map.get(name);

}

Diese Methode kann wie folgt genutzt werden:

String isbn = get(map, "isbn"); String title = get(map, "title"); int year = get(map, "year"); double price = get(map, "price");

Der Trick besteht natürlich darin, dass der Typ-Paramater T in der Überschrift der get- Methode nur ein einziges Mal genutzt wird: als Return-Typ der Methode. Und natürlich findet weiterhin ein Downcast statt: dieser ist nun aber in der get-Methode versteckt.

# 10 Enums - Erweiterungen

### Voraussetzungen

* Elementare enum-Definitionen und Verwendung von enums

Enum-Konstanten sind bekanntlich keine einfachen (Zahlen-)Werte, sondern Referenzen auf Objekte. Und eine enum-Definition ist – genauer besehen – eine Klassendefinition.

### Übersicht

* Alte "enums"
* Neue enums
* Basiskalsse von enum-Klassen
* EnumMap.
* Erweiterbarkeit von enums: Attribute, Methoden, Implementierung von Interfaces
* Erweiterungen um statische Elemente
* Serialisierung von enums.

## Eine "alte" Implementierung

Sei folgender enum-Typ gegeben:

enum Color { RED, GREEN, BLUE }

Was verbirgt sich hinter dieser Definition? Wie könnten wir eine gewöhnliche Klasse definieren, die auf dasselbe hinausläuft wir die obige enum-Definition?

Ein Objekt-Diagramm:

values

: Color

"BLUE" 2

: Color

"GREEN" 1

: Color

"RED" 0

Color.RED

Color.GREEN

Color.BLUE

Wir bauen eine Klasse Color mit drei öffentlichen, statischen konstanten Referenzen: RED, GREEN und BLUE. Alle drei Referenzen sind vom Typ Color. Diese Referenz- Variablen werden unmittelbar bei ihrer Definition initialisiert – mit Objekten des Typs Color.

Hier zunächst der komplette Quellcode:

public class **Color** {

public static final Color **RED** = new Color("RED", 0); public static final Color **GREEN** = new Color("GREEN", 1); public static final Color **BLUE** = new Color("BLUE", 2);

public static final Color[] **values** = { RED, GREEN, BLUE }; private final String **name**;

private final int **ordinal**;

private **Color**(String name, int ordinal) { this.name = name;

this.ordinal = ordinal;

}

public final String **name**() { return this.name;

}

public final int **ordinal**() { return this.ordinal;

}

@Override

public String **toString**() { return this.name;

}

public static Color **valueOf**(String name) { for(Color color : values)

if (color.name.equals(name)) return color;

return null;

}

public static Color[] **values**() { return values.clone();

}

}

Bei der Erzeugung eines Color-Objekts wird ein privater(!) Konstruktor genutzt, dem ein "Ordinal-Wert" und ein Name übergeben werden – wobei der Name identisch ist mit dem Namen der Referenz-Variablen.

Der Konstruktor initialisiert aufgrund seiner Parameter zwei Attribute der Klasse: ordinal und name. Diese beiden Attribute können via ordinal() und name() ausgelesen werden.

Zusätzlich trägt der Konstruktor das aktuelle Color-Objekt ein einen Color-Array namens values ein. Dieser Array (genauer: eine Kopie dieses Arrays) wird von values() zurückgeliefert.

Schließlich enthält die Klasse eine statische valueOf-Methode, die aufgrund des Namens eines Color-Objekts eben dieses Color-Objekt liefert.

Hier eine beispielhafte Anwendung der obigen Klasse:

Color red = Color.RED; System.out.println(red);

System.out.println(red.name()); System.out.println(red.ordinal()); System.out.println(Arrays.toString(Color.values())); System.out.println(Color.valueOf("BLUE"));

Die Ausgaben:

RED RED 0

[RED, GREEN, BLUE] BLUE

Enum-Klasse auf diese Weise zu implementieren, erfordert offensichtlich einen großen Schreibaufwand…

## Die "neue" Implementierung

Seit Java 1.5 kann die im letzten Abschnitt implementierte Klasse wie folgt abgekürzt werden:

public enum **Color** {

**RED**, **GREEN**, **BLUE**

}

Aufgrund dieser enum-Definition erzeugt der Compiler eine class-Datei, welche all diejenigen Features beinhaltet, die im letzten Abschnitt explizit definiert wurden.

Hier eine Anwendung der obigen Definition:

Color red = Color.RED; System.out.println(red); System.out.println(red.name()); System.out.println(red.ordinal());

System.out.println(Arrays.toString(Color.values())); System.out.println(Color.valueOf("BLUE"));

Die Ausgaben:

RED RED 0

[RED, GREEN, BLUE] BLUE

Um zu sehen, welche Elemente die vom Compiler generierte Klasse erzeugt, verwenden wird wieder Members.print. Dabei geben wird zunächst die Elemente der Klasse Enum aus (Enum ist die implizite gemeinsame Basisklasse aller enum-Klassen). Dann geben wir die Elemente von Color.class aus:

Members.print(Enum.class); Members.print(Color.class);

Die Ausgaben:

class **Enum** extends Object Constructors

protected **Enum**(String name, int ordinal)

Fields

private final String **name**

private final int **ordinal**

Methods

protected final Object clone() public final int compareTo(Enum arg0)

@Overrides public final boolean equals(Object arg0) protected final void finalize()

public final Class getDeclaringClass() @Overrides public final int hashCode() public final String **name**()

public final int **ordinal**()

private void readObject(ObjectInputStream arg0) private void readObjectNoData()

@Overrides public String **toString**()

public static Enum valueOf(Class cls, String name)

class **Color** extends Enum Constructors

private **Color**(String name, int ordinal)

Fields

public static final Color **BLUE** public static final Color **GREEN** public static final Color **RED**

private static final Color[] **ENUM$VALUES**

Methods

public static Color **valueOf**(String name) public static Color[] **values**()

Die Basisklasse Enum enthält bereits einen Konstruktor, dem ordinal und name übergeben werden. Sie enthält auch die Attribute ordinal und name und die Zugriffsmethoden auf diese Attribute.

Die Klasse Color ist von Enum abgeleitet. Sie enthält einen private Konstruktor, der auf den Konstruktor der Basisklasse abgebildet wird. Sie enthält die drei statischen Referenzen BLUE, GREEN und RED – und einen Array dieser Elemente. Und schließlich enthält sie die statischen Methoden valueOf und values.

## Die Basisklasse Enum

Im folgenden wird die Basisklasse Enum etwas genauer untersucht. Die Interpretation der folgenden Demo-Methoden sei dem Leser / der Leserin überlassen.

static void **demoBasics**() { System.out.println(Enum.class.isAssignableFrom(Color.class)); System.out.println(Color.class.getSuperclass());

final Color e1 = Enum.valueOf(Color.class, "RED"); final Color e2 = Color.valueOf("RED");

final Color e3 = Color.RED; System.out.println(e1 == e2 && e2 == e3);

final Enum<Color> e4 = Enum.valueOf(Color.class, "RED"); final Enum<Color> e5 = Color.valueOf("RED");

final Enum<Color> e6 = Color.RED; System.out.println(e4 == e5 && e5 == e6);

System.out.println(e1 == e4);

}

Die Ausgaben:

true

class java.lang.Enum true

true true

static void **inspectClass**() throws Exception { Class<? extends Enum<?>> cls = Color.class;

System.out.println(cls.getSimpleName()); System.out.println(cls.getSuperclass().getSimpleName());

final Enum<?>[] constants2 = (Enum<?>[]) cls.getMethod("values").invoke(null);

for (final Enum<?> constant : constants2) System.out.print(constant + " ");

System.out.println();

final Enum<?>[] constants1 = cls.getEnumConstants(); for (final Enum<?> constant : constants1)

System.out.print(constant + " "); System.out.println();

for (final Constructor<?> constructor : cls.getDeclaredConstructors()) {

System.out.println(constructor);

}

final Constructor<?> constructor = cls.getDeclaredConstructor(String.class, int.class);

constructor.setAccessible(true);

// constructor.newInstance("abc", 42);

// will throw an exception:

// Cannot reflectively create enum objects

}

Die Ausgaben:

Color Enum

RED GREEN BLUE RED GREEN BLUE

private appl.Color(java.lang.String,int)

static void **inspectConstant**() { Enum<?> constant = Color.BLUE;

System.out.println(constant.getDeclaringClass().getSimpleName()); System.out.println(constant.name()); System.out.println(constant.ordinal());

try {

System.out.println(Enum.valueOf(constant.getClass(), "BLUE"));

}

catch (IllegalArgumentException ex) { System.out.println(ex.getMessage());

}

}

Die Ausgaben:

Color BLUE 2 BLUE

## EnumMap

Wenn enum-Objekte als Schlüssel für Maps fungieren sollen, sollte die Klasse EnumMap

verwendet werden.

Sei z.B. die folgende enum-Definition gegeben:

public enum **Foo** {

**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**

}

Im folgenden wird gezeigt, dass EnumMaps in einem solchen Fall wesentlich performanter sind als z.B. HashMaps. Zur Performance-Messung verwenden wir wieder unsere Ulitity-Klasse PerformanceTest:

// ...

public class **Application** {

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

final Foo[] values = Foo.values(); final int N = values.length;

final EnumMap<Foo,String> enumMap = new EnumMap<>(Foo.class); for(Foo value : values)

enumMap.put(value, value.name());

final HashMap<Foo,String> hashMap = new HashMap<>(); for(Foo value : values)

hashMap.put(value, value.name());

new PerformanceTest<Integer>("Maps",10\_000\_000)

.add("EnumMap", N, () -> { int n = 0;

for(int i = 0; i < N; i++)

n += enumMap.get(values[i]).length(); return n;

})

.add("HashMap", N, () -> { int n = 0;

for(int i = 0; i < N; i++)

n += hashMap.get(values[i]).length(); return n;

})

.test();

}

}

Hier die Messergebnisse:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Maps | | | Loops | | Millis | | | AVG | (nano) |
| EnumMap | | | 10000000 | | 591 | | |  | 59 |
| HashMap | | | 10000000 | | 2116 | | |  | 211 |

Zur Speicherung der Elemente verwendet EnumMap einen einfachen Java-Array. Wird

z.B. die put(key, value) aufgerufen, wird einfach der Ordinal-Wert von key als Index für diesen Array verwendet.

## Implementierung von Interfaces

Enum-Klassen können zwar von keiner Klasse abgeleitet werden (sie sind ja bereits implizit von der Basisklasse Enum abgeleitet), aber sie können – wie jede andere Klasse auch – Interfaces implementieren.

Im folgenden implementieren wir IntBinaryOperators als BinaryOps:

import java.util.function.IntBinaryOperator; public enum **BinaryOp** implements IntBinaryOperator {

**PLUS**("+") {

@Override

public int **applyAsInt**(int x, int y) { return x + y;

}

},

**MINUS**("-") {

@Override

public int **applyAsInt**(int x, int y) { return x - y;

}

};

private final String **symbol**; private **BinaryOp**(String symbol) {

this.symbol = symbol;

}

public String **symbol**() { return this.symbol;

}

@Override

public String **toString**() { return this.symbol;

}

}

Ein BinaryOp enthält neben name und ordinal nun ein weiteres Attribut symbol (und eine Zugriffsmethode symbol()). Dieses Attribut wird über den (privaten!) Konstruktor initialisiert.

Bei der Definition der BinaryOp-Referenzen wird dann das jeweilige Symbol ("+", "-") übergeben.

Zusätzlich wird bei der Definition einer BinaryOp-Referenz jeweils ein Block hinterlegt, in welchem dann die im Interface IntBinaryOperator spezifizierte applyAsInt- Methode implementiert wird.

Eine Anwendung:

BinaryOp binaryOp = BinaryOp.PLUS; System.out.println(binaryOp); System.out.println(binaryOp.applyAsInt(40, 2)); IntBinaryOperator op = binaryOp; System.out.println(op.applyAsInt(40, 2));

Die Ausgaben:

+ 42

42

Eine zweite, alternative Variante:

import java.util.function.IntBinaryOperator; public enum **BinaryOp** implements IntBinaryOperator {

**PLUS**("+", (x, y) -> x + y),

**MINUS**("-", (x, y) -> x - y);

private final String **symbol**; private final IntBinaryOperator **op**;

private **BinaryOp**(String symbol, IntBinaryOperator op) { this.symbol = symbol;

this.op = op;

}

public String **symbol**() { return this.symbol;

}

@Override

public String **toString**() { return this.symbol;

}

@Override

public int **applyAsInt**(int x, int y) { return this.op.applyAsInt(x, y);

}

}

Der Konstruktor verlangt nun neben dem Symbol einen zweiten Parameter: einen IntBinaryOperator. Dieser kann dann bei der Definition der enum-Referenzen in Form eines Lambdas übergeben werden.

Und das Interface IntBinaryOperator wird nun direkt im Rumpf der enum-Klasse definiert – dort wird nun die applyAsInt-Methode des IntBinaryOperator aufgerufen.

Eine Anwendung:

BinaryOp binaryOp = BinaryOp.MINUS; System.out.println(binaryOp); System.out.println(binaryOp.applyAsInt(80, 3)); IntBinaryOperator op = binaryOp; System.out.println(op.applyAsInt(80, 3));

Die Ausgaben:

- 77

77

## Statische Elemente

Im folgenden erweitern wir die Color-Klasse um ein Attribut displayName (RED hat den

displayName "rot", etc.).

Zusätzlich soll eine statische Methode forDisplayName das Color-Objekt für einen vorgegebenen displayName zurückliefern.

public enum **Color** {

**RED**("rot"), **GREEN**("gruen"), **BLUE**("blau"); private final String **displayName**;

private **Color**(String displayName) { this.displayName = displayName;

}

public String **displayName**() { return this.displayName;

}

public static Color **forDisplayName**(String displayName) { ... }

}

Die letzte Methode soll etwa wie folgt aufgerufen werden können:

Color red = Color.forDisplayName("rot");

Wir präsentieren vier Varianten (von denen allerdings die erste nicht funktioniert…):

#### Variante 1

Wir erzeugen eine HashMap, die Strings auf Color-Objekte abbildet: colors. Im Konstruktor von Color versuchen wir, via colors.put(displayName, this) das aktuelle Color-Objekt unter dem Namen displayName zu colors hinzuzufügen:

public enum **Color** {

**RED**("rot"), **GREEN**("gruen"), BLUE("blau");

private static final Map<String, Color> **colors** = new HashMap<>(); private final String **displayName**;

private **Color**(String displayName) { this.displayName = displayName;

colors.put(displayName,this); // illegal

}

public String **displayName**() { return this.displayName;

}

public static Color **forDisplayName**(String displayName) { return colors.get(displayName);

}

}

Warum weist der Compiler den Aufruf von put zurück? Der Konstruktor von Color wird bereits drei Mal aufgerufen, bevor die colors-Variable überhaupt initialisiert wird. Denn die statischen Element werden natürlich in der Reihenfolge ihrer Definition initialisiert. Zunächst also werden die Referenzen RED, GREEN und BLUE initialisiert (wobei jeweils der Konstruktor aufgerufen wird) – und dann erst colors.

Und leider können wird die Definition von colors auch nicht oberhalb der Color- Konstanten vornehmen…

### Color2

Wir bauen einen statischen Block, der erst dann ausgeführt wird, nachdem alle Color- Referenzen und auch colors bereits initialisiert sind. In diesem statischen Block iterieren wird über die Color-Objekte und erzeugen für jedes Color-Objekt einen entsprechenden Map-Eintrag. Im Konstruktor wird nur displayName initialisiert:

public enum **Color** {

**RED**("rot"), **GREEN**("gruen"), **BLUE**("blau");

private static final Map<String, Color> **colors** = new HashMap<>();

**static** {

for (Color value : values()) colors.put(value.displayName, value);

}

private final String **displayName**; private **Color**(String displayName) {

this.displayName = displayName;

}

public static **Color** forDisplayName(String displayName) { return colors.get(displayName);

}

public String **displayName**() {

return this.displayName;

}

}

#### Variante 3

Die Interpretation der folgenden Variante sei dem Leser / der Leserin überlassen:

public enum **Color** {

**RED**("rot"), **GREEN**("gruen"), **BLUE**("blau"); private **static** class Colors {

**static** {

System.out.println("static of Inner");

}

static final Map<String, Color> **map** = new HashMap<>();

}

**static** {

System.out.println("static of Color");

}

private final String **displayName**;

private **Color**(String displayName) { this.displayName = displayName; Colors.map.put(displayName, this);

}

public static Color **forDisplayName**(String displayName) { return Colors.map.get(displayName);

}

public String **displayName**() { return this.displayName;

}

}

#### Variante 3

Auch die Interpretation der letzten Variante sei dem Leser / der Leserin überlassen:

public enum **Color** {

**RED**("rot"), **GREEN**("gruen"), **BLUE**("blau"); private static Map<String, Color4> **colors**; private final String **displayName**;

private **Color**(String displayName) {

this.displayName = displayName;

}

public static Color **forDisplayName**(String displayName) { synchronized (Color.class) {

if (colors == null) {

colors = new HashMap<>(); for (Color color : values())

colors.put(color.displayName(), color);

}

return colors.get(displayName);

}

}

public String **displayName**() { return this.displayName;

}

}

## Serialisierung

Enum-Objekte sind serialisierbar (die Basisklasse Enum ist von Serializable

abgeleitet).

Dabei gibt's ein mögliches Problem: Angenommen, wir serialisieren Color.GREEN. Was passiert bei der Deserialisierung? Entsteht dort neben dem bereits existierenden Color.GREEN-Objekt ein weiteres Color.GREEN-Objekt? Hoffentlich nicht…

Wir serialisieren:

try(ObjectOutputStream oos = new ObjectOutputStream( new FileOutputStream("demo.ser"))) {

oos.writeObject(Color.GREEN);

}

Und deserialisieren:

try(ObjectInputStream ois = new ObjectInputStream( new FileInputStream("demo.ser"))) {

Color c = (Color)ois.readObject(); System.out.println(c == Color.GREEN);

}

Der Referenz-Vergleich von c und Color.GREEN liefert zum Glück true. Das Problem existiert also nicht – was aber nicht selbstverständlich ist. (Hierauf werden wir im Kapitel zum Thema Serialisierung ausführlich eingehen.)

# 11 Reflection - Erweiterungen

### Voraussetzungen

* Elementare Kenntnisse der Klassen Class, Field, Method und Constructor

Im Folgenden geht's um eine Vertiefung des Themas Reflection (dabei wird ein bestimmtes Basiswissen von Reflection vorausgesetzt).

### Übersicht

* Introspection: eine Features-Klasse
* Zugriff auf Felder, Aufruf von Methoden und Konstruktoren
* "Deep-Reflection"
* "Properties"

## Introspektion

Im folgenden wird eine kleine Utility-Klasse namens Inspector vorgestellt, welche alle Elemente einer Klasse ausgibt: alle Felder, alle Konstruktoren und alle Methoden.

Dabei werden sowohl die öffentlichen als auch die nicht öffentlichen Elemente berücksichtigt.

Dabei werden nicht nur die Elemente der in Frage stehenden Klasse ausgegeben, sondern auch alle Elemente der Basisklasse dieser Klasse.

Zudem werden die Basisklasse und die von der Klasse implementierten Interfaces ausgegeben.

Als Demo-Klassen werden zwei Klassen verwendet: eine Klasse Book und eine davon abgeleitete Klasse SchoolBook:

package **appl**;

import java.io.Serializable;

public class **Book** implements Serializable { private String **isbn**;

private String **title**;

private int **year**; public **Book**() { }

public **Book**(String isbn, String title, int year) { ... }

public String **getIsbn**() { ... }

public void **setIsbn**(String isbn) { ... }

public String **getTitle**() { ... }

public void **setTitle**(String title) { ... }

public int **getYear**() { ... }

public void **setYear**(int year) { ... }

@Override

public String **toString**() { ... }

}

package **appl**;

public class **SchoolBook** extends Book { private String **subject**;

public **SchoolBook**() { }

public **SchoolBook**(String isbn, String title, int year, String subject) {

// ...

}

public String **getSubject**() { ... }

public void **setSubject**(String subject) { ... }

@Override

public String **toString**() { ... }

}

Die printClass-Methode der Inspector-Klasse kann wie folgt aufgerufen werden:

Inspector.printClass(SchoolBook.class);

Die Ausgaben:

public **SchoolBook** extends **Book**

interfaces

**Serializable**

fields

private String **isbn** <Book> private String **title** <Book> private int **year** <Book>

private String **subject** <SchoolBook> constructors

public **SchoolBook**()

public **SchoolBook**(String isbn, String title,

int year, String subject)

methods

protected void **finalize**() <Object>

final public void **wait**(long arg0, int arg1) <Object> final public void **wait**() <Object>

final native public void **wait**(long arg0) <Object> public boolean **equals**(Object arg0) <Object> public String **toString**() <Object>

native public int **hashCode**() <Object>

final native public Class **getClass**() <Object> native protected Object **clone**() <Object> final native public void **notify**() <Object> final native public void **notifyAll**() <Object>

native private static void **registerNatives**() <Object> public String **toString**() <Book>

public String **getIsbn**() <Book> public String **getTitle**() <Book> public int **getYear**() <Book>

public void **setIsbn**(String isbn) <Book> public void **setTitle**(String title) <Book>

public void **setYear**(int year) <Book> public String **toString**() <SchoolBook> public String **getSubject**() <SchoolBook>

public void **setSubject**(String subject) <SchoolBook>

Nun zur Implementierung der Klasse Inspector. Die Klasse benötigt folgende imports:

package **util**;

import java.lang.reflect.Constructor; import java.lang.reflect.Field; import java.lang.reflect.Method; import java.lang.reflect.Modifier; import java.lang.reflect.Parameter; import java.util.ArrayList;

import java.util.Collections; import java.util.HashMap; import java.util.List;

import java.util.Map;

public class **Inspector** {

// ...

}

Sie enthält nur eine einzige öffentliche Methode printClass, der ein Class-Objekt übergeben wird (diese ist static – wie alle anderen Methoden der Klasse auch).

Die Methode gibt den Kopf der Klasse aus (Modifizierer, Name der Klasse und Name der Basisklasse). Und ruft dann die Methoden printInterfaces, printFields, printConstructors und printMethods auf:

public static void **printClass**(Class<?> cls) { printModifiers(cls.getModifiers()); print(cls.getSimpleName() + " extends "); println(cls.getSuperclass().getSimpleName()); printInterfaces(cls);

printFields(cls); printConstructors(cls); printMethods(cls);

}

Zur Erzeugung einer lesbaren Ausgabe von Modifizieren existiert eine kleine HashMap:

static Map<Integer, String> **modifiersMap** = new HashMap<>(); static {

modifiersMap.put(Modifier.ABSTRACT, "abstract"); modifiersMap.put(Modifier.FINAL, "final"); modifiersMap.put(Modifier.PUBLIC, "public"); modifiersMap.put(Modifier.PROTECTED, "protected");

modifiersMap.put(Modifier.PRIVATE, "private"); modifiersMap.put(Modifier.STATIC, "static"); modifiersMap.put(Modifier.NATIVE, "native");

// ...

}

Die Methode printInterfaces ermittelt via getInterfaces die Interfaces einer Klasse und gibt jedes dieser Interfaces aus:

static void **printInterfaces**(Class<?> cls) { println("interfaces"); getInterfaces(cls).forEach(

iface -> println("\t\t" + iface.getSimpleName()));

}

Die (Plural-)Methoden printFields, printConstruktors und printMethods ermitteln zunächst jeweils alle Fields, Constructors und Methods (via get…-Methoden) und rufen dann für jedes der zurückgelieferten Elemente jeweils die (Singular-)Methoden printField, printConstructor und printMethod auf:

static void **printFields**(Class<?> cls) { println("fields"); getFields(cls).forEach(

field -> printField(field));

}

static void **printConstructors**(Class<?> cls) { println("constructors"); getConstructors(cls).forEach(

constructor -> printConstructor(constructor));

}

static void **printMethods**(Class<?> cls) { println("methods"); getMethods(cls).forEach(

method -> printMethod(method));

}

Die Methoden printField, printConstructor und printMethod geben jeweils die relevanten Eigenschaften des jeweiligen Elements aus (wobei u.a. die Helper-Methoden printParameters und printModifiers aufgerufen werden):

static void **printField**(Field field) { print("\t\t"); printModifiers(field.getModifiers()); println(field.getType().getSimpleName() + " " +

field.getName() +

" <" + field.getDeclaringClass().getSimpleName() + ">");

}

static void **printConstructor**(Constructor<?> constructor) { print("\t\t"); printModifiers(constructor.getModifiers());

print(constructor.getDeclaringClass().getSimpleName()); printParameters(constructor.getParameters()); println("");

}

static void **printMethod**(Method method) { print("\t\t"); printModifiers(method.getModifiers());

print(method.getReturnType().getSimpleName() + " "); print(method.getName()); printParameters(method.getParameters());

println(" <" + method.getDeclaringClass().getSimpleName() + ">");

}

static void **printParameters**(Parameter[] parameters) { print("(");

for (int i = 0; i < parameters.length; i++) { if (i > 0)

print(", ");

final Parameter parameter = parameters[i]; print(parameter.getType().getSimpleName() + " "); print(parameter.getName());

}

print(")");

}

static void **printModifiers**(int modifiers) { modifiersMap.forEach((k, v) -> {

if ((modifiers & k) != 0) print(v + " ");

});

}

Die Methoden getInterfaces, getFields, getConstructors und getMethods liefern für eine gegebene Klasse alle Elemente des jeweiligen Typs zurück. Dabei sind getInterfaces, getFields und getMethods rekursiv implementiert.

getFields(cls) z.B. ermittelt durch den rekursiven Aufruf zunächst alle Field-Objekte der Basisklasse von cls. Zu diesen Field-Objekten fügt sie dann alle Field-Objekte hinzu, welche in cls definiert sind – sowohl die öffentlichen als auch die nicht- öffentlichen (man beachte, dass die Class-Methode getDeclaredField genutzt wird).

static List<Class<?>> **getInterfaces**(Class<?> cls) { final List<Class<?>> list = new ArrayList<>(); if (cls.getSuperclass() != null)

list.addAll(getInterfaces(cls.getSuperclass())); list.addAll(List.of(cls.getInterfaces()));

return Collections.unmodifiableList(list);

}

static List<Field> **getFields**(Class<?> cls) { final List<Field> list = new ArrayList<>(); if (cls.getSuperclass() != null)

list.addAll(getFields(cls.getSuperclass())); list.addAll(List.of(cls.getDeclaredFields())); return Collections.unmodifiableList(list);

}

static List<Constructor<?>> **getConstructors**(Class<?> cls) { final List<Constructor<?>> list = new ArrayList<>(); list.addAll(List.of(cls.getDeclaredConstructors())); return Collections.unmodifiableList(list);

}

static List<Method> **getMethods**(Class<?> cls) { final List<Method> list = new ArrayList<>(); if (cls.getSuperclass() != null)

list.addAll(getMethods(cls.getSuperclass())); list.addAll(List.of(cls.getDeclaredMethods())); return Collections.unmodifiableList(list);

}

Schließlich werden die eigentlichen Ausgaben durch Aufrufe von zwei kleinen Helper- Methoden erzeugt:

static void **println**(Object msg) { System.out.println(msg);

}

static void **print**(Object msg) { System.out.print(msg);

}

## Set/Get, Invoke, NewInstance

Im folgenden wird gezeigt, wie mittels der Field-Methoden get und set auf Felder lesend resp. schreibend zugegriffen werden kann, wie mittels der Method-Methode invoke eine Methode aufgerufen werden kann und mittels der Constructor-Methode newInstance Objekte erzeugt werden können.

Als Beispiel-Klasse dient eine Klasse Foo:

package **appl**;

import java.io.Serializable; public class **Foo** {

public int **alpha**; public String **beta**;

public **Foo**() { System.out.println("Constructor of C"); this.alpha = 4711;

this.beta = "Hello";

}

public **Foo**(int alpha, String beta) {

System.out.println("Constructor of C (" + alpha + ", " + beta + ")"); this.alpha = alpha;

this.beta = beta;

}

public void **doThis**() {

System.out.println("doThid: " + this.alpha + " " + this.beta);

}

public int **doThat**(int alpha, String beta) { System.out.println("doThat: " + alpha + " " + beta); this.alpha = alpha;

this.beta = beta; return this.alpha;

}

public int **getAlpha**() { return alpha;

}

public String **getBeta**() { return beta;

}

@Override

public String **toString**() {

return this.getClass().getSimpleName() +

" [" + this.alpha + ", " + this.beta + "]";

}

}

Die Class.getField-Methode und die get-Methode der Klasse Field:

static void **demoGet**() throws Exception { final Object obj = new Foo();

final Field alphaField = obj.getClass().getField("alpha"); final Object alphaValue = alphaField.get(obj); System.out.println(alphaValue);

final Field betaField = obj.getClass().getField("beta"); final Object betaValue = betaField.get(obj); System.out.println(betaValue)

}

Die Ausgaben:

Constructor of C 4711

Hello

Die Class.getField-Methode und die set-Methode der Klasse Field:

static void **demoSet**() throws Exception { final Object obj = new Foo();

final Field alphaField = obj.getClass().getField("alpha"); alphaField.set(obj, 5000);

final Field betaField = obj.getClass().getField("beta"); betaField.set(obj, "World");

System.out.println(obj);

}

Die Ausgaben:

Constructor of C Foo [5000, World]

Die Class.getFields-Methode und die get-Methode der Klasse Field:

static void **demoGetAll**() throws Exception { final Object obj = new Foo();

final Field[] fields = obj.getClass().getFields(); for (final Field field : fields) {

final String name = field.getName(); final Object value = field.get(obj); System.out.println(name + " = " + value);

}

}

Die Ausgaben:

Constructor of C alpha = 4711 beta = Hello

Die Class.getMethod-Methode und die invoke-Methode der Klasse Method (Aufruf einer parameterlose Methode)

static void **demoInvokeParamerterlessMethod**() throws Exception { final Object obj = new Foo();

final Method method = obj.getClass().getMethod("doThis"); final Object result = method.invoke(obj); System.out.println(result);

}

Die Ausgaben:

Constructor of C doThid: 4711 Hello null

Die Class.getMethod-Methode und die invoke-Methode der Klasse Method (Aufruf einer parametrisierten Methode):

static void **demoInvokeParamerterizedMethod**() throws Exception { final Object obj = new Foo();

final Method method =

obj.getClass().getMethod("doThat", int.class, String.class); final Object result = method.invoke(obj, 5000, "World"); System.out.println(result); }

Die Ausgaben:

Constructor of C doThat: 5000 World

5000

Aufrufe aller getter-Methode einer Klasse:

static void **demoInvokeAllGetterMethods**() throws Exception { final Object obj = new Foo();

for (final Method method : obj.getClass().getMethods()) { final String name = method.getName();

if (name.equals("get")) continue;

if (!name.startsWith("get")) continue;

if (method.getParameterTypes().length != 0) continue;

if (method.getReturnType() == void.class) continue;

if (Modifier.isStatic(method.getModifiers())) continue;

final Object result = method.invoke(obj); System.out.println(name + " ==> " + result);

}

}

Die Ausgaben:

Constructor of C getAlpha ==> 4711 getBeta ==> Hello

getClass ==> class appl.Foo

Die Class.getConstructor-Methode und die Constructor-Methode newInstance

(Aufruf des parameterlosen Konstruktors):

static void **demoNewInstanceParameterless**() throws Exception { final Class<?> cls = Foo.class;

final Constructor<?> constructor = cls.getConstructor(); final Object obj = constructor.newInstance(); System.out.println(obj);

}

Ausgaben:

Constructor of C Foo [4711, Hello]

Hinweis: die Class-Methode newInstance ist seit Java 9 deprecated!

Die Class.getConstructor-Methode und die Constructor-Methode newInstance

(Aufruf des parametrisierten Konstruktors):

static void **demoNewInstanceParameterized**() throws Exception { final Class<?> cls = Foo.class;

final Constructor<?> constructor = cls.getConstructor(int.class, String.class);

final Object obj = constructor.newInstance(1000, "Hello World"); System.out.println(obj);

}

Die Ausgaben:

Constructor of C (1000, Hello World) Foo [1000, Hello World]

## Zugriff auf private Member

Mittels Reflection können wir auch auf private Elemente einer Klasse zugreifen: private Felder auslesen / neu setzen, private Methode aufrufen und private Konstruktoren nutzen). Man bezeichnet dies auch als "Deep Reflection".

Hinweis: Allerdings kann eine Anwendung mittels eine SecurityManagers gestartet werden, der Deep-Reflection verbietet.

Zur Demonstration verwenden wir eine Klasse Foo mit einem privaten Feld, mit einem privaten Konstruktor und einer privaten Methode:

package **appl**; public class **Foo** {

private int **alpha**;

public **Foo**() { System.out.println("Constructor of C"); this.alpha = 4711;

}

private **Foo**(int alpha) {

System.out.println("Constructor of C (" + alpha + ")"); this.alpha = alpha;

}

private void **doSomething**() { System.out.println("doSomething: " + this.alpha);

}

@Override

public String **toString**() {

return this.getClass().getSimpleName() + " [" + this.alpha + "]";

}

}

Wir greifen auf das private alpha-Feld zu (müssen zu diesem Zweck zuvor die Field- Methode setAccessible aufrufen):

static void **demoGet**() throws Exception { final Object obj = new Foo();

final Field field = obj.getClass().getDeclaredField("alpha");

field.setAccessible(true);

final Object value = field.get(obj); System.out.println(value);

}

Die Ausgaben:

Constructor of C 4711

Wir greifen auf die private doSomething-Methode zu (müssen zu diesem Zweck zuvor die Method-Methode setAccessible aufrufen):

static void **demoInvoke**() throws Exception { final Object obj = new Foo();

final Method method = obj.getClass().getDeclaredMethod("doSomething");

method.setAccessible(true); method.invoke(obj);

}

Die Ausgaben:

Constructor of C doSomething: 4711

Und schließlich benutzen wir zur Initialisierung eines Objekts den privaten Konstruktor (und müssen zu diesem Zweck zuvor die Constructor-Methode setAccessible aufrufen):

static void **demoNewInstance**() throws Exception { final Class<?> cls = Foo.class;

final Constructor<?> constructor = cls.getDeclaredConstructor(int.class);

constructor.setAccessible(true);

final Object obj = constructor.newInstance(1000); System.out.println(obj);

}

Die Ausgaben:

Constructor of C (1000) Foo [1000]

Man beachte, dass das Field- resp. das Method- resp. das Constructor-Objekt jeweils via getDeclaredField, getDeclaredMethod und getDeclaredConstructor ermittelt wurden.

## PropertyDescriptors

Die Sprache Java kennt keine "Properties" – Java kennt nur Attribute, Konstruktoren und Methoden. Im Kontext der Java-Beans-Spezifikation ist aber der Begriff Property eingeführt worden. Eine Property im Sinne dieser Spezifikation ist ein getter-setter-Paar.

Seien z. B. in einer Klasse folgende Methoden definiert:

int getSchallUndRauch()

void setSchallUndRauch(int value)

So bezeichnet man diese beiden Methoden als "Property".

Eine Property hat einen Namen – hier: "schallUndRauch" (man beachte die Groß-

/Klein-Schreibung). Und sie hat eine Typ: den Return-Typ der getter-Methode resp. den Parameter-Typ der setter-Methode: hier: int (diese beiden Typen müssen übereinstimmen).

Damit ein getter-Setter-Paar als Property bezeichnet werden kann, müssen die Methoden public sein – und dürfen nicht static sein.

Das Präfix einer getter-Method ist "get", das Präfix einer setter-Method ist "set". Ausnahme: das Präfix einer boolschen getter-Method ist "is".

Man beachte, dass Properties keine Attribute sind. Sie können auf Attribute abgebildet werden (und eben dies ist die Regel), müssen aber nicht auf Attribute abgebildet werden. Werden sie auf Attribute abgebildet, haben sie i.d.R. denselben Namen wie diese Attribute - aber auch die muss nicht zwingend.

Der ("öffentliche") Zustand eines Objekts kann dann als die Gesamtheit aller Werte bestimmt werden, die von den getter-Methoden der entsprechenden Klasse geliefert werden.

Reflection-basierte Tools greifen häufig auf solche Properties zu. Um diese Properties zu berechnen, könnten wir natürlich via Reflection alle Methoden ermitteln, die mit "get" beginnen, public und nicht static und parameterlos sind. Zu diesen gettern müssten wir jeweils die passenden setter ermitteln.

Ein solches Verfahren ist allerdings recht aufwendig. Deshalb existiert ein Paket java. Beans, dessen Klassen den Umgang mit solchen Properties erleichtern.

Das Paket enthält insbesondere eine Klasse namens PropertyDescriptor. Ein solcher Deskriptor enthält den Namen und den Typ der Property, die er beschreibt. Und er enthält zwei Method-Referenzen: eine Referenz auf dasjenige Method-Objekt, welches

die getter-Methode beschreibt, und eine zweite Referenz auf dasjenige Method-Objekt, welches die setter-Methode beschreibt.

Wir benutzen als Demo-Klasse die Klasse Book – eine Klasse mit den Properties "isbn", "title", "year", "price" und "author":

package **appl**; public class **Book** {

private String **isbn**; private String **title**; private int **year**; private double **price**; private String **author**;

public **Book**() { ... }

public **Book**(String isbn, String title, int year, double price, String author) { ... }

public String **getIsbn**() { ... }

public void **setIsbn**(String isbn) { ... }

public String **getTitle**() { ... }

public void **setTitle**(String title) { ... }

public int **getYear**() { ... }

public void **setYear**(int year) { ... }

public double **getPrice**() { ... }

public void **setPrice**(double price) { ... }

public String **getAuthor**() { ... }

public void **setAuthor**(String author) { ... }

@Override

public String **toString**() { ... }

}

Mit den Mittels von java.beans können wir uns einen Array von PropertyDescriptors besorgen, welche die Properties der Klasse Book beschreiben:

import java.beans.BeanInfo; import java.beans.Introspector;

import java.beans.PropertyDescriptor;

final BeanInfo beanInfo = Introspector.getBeanInfo(Book.class, Object.class);

final PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors();

Hinweis: das zweite Argument von getBeanInfo ist die sog. "Stop-Klasse". getBeanInfo liefert auch geerbte Elemente – bis hin zu eben dieser Stop-Klasse (deren Elemente werden unberücksichtigt gelassen).

Wir erhalten folgende PropertyDescriptors:

PropertyDescriptor [ ]

readMethod

: PropertyDescriptor

name = "author" propertyType = String.class

: PropertyDescriptor

name = "price" propertyType = double.class

: PropertyDescriptor

name = "year" propertyType = int.class

: PropertyDescriptor

name = "title" propertyType = String.class

: PropertyDescriptor

name = "isbn" propertyType = String.class

: Method

name = "getTitle"

: Method

name = "setIsbn"

: Method

name = "getIsbn"

writeMethod readMethod

writeMethod

readMethod

: Method

name = "getYear"

: Method

name = "setTitle"

writeMethod

readMethod

: Method

name = "getPrice"

: Method

name = "setYear"

writeMethod

readMethod

: Method

name = "getAuthor"

: Method

name = "setPrice"

writeMethod

Wir geben alle PropertyDescriptors der Book-Klasse aus:

: Method

name = "setAuthor"

for (final PropertyDescriptor pd : pds) { System.out.println(pd.getName() + ": " +

pd.getPropertyType().getSimpleName()); System.out.println(pd.getReadMethod()); System.out.println(pd.getWriteMethod()); System.out.println();

}

Die Ausgaben:

author: String

public java.lang.String appl.Book.getAuthor() public void appl.Book.setAuthor(java.lang.String)

isbn: String

public java.lang.String appl.Book.getIsbn() public void appl.Book.setIsbn(java.lang.String)

price: double

public double appl.Book.getPrice() public void appl.Book.setPrice(double)

title: String

public java.lang.String appl.Book.getTitle() public void appl.Book.setTitle(java.lang.String)

year: int

public int appl.Book.getYear() public void appl.Book.setYear(int)

# 12 Reflection und Generics

*TypeVariable*

*GenericArrayType*

*WildcardType*

*ParameterizedType*

Die virtuelle Maschine kümmert sich bekanntlich nicht um Generics – sie kennt nur einfache Typen und Objects. Die generischen Informationen sind nur für den Compiler relevant – und sie "verdampfen" im Prozess der Kompilation.

Dennoch können wir im begrenzten Umfang zur Laufzeit generische Informationen ermitteln – natürlich via Reflection. Im Zuge der Einführung von Generics sind also auch die Reflection-basierten Möglichkeiten erweitert worden. Hier ein Klassendiagramm:

GenericDeclaration

*Generic Declaration*

Class

*Type*

**\*** Bounds

Constructor

Method



GenericTypeComponent

* LowerBounds
* UpperBounds
* ActualTypeArguments RawType

TypeParameters

**\***

### Eine Übersicht

* + - Ermittlung von parametrisierten Typen
    - Parametrisierte Typen im Kontext von Vererbungen
    - Parametrisierte Typen im Kontext der Implementierung von Interfaces
    - Beispiel: Iterationen über polymorphe Listen
    - Eine Utility-Klasse zur Ermittlung von Typ-Parameter (auch bei der Verwendung von Lambdas)
    - Beispiel: Parser-Registrierungen

Darüber hinaus enthält der Eclipse-Workspace zwei weitere Projekte, die aus Platzgründen in diesem Skript aber nicht weiter vorgestellt werden. Sie sind zum Selbststudium gedacht.

## Parametrisierte Typen von Feldern und Methoden

Sei folgende Klasse gegeben:

public class **Foo** {

public Set<Long> **aField** = new HashSet<>();

public Map<Integer, Double> **aMethod**(List<String> list) { return null;

}

}

Die Klasse definiert eine Feld aField vom Typ Set<Long>. Und sie definiert eine Methode aMethod, der eine List<String> übergeben wird und die eine Map<Integer, Double> zurückliefert.

Wir wollen nun den "genauen" Typ von aField, den "genauen" Return-Typ von

aMethod und die "genauen" Parameter-Typen von aMethod ermitteln. Wir benötigen die Interfaces Type und ParameterizedType:

import java.lang.reflect.ParameterizedType; import java.lang.reflect.Type;

Wir benutzen eine Hilfsmethode, der eine Type-Referenz übergeben wird:

private static void **printGenericType**(Type genericType) { final ParameterizedType parameterizedType =

(ParameterizedType) genericType; System.out.println("parameterizedType = " + parameterizedType);

final Type rawType = parameterizedType.getRawType();

System.out.println("rawType = " + rawType);

final Type[] typeArguments = parameterizedType.getActualTypeArguments();

for (Type typeArgument : typeArguments) System.out.println("typeArgument = " + typeArgument);

}

Und untersuchen zunächst das Feld aField:

public static void **demoFieldType**() throws Exception { final Field field = Foo.class.getField("aField"); final Type type = field.getGenericType(); printGenericType(type);

}

Die Ausgaben:

parameterizedType = java.util.Set<java.lang.Long> rawType = interface java.util.Set typeArgument = class java.lang.Long

Wir untersuchen den Return-Typ der aMethod-Mehode:

public static void **demoMethodReturnType**() throws Exception {

final Method method = Foo.class.getMethod("aMethod", List.class); final Type type = method.getGenericReturnType(); printGenericType(type);

}

Die Ausgaben:

parameterizedType = java.util.Map<java.lang.Integer, java.lang.Double> rawType = interface java.util.Map

typeArgument = class java.lang.Integer typeArgument = class java.lang.Double

Und wir untersuchen schließlich den Typ des Parameters von aMethod:

public static void **demoMethodParameterTypes**() throws Exception { Method method = Foo.class.getMethod("aMethod", List.class); final Type type = method.getGenericParameterTypes()[0]; printGenericType(type);

}

Die Ausgaben:

parameterizedType = java.util.List<java.lang.String> rawType = interface java.util.List typeArgument = class java.lang.String

## Generische Basisklassen

Seien eine generische Basisklasse Foo und eine davon abgeleitete Klasse FooExt

gegeben:

public class **Foo**<R, S> {

}

public class **FooExt** extends Foo<Integer, String> {

}

Wir können zunächst die Typ-Variablen von Foo ermitteln (R, S):

public static void **demoTypeVariable**() { final Class<?> cls = Foo.class;

TypeVariable<?>[] typeVariables = cls.getTypeParameters(); for (TypeVariable<?> typeVariable : typeVariables)

System.out.println(typeVariable);

}

Die Ausgaben:

R S

Dann ermitteln wir den parametrisierten Typ von FooExt (und verwenden dabei wieder die bereits im letzten Abschnitt benutzte Methode printGenericType):

public static void **demoSuperclass**() { final Class<?> cls = FooExt.class;

final Type genericSuperclass = cls.getGenericSuperclass(); printGenericType(genericSuperclass);

}

Die Ausgaben:

parameterizedType = appl.Foo<java.lang.Integer, java.lang.String> rawType = class appl.Foo

typeArgument = class java.lang.Integer typeArgument = class java.lang.String

## Generische Interfaces

Wir verwenden das funktionale Interface Consumer:

package java.util.function;

public interface **Consumer**<T> { public abstract void **accept**(T);

}

Foo implementiert das Interface:

public class **Foo** implements Consumer<Float> { @Override

public void **accept**(Float value) {

}

}

Wir geben die Typ-Variablen von Consumer aus:

public static void **demoTypeVariable**() { final Class<?> cls = Consumer.class;

TypeVariable<?>[] typeVariables = cls.getTypeParameters(); for (TypeVariable<?> typeVariable : typeVariables)

System.out.println(typeVariable);

}

Die Ausgaben:

T

Und geben dann den parametrisierten Typ von Foo aus:

public static void **demoInterfaces**() throws Exception { final Class<?> cls = Foo.class;

final Type genericIface = cls.getGenericInterfaces()[0]; printGenericType(genericIface);

}

Die Ausgaben:

parameterizedType = java.util.function.Consumer<java.lang.Float> rawType = interface java.util.function.Consumer typeArgument = class java.lang.Float

## Ein Beispiel: Iteration über Drinks

Wir kommen auf die Hierarchie der Drink-Klassen zurück (siehe das Generics-Kapitel):

Drink, Beer, Wine, RedWine und WhiteWine. Wir bauen eine polymorphe Drink-Liste:

final List<Drink> drinks = new ArrayList<>();

drinks.add(new Beer()); drinks.add(new Wine()); drinks.add(new WhiteWine()); drinks.add(new Beer()); drinks.add(new RedWine()); drinks.add(new Drink());

Wir bauen eine iterate-Methode, welche über eine Liste iteriert und dabei nur solche Objekte an einen Consumer übergibt, die kompatibel zu einem bestimmten Typ sind.

Z.B. wollen wir nur alle Weine ausgeben (also Objekte vom Typ Wine, RedWine und

WhiteWine).

Eine erste Version:

static <T> void **iterate**(

List<? super T> list, Class<T> type,

Consumer<? super T> consumer) { for (Object element : list) {

if (type.isAssignableFrom(element.getClass())) consumer.accept((T)element);

}

}

Der Methode wird eine Liste übergeben, ein Class-Objekt und ein Consumer. Aufgrund der Parametrisierung mit T kann der Compiler garantieren, dass das Class-Objekt zum Consumer "passt". Das Class-Objekt wird in der Schleife genutzt, um nur diejenigen Objekte auszuwählen, kompatibel zu eben diesem Class-Objekt sind.

Diese Methode kann mit der oben erstellten Drink-Liste wie folgt aufgerufen werden:

iterate(

drinks, Wine.class,

new **Consumer**<Wine>() { @Override

public void **accept**(Wine drink) { System.out.println(drink);

}

}

);

Die Ausgaben:

drinks.Wine@9e89d68 drinks.WhiteWine@11028347 drinks.RedWine@14899482

Beer- und Drink-Objekte sind – wie beabsichtigt – übergangen worden.

Warum müssen aber an iterate drei Parameter übergeben werden? Reichen zwei nicht aus? Kann der type-Parameter entfallen?

Wie können den in Frage stehenden Typ tatsächlich aus dem Parametrisierung des

Consumer-Interfaces berechnen. Hier eine zweite Variante von iterate:

static <T> void **iterate**(

List<? super T> list, Consumer<T> consumer) {

final ParameterizedType pt = (ParameterizedType)consumer.getClass()

.getGenericInterfaces()[0];

final Class<T> type = (Class<T>) pt.getActualTypeArguments()[0]; for (final Object element : list) {

if (type.isAssignableFrom(element.getClass())) consumer.accept((T)element);

}

}

Hier ein möglicher Aufruf der Methode:

iterate(

drinks,

new **Consumer**<Wine>() { @Override

public void **accept**(Wine drink) { System.out.println(drink);

}

}

);

Die Ausgaben sind dieselben wie beim Aufruf der ersten iterate-Varianten.

Consumer ist ein funktionales Interfaces. Wir haben das Interfaces bislang in Form einer anonymen Klasse implementiert. Wir können es natürlich auch in Form eines Lambdas implementieren.

Hier der Aufruf der ersten iterate-Methode:

iterate(

drinks, Wine.class,

(Wine drink) -> System.out.println(drink));

Die Ausgaben sind natürlich dieselben wie bei der Verwendung einer anonymen Klassen.

Dann versuchen wir, auch die zweite iterate-Methode mittels eines Lambdas aufzurufen:

iterate(

drinks,

(Wine drink) -> System.out.println(drink));

Der Compiler ist natürlich zufrieden – zur Laufzeit aber wird nun eine

ClassCastException geworfen:

java.base/java.lang.Class cannot be cast to java.base/java.lang.reflect.ParameterizedType

Dieses Problem wird im nächsten Abschnitt gelöst werden…

## Eine Utility-Klasse ActualTypeArguments

Um auch dann die parametrisierten Typen von Interfaces ermitteln zu können, wenn diese in Form von Lambdas implementiert sind, definieren wir folgende Utilitiy-Klasse (deren genaues Studium dem Leser / der Lesering überlassen bleiben soll…):

package **util**;

// ...

public class **ActualTypeArguments** {

public static List<Class<?>> **of**(Object object) { try {

return ofStandard(object);

}

catch (Exception e) { try {

return ofLambda(object);

}

catch(Exception ex) {

throw new RuntimeException(ex);

}

}

}

private static List<Class<?>> **ofStandard**(Object object) { final Class<?> cls = object.getClass();

final Type[] ifaces = cls.getGenericInterfaces(); if (ifaces.length != 1)

throw new RuntimeException("class '" + cls.getName() + "' must implement exactly one generic interface");

final Type iface = ifaces[0];

if (!(iface instanceof ParameterizedType)) throw new RuntimeException("interface '" +

iface.getTypeName() + "' is not parameterized"); final ParameterizedType type = (ParameterizedType) iface; final Type[] typeArguments = type.getActualTypeArguments(); final Class<?>[] types = new Class<?>[typeArguments.length]; for (int i = 0; i < typeArguments.length; i++) {

final Type typeArgument = typeArguments[i]; if (!(typeArgument instanceof Class))

throw new RuntimeException(

"not a class: " + typeArgument); types[i] = (Class<?>) typeArgument;

}

return List.of(types);

}

private static List<Class<?>> **ofLambda**(Object object) throws Exception { final Method replaceMethod =

object.getClass().getDeclaredMethod("writeReplace"); replaceMethod.setAccessible(true);

final SerializedLambda lambda =

(SerializedLambda) replaceMethod.invoke(object);

final Class<?> ifaceClass = Class.forName( lambda.getFunctionalInterfaceClass().replace('/', '.'));

final Method ifaceMethod = findAbstractMethodOfFunctionalInterface(ifaceClass);

final Class<?> implClass = Class.forName(lambda.getImplClass().replace('/', '.'));

final String implMethodName = lambda.getImplMethodName(); final Method implMethod =

findLambdaImplMethod(implClass, implMethodName);

final Type[] ifaceTypeParameters = ifaceMethod.getDeclaringClass().getTypeParameters();

final Type ifaceReturnType = ifaceMethod.getGenericReturnType(); final Type[] ifaceParameterTypes =

ifaceMethod.getGenericParameterTypes();

final Class<?> implReturnType = implMethod.getReturnType(); final Class<?>[] implParameterTypes =

implMethod.getParameterTypes();

final Map<Type, Class<?>> actualTypes = new LinkedHashMap<>(); for (Type ifaceTypeParameter : ifaceTypeParameters) {

if (ifaceTypeParameter == ifaceReturnType) actualTypes.put(ifaceTypeParameter, implReturnType);

else {

int index =

indexOf(ifaceTypeParameter, ifaceParameterTypes); actualTypes.put(ifaceTypeParameter,

implParameterTypes[index]);

}

}

return List.of(actualTypes.values().toArray(

new Class<?>[actualTypes.values().size()]));

}

private static int **indexOf**(

Type typeParameter, Type[] parameterTypes) { for (int i = 0; i < parameterTypes.length; i++)

if (parameterTypes[i] == typeParameter) return i;

throw new RuntimeException();

}

public static Method **findLambdaImplMethod**( Class<?> impl, String name) {

for (Method method : impl.getDeclaredMethods()) { if (method.getName().equals(name))

return method;

}

return null;

}

public static Method **findAbstractMethodOfFunctionalInterface**( Class<?> iface) {

for (Method method : iface.getMethods()) {

if (Modifier.isAbstract(method.getModifiers())) return method;

}

return null;

}

}

Damit die of-Methode der obigen Klasse wie erwartet funktioniert, müssen die funktionalen Interfaces, die wir implementieren wollen, Serializable sein. Wir definieren also ein Interface SerializableConsumer:

@FunctionalInterface

public interface **SerializableConsumer**<T> extends Consumer<T>, Serializable {

}

Wir benutzen nun folgende iterate-Methode (statt Consumer verwenden wir als Typ des zweiten Parameters SerializableConsumer):

static <T> void **iterate**(

List<? super T> list, SerializableConsumer<T> consumer) {

final Class<?> type = ActualTypeArguments.of(consumer).get(0); for (final Object element : list) {

if (type.isAssignableFrom(element.getClass())) consumer.accept((T)element);

}

}

Man beachte den Aufruf von ActualTypeArguments.of.

Dieser Methode kann nun sowohl ein SerializableConsumer übergeben werden, der in Form einer anonymen Klasse implementiert ist als auch ein Lambda-Ausdruck:

iterate(drinks, new **SerializableConsumer**<Wine>() { @Override

public void **accept**(Wine drink) { System.out.println(drink);

}

});

iterate(drinks, (Wine drink) -> System.out.println(drink));

Um die Utilitiy-Klasse ActualTypeArguments allgemein verwenden zu können, müssten wird natürlich zu jedem funktionalen Interface dessen Serializable-Variante definieren…

## Beispiel: Traversierung eines GUI-Baumes

Das Beispiel soll hier nicht weiter beschrieben werden. Das Studium des Quellcodes sei dem Leser / der Leserin überlassen.

(Es handelt sich um ein interessantes Beispiel…)

## Beispiel: Restful

Auch dieses Beispiel soll hier nicht weiter beschrieben werden. Das Studium des Quellcodes sei dem Leser / der Leserin überlassen.

(Auch hier handelt es sich um ein interessantes Beispiel…)

# 13 Dynamic Proxies

Ein Proxy-Objekt ist ein Objekt, welches einem anderen Objekt – dem "realen Objekt" – vorgeschaltet wird. Die Klasse des Proxy-Objekts implementiert dabei dasselbe (dieselben) Interface(s) wie die Klasse des realen Objekts. Ein Client kann also ein Proxy-Objekt genauso behandeln wie das reale Objekt – sofern er sich über Interface- Referenzen entweder auf das Proxy oder auf das reale Objekt bezieht.

Eine Proxy-Klasse wird implementiert, indem alle ihre Methoden abgebildet werden auf die entsprechenden Methoden der "realen" Klasse. Alle Methodenaufrufe auf das Proxy- Objekt werden von diesem an das reale Objekt delegiert. Vor und nach der entsprechenden Delegation können aber jeweils Proxy-spezifische Operationen ausgeführt werden.

Proxy-Objekte können somit genutzt werden, um in den Aufruf der Methoden auf die realen Objekte einzugreifen. Sie können als "Interzeptoren" fungieren. Sie können eingesetzt werden, um fachfremde, aber übergreifende "Aspekte" ("crosscutting concerns") von der "eigentlichen" Funktionalität zu trennen. (Dies ist auch das Anliegen der sog. aspekt-orientierten Programmierung.) Hier einige Beispiele für solche "Aspekte":

* Tracen von Methodenaufrufen
* Berechtigungsprüfungen,
* Transaktionssteuerung
* Performance-Messungen
* Synchronisation

Die Implementierung einer Proxy-Methode besteht zumeist auf dem folgendem "Dreisatz":

* Pre-Invoke (Proxy-spezifische Operationen)
* Invoke (Delegation an die entsprechende Methode des realen Objekts)
* Post-Invoke (Proxy-spezifische Operationen)

Proxy-Klassen können natürlich für jedes Interface und für jeden Aspekt manuell erstellt werden. Die Klasse java.lang.reflect.Proxy geht einen Schritt weiter: sie ermöglicht die Erstellung von Proxy-Klassen zur Laufzeit. Dabei delegiert das eigentliche Proxy-Objekt (dessen Klasse generiert wird) nicht direkt an das reale Objekt, sondern an einen InvocationHandler. Es ist dann dieser Handler, der den obigen Dreisatz implementiert. Für jeden zu unterstützenden Aspekten muss dann nur noch eine entsprechende Handler-Klasse bereitgestellt werden (welche unabhängig ist von dem jeweiligen fachlichen Interface).

Insbesondere im Kontext eines EJB- oder Spring-Containers sind die Dynamic-Proxies von herausragender Bedeutung.

### Übersicht

* + Vermischung von Fachlichkeit und fachfremden Aspekten
  + Das Proxy-Pattern
  + InvocationHandler
  + Dynamic-Proxies
  + Verfeinerungen von InvocationHandler
  + Eine kleine Client-Server-Anwendung
  + Grundlegende Interfaces der AOPAlliance

## Das Problem

Jede nicht-triviale Anwendung besteht i.d.R. aus einer Reihe von "Service"-Klassen – Klassen, die eine fachliche Funktionalität ("Geschäftslogik") implementieren. Solche Klassen sind zumeist zustandslos. Von jeder dieser Service-Klassen existiert zur Laufzeit genau ein einziges Objekt (welches – da zustandslos – von beliebig vielen Threads gleichzeitig genutzt werden kann). Im Kontext von EJB (Enterprise Java Beans) werden solche Objekte als "stateless session beans" bezeichnet – im Spring-Kontext handelt es sich um Beans, die als "Singletons" registriert werden.

Als durchgängiges Beispiel wird im folgenden ein sehr einfacher Service benutzt – ein Mathematik-Service. Dieser Service bietet zwei Methoden an, mittels derer die Summe resp. die Differenz zweier Zahlen berechnet werden kann (einfacher geht's kaum…)

Hier eine Implementierung:

public class **MathService** {

public int **sum**(int x, int y) { return x + y;

}

public int **diff**(int x, int y) { return x - y;

}

}

Nun kommt eine neue Anforderung hinzu: Unter bestimmten Bedingungen soll der Einstieg in und der Ausstieg aus den Methoden protokolliert werden. Kein Problem:

public class **MathService** {

public boolean **mustTrace** = true; public int **sum**(int x, int y) {

if (this.mustTrace)

System.out.println(">> sum(" + x + ", " + y + ")"); int result = x + y;

if (this.mustTrace)

System.out.println("<< sum(" + x + ", " + y + ") --> " + result); return result;

}

public int **diff**(int x, int y) { if (this.mustTrace)

System.out.println(">> diff(" + x + ", " + y + ")"); int result = x - y;

if (this.mustTrace)

System.out.println("<<diff (" + x + ", " + y + ") --> " + result); return result;

}

}

Technisch war die Erweiterung problemlos – konzeptionell ist der obige Code aber eine Katastrophe. Die Klasse ist nun alles andere als "clean" – sie würde Fachlichkeit vermischen mit dem ganz fachfremden Trace-Aspekt. Eine Klasse soll nur genau eine einzige Verantwortlichkeit haben – die obige Klasse hat zwei Verantwortlichkeiten.

Einmal ganz abgesehen davon, dass die Erweiterung nun an allen Stellen "denselben" dummen Trace-Code enthält (wir schreiben uns die Finger wund…).

## Ein einfaches Proxy

Zunächst ein Objekt-Diagramm:

*MathService*

*MathService*

: MathService TraceProxy

: MathService Impl

target

ff

di

um

s

ff

di

um

s

Um die Angelegenheiten sauber zu trennen, beginnen wir mit einem Interface (dies ist die Voraussetzung für alles weitere…):

public interface **MathService** {

public abstract int **sum**(int x, int y); public abstract int **diff**(int x, int y);

}

Die Implementierung des Interfaces beschränkt sich auf die eigentliche Fachlogik:

public class **MathServiceImpl** implements MathService { public int **sum**(int x, int y) {

return x + y;

}

public int **diff**(int x, int y) { return x - y;

}

}

Der Trace-Aspekt wird in einer Proxy-Klasse implementiert – in einer Klasse, die dasselbe Interface implementiert wie die "eigentliche" Klasse:

public class **MathServiceTraceProxy** implements MathService { private final MathService **target**;

public **MathServiceTraceProxy**(MathService target) { this.target = target;

}

public int **sum**(int x, int y) {

// PRE-INVOKE

System.out.println(">> sum(" + x + ", " + y + ")");

// INVOKE

int result = this.target.sum(x, y);

// POST-INVOKE

System.out.println("<< sum(" + x + ", " + y + ") -> " + result); return result;

}

public int **diff**(int x, int y) {

// PRE-INVOKE

System.out.println(">> diff(" + x + ", " + y + ")");

// INVOKE

int result = this.target.diff(x, y);

// POST-INVOKE

System.out.println("<< diff(" + x + ", " + y + ") -> " + result); return result;

}

}

Ein MathServiceTraceProxy-Objekt enthält eine Referenz auf das "Zielobjekt" – die target Referenz. Man beachte, dass die Referenz vom Interface-Typ ist (und nicht vom Implementierung-Typ). Diese Referenz wird mittels des Konstruktors initialisiert.

Die Methoden sind allesamt nach demselben Muster gestrickt: Protokollierung des Einstiegs in die Methode, Aufruf der Methode auf das Zielobjekt und Protokollierung des Ausstiegs. Man könnte die Schritte dieses "Dreisatzes" allgemein als Pre-Invoke, Invoke und Post-Invoke bezeichnen.

(Ein Transaktions-Proxy z.B. würde im Pre-Invoke die Transaktion starten und im Post- Invoke terminieren – via commit oder rollback. Ein Zeitmessungs-Proxy würde im Pre- Invoke eine Stoppuhr starten und diese im Post-Invoke ablesen. Etc.).

Hier eine mögliche Applikation:

public class **Application** {

public static void **main**(String[] args) throws Exception {

// final MathService mathService = new MathServiceImpl(); final MathService mathService = new MathServiceTraceProxy(

new MathServiceImpl()); theWholeApplication(mathService);

}

static void **theWholeApplication**(MathService mathService) { System.out.println(mathService.sum(40, 2));

System.out.println(mathService.diff(80, 3));

}

}

Zunächst wird das Impl-Objekt erzeugt und dieses dann dem Konstruktor der Proxy- Klasse übergeben. Das resultierende Proxy-Objekt wird einer Variablen des Interface- Typs (MathSevice) zugewiesen. Der komplette Rest der Anwendung bezieht sich auf die erzeugten Objekte nur über das Interface. Und soll morgen nicht mehr protokolliert

werden, wird statt der Kombination Proxy-Impl nur das Impl-Objekt erzeugt und an die MathService-Variable gebunden (s. den oben auskommentierten Code). Der komplette Rest der Anwendung bleibt unverändert.

Die Ausgaben des obigen Programms:

>> sum(40, 2)

<< sum(40, 2) --> 42

42

>> diff(80, 3)

<<diff (80, 3) --> 77

77

Schließlich können mehrere Proxy-Objekte "in Reihe" geschaltet werden. Da ein Proxy- Objekt das Zielobjekt wieder nur über eine Referenz des Interface-Typs (MathService) anspricht, kann es einem Proxy gleichgültig sein, ob es tatsächlich bereits an das "eigentliche" Zielobjekt (hier: an der MathServiceImpl) delegiert oder aber an ein weiteres Proxy, welches ebenfalls das MathService-Interface implementiert. Auf diese Weise können mehrere Aspekte der Reihe nach behandelt werden, bevor die eigentliche Fachlogik zum Zuge kommt (und nachdem diese zum Zuge gekommen ist).

Doppelt protokolliert hält besser:

*MathService*

*MathService*

*MathService*

: MathService TraceProxy

: MathService TraceProxy

: MathService Impl

target

target

ff

di

um

s

ff

di

um

s

ff

di

um

s

final MathService mathService = new MathServiceTraceProxy(

new MathServiceTraceProxy(

new MathServiceImpl()));

Die Ausgaben:

>> sum(40, 2)

>> sum(40, 2)

<< sum(40, 2) --> 42

<< sum(40, 2) --> 42

42

>> diff(80, 3)

>> diff(80, 3)

<<diff (80, 3) --> 77

<<diff (80, 3) --> 77

77

Die Verwendung des Proxy-Musters ermöglicht also eine saubere Trennung der Fachlichkeit von fachfremden Aspekten.

Leider aber hat die Sache einen Haken. Angenommen, eine Anwendung besteht aus

100 Service-Klassen, die allesamt über ein Interface spezifiziert sind. Und weiterhin angenommen, es gäbe 5 Aspekte, die für alle diese Interfaces implementiert werden müssten (Tracen, Transaktionen steuern, Zeiten messen etc.):

Dann müssten wird 5 \* 100 Proxy-Klassen schreiben. Und das ist natürlich völlig unzumutbar…

Schön wäre es, wenn wir nur 5 + 100 Proxy-Klassen müssten, um alle Aspekte mit allen Interfaces verheiraten zu können.

Und noch schöner wäre es, wenn wird nur 5 Klassen schreiben müssten, um dasselbe Ziel zu erreichen.

## Das Interface InvocationHandler

Um das oben skizzierte Ziel zu erreichen – damit sich also alles in Wohlgefallen auflöst

* müssen wir die Sache zunächst einmal etwas verkomplizieren.

Ein Objekt-Diagramm:

*MathService*

*Invocation Handler*

: Trace Handler

*MathService*

: MathService Proxy

: MathService Impl

handler

target

ff

di

um

s

vke

ino

ff

di

um

s

Die Proxy-Klasse heißt nun nicht mehr MathServiceTraceProxy, sondern MathServiceProxy. Offenbar wird dann in den Methoden der Klasse nicht mehr protokolliert.

Stattdessen delegieren alle Methoden der Proxy-Klasse an einen InvocationHandler

* z.B. an die invoke-Methoden eines TraceHandlers (oder eines

TransactionHandlers etc.). InvocationHandler ist ein Standard-Java-Interface:

package java.lang.reflect;

public interface **InvocationHandler** {

public abstract Object **invoke**(Object proxy, Method method, Object[] args)

throws Throwable;

}

Die Methoden der Proxy-Klassen müssen dasjenige Method-Objekt berechnen, welches die jeweilige Interface-Methode repräsentiert. Dann müssen sie ihre individuellen Argumente zu einem Object-Array flachklopfen. Mit diesen beiden Argumente – das Method-Objekt und der Object-Array – kann dann die invoke-Methode des InvocationHandlers aufgerufen werden.

Man beachte, dass die Proxy-Klasse völlig unabhängig ist von der tatsächlichen Handler-Klasse (die handler-Referenz der Proxy-Klasse ist vom Typ InvocationHandler).

Der TraceHandler kennt das eigentliche Zielobjekt – über sein target-Attribute. Dieses wird vom allgemeinsten Typ Object sein. Dem TraceHandler kann es also egal sein, ob das Zielobjekt vom Typ MathServicImpl oder vom Typ AccountServiceImpl etc. ist.

Nun zur Implantierung der obigen Überlegungen.

Wir können folgende TraceHandler-Klasse schreiben:

import java.lang.reflect.InvocationHandler; import java.lang.reflect.Method;

import java.util.Arrays;

public class **TraceHandler** implements InvocationHandler { private final Object **target**;

public **TraceHandler**(Object target) { this.target = target;

}

@Override

public Object **invoke**(Object proxy, Method method, Object[] args) throws Throwable {

final String call = method.getName() + Arrays.toString(args);

// PRE-INVOKE

System.out.println(">> " + call);

// INVOKE

final Object result = method.invoke(this.target, args);

// POST-INVOKE

System.out.println("<< " + call + " --> " + result);

return result;

}

}

Über den Konstruktor der Klasse wird einem TraceHandler sein Zielobjekt injiziert und dieses an die Object-Variablen target gebunden.

Der invoke-Methode wird zusätzlich zu den beiden oben bereits genannten Argumenten noch die Referenz auf das Proxy-Objekt übergeben werden, welches die invoke-Methode aufruft.

Der Pre-Invoke- und der Post-Invoke sind trivial: wir geben jeweils den Namen der uns übergebenen Methode und die uns übergebenen Parameter aus (und beim Post-Invoke noch das Resultat).

Der Invoke-Schritt wird wie folgt implementiert: wir benutzen das uns übergebene Method-Objekt, um mittels dessen(!) invoke-Methode via Reflection die Methode auf das eigentliche Ziel-Objekt aufzurufen. Diese Methode liefert das Resultat dieser von ihr aufgerufenen Methode zurück, welches dann als Resultat auch der Handler-eigenen invoke-Methode zurückgeliefert wird.

Nun zur MathServiceProxy-Klasse:

import java.lang.reflect.InvocationHandler; import java.lang.reflect.Method;

public class **MathServiceProxy** implements MathService { private final InvocationHandler **handler**;

public **MathServiceProxy**(InvocationHandler handler) { this.handler = handler;

}

public int **sum**(int x, int y) { try {

final Method method = MathService.class.getMethod( "sum", int.class, int.class);

final Object[] args = { x, y };

final Object result = handler.invoke(this, method, args); return (Integer)result;

}

catch(Throwable e) {

throw new RuntimeException(e);

}

}

public int **diff**(int x, int y) { try {

final Method method = MathService.class.getMethod( "diff", int.class, int.class);

final Object[] args = { x, y };

final Object result = handler.invoke(this, method, args); return (Integer)result;

}

catch(Throwable e) {

throw new RuntimeException(e);

}

}

}

Dem MathService wird mittels des Konstruktors ein InvocationHandler injiziert. Alle Methoden dieser Klasse berechnen das Method-Objekt, klopfen ihrer Parameter flach und rufen dann die invoke-Methode des InvocationHandlers auf.

Die folgende Anwendung erzeugt ein MathServicImpl-Objekt, steckt dieses in einen

TraceHandler und steckt diesen TraceHandler in ein MathServiceProxy:

public class **Application** {

public static void **main**(String[] args) throws Exception { final MathService mathService =

new MathServiceProxy( new TraceHandler(

new MathServiceImpl())); theWholeApplication(mathService);

}

static void **theWholeApplication**(MathService mathService) { System.out.println(mathService.sum(40, 2));

System.out.println(mathService.diff(80, 3));

}

}

Das MathServiceProxy delegiert an den TraceHandler, der TraceHandler delegiert an das MathServiceImpl. Der "Dreisatz" ist aus dem Proxy in den Handler verlagert worden.

Der gesamte Rest der Anwendung ist natürlich auch hier nur abhängig von dem Service-Interface.

Resultat:

Die MathServiceProxy-Klasse ist unabhängig geworden vom konkreten Aspekt. Sie ist nur noch abhängig von der Standardklasse InvocationHandler (und natürlich vom fachlichen Interface: MathService). Sie kann daher mit jedem InvocationHandler (TraceHandler, TransactionHandler, etc.) verwendet werden. Die Klasse TraceHandler ist ihrerseits unabhängig von dem fachlichen Interface. Ein TraceHandler z.B. kann daher mit jedem fachlichen Interface zusammengebracht werden.

Um also 100 Interfaces mit allen 5 Aspekten beliebig verbinden zu können, bedarf es nurmehr 105 Klassen: 100 Proxy-Klassen und 5 Handler-Klassen.

## Proxy.newProxyInstance

Betrachtet man die oben benutzte MathServiceProxy einmal genauer, so fällt auf, dass sie ziemlich "dumm" ist.

Egal, welches fachliche Interface eine solche Proxy-Klasse implementiert: die Implementierung folgt immer wieder demselben Schema. Jede solche Proxy-Klasse benötigt eine Instanzvariable, welche auf einen InvocationHandler zeigt (und einen Konstruktor, welcher diese Variable initialisiert). Dann müssen alle Interface-Methoden implementiert werden. In jeder dieser Implementierungen wird jeweils ein Method-Objekt berechnet (welches eben die entsprechende Interface-Methode beschreibt), die individuellen Parameter der Methode in einen Object-Array verpackt und anschließend die invoke-Methode vom InvocationHandler aufgerufen - deren Ergebnis schließlich

i.d.R. downgecastet wird und dann zurückgeliefert wird.

Angenommen, es existiere bereits die class-Datei eines fachlichen Interfaces (hier also: MathService.class). Dann könnte ein Reflection-basierter Automat dieses Interface analysieren und den Quellcode der diesem Interface entsprechenden Proxy- Klasse automatisch generieren (denn diese Klasse lässt sich offenbar automatisch aus dem Interface ableiten). Es bedarf nur relativ wenig Aufwand, einen solchen Automaten zu schreiben.

Allerdings benötigt man dann eine zweistufige Übersetzung: zunächst müsste per javac aus MathService.java die Daten MathService.class erzeugt werden – welche dann vom Automaten als Input benutzt würde. Im zweiten Schritt müsste dann der Output des Automaten (MathServiceProxy.java) in die entsprechende class-Datei übersetzt erden.

Die zweite Übersetzung ließe sich umgehen, wenn der Automat – statt Java-Quellcode - direkt Bytecode erzeugen würde – also direkt die class-Datei erzeugen würde. Allerdings müsste der Automat dann beim Build immer noch explizit aufgerufen werden...

Aber muss der produzierte Bytecode überhaupt im Filesystem gespeichert werden? Es handelt sich bei einer class-Datei doch nur um eine Folge von Bytes – welche auch einfach direkt im Hauptspeicher abgelegt werden könnte. Der Bytecode könnte also zur Laufzeit erzeugt werden – und mittels eines entsprechenden Klassenladers geladen werden. Der Vorteil läge darin, dass man sich beim Build des Systems um nichts mehr kümmern muss…

Glücklicherweise brauchen wir einen solchen Automaten nicht erst zu schreiben – er existiert bereits: in Form der Klasse java.lang.reflect.Proxy.

Hier eine Anwendung, welche diesen Automatismus nutzt:

import java.lang.reflect.InvocationHandler; import java.lang.reflect.Proxy;

public class **Application** {

public static void **main**(String[] args) throws Exception {

final InvocationHandler handler = new TraceHandler( new MathServiceImpl());

final MathService mathService = (MathService) Proxy.newProxyInstance( ClassLoader.getSystemClassLoader(),

new Class[] { MathService.class }, handler);

System.out.println(mathService.getClass().getName()); theWholeApplication(mathService);

}

static void **theWholeApplication**(MathService mathService) { System.out.println(mathService.sum(40, 2));

System.out.println(mathService.diff(80, 3));

}

}

Zunächst wird – wie gehabt – ein TraceHandler erzeugt (mit einem MathServiceImpl

als Zielobjekt).

Dann wird die statische Proxy-Methode newProxyInstance aufgerufen. Diese Methode wird zunächst die erforderliche Proxy-Klasse generieren. Zu diesem Zwecke muss die Methode zwei Dinge kennen: den ClassLoader, mittels dessen die Klasse geladen werden soll (im obigen Beispiel wird der System Class Loader verwendet) und das Interface, welches diese Klasse implementieren soll. Da eine Klasse (also auch eine generierte Proxy-Klasse) nicht nur ein einziges, sondern beliebig viele Interfaces implementieren kann, wird (werden) die Interface-Klasse(n) in einem Array übergeben. Im obigen Falle soll die Proxy-Klasse soll nur ein einziges Interface implementieren (MathService).

Aufgrund dieser ersten beiden Parameter von newProxyInstance kann also die erforderliche Proxy-Klasse generiert werden. Diese Klasse ist quasi "namenlos"– intern aber werden solche Klassen einfach als $Proxy0, $Proxy1 etc. benannt.

Wenn die Proxy-Klasse generiert wurde, muss anschließend ein Objekt dieser Klasse erzeugt werden. Dabei wird dem Konstruktor dieser Klasse die Referenz auf denjenigen InvocationHandler übergeben, welcher beim Aufruf von newProxyInstance als dritter Parameter mitgegeben wird.

newProxyInstance liefert dann die Referenz auf das so erzeugt Proxy-Objekt als Object zurück – welche dann auf eines der von der Proxy-Klasse implementierten Interfaces downgecastet werden kann (hier: MathService).

Die Ausgaben des obigen Programms:

com.sun.proxy.$Proxy0

>> sum[40, 2]

<< sum[40, 2] --> 42

42

>> diff[80, 3]

<< diff[80, 3] --> 77

77

## Verfeinerungen

Die im vorletzten Abschnitt entwickelte TraceHandler-Klasse ist ein wenig simple- minded. Zwei Probleme wurden noch nicht weiter beachtet:

Was passiert, wenn ein TraceHandler noch nicht an das eigentliche Zielobjekt, sondern an einen weiteren InvocationHandler delegiert?

Und was passiert, wenn die von der invoke-Methode des TraceHandlers aufgerufene Methode des Zielobjekts eine Exception wirft?

Diese beiden Problems sind nun in der folgenden erweiterten TraceHandler-Klasse berücksichtigt:

import java.lang.reflect.InvocationHandler;

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

import java.util.Arrays;

public class **TraceHandler** implements InvocationHandler { private final Object **target**;

public **TraceHandler**(Object target) { this.target = target;

}

@Override

public Object **invoke**(Object proxy, Method method, Object[] args) throws Throwable {

final String call = method.getName() + Arrays.toString(args);

// PRE-INVOKE

System.out.println(">> " + call); try {

// INVOKE

final Object result;

if (this.target instanceof InvocationHandler)

result = ((InvocationHandler) this.target).invoke( proxy, method, args);

else

result = method.invoke(this.target, args);

// POST-INVOKE

System.out.println("<< " + call + " --> " + result); return result;

}

catch (InvocationTargetException e) { System.out.println("<< " + call + " --> " + e); throw e.getTargetException();

}

}

}

invoke prüft zunächst, ob target auf einen weiteren InvocationHandler zeigt. Wenn ja, dann wird einfach die invoke-Methode eben dieses InvocationHandlers aufgerufen. Ansonsten wird – wie gehabt – via Reflection – die Methode des entsprechenden Zielobjekts aufgerufen.

Die Delegation an einen weiteren InvocationHandler oder an das Zielobjekt geschieht in einem try-Block. Wenn die Methode des Zielobjekts eine Exception wirft, so wickelt die Method-eigene invoke-Methode diese Exception in eine InvocationTragetException ein – mit der Konsequenz, dass die Handler-eigene invoke-Methode diese eingewickelte Exception in ihrem catch-Block wieder auswickeln muss.

Hier eine Applikation, welche zwei TraceHandler in Reihe schaltet:

public class **Application** {

public static void **main**(String[] args) throws Exception {

final InvocationHandler handler = new TraceHandler( new TraceHandler(

new MathServiceImpl()));

final MathService mathService = (MathService) Proxy.newProxyInstance( ClassLoader.getSystemClassLoader(),

new Class[] { MathService.class }, handler);

theWholeApplication(mathService);

}

static void **theWholeApplication**(MathService mathService) { System.out.println(mathService.sum(40, 2));

System.out.println(mathService.diff(80, 3));

}

}

Die Ausgaben:

>> sum[40, 2]

>> sum[40, 2]

<< sum[40, 2] --> 42

<< sum[40, 2] --> 42

42

>> diff[80, 3]

>> diff[80, 3]

<< diff[80, 3] --> 77

<< diff[80, 3] --> 77

77

## Beispiel: Client-Server

Zunächst ein Objektdiagramm:



*MathService*

*Invocation Handler*

: MathService Proxy

: RMI

Handler

handler

out

in

*Serializable*

creates

vke

ino

ff

di

um

s

: Object InputStream

: Object OutputStream

: MethodCall Method m Object [ ] args

: Output Stream

: Input Stream

: Socket

Client

Serializing MethodCall

Deserializing Result

Server

Deserializing MethodCall

Deserializing Result

creates

*Serializable*

: ServerSocket

accept

: MethodCall Method m Object [ ] args

*MathService*

: MathService Impl

ff

di

um

s

: Object OutputStream

: Object InputStream

: Output Stream

: Input Stream

: Socket

invoke

Die folgende Anwendung demonstriert die grundlegenden Mechanismen des Remote Method Calls. (Die "richtige" Standard-RMI-Implementierung enthält denselben Kern wie die hier vorgestellt simple Implementierung.)

Das Package common enthält das Interface MathService. Dieses Interface wird sowohl vom Client als auch vom Server genutzt:

package **common**;

public interface **MathService** {

public abstract int **sum**(int x, int y); public abstract int **diff**(int x, int y);

}

Das Package server enthält die Implementierungsklasse MathServiceImpl. Diese wird nur Server-seitig verwendet:

package **server**;

import common.MathService;

public class **MathServiceImpl** implements MathService { public int **sum**(int x, int y) {

return x + y;

}

public int **diff**(int x, int y) { return x - y;

}

}

Das Package util enthält die (wiederverwendbare) Klasse MethodCall. Ein MethodCall repräsentiert einen Methodenaufruf und muss deshalb ein Method-Objekt und die Aufrufargumente (in Form eines Object-Arrays) enthalten. Ein MethodCall muss Serializable sein.

Leider aber sind Method-Objekte nicht serialisierbar. Also wird das Method-Objekt zwar dem Konstruktor der Klasse übergeben – dieser aber "zerlegt" das ihm übergebene Method-Objekt in "Einzelteile", die ihrerseits serialisierbar sind. Die Klasse enthält eine zum Konstruktor "inverse" Methode namens getMethod, welche diese Einzelteile wieder zu einem Method-Objekt zusammenbaut. Nach außen sieht es also so aus, als würde tatsächlich das Method-Objekt serialisiert:

package **util**;

import java.io.Serializable; import java.lang.reflect.Method;

public class **MethodCall** implements Serializable { private final Class<?> **declaringClass**;

private final String **methodName**; private final Class<?>[] **paramterTypes**; private final Object[] **arguments**;

public **MethodCall**(Method method, Object... arguments) { this.declaringClass = method.getDeclaringClass(); this.methodName = method.getName(); this.paramterTypes = method.getParameterTypes(); this.arguments = arguments;

}

public Object[] **getArguments**() { return this.arguments;

}

public Method **getMethod**() { try {

return this.declaringClass.getMethod( this.methodName, this.paramterTypes);

}

catch(Exception e) {

throw new RuntimeException(e);

}

}

}

Im server-Package ist neben der MathServiceImpl-Klasse die Startklasse des Servers enthalten – die Klasse Server. Diese erzeugt ein MathServiceImpl-Objekt.

Die main-Methode erzeugt ein ServerSocket erzeugt und wartet dann via accept auf einen Verbindungswunsch eines Clients. Wenn ein Client dann ein Socket erzeugt, kehrt die accept-Methode zurück und liefert dabei ein Server-seitiges Socket-Objekt, welches via TCP/IP mit dem Client-seitigen Socket-Objekt verbunden ist.

An den InputStream des Sockets wird dann ein Deserialisierer angeschlossen (ein ObjectInputStream) – und an den OutputStream des Sockets ein Serialisierer (ein ObjectOutputStream).

Dann geht die main-Methode in eine scheinbare Endlosschleife über. In jedem Schleifendurchlauf wird jeweils ein Request verarbeitet. Der ObjectInputStream liefert ein MethodCall, der dann für den Reflection-basierten Zugriff auf das MathServiceImpl-Objekt benutzt wird. Das Resultat dieses Aufrufs wird dann mittels des OutputStreams des Sockets zum Client zurückgeschickt.

Schließt der Client sein Socket, liefert der ObjectInputStream eine EOFException – was dazu führt, dass die Schleife verlassen wird und die main-Methode terminiert.

(Es wäre natürlich wünschenswert, wenn der Server mehrere Clients bedienen könnte – entweder nacheinander oder aber – noch schöner – parallel. Aber das steht hier nicht zur Debatte.)

Hier der komplette Quellcode des Servers:

package **server**;

import java.io.EOFException; import java.io.ObjectInputStream; import java.io.ObjectOutputStream; import java.lang.reflect.Method; import java.net.ServerSocket; import java.net.Socket;

import util.MethodCall; public class **Server** {

private static final Object **service** = new MathServiceImpl();

public static void **main**(final String[] args) throws Exception { try(ServerSocket serverSocket = new ServerSocket(8000)) {

System.out.println("waiting...");

final Socket socket = serverSocket.accept(); System.out.println("accepted!");

final ObjectInputStream in =

new ObjectInputStream(socket.getInputStream()); final ObjectOutputStream out =

new ObjectOutputStream(socket.getOutputStream()); while(true) {

try { System.out.println("reading..."); final MethodCall call =

(MethodCall)in.readObject();

final Method method = call.getMethod(); final Object result =

method.invoke(service, call.getArguments()); out.writeObject(result);

}

catch(EOFException e) { System.out.println("EOF!"); break;

}

}

}

}

}

Im util-Package ist neben der Klasse MethodCall auch die Klasse RMIHandler

enthalten (diese Klasse wird natürlich nur Client-seitig genutzt).

Bei der Erzeugung eines RMIHandlers wird dem Konstruktor ein Socket übergeben. Aufgrund dieses Sockets erzeugt er dann seinen Serialisierer und seinen Deserialisierer.

Ein RMIHandler ist ein InvocationHandler. Er implementiert also die invoke- Methode. Aufgrund der an invoke übergebenen Parameter kann ein MethodCall gebaut werden und dieses über den Serialisierer an das Socket übergeben werden. Dann wartet die invoke-Method auf den Response des Servers. Und dieser wird schließlich als Resultat von invoke zurückgeliefert.

package **util**;

import java.io.ObjectInputStream; import java.io.ObjectOutputStream;

import java.lang.reflect.InvocationHandler; import java.lang.reflect.Method;

import java.net.Socket;

public class **RMIHandler** implements InvocationHandler { private final ObjectOutputStream **out**;

private final ObjectInputStream **in**;

public **RMIHandler**(Socket socket) { try {

this.out = new ObjectOutputStream(socket.getOutputStream()); this.in = new ObjectInputStream(socket.getInputStream());

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

@Override

public Object **invoke**(Object proxy, Method method, Object[] args) throws Throwable {

final MethodCall call = new MethodCall(method, args); this.out.writeObject(call);

return in.readObject();

}

}

Das client-Package enthält nur die Hauptklasse des Clients – die Klasse Client.

Die main-Method erzeugt ein Socket-Objekt. Dann wird ein Proxy erzeugt, dessen Klasse das Interface MathService implementiert. Als InvocationHandler wird diesem Proxy ein RMIHandler übergeben.

Anschließend können dann auf das Proxy die im MathService-Interfaces spezifizierten Methoden aufgerufen werden.

package **client**;

import java.lang.reflect.Proxy; import java.net.Socket;

import common.MathService; import util.RMIHandler;

public class **Client** {

public static void **main**(final String[] args) throws Exception { try(Socket socket = new Socket("localhost", 8000)) {

final MathService service = (MathService)Proxy.newProxyInstance(

ClassLoader.getSystemClassLoader(), new Class<?>[] { MathService.class }, new RMIHandler(socket));

System.out.println(service.sum(40, 2));

System.out.println(service.diff(80, 3));

}

}

}

Die Ausgaben des Servers:

waiting... accepted! reading... reading... reading... EOF!

Und die Ausgaben des Clients:

42

77

## Das Konzept der AOPAlliance

Die APOAlliance (ein Open-Source-Projekt) spezifiziert zwei Interfaces: MethodInterceptor und MethodInvocation. Auf diesen Interfaces baut z.B. das SpringFramework auf.

Im folgenden wird gezeigt, wie diese beiden Interfaces auf den Dynamic-Proxy- Mechanismus abgebildet werden kann.

Wir bauen einen TraceInterceptor, der MethodInterceptor implementiert:

package **interceptors**;

// ...

import org.aopalliance.intercept.MethodInterceptor; import org.aopalliance.intercept.MethodInvocation;

public class **TraceInterceptor** implements MethodInterceptor { @Override

public Object **invoke**(MethodInvocation invocation)

throws Throwable {

final Method method = invocation.getMethod(); final Object[] args = invocation.getArguments(); final String call =

method.getName() + Arrays.toString(args); System.out.println(">> " + call);

try {

final Object result = invocation.proceed(); System.out.println("<< " + call + " --> " + result); return result;

}

catch (final Exception e) {

System.out.println("<< " + call + " --> " + e); throw e;

}

}

}

Der invoke-Method des MethodInterceptor-Interfaces wird ein MethodInvocation- Objekt übergeben. Dieses enthält u.a. ein Method-Objekt und die Aufruf-Parameter (in Form einen Object-Arrays). Mittels der process-Methode von MethodInvocation kann dann der Aufruf weitergeleitet werden an das eigentliche Zielobjekt. Das Resultat des Aufrufs der Zielmethode wird dann auch von invoke zurückgeliefert.

Nun müssen wir das Interface MethodInvocation implementieren:

package **util**;

import java.lang.reflect.AccessibleObject;

import java.lang.reflect.Method; import java.util.function.Supplier;

import org.aopalliance.intercept.MethodInterceptor; import org.aopalliance.intercept.MethodInvocation;

class **MethodInvocationImpl** implements MethodInvocation {

private final Method **method**; private final Object[] **arguments**; private final Object **target**;

private final Supplier<MethodInterceptor> **nextInterceptorSupplier**;

public **MethodInvocationImpl**( Method method, Object[] arguments,

Supplier<MethodInterceptor> nextInterceptoSupplier, Object target) {

this.method = method; this.arguments = arguments; this.target = target;

this.nextInterceptorSupplier = nextInterceptoSupplier;

}

@Override

public Object[] **getArguments**() { return this.arguments;

}

@Override

public Object **proceed**() throws Throwable { final MethodInterceptor nextInterceptor =

this.nextInterceptorSupplier.get(); return nextInterceptor == null

? this.method.invoke(this.target, this.arguments)

: nextInterceptor.invoke(this);

}

@Override

public Object **getThis**() { return this.target;

}

@Override

public AccessibleObject **getStaticPart**() { return this.method;

}

@Override

public Method **getMethod**() { return this.method;

}

}

Die Implementierung soll hier nicht näher erläutert werden.

Schließlich bauen wir eine DynamicProxyFactory:

package **util**;

import java.lang.reflect.Method; import java.lang.reflect.Proxy; import java.util.ArrayList; import java.util.Iterator; import java.util.List;

import java.util.function.Supplier;

import org.aopalliance.intercept.MethodInterceptor; import org.aopalliance.intercept.MethodInvocation;

public class **DynamicProxyFactory**<T> {

private final List<MethodInterceptor> **interceptors** = new ArrayList<>();

private final List<Class<?>> **ifaces** = new ArrayList<>();

private final T **target**;

public **DynamicProxyFactory**(Class<T> mainIFace, T target) { this.target = target;

this.ifaces.add(mainIFace);

}

@SuppressWarnings("unchecked") public T **proxy**() {

final Class<?>[] ifaces =

this.ifaces.toArray(new Class<?>[this.ifaces.size()]); return (T)Proxy.newProxyInstance(

Thread.currentThread().getContextClassLoader(), ifaces,

this::doInvoke);

}

@SuppressWarnings("unchecked")

private T **doInvoke**(Object proxy, Method method, Object[] args) throws Throwable {

if (method.getDeclaringClass() == Object.class) return (T)method.invoke(this.target, args);

final Iterator<MethodInterceptor> iterator = this.interceptors.iterator();

final Supplier<MethodInterceptor> interceptorSupplier = () -> iterator.hasNext() ? iterator.next() : null;

final MethodInterceptor interceptor = interceptorSupplier.get(); if (interceptor == null)

return (T)method.invoke(this.target, args);

final MethodInvocation invocation = new MethodInvocationImpl( method, args, interceptorSupplier, this.target);

return (T)interceptor.invoke(invocation);

}

public DynamicProxyFactory<T> **addMethodInterceptor**( MethodInterceptor interceptor) {

this.interceptors.add(interceptor); return this;

}

public DynamicProxyFactory<T> **addInterface**(Class<?> iface) { this.ifaces.add(iface);

return this;

}

}

Hier einige Beispiel-Anwendungen (natürlich benutzen wir wieder unser Interface

MathService und die Impl-Klasse MathServiceImpl):

static void **demo1**() {

final TraceInterceptor traceInterceptor = new TraceInterceptor();

final MathService mathService =

new DynamicProxyFactory<MathService>( MathService.class, new MathServiceImpl())

.addMethodInterceptor(traceInterceptor)

.proxy(); System.out.println(mathService.sum(40, 2));

System.out.println(mathService.diff(80, 3));

}

Die Ausgaben:

>> sum[40, 2]

<< sum[40, 2] --> 42

42

>> diff[80, 3]

<< diff[80, 3] --> 77

77

static void **demo2**() {

final TraceInterceptor traceInterceptor = new TraceInterceptor();

final MathService mathService =

new DynamicProxyFactory<MathService>( MathService.class, new MathServiceImpl())

.addMethodInterceptor(traceInterceptor)

.addMethodInterceptor(traceInterceptor)

.addInterface(Closeable.class)

.proxy(); System.out.println(mathService.sum(40, 2));

System.out.println(mathService.diff(80, 3));

}

Die Ausgaben:

>> sum[40, 2]

>> sum[40, 2]

<< sum[40, 2] --> 42

<< sum[40, 2] --> 42

42

>> diff[80, 3]

>> diff[80, 3]

<< diff[80, 3] --> 77

<< diff[80, 3] --> 77

77

static void **demo3**() {

final MathService mathService =

new DynamicProxyFactory<MathService>( MathService.class, new MathServiceImpl())

.addMethodInterceptor(invocation -> { System.out.println(invocation.getMethod().getName()); return invocation.proceed();

})

.proxy(); System.out.println(mathService.sum(40, 2));

System.out.println(mathService.diff(80, 3));

}

}

Die Ausgaben:

sum 42

diff 77

# 14 Annotations

Mittels den in Java 5 eingeführten Annotations können die Metadaten einer Klasse durch benutzerdefinierte Metadaten erweitert werden – welche dann zur Laufzeit per Reflection ausgelesen werden können.

(Früher hat man solche Metadaten gern in XML-Dateien hinterlegt – mit der Konsequenz, dass man Glück haben musste, dass diese Daten auch zu den sie beschreibenden Java-Konstrukten passten…)

### Übersicht

* Beispiel: Objekt-relationales Mapping
* Details
* Beispiel: ein Annotations-basiertes Dokumentations-Tool

## Beispiel: OR-Mapping

Als Einstieg dient ein Problem des Object-relationalen Mappings.

Beans sollen in einer Datenbank persistiert werden können. Man kann einen Reflection- basiertes Instrument bauen, welches die Properties einer Bean ausliest, den erforderlichen SQL-INSERT generiert, das PreparedStatement aufbaut und mit den Bean-Daten parametrisiert, und schließlich dieses PreparedStatement ausführt. Ein solches Instrument wird als "Objekt-relationaler Mapper" bezeichnet.

Ein OR-Mapper muss allerdings wissen, in welcher Tabelle Objekte einer bestimmten Klasse gespeichert werden sollen (er muss den Namen der Tabelle kennen); und er muss wissen, auf welche Spalten die Properties der Klasse abgebildet werden müssen (er muss die Property-Namen Spaltennamen zuordnen können). Im einfachsten Falle könnte man natürlich einfach vereinbaren, dass der Klassename identisch ist mit dem Tabellennamen und die Property-Namen identisch sind mit den Spaltennamen. Wenn diese Vereinbarung aber nicht möglich ist (weil die Datenbank bereits existiert und man deren Namen nicht die Java-Welt übernehmen möchte), ist eine explizite Abbildung der Java-Namen auf die Datenbank-Namen erforderlich.

Eine solche Abbildung könnte z.B. in Form einer XML-Datei vorliegen. Oder in Form von Annotations, welche im Java-Quellcode direkt eingebettet werden.

### Die annotierte Klasse

Hier eine beispielhafte Verwendung von Annotations (die Definition dieser Annotations wird später erläutert):

package **appl**;

import java.io.Serializable;

import annotations.Basic; import annotations.Column; import annotations.Entity; import annotations.Id; import annotations.Table;

@Entity

@Table(name = "BUCH")

public class **Book** implements Serializable {

private static final long **serialVersionUID** = 1L; private String **isbn**;

private String **title**; private double **price**;

private String **author**; public **Book**() { }

public **Book**(String isbn, String title, double price, String author) { ... }

@Id

@Column(name = "ISBN")

public String **getIsbn**() { ... }

public void **setIsbn**(String isbn) { ... }

@Basic

@Column(name = "TITEL")

public String **getTitle**() { ... }

public void **setTitle**(String title) { ... }

@Basic

@Column(name = "PREIS")

public double **getPrice**() { ... }

public void **setPrice**(double price) { ... }

@Basic

@Column(name = "AUTOR")

public String **getAuthor**() { ... }

public void **setAuthor**(String author) { ... }

@Override

public String **toString**() { ... }

}

Die Namen von Annotations beginnen jeweils mit einem @. Annotations können in unterschiedlichen Kontexten verwendet werden: z.B. auf Klassenebene, auf Attribut- und auf Methoden-Ebene. Jede Klasse, jedes Feld, jede Methode etc. kann jeweils mehrere Annotations besitzen.

Diese obige Klasse verwendet selbstdefinierte Annotations, welche im annotation- Projekt definiert sind. Bevor diese Definition vorgestellt wird, hier zunächst einige Anmerkungen zum Sinn dieser Annotations.

Die @Entity-Annotation wird (auf Klassenebene) verwendet, um die Klasse als "persistente Klasse" zu annotieren (also als Klasse, deren Elemente persistiert werden können). Es handelt sich hier um eine reine "Dokumentation".

Die @Table- Annotation ermöglicht durch ihr name-Attribut die Definition eines Tabellennames. Objekte vom Typ Book sollen in einer Tabelle namens "BUCH" persistiert werden.

Mit der @Id-Annotation wird diejenige Property ausgestattet, deren entsprechende Tabellenspalte der Primary Key der Tabelle ist.

Alle anderen zu persistierenden Properties werden mit @Basic annotiert.

Schließlich ermöglicht die @Column-Annotation mit ihrem name-Attribut die Zuordnung von einer Property zu einem Spaltennamen. Die Property "price" z.B. soll abgebildet werden auf die Spalte "Preis".

Annotations, welche sich auf Properties beziehen, sollen – per Vereinbarung – jeweils die getter-Methoden der Properties auszeichnen.

### Die Annotations-Definitionen

Zunächst nun zur Definition dieser Annotationen. (Später wird dann gezeigt, wie zur Laufzeit mittels Reflection auf diese zusätzlichen Metadaten zugegriffen werden kann).

Die @Entity-Annotation ist wie folgt implementiert:

package **annotations**;

// ... @Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME) public @interface **Entity** {

}

Eine Annotation wird als "Spezial-Interface" implementiert – als @interface. Das Entity- Interface enthält keine weitere Methoden. Es ist ausgestattet mit zwei "Meta- Annotations" – Annotations, welche eine Annotation auszeichnen:

@Target legt fest, auf welcher Ebene eine Annotation des entsprechenden Typs verwendet werden kann. ElementType.TYPE legt z.B. fest, dass die Annotation nur auf Typ-Ebene (Klassen, Interface- und Enum-Ebene) verwendet werden kann. (Neben ElementType.TYPE kann hier z.B. ElementType.METHOD oder ElementType.FIELD festgelegt werden. Auch ein Kombination von ElementTypes ist möglich – dann müssen die Werte in einer Liste angegeben werden, welche durch geschweifte Klammern begrenzt wird).

@Retention legt fest, was mit einer konkreten Verwendung der Annotation zu geschehen hat. RetentionPolicy.RUNTIME weist den Compiler an, die entsprechende Annotation in die class-Datei zu schreiben – und zwar derart, dass zur Laufzeit per Reflection auf diese Annotation zugegriffen werden kann. RetentionPolicy.CLASS führt dazu, dass die Annotation zwar in der class-Datei landet, zur Laufzeit aber nicht per Reflection zugänglich sein wird. Und RetentionPolicy.SOURCE führt dazu, dass die Annotation nur über den Quelltext zugänglich sein wird – also nur von Instrumenten genutzt werden kann, die sich direkt mit dem Quelltext befassen (z.B. also vom Compiler – siehe @Override!).

Die @Table-Annotation hat dieselben Meta-Annotations wie @Entity – ist aber mit einer "Methode" ausgestattet:

package **annotations**;

// ... @Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME) public @interface **Table** {

String **name**();

}

Weil nun diese Annotation die name-Methode (vom Typ String) besitzt, wird der Compiler bei der Verwendung der Annotation verlangen, dass ein Attribut name angegeben wird – mit einem Wert vom Typ String:

@Table(name="BUCH")

Die @Id-Annotation ist nur auf Methoden-Ebene gültig. Sie besitzt keine Methoden:

package **annotations**;

// ... @Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME) public @interface **Id** {

}

Die @Basic-Annotation ist ebenfalls nur auf Methoden-Ebene gültig;

package **annotations**;

// ... @Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME) public @interface **Basic** {

}

Die @Column-Annotation hat wie auch @Table eine name-Methode. Die Annotation kann nur auf Methoden-Ebene verwendet werden:

package **annotations**;

// ... @Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME) public @interface **Column** {

String **name**();

}

### Reflection-basierter Zugriff

Wie können wir via Reflection zur Laufzeit die jeweils verwendeten Annotations ermitteln?

Wir ermitteln, ob die Klasse Book mit @Entity ausgestattet ist:

static void **demoEntity**() throws Exception { final Class<?> cls = Book.class;

final Entity entity = cls.getAnnotation(Entity.class); if (entity == null)

System.out.println("missing @Entity"); else {

System.out.println(entity.getClass().getName()); System.out.println("@Entity");

}

}

Wir verwenden die Class-Methode getAnnotation. Ihr wird dasjenige Class-Objekt übergeben, welches das Annotations-Interface beschreibt (hier: Entity.class). Die Methode liefert eine Referenz eben dieses Annotations-Typs zurück. Ist die Klasse entsprechend annotiert, wird eine Referenz ungleich null geliefert – ansonsten null.

Die Ausgaben:

com.sun.proxy.$Proxy1 @Entity

Die erste Zeile zeigt, dass das von uns definierte Annotation-Interface offensichtlich von einer zur Laufzeit erzeugten Klasse implementiert wurde (Dynamic-Proxy!) – und uns im positiven Falle dann eine Referenz eben auf ein Objekt dieser Klasse (auf ein Proxy) zurückgeliefert wird.

Wir wollen ermitteln, ob die Klasse Book mit @Table annotiert ist – und wenn ja, welchen Wert das name-Attribut der Annotation besitzt:

static void **demoTable**() throws Exception { final Class<?> cls = Book.class;

final Table table = cls.getAnnotation(Table.class); if (table == null)

System.out.println("missing @Table"); else {

System.out.println(table.getClass().getName()); String name = table.name(); System.out.println("@Table name = " + name);

}

}

Da das @Table-Interface die Methode name spezifiziert, kann eben diese Methode auf das uns von getAnnotation gelieferte Proxy aufgerufen werden.

Die Ausgaben:

com.sun.proxy.$Proxy2 @Table name = BUCH

Ist die Book-Methode getIsbn mit @Id annotiert?:

static void **demoId**() throws Exception { final Class<?> cls = Book.class;

final Method m = cls.getMethod("getIsbn"); final Id id = m.getAnnotation(Id.class); if (id == null)

System.out.println("missing @Id"); else {

System.out.println(id.getClass().getName()); System.out.println("@Id");

}

}

Auch die Klasse Method besitzt eine Methode namens getAnnotation - die genauso funktioniert wie die Class-eigene getAnnoation-Methode. (Und auch die Reflection- Klassen Field, Construktor, Package und Parameter besitzen jeweils eine eigene getAnnotation-Methode.)

Die Ausgaben:

com.sun.proxy.$Proxy3 @Id

Ist die getTitle-Methode mit @Basic annotiert?:

static void **demoBasic**() throws Exception { final Class<?> cls = Book.class;

final Method m = cls.getMethod("getTitle");

final Basic basic = m.getAnnotation(Basic.class); if (basic == null)

System.out.println("missing @Basic"); else {

System.out.println(basic.getClass().getName()); System.out.println("@Basic");

}

}

Die Ausgaben:

com.sun.proxy.$Proxy5 @Basic

Und schließlich: Ist getTitle auch mit @Column annotiert – und welchen Wert hat dann das name-Attribut:

static void **demoColumn**() throws Exception { final Class<?> cls = Book.class;

final Method m = cls.getMethod("getTitle");

final Column column = m.getAnnotation(Column.class); if (column == null)

System.out.println("missing @Column"); else {

System.out.println(column.getClass().getName()); final String name = column.name(); System.out.println("@Column name = " + name);

}

}

Die Ausaben:

com.sun.proxy.$Proxy4 @Column name = TITEL

### Generierung von SQL-Strings

Die Methode demoAlltogether nutzt einige Klassen des util-Packages, die weiter unten vorgestellt werden. Sie erzeugt aufgrund der Klasse Book einige SQL-Strings, die zum INSERT, UPDATE etc. eines Books verwendet werden könnten:

static void **demoAlltogether**() throws Exception { final TableInfo tableInfo =

TableInfoCreator.createTableInfoFor(Book.class);

System.out.println(SQLBuilder.buildInsert(tableInfo)); System.out.println(SQLBuilder.buildUpdate(tableInfo)); System.out.println(SQLBuilder.buildDelete(tableInfo)); System.out.println(SQLBuilder.buildSelect(tableInfo));

}

Die Ausgaben:

insert into BUCH (AUTOR, ISBN, PREIS, TITEL)

values (?, ?, ?, ?)

update BUCH set AUTOR = ?, ISBN = ?, PREIS = ?, TITEL = ?

where ISBN = ?

delete from BUCH where ISBN = ?

select AUTOR, ISBN, PREIS, TITEL from BUCH

Nun zu den util-Klassen.

Ein ColumnInfo-Objekt beschreibt eine Spalte einer Datenbank-Tabelle:

package **util**;

public class **ColumnInfo** {

public final String **name**; public final boolean **isKey**;

public **ColumnInfo**(String name, boolean isKey) { this.name = name;

this.isKey = isKey;

}

}

Ein TableInfo-Objekt beschreibt eine komplette Datenbank-Tabelle (und hat deshalb

u.a. Listen von ColumnInfo-Objekten):

package **util**;

// ...

public class **TableInfo** {

public final String **name**;

public final List<ColumnInfo> **columnInfos**; public final ColumnInfo **keyColumnInfo**;

public final List<ColumnInfo> **nonKeyColumnInfos**;

public **TableInfo**(

String name,

List<ColumnInfo> columnInfos, ColumnInfo keyColumnInfo, List<ColumnInfo> nonKeyColumnInfos) {

this.name = name; this.columnInfos =

Collections.unmodifiableList(columnInfos); this.keyColumnInfo = keyColumnInfo; this.nonKeyColumnInfos =

Collections.unmodifiableList(nonKeyColumnInfos);

}

}

Die folgende Klasse erzeugt aufgrund einer Entity-Klasse ein TableInfo-Objekt (das genaue Studium sei dem Leser / der Leserin überlassen):

package **util**;

// ...

public class **TableInfoCreator** {

public static TableInfo **createTableInfoFor**(Class<?> cls) throws Exception {

checkIfClassIsEntityClass(cls);

final String tableName = getTableName(cls);

final BeanInfo beanInfo = Introspector.getBeanInfo(cls, Object.class); final PropertyDescriptor[] descriptors =

beanInfo.getPropertyDescriptors();

final List<ColumnInfo> columnInfos = new ArrayList<>(); ColumnInfo keyColumnInfo = null;

for (final PropertyDescriptor descriptor : descriptors) { final ColumnInfo columnInfo = createColumnInfo(descriptor); if (columnInfo == null)

continue; columnInfos.add(columnInfo); if (columnInfo.isKey)

keyColumnInfo = columnInfo;

}

if (keyColumnInfo == null)

throw new RuntimeException("missing key column"); final List<ColumnInfo> nonKeyColumnInfos =

createNonKeyColumnList(columnInfos, keyColumnInfo); return new TableInfo(

tableName, columnInfos, keyColumnInfo, nonKeyColumnInfos);

}

private static List<ColumnInfo> **createNonKeyColumnList**( final List<ColumnInfo> columnInfos,

ColumnInfo keyColumnInfo) {

final List<ColumnInfo> nonKeyColumnInfos = new ArrayList<>(); nonKeyColumnInfos.addAll(columnInfos); nonKeyColumnInfos.remove(keyColumnInfo);

return nonKeyColumnInfos;

}

private static ColumnInfo **createColumnInfo**( PropertyDescriptor descriptor) {

final Method getter = descriptor.getReadMethod(); if (getter == null)

return null;

final Id id = getter.getAnnotation(Id.class);

final Basic basic = getter.getAnnotation(Basic.class); if (id == null && basic == null)

return null;

final Column column = getter.getAnnotation(Column.class); final String columnName =

column != null ? column.name() : descriptor.getName(); return new ColumnInfo(columnName, id != null);

}

private static String **getTableName**(Class<?> cls) { final Table table = cls.getAnnotation(Table.class); final String tableName =

table != null ? table.name() : cls.getSimpleName(); return tableName;

}

private static void **checkIfClassIsEntityClass**(Class<?> cls) { final Entity entity = cls.getAnnotation(Entity.class);

if (entity == null)

throw new RuntimeException("@Entity-Annotation missing");

}

}

Und hier schließlich die Klasse SQLBuilder, deren Methoden aufgrund eine TableInfo- Objekts jeweils einen SQL-String erzeugen:

package **util**;

// ...

public class **SQLBuilder** {

public static String **buildInsert**(TableInfo tableInfo) { final StringBuilder builder = new StringBuilder(); builder.append("insert into "); builder.append(tableInfo.name);

builder.append(" ("); builder.append(join(tableInfo.columnInfos, ", ",

columnInfo -> columnInfo.name)); builder.append(") values ("); builder.append(join(tableInfo.columnInfos, ", ",

columnInfo -> "?"));

builder.append(")"); return builder.toString();

}

public static String **buildUpdate**(TableInfo tableInfo) { final StringBuilder builder = new StringBuilder(); builder.append("update "); builder.append(tableInfo.name);

builder.append(" set "); builder.append(join(tableInfo.columnInfos, ", ",

columnInfo -> columnInfo.name + " = ?")); builder.append(" where "); builder.append(tableInfo.keyColumnInfo.name); builder.append(" = ?");

return builder.toString();

}

public static String **buildDelete**(TableInfo tableInfo) { final StringBuilder builder = new StringBuilder(); builder.append("delete from "); builder.append(tableInfo.name);

builder.append(" where "); builder.append(tableInfo.keyColumnInfo.name); builder.append(" = ?");

return builder.toString();

}

public static String **buildSelect**(TableInfo tableInfo) { final StringBuilder builder = new StringBuilder();

builder.append("select "); builder.append(join(tableInfo.columnInfos, ", ",

columnInfo -> columnInfo.name)); builder.append(" from "); builder.append(tableInfo.name); return builder.toString();

}

private static <T> String **join**(

List<T> list, String delimiter, Function<T, String> extractor) { return list.stream().map(extractor).collect(

Collectors.joining(delimiter));

}

}

## Mehr zu Annotationen

Im folgenden werden die Möglichkeiten, die Annotationen bieten, etwas genauer betrachtet.

### value

Sei folgende Annotation gegeben:

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.FIELD)

public @interface **Bar** {

public abstract String **value**();

}

Die Annotation hat eine einzige Methode namens value. Bei der Verwendung einer solchen Annotation kann dann der Name der Methode entfallen:

public class **Foo** { @Bar(value = "hello") public int **alpha**;

@Bar("world") public int **beta**;

}

value() liefert bei alpha den Wert "hello", bei beta den Wert "world". Der Reflection-basierte Zugriff (bezogen auf das alpha-Feld):

Bar bar = Foo.class.getField("alpha").getAnnotation(Bar.class); if (bar != null) {

String value = bar.value();

// ...

}

### default

Eine Methode einer Annotation kann mit default qualifiziert sein:

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.FIELD)

public @interface **Bar** {

public abstract int **magic**() default 42;

}

Eine solche Methode kann mit oder ohne ein entsprechenden Attributwert verwendet werden:

public class **Foo** { @Bar(magic = 77) public int **alpha**;

@Bar

public int **beta**;

}

Die @Bar-Annotation von beta hat nun implizit den Wert 42. Der Reflection-basierte Zugriff (bezogen auf alpha):

Bar bar = Foo.class.getField("alpha").getAnnotation(Bar.class); if (bar != null) {

int magic = bar.magic();

// ...

}

### PARAMETER

Annotationen können auch zur Auszeichnung von Parametern verwendet werden:

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.PARAMETER) public @interface **Bar** {

public abstract int **magic**() default 42;

}

Diese Annotation kann wie folgt verwendet werden:

public class **Foo** {

public void **alpha**(@Bar(magic = 42) int arg) { System.out.println(arg);

}

}

Der Reflection-basierte Zugriff:

Bar bar = Foo.class.getMethod("alpha", int.class)

.getParameters()[0].getAnnotation(Bar.class); if (bar != null) {

int magic = bar.magic();

// ...

}

### Annotationen als Attribute von Annoationen

Eine Annotation kann sich auf andere Annotationen beziehen:

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.FIELD)

public @interface **Bar** {

public abstract int **magic**();

}

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.FIELD)

public @interface **Bars** {

public abstract Bar[] **entries**();

}

Die beiden Annotationen können wie folgt verwendet werden:

public class **Foo** { @Bars(entries = {

@Bar(magic = 42),

@Bar(magic = 77)

})

public int **alpha**;

@Bar(magic = 66) public int **beta**;

}

Der Reflection-basierte Zugriff (bez. des alpha-Feldes):

final Bars bars = Foo.class.getField("alpha").getAnnotation(Bars.class);

if (bars != null) { System.out.println("@Bars"); Bar[] barArray = bars.entries(); for (Bar bar : barArray)

System.out.println("\t@Bar magic = " + bar.magic());

}

Die Ausgaben:

@Bars

@Bar magic = 42 @Bar magic = 77

### Repeatable

@Bars enthält eine value-Methode vom Typ Bar[] (wichtig ist, dass die Methode value heißt – es ist daher ausgeschlossen, dass die Annotation noch weitere Methode enthält):

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.FIELD)

public @interface **Bars** {

public abstract Bar[] **value**();

}

Die @Bar-Annotation kann mit @Repeatable annotiert sein:

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.FIELD) @Repeatable(Bars.class)

public @interface **Bar** {

public abstract int **magic**();

}

Nun kann @Bar für ein und dasselbe Element mehrfach verwendet werden (ohne explizit die @Bars-Annotation mit value() zu nutzen):

public class **Foo** { @Bars(value = {

@Bar(magic = 42),

@Bar(magic = 77)

})

public int **alpha**;

@Bar(magic = 42)

@Bar(magic = 77) public int **beta**;

}

Der Reflection-basierte Zugriff ist derselbe wie im letzten Abschnitt (er beginnt mit dem Zugriff auf @Bars).

### Zugriff auf alle Annotationen eines Elements

Seien zwei Annotations-Definitionen gegeben:

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.FIELD)

public @interface **Bar1** {

public abstract int **magic**();

}

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.FIELD)

public @interface **Bar2** {

public abstract String **name**();

}

Und folgende Klasse, deren alpha-Feld sowohl mit @Bar1 als auch mit @Bar2 annotiert ist:

public class **Foo** { @Bar1(magic = 42) @Bar2(name = "Hello") public int **alpha**;

}

Mittels der Field-Methode getAnnoations (Plural!) können alle Annotationen eines Feldes ermittelt werden (diese Plural-Methode gibt's natürlich auch bei Class, Method, Constructor etc.):

Annotation[] annotations = Foo.class.getField("alpha").getAnnotations();

for (Annotation annotation : annotations) {

Class<? extends Annotation> type = annotation.annotationType(); System.out.println(type);

if (type == Bar1.class)

System.out.println("\t" + ((Bar1)annotation).magic()); else if (type == Bar2.class)

System.out.println("\t" + ((Bar2)annotation).name());

}

Die Ausgaben:

interface appl6.Bar1 42

interface appl6.Bar2 Hello

Wie man sieht, sind alle Annotations-Interfaces implizit von Annotation abgeleitet – ähnlich, wie auch alle enum-Definitionen von Enum abgeleitet sind.

### Vererbung

Annotationen werden nur dann "vererbt", wenn sie als @Inherited markiert sind. Seien folgende zwei Annotations-Definitionen gegeben:

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.TYPE)

public @interface **Bar1** {

public abstract int **magic**();

}

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.TYPE) @Inherited

public @interface **Bar2** {

public abstract String **name**();

}

Bar2 ist im Unterschied zu Bar1 mit @Inherited annotiert. Sei dann folgende Basisklasse gegeben:

@Bar1(magic = 42) @Bar2(name = "Hello") public class **Foo** {

}

Und folgende von Foo abgeleitete Klasse:

public class **FooExt** extends Foo {

}

FooExt hat nur zwar die Bar2-Annotation (von Foo "geerbt"), nicht aber die Bar1- Annoation:

Bar1 bar1 = FooExt.class.getAnnotation(Bar1.class); System.out.println(bar1);

Bar2 bar2 = FooExt.class.getAnnotation(Bar2.class); System.out.println(bar2);

Die Ausgaben:

null @appl7.Bar2(name="Hello")

### Abschließendes

Als Werte für Attribute von Annotationen dürfen nur solche Elemente verwendet werden, welche zur Kompilations-Zeit bekannt sind:

* Literale primitiver Typen (42, 3.14)
* String-Literate ("Hello")
* Class-Literale (z.B. Foo.class)
* Annotationen
* statische finale Variablen, deren Initialisierungswert zur Kompilationszeit feststeht.

Was die letzte Möglichkeit angeht, hier ein Beispiel:

@Bar(name = Foo.NAME) class **Foo** {

public static final String **NAME** = "Hase";

...

}

## Ein abschließendes Beispiel: Ein Dokumentations-Tool

Sei folgendes Interface gegeben:

package **appl**;

import annotations.Doc; import annotations.MethodDoc; import annotations.Parameter;

@Doc("a stateless Mathematician") public interface **MathService** {

@MethodDoc(text = "calculates the sum of x and y", result = "sum of x and y",

parameters = {

@Parameter(name = "x", description = "some value"), @Parameter(name = "y", description = "some value")

}

)

public abstract double **sum**(double x, double y);

@MethodDoc(text = "calculates the differecne of x and y", result = "difference of x and y",

parameters = {

@Parameter(name = "x", description = "some value"), @Parameter(name = "y", description = "some value")

}

)

public abstract double **diff**(double x, double y);

}

Die hier verwendeten Annotationen dienen offenbar dazu, eine Klasse oder ein Interface zu dokumentieren (und zwar so, dass der Compiler die Dokumentation auf formale Richtigkeit hin prüfen kann – was bei JavaDoc bekanntlich nicht der Fall ist…)

Hier die verwendeten Annotationen:

package **annotations**;

// ... @Retention(RetentionPolicy.RUNTIME)

@Target({ ElementType.FIELD, ElementType.TYPE }) public @interface **Doc** {

String **value**();

}

package **annotations**;

// ... @Retention(RetentionPolicy.RUNTIME) @Target(ElementType.ANNOTATION\_TYPE)

public @interface **Parameter** {

String **name**();

String **description**();

}

package **annotations**;

// ... @Retention(RetentionPolicy.RUNTIME) @Target(ElementType.METHOD)

public @interface **MethodDoc** { String **text**();

String **result**() default ""; Parameter[] parameters();

}

Und hier die Klasse DocTool (ein Doc-Generator), welche zur Laufzeit diese Annotationen auswertet:

package **util**;

// ...

public class **DocTool** {

public static void **document**(Class<?> cls) { documentType(cls);

for (final Field field : cls.getFields()) documentField(field);

for (final Method method : cls.getMethods()) documentMethod(method);

}

private static void **documentType**(Class<?> cls) { final Doc doc = cls.getAnnotation(Doc.class); System.out.println(cls.getName());

if (doc == null) return;

System.out.println("\t" + doc.value());

}

private static void **documentField**(Field field) { System.out.println(field.getName());

final Doc d = field.getAnnotation(Doc.class); if (d == null)

return;

System.out.println("\t" + d.value());

}

private static void **documentMethod**(Method method) { System.out.println(method.getName());

final MethodDoc d = method.getAnnotation(MethodDoc.class); if (d == null)

return;

System.out.println("\t" + d.text()); System.out.println("\tResult:"); System.out.println("\t\t" + d.result()); System.out.println("\tParameters:");

final Parameter[] parameters = d.parameters();

for (final Parameter p : parameters) {

System.out.println("\t\t" + p.name() + " : " + p.description());

}

}

}

Das DocTool kann wie folgt verwendet werden:

DocTool.document(MathService.class);

Die beispielhaften Ausgaben:

appl.MathService

a stateless Mathematician sum

calculates the sum of x and y Result:

sum of x and y Parameters:

x : some value y : some value

diff

calculates the differecne of x and y Result:

difference of x and y Parameters:

x : some value y : some value

# 15 Serialisierung - Erweiterungen

### Voraussetzungen

* + - Serialisierung einfacher Objekte
    - Das Interface Serializable
    - Die Klassen ObjectInputStream und ObjectOutputStream
    - Die Bedeutung des statischen Attributs serialVersionUID

Objekte, deren Klassen das Marker-Interface java.io.Serializable implementieren, können bekanntlich mittels eines ObjectOutputStreams serialisiert und mittels eines ObjectInputStreams deserialisiert werden.

Dabei spielt es keine Rolle, ob es sich um "einfache" Objekte oder um komplette "Objekt-Wolken" handelt.

Zum Thema Serialisierung gibt's allerdings noch etwas mehr zu sagen.

### Übersicht

* + - Utilitiy-Methoden
    - Serialisierung komplexer "Objektwolken"
    - Vererbung und Serialisierung
    - Transiente und statische Attribute
    - readObject und writeObjekt
    - readResolve und writeReplace
    - Serialisierung von Class-, Method-, Field- und Constructor-Objekten

## Utilities

Im Folgenden stellen wir eine Serializer-Klasse vor, die in den folgenden Abschnitten immer wieder genutzt wird (die Klasse ist im shared-Projekt hinterlegt).

Als Beispiel für eine Klasse, deren Objekte serialisierbar sind, benutzen wir die Klasse

Book:

package **appl**;

import java.io.Serializable;

public class **Book** implements Serializable { private String **isbn**;

private String **title**;

public **Book**() {

System.out.println(Book.class.getSimpleName() + ".<init>()");

}

public **Book**(String isbn, String title) { System.out.println(Book.class.getSimpleName() +

".<init>(" + isbn + ", " + title + ")"); this.isbn = isbn;

this.title = title;

}

public String **getIsbn**() { ... ]

public void **setIsbn**(String isbn) { ... ]

public String **getTitle**() { ... ]

public void **setTitle**(String title) { ... ]

@Override

public String **toString**() { ... ]

}

Zunächst zeigen wir, wie Book-Objekte ohne Zuhilfenahme der Utility-Klasse serialisiert / deserialisiert werden können. Wir zeigen zwei Varianten: die File-basierte Variante und die Byte-Array basierte Variante.

### Variante 1

Wir erzeugen einen ObjectOutputStream (also einen Serialisierer), der seine Ausgaben in einen FileOutputStream schreibt. Mittels dieses Serialisierers serialisieren wir ein Book-Objekt (mittels der Methode writeObject).

Anschließend erzeugen wir einen ObjectInputStream (einen Deserialisierer), der seine Ausgaben aus einem FileInputStream liest (der seine Daten aus genau derjenigen Datei liest, in welcher der FileOutputStream die Daten geschrieben hat). Der Aufruf von readObject wird ein neues Book zurückliefern, dessen Zustand derselbe ist wie der Zustand des zuvor serialisierten Books:

static void **demoFileWithoutUtils**() throws Exception {

try (final ObjectOutputStream out = new ObjectOutputStream( new FileOutputStream("zzz.ser"))) {

final Book book = new Book("1111", "Pascal"); out.writeObject(book);

}

System.out.println("<-->");

try (final ObjectInputStream in = new ObjectInputStream( new FileInputStream("zzz.ser"))) {

final Book book = (Book) in.readObject(); System.out.println(book);

}

}

Die Ausgaben zeigen, dass bei der Deserialisierung weder der parametrisierte noch der parameterlose Konstruktor von Book aufgerufen wird (hierzu später mehr):

Book.<init>(1111, Pascal)

<-->

Book [1111, Pascal]

### Variante 2

Statt eines FileOutputStreams benutzten wird nun einen ByteArrayOutputStream (wir serialisieren also in eine Hauptspeicher-Struktur) – und statt eines FileInputStreams einen ByteArrayInputStream:

static void **demoByteArrayWithoutUtils**() throws Exception { final ByteArrayOutputStream ostream =

new ByteArrayOutputStream(); try (final ObjectOutputStream out =

new ObjectOutputStream(ostream)) {

final Book book = new Book("1111", "Pascal"); out.writeObject(book);

}

System.out.println("<-->");

final ByteArrayInputStream istream =

new ByteArrayInputStream(ostream.toByteArray()); try (final ObjectInputStream in =

new ObjectInputStream(istream)) { final Book book = (Book) in.readObject(); System.out.println(book);

}

}

Die Ausgaben sind dieselben wie beim Aufruf der ersten demo-Methode.

Beide Varianten können einfacher formuliert werden – mit Hilfe von Methoden der Utilitiy-Klasse Serializer.

### Variante 1

static void **demoFileWithUtils**() throws Exception { final Book book1 = new Book("1111", "Pascal"); Serializer.serialize(book1, "zzz.ser"); System.out.println("<-->");

final Book book2 = Serializer.deserialize("zzz.ser"); System.out.println(book2);

}

Der serialize-Methode wird das zu serialisierende Objekt und ein Dateiname übergeben; der deserialize-Methode wird ein Dateiname übergeben – und sie liefert das deserialisierte Objekt zurück.

### Variante 2

static void **demoByteArrayWithUtils**() throws Exception { final Book book1 = new Book("1111", "Pascal"); final byte[] array = Serializer.serialize(book1); System.out.println("<-->");

final Book book2 = Serializer.deserialize(array); System.out.println(book2);

}

Diese Variante nutzt überlade Varianten der serialize- resp. deserialize-Methoden. Der serialize-Methode wird nur das zu serialisierende Objekt übergeben – sie liefert eine byte-Array zurück (der das serialisierte Objekt enthält). Der deserialize-Methode wird dieses byte-Array übergeben – und sie liefert das deserialisierte Objekt zurück.

In den folgenden Demo-Anwendungen geht's immer darum, das Resultat einer Serialisierung sofort wieder zu deserialisieren. Hierzu bietet die Serializer-Klasse eine Methode namens clone. Diese kombiniert die Aufrufe der serialize- und deserialize-Methoden (diejenigen der byte-Array-Variante):

static void **demoClone**() throws Exception {

final Book book1 = new Book("1111", "Pascal"); Book book2 = Serializer.clone(book1); System.out.println(book2);

}

Auch hier sind die Ausgaben dieselben wie bei den anderen demo-Methoden. Genau diese clone-Methode wird im folgenden immer wieder genutzt werden. Hier die Klasse Serializer:

package **jn.util.serialization**;

// ...

public class **Serializer** {

public static void **serialize**(Object obj, String filename) {

try (final ObjectOutputStream out = new ObjectOutputStream( new FileOutputStream(filename))) {

out.writeObject(obj);

}

catch (IOException e) {

throw new RuntimeException(e);

}

}

public static <T> T **deserialize**(String filename) {

try (final ObjectInputStream in = new ObjectInputStream( new FileInputStream(filename))) {

return (T) in.readObject();

}

catch (IOException | ClassNotFoundException e) { throw new RuntimeException(e);

}

}

public static byte[] **serialize**(Object obj) { try (final ByteArrayOutputStream stream =

new ByteArrayOutputStream()) { try (final ObjectOutputStream out =

new ObjectOutputStream(stream)) { out.writeObject(obj);

return stream.toByteArray();

}

}

catch (IOException e) {

throw new RuntimeException(e);

}

}

public static <T> T **deserialize**(byte[] bytes) { try (final ByteArrayInputStream stream =

new ByteArrayInputStream(bytes)) { try (ObjectInputStream in =

new ObjectInputStream(stream)) { return (T) in.readObject();

}

}

catch (IOException | ClassNotFoundException e) { throw new RuntimeException(e);

}

}

public static <T> T **clone**(T obj) { byte[] bytes = serialize(obj); System.out.println("<-->"); return deserialize(bytes);

}

}

## Serialisierung von "Objektwolken"

Im Folgenden zeigen wir, dass auch komplexe "Objektwolken" serialisierbar sind (und dass dasselbe auch für Collections gilt).

Ein Author ist Serializable. Er hat einen Namen:

package **appl**;

import java.io.Serializable;

public class **Author** implements Serializable { private String **name**;

public **Author**() {

System.out.println(Author.class.getSimpleName() + ".<init>()");

}

public **Author**(String name) { System.out.println(Author.class.getSimpleName() + ".<init>("

+ name + ")"); this.name = name;

}

public String **getName**() { ... }

public void **setName**(String name) { ... }

@Override

public String **toString**() { ... }

}

Ein Book ist ebenfalls Serializable. Es hat eine Isbn-Nummer, einen Titel und eine Referenz auf einen Author:

package **appl**;

import java.io.Serializable;

public class **Book** implements Serializable { private String **isbn**;

private String **title**;

private Author **author**;

public **Book**() {

System.out.println(Book.class.getSimpleName() + ".<init>()");

}

public **Book**(String isbn, String title, Author author) { System.out.println(Book.class.getSimpleName() + ".<init>("

+ isbn + ", " + title + ", " + author + ")");

this.title = title; this.author = author;

}

public String **getIsbn**() { ... }

public void **setIsbn**(String title) { ... }

public String **getTitle**() { ... }

public void **setTitle**(String title) { ... }

public Author **getAuthor**() { ... }

public void **setAuthor**(Author author) { ... }

@Override

public String **toString**() { ... }

}

Wir erzeugen ein Book mit einem Author – und serialisieren / deserialisieren das Book (wobei nicht nur das Book serialisiert wird, sondern auch der mit dem Book assoziierte Author):

static void **demoBookWithAuthor**() throws Exception {

final Book book1 = new Book("1111", "Pascal", new Author("Wirth")); Book book2 = Serializer.clone(book1);

System.out.println(book2);

}

Die Ausgaben:

Author.<init>(Wirth)

Book.<init>(1111, Pascal, Author [Wirth])

<-->

Book [1111, Pascal, Author [Wirth]]

Wir erzeugen eine Liste von Books (die jeweils mit einem Author assoziiert sind) – und serialisieren / deserialisieren diese Liste. Das Beispiel zeigt, dass auch ArrayLists serialisierbar sind (und dasselbe gilt auch für alle anderen Collections):

static void **demoListOfBooksWithAuthors**() throws Exception { final List<Book> bookList1 = new ArrayList<>();

bookList1.add(new Book("1111", "Pascal", new Author("Wirth"))); bookList1.add(new Book("2222", "Eiffel", new Author("Meyer"))); List<Book> bookList2 = Serializer.clone(bookList1); bookList2.forEach(System.out::println);

}

Die Ausgaben:

Author.<init>(Wirth)

Book.<init>(1111, Pascal, Author [Wirth]) Author.<init>(Meyer)

Book.<init>(2222, Eiffel, Author [Meyer])

<-->

Book [1111, Pascal, Author [Wirth]] Book [2222, Eiffel, Author [Meyer]]

## Vererbung

Eine Klasse, die von einer serialisierbaren Klasse abgeleitet ist, ist natürlich ihrerseits ebenfalls serialisierbar (auch dann, wenn sie ihre Serialisierbarkeit nicht mittels implements Serializable ankündigt). Bei der Serialisierung eines Objekts einer abgeleiteten Klassen werden natürlich auch alle Felder serialisiert, die in der Basisklasse definiert sind.

Foo ist eine Basisklasse, die Serializable ist:

package **appl**;

// ...

public class **Foo** implements Serializable { public int **alpha**;

public **Foo**() { System.out.println(Foo.class.getSimpleName() +

".<init>()"); this.alpha = 9999;

}

public **Foo**(int alpha) { System.out.println(Foo.class.getSimpleName() +

".<init>(" + alpha + ")"); this.alpha = alpha;

}

@Override

public String **toString**() { ... }

}

ExtendedFoo ist von Foo abgeleitet (und implementiert noch einmal Serializable – obgleich das natürlich nicht nötig wäre):

package **appl**;

// ...

public class **ExtendedFoo** extends Foo implements Serializable { public String **beta**;

public **ExtendedFoo**(int alpha, String beta) { super(alpha); System.out.println(ExtendedFoo.class.getSimpleName() +

".<init>(" + alpha + ", " + beta + ")"); this.beta = beta;

}

@Override

public String **toString**() { ... }

}

Wir serialisieren / deserialisieren ein ExtendedFoo-Objekt:

static void **demoFoo**() throws Exception {

final ExtendedFoo foo1 = new ExtendedFoo(42, "Hello"); System.out.println();

final ExtendedFoo foo2 = Serializer.clone(foo1); System.out.println(foo2);

}

Die Ausgaben zeigen, dass bei der Deserialisierung weder ein Konstruktor der ExtendedFoo-Klasse noch ein Konstruktor der Basisklasse Foo aufgerufen wird – und sie zeigen, dass bei der Serialisierung sowohl die in der Basisklasse definierten Daten als auch die in der abgeleiteten Klasse definierten Daten serialisiert werden:

Foo.<init>(42) ExtendedFoo.<init>(42, Hello)

<-->

ExtendedFoo [42, Hello]

Die Serialisierung und die Deserialisierung geschieht mittels nativer Methoden. Mittels solcher Methoden lassen sich Objekte erzeugen, ohne dass bei dieser Erzeugung ein Konstruktor (nicht einmal der parameterlose!) aufgerufen wird.

Was passiert nun aber, wenn die Basisklasse nicht(!) Serializable ist? Die Basisklasse sei Bar:

package **appl**;

// ...

public class **Bar** { // NOT Serialiable ! public int **alpha**;

public **Bar**() { System.out.println(Bar.class.getSimpleName() +

".<init>()"); this.alpha = 9999;

}

public **Bar**(int alpha) { System.out.println(Bar.class.getSimpleName() +

".<init>(" + alpha + ")"); this.alpha = alpha;

}

@Override

public String **toString**() { ... }

}

Man beachte, dass der parameterlose Konstruktor das alpha-Feld mit 9999 initialisiert.

Von Bar sei ExtendedBar abgeleitet – wobei diese Klasse nun Serializable ist:

package **appl**;

// ...

public class **ExtendedBar** extends Bar implements Serializable { public String **beta**;

public **ExtendedBar** (int alpha, String beta) { super(alpha); System.out.println(ExtendedBar.class.getSimpleName() +

".<init>(" + alpha + ", " + beta + ")"); this.beta = beta;

}

@Override

public String **toString**() { ... }

}

Wir serialisieren / deserialisieren ein ExtendedBar-Objekt:

static void **demoBar**() throws Exception {

final ExtendedBar bar1 = new ExtendedBar(77, "World"); System.out.println();

final ExtendedBar bar2 = Serializer.clone(bar1); System.out.println(bar2);

}

Die Ausgaben zeigen, dass bei der Deserialisierung zwar kein Konstruktor von ExtendedBar aufgerufen wird, wohl aber der parameterlose Konstruktor der Basisklasse Bar - und dass die in der Basisklasse definierten Daten nicht(!) serialisiert werden:

Bar.<init>(77) ExtendedBar.<init>(77, World)

<-->

Bar.<init>()

ExtendedBar [9999, World]

Das deserialisierte Objekts hat also dieselben Basisdaten, die ein Bar-Objekt hätte, wenn es mittels des parameterlosen Konstruktors initialisiert worden wäre.

Hätte die Basisklasse Bar keinen parameterlosen Konstruktor, würde also die Deserialisierung scheitern.

## Statische und transiente Attribute

Im Folgenden zeigen wir, was es mit statischen und transienten Attributen bezüglich der Serialisierung / Deserialisierung auf sich hat.

Die folgende Foo-Klasse definiert drei Attribute: eine Instanzvariable (instanceVar), eine statische (also globale) Variable staticVar und eine weitere Variablen transientVar, die mit dem Schlüsselwort transient ausgezeichnet ist:

package **appl**;

// ...

public class **Foo** implements Serializable {

public String **instanceVar**; public static String **staticVar**;

public transient String **transientVar**;

public **Foo**(String instanceVar, String transientVar) { this.instanceVar = instanceVar; this.transientVar = transientVar;

}

@Override

public String **toString**() { ... }

}

Wir serialisieren ein Foo-Objekt. Dann ändern wir den Inhalt der statischen Variable

staticVar. Und deserialisieren das Foo-Objekt:

static void **demo**() {

Foo.staticVar = "alpha";

final Foo foo1 = new Foo("beta", "gamma");

System.out.println("Foo.staticVar = " + Foo.staticVar); System.out.println("foo1.instanceVar = " + foo1.instanceVar); System.out.println("foo1.transientVar = " + foo1.transientVar); System.out.println();

final byte[] bytes = Serializer.serialize(foo1); Foo.staticVar = "ALPHA";

final Foo foo2 = Serializer.deserialize(bytes);

System.out.println("Foo.staticVar = " + Foo.staticVar); System.out.println("foo2.instanceVar = " + foo2.instanceVar); System.out.println("foo2.transientVar = " + foo2.transientVar);

Die Ausgaben:

Foo.staticVar = alpha foo1.instanceVar = beta foo1.transientVar = gamma

<-->

Foo.staticVar = ALPHA foo2.instanceVar = beta foo2.transientVar = null

Die statische Variable bleibt von der Deserialisierung unberührt.

Und der Wert von transientVar ist null – dieses Attribut hat an der Serialisierung / Deserialisierung also nicht teilgenommen (obwohl auch transientVar im Gegensatz zu staticVar eine Instanzvariable ist).

Mit transient sollten also alle Instanzvariablen gekennzeichnet sein, die auf Objekte zeigen, welche nicht(!) serialisierbar sind (etwa: Referenzen auf Datenbank- Connections oder auf irgendwelche Input-/OutputStreams…)

## writeObject / readObject

Wir können in den Ablauf der Serialisierung und Deserialisierung eingreifen.

In der Klasse der zu serialisiernden Objekt können wir zwei Methoden implementieren: writeObject und readObject. Keine der beiden Methoden ist in einem Interface spezifiziert (Serializable ist bekanntlich ein reines Marker-Interface).

Der Deserialisierer ermittelt zur Laufzeit, ob diese Methoden vorhanden sind. Sofern writeObject implementiert ist, wird diese bei der Serialisierung auf das zu serialisierende Objekt aufgerufen – und zwar mit einem ObjectOutputSteam. Sofern readObject implementiert ist, wird sie bei der Deserialisierung aufgerufen (wobei ihr der ObjectInputStream übergeben wird).

Es liegt dann komplett in Verantwortung dieser beiden Methoden, wie die jeweiligen Objekte serialisiert resp. deserialisiert werden.

Ein Beispiel:

package **appl**;

// ...

public class **Foo** implements Serializable { public int **value**;

public **Foo**(int value) { this.value = value;

}

private void **writeObject**(ObjectOutputStream out) throws IOException { System.out.println(Foo.class.getSimpleName() + ".writeObject(...)"); out.writeInt(2 \* this.value);

}

private void **readObject**(ObjectInputStream in) throws IOException, ClassNotFoundException {

System.out.println(Foo.class.getSimpleName() + ".readObject(...)"); this.value = in.readInt() / 2;

}

}

Bei der Serialisierung der Foo-Klasse wird nicht der value serialisiert, sondern das Doppelte des values – um bei der Deserialisierung wieder die Hälfte des im Serialisierung-Strom enthaltenen Wertes ermittelt und dem value-Attribut zugewiesen wird.

Die Serialisierung hat also das Objekt "verschlüsselt", die Deserialisierung musste es wieder "entschlüsselt".

Der verschlüsselte Wert kann dann einfach mittels writeInt geschrieben werden; der zu entschlüssende Wert kann via readInt gelesen werden.

static void **demo**() {

Foo foo1 = new Foo(42);

Foo foo2 = Serializer.clone(foo1); System.out.println(foo2.value);

}

Die Ausgaben zeigen, dass readObject und writeObject tatsächlich aufgerufen werden – und das die Ver- und Entschlüsselung funktioniert hat:

Foo.writeObject(...) Foo.readObject(...) 42

## readResolve

Die Deserialisierung kann zuweilen merkwürdige Resultate liefern. Betrachten wir folgende Singleton-Klasse Universe:

package **appl1**;

// ...

public class **Universe** implements Serializable {

public static final Universe **instance** = new Universe(); private **Universe**() {

}

}

Und folgende Serialiserung / Deserialsierung des Universums:

static void **demo**() {

Universe u = Serializer.clone(Universe.instance); System.out.println(u == Universe.instance);

}

Die Ausgabe zeigt, dass nach der Deserialisierung zwei Universe-Objekte existieren – das "eigentliche" Universum, das bereits vor der Serialisierung existierte (dasjenige, welches in der Universe-Klasse erzeugt wurde), und ein weiteres, das als Resultat der Deserialiserung das Licht der Welt erblickt hat:

false

Wir benötigen einen Mechanismus, dass Universe-Objekt, welches bei Deserialsierung erzeugt wurde, durch "eigentliche" Universum zu ersetzen – so dass ersteres erst gar nicht das Licht der Welt erblickt (und beim nächsten GC wieder eingesammelt werden kann. Hierzu implementieren wir eine readResolve-Methode (die ebenso wenig in einem Interface spezifiziert ist wie readObject resp. writeObject):

package **appl2**;

// ...

public class **Universe** implements Serializable {

public static final Universe **instance** = new Universe(); private **Universe**() {

}

private Object **readResolve**() throws ObjectStreamException { System.out.println(Universe.class.getSimpleName() +

".readResolve() : " + (this == Universe.instance)); return Universe.instance;

}

}

Wir serialisieren und deserialisieren:

static void **demo**() {

Universe u = Serializer.clone(Universe.instance); System.out.println(u == Universe.instance);

}

Die Ausgaben zeigen zunächst, dass readResolve tatsächlich aufgerufen wurde – und dass die Deserialisierung dann genau dasjenige Objekt liefert, welches readResolve zurückgeliefert hat:

Universe.readResolve() : false true

Wie verhält es sich, wenn das Universe-Singleton in Form eines einwertigen enums definiert wird:

package **appl3**;

public enum **Universe** {

**instance**

}

Die Ausgabe liefert dann true - d.h.: ohne unser Zutun ist ein readResolve erfolgt. Bei der Serialisierung von enums gibt's also keinerlei Probleme.

## writeReplace

Im folgenden zeigen wir, wie sich Objekte bei der Serialisierung aus dem Serialisierung- Strom "heraussubtrahieren" können.

Wir verwenden ein Proxy-Beispiel. Sie folgendes Interface gegeben:

package **appl1**;

// ...

public interface **Foo** extends Serializable { public abstract int **getValue**();

public abstract void **setValue**(int value);

}

Und folgende Implementierung dieses Interfaces:

package **appl1**;

public class **FooImpl** implements Foo { private int **value**;

@Override

public int **getValue**() { return this.value;

}

@Override

public void **setValue**(int value) { this.value = value;

}

}

Dann können wir eine Proxy-Klasse schreiben, die ebenfalls das Interface Foo implementiert. Bei der Erzeugung eines Proxy-Objekts wird ein "Ziel-Objekt" übergeben (dessen Klasse Foo implementiert). Jede Proxy-Methode diagnostiziert den Eintritt in die Methode und delegiert dann an das Ziel-Objekt:

package **appl1**;

public class **FooTraceProxy** implements Foo { private final Foo **target**;

public **FooTraceProxy**(Foo target) { this.target = target;

}

@Override

public int **getValue**() {

System.out.println("> getValue"); return this.target.getValue();

}

@Override

public void **setValue**(int bar) { System.out.println("> setValue"); this.target.setValue(bar);

}

}

Wir erzeugen ein FooImpl und übergeben dieses an ein FooTraceProxy-Objekt. Der Aufruf von setValue wird dann auf der Konsole protokolliert werden. Wir serialisieren / deserialisieren das Objekt. Als Resultat der Deserialisierung erhalten wir das Proxy- Objekt und das daran angeschlossene FooImpl-Objekt zurück. Auch wenn nun eine Methode auf das deserialisierte Objekt aufgerufen wird, wir der Aufruf protokolliert:

static void **demo**() {

final Foo foo1 = new FooTraceProxy(new FooImpl()); System.out.println(foo1 instanceof FooTraceProxy); foo1.setValue(4711);

final Foo foo2 = Serializer.clone(foo1); System.out.println(foo2 instanceof FooTraceProxy); System.out.println(foo2.getValue());

}

true

* setValue

<-->

true

* getValue 4711

Die folgende Variante von FooTraceProxy enthält eine writeReplace-Methode. Wenn im Zuge der Serialisierung das FooTraceProxy an der Reihe ist, dann "ersetzt" sich dieses Objekt durch das an ihm angeschlossene Ziel-Objekt. Dieses Ziel-Objekt wird als Resultat von writeReplace zurückgeliefert:

package **appl2**;

// ...

public class **FooTraceProxy** implements Foo { private final Foo **target**;

public **FooTraceProxy**(Foo target) { ... }

@Override

public int **getValue**() { ... }

@Override

public void **setValue**(int bar) { ... }

protected Object **writeReplace**() throws ObjectStreamException {

System.out.println(FooTraceProxy.class + ".writeReplace()"); return this.target;

}

}

Wir verwenden wieder dieselbe Serialisierungs- / Deserialisierung-Routine wie im ersten Beispiel:

static void **demo**() {

System.out.println(foo1 instanceof FooTraceProxy); foo1.setValue(4711);

final Foo foo2 = Serializer.clone(foo1); System.out.println(foo2 instanceof FooTraceProxy); System.out.println(foo2.getValue());

}

Die Ausgaben zeigen, dass das von der Deserialisierung geliefert Objekt das FooImpl- Objekt ist (und eben nicht mehr das Proxy-Objekt): Der Aufruf von getValue wird nun nicht mehr diagnostiziert…

true

* setValue

<-->

class appl2.FooTraceProxy.writeReplace() false

4711

## Classes, Fields, Methods und Constructors

Die Klasse Class ist serialisierbar; Objekte der Typen Field, Constructor und Method

sind es nicht:

static void **demo**() { System.out.println(

Serializable.class.isAssignableFrom(Class.class)); System.out.println(

Serializable.class.isAssignableFrom(Field.class)); System.out.println(

Serializable.class.isAssignableFrom(Constructor.class)); System.out.println(

Serializable.class.isAssignableFrom(Method.class));

}

Die Ausgaben:

true false false false

Sei eine Klasse Foo gegeben:

package **appl**; public class **Foo** {

public String **value**;

public **Foo**(String value) { this.value = value;

}

public void **setValue**(String value) { this.value = value;

}

public String **setValue**() { return this.value;

}

@Override

public String **toString**() { ... }

}

Die folgende Methode zeigt, dass das Resultat der Serialisierung / Deserialisierung eines Class-Objekts wieder das ursprüngliche Class-Objekt ist (es wäre schlimm, wenn

es anschließend zwei verschiedene Class-Objekte gäbe, die die Klasse Foo

repräsentieren):

static void **demoClass**() {

final Class<?> cls1 = Foo.class;

final Class<?> cls2 = Serializer.clone(cls1); System.out.println(cls1 == cls2);

}

Die Ausgabe: true.

Wie können wir Method-Objekte serialisieren (wenn doch Method-Objekte nicht serialisierbar sind)?

Wie können eine Klasse MethodData schreiben, welche nur serialisierbare Attribute enthält. Dem Konstruktor der Klasse wird ein Method-Objekt übergeben, welches in seine serialisierbaren Bestandteile "zerlegt" wird. Die createMethod-Methode kann dann aus diesen Bestandteilen ein neues Method-Objekt erzeugen und zurückliefern:

package **util**;

import java.io.Serializable; import java.lang.reflect.Method;

public class **MethodData** implements Serializable {

private final Class<?> **declaringClass**; private final String **methodName**;

private final Class<?>[] **parameterTypes**;

public **MethodData**(Method method) { this.declaringClass = method.getDeclaringClass(); this.methodName = method.getName(); this.parameterTypes = method.getParameterTypes();

}

public Method **createMethod**() { try {

return this.declaringClass.getMethod( this.methodName, this.parameterTypes);

}

catch(final Exception e) {

throw new RuntimeException(e);

}

}

}

Die folgende Demo-Methode demonstriert die Verwendung der Klasse MethodData:

static void **demoMethod**() throws Exception {

final Method method1 = Foo.class.getMethod("setValue", String.class); System.out.println(method1);

final MethodData methodData1 = new MethodData(method1);

final MethodData methodData2 = Serializer.clone(methodData1); Method method2 = methodData2.createMethod(); System.out.println(method2);

System.out.println(method1 == method2);

}

Die Ausgaben:

public void appl.Foo.setValue(java.lang.String) public void appl.Foo.setValue(java.lang.String) false

Nun aber haben wir zwei(!) Method-Objekte, welche dieselbe Methode repräsentieren. Auf ähnliche Weise können wir eine Klasse FieldData und ConstruktorData

schreiben.

# 16 Multithreading - Erweiterungen

### Voraussetzungen

* Das Konzept des Multithreading
* Die Klasse Thread
* Überschreiben der run-Methode
* Starten eines Threads
* Warten auf Terminierung
* Interface Runnable

Dieses Kapitel behandelt Thread-bezogene Features, die bereits vor Java 5 existierten. Java 5 führte darüber hinaus das sog. concurrent-Package ein, welches eine Vielzahl weiterer Thread-Klassen beinhaltet. Diese Klassen werden erst im nächsten Kapitel behandelt werden.

### Übersicht

* Utility-Klassen
* Synchronisation
* volatile: "Ich sehe was, was Du nicht siehst…"
* Das wait-notify-Konzept: Producer und Consumer
* Eine BlockingQueue
* Ein kleiner, einfacher Thread-Pool für Runnables
* Futures und Callables
* Ein Thread-Pool, der Futures liefert.
* Exception-Handling
* ThreadLocal
* Zustände eines Thread.
* Ermittlung der aktuell laufenden Threads
  1. **Uitlities**

Wir erzeugen zwei Runnables, erzeugen zwei Threads, von denen jeder einer der Runnables ausführen wird, starten dieses Threads und warten via join auf deren Terminierung:

static void **demoWithoutUtils**() {

System.out.println("\tdemo starts"); final Runnable r1 = () -> {

try {

for (int i = 0; i < 5; i++) { Thread.sleep(1000); System.out.println(i);

}

}

catch (final InterruptedException e) { throw new RuntimeException(e);

}

};

final Runnable r2 = () -> { try {

for (int i = 0; i < 5; i++) { Thread.sleep(600); System.out.println("\t\t" + i);

}

}

catch (final InterruptedException e) { throw new RuntimeException(e);

}

};

final Thread t1 = new Thread(r1); final Thread t2 = new Thread(r2); t1.start();

try { Thread.sleep(500);

}

catch (InterruptedException e) { throw new RuntimeException(e);

}

t2.start(); try {

t1.join();

}

catch (InterruptedException e) { throw new RuntimeException(e);

}

try { t2.join();

}

catch (InterruptedException e) { throw new RuntimeException(e);

}

System.out.println("\tdemo terminates");

}

Die Ausgaben:

demo starts

0

0

1

1

2

3

2

4

3

4

demo terminates

Die Methode enthält viel Boilerplate-Code. Insbesondere ist ärgerlich, dass sleep und join checked Exceptions (InterruptedExceptions) werfen können – und wir also stets try/catch-Anweisungen formulieren müssen.

Die folgende Methode benutzt eine sleep- und eine join-Methode der Util-Klasse – und liest sich bereits wesentlich besser als die obige Variante:

static void **demoWithSimpleUtils**() {

System.out.println("\tdemo starts"); final Runnable r1 = () -> {

for (int i = 0; i < 5; i++) { Util.sleep(1000); System.out.println(i);

}

};

final Runnable r2 = () -> {

for (int i = 0; i < 5; i++) { Util.sleep(600); System.out.println("\t\t" + i);

}

};

final Thread t1 = new Thread(r1); final Thread t2 = new Thread(r2); t1.start();

Util.sleep(500); t2.start(); Util.join(t1); Util.join(t2);

System.out.println("\tdemo terminates");

}

Hier werden folgende Methoden der Util-Klasse benutzt:

public class **Util** {

public static void **sleep**(int millis) { try {

Thread.sleep(millis);

}

catch(InterruptedException e) { throw new RuntimeException(e);

}

}

public static void **join**(Thread thread) { try {

thread.join();

}

catch(InterruptedException e) { throw new RuntimeException(e);

}

}

// ...

}

Die folgende Methode ist noch etwas kürzer – sie nutzt die startAndJoin-Methode der

Util-Klasse:

static void **demoWithStartJoin**() { System.out.println("\tdemo starts"); final Runnable r1 = () -> {

for (int i = 0; i < 5; i++) { Util.sleep(1000); System.out.println(i);

}

};

final Runnable r2 = () -> {

for (int i = 0; i < 5; i++) { Util.sleep(600); System.out.println("\t\t" + i);

}

};

Util.startAndJoin(500, r1, r2); System.out.println("\tdemo terminates");

}

Und hier die oben benutzte startAndJoin-Methode:

public class **Util** {

// ...

public static void **startAndJoin**(

int distanceMillis, Runnable... runnables) {

final Thread[] threads = new Thread[runnables.length]; for (int i = 0; i < runnables.length; i++) {

final Thread t = new Thread(runnables[i]); t.start();

threads[i] = t; sleep(distanceMillis);

}

for (Thread t : threads) { join(t);

}

}}

Der Methode werden ein Zeitintervall und beliebig viele Runnables übergeben. Für jedes Runnable wird ein Thread erzeugt und gestartet (wobei die Threads jeweils im Abstand von distanceMillis-Millisekunden gestartet werden).

Und da es schließlich unschön ist, in einem Lambda-Ausdruck mehrere Anweisungen zu hinterlegen, lagern wird diese in zwei separate Methoden aus:

static void **demoSeparateMethods**() { Util.startAndJoin(500, () -> run1(), () -> run2()); System.out.println("\tdemo terminates");

}

static void **run1**() {

for (int i = 0; i < 5; i++) { Util.sleep(1000); System.out.println(i);

}

}

static void **run2**() {

for (int i = 0; i < 5; i++) { Util.sleep(600); System.out.println("\t\t" + i);

}

}

run1 und run2 werden in jeweils in einem eigenen Thread ausgeführt.

* 1. **Synchronized**

Die Threads einer Anwendung teilen sich deren Adressraum (sie laufen alle in derselben VM, also innerhalb ein- und desselben Betriebssystem-Prozesses). Sie können daher auch auf gemeinsame Daten zugreifen.

Dies ist gegenüber Multiprocessing ein entscheidender Vorteil: man benötigt keine zusätzlichen Mechanismen zum Austausch von Daten (etwa: pipes, shared memory etc).

In der Möglichkeit des Zugriffs auf gemeinsame Daten steckt aber auch ein Problem: greifen zwei Threads "gleichzeitig" auf solche Daten zu, kann es zu Problemen kommen. In solchen Fällen muss dann garantiert sein, dass eine bestimmte Anweisungsfolge (eben die Folge derjenigen Anweisungen, welche die gemeinsamen Daten manipulieren) zu einer Zeit immer nur von einem Thread ausgeführt wird. Will ein zweiter Thread diese Anweisungsfolge ausführen, muss er gegebenenfalls warten – solange warten, bis der Thread, der aktuell diese Anweisungsfolge ausführt, das Ende dieser Anweisungsfolge erreicht hat. Der Zugriff muss synchronisiert werden.

Wenn hier von "Daten" die Rede ist, sind entweder globale Daten (also statische Attribute) oder Instanzvariablen eines Objekts gemeint. Lokale Variablen und Methodenparameter sind völlig unkritisch: denn jeder Thread hat seinen eigenen Stack

* verfügt also über sein eigenes Set an lokalen Variablen und Parametern. Nur die Attribute einer Klasse (die Instanz- und die Klassen-Attribute) sind kritisch.

Im folgenden betrachten wir drei Konto-Klassen: UnsafeAccount, SafeAccpont1 und

SafeAccount2. Alle drei Klassen implementieren das Interface Account:

public interface **Account** {

public abstract int **getNumber**();

public abstract void **deposit**(int amount); public abstract void **withdraw**(int amount); public abstract int **getBalance**();

}

Jede der drei Konto-Klassen definiert ein Feld balance. Die Klassen sollen garantieren, dass balance nie negativ sein darf.

Hier zunächst die Klasse UnsafteAccount:

import jn.util.threading.Util;

public class **UnsafeAccount** implements Account { private final int **number**;

private int **balance**;

public **UnsafeAccount**(int number) { this.number = number;

}

@Override

public int **getNumber**() { return this.number;

}

@Override

public void **deposit**(int amount) { checkAmount(amount); Util.sleep(1000); this.balance += amount;

}

@Override

public void **withdraw**(int amount) { checkAmount(amount);

if (this.getBalance() < amount)

throw new IllegalArgumentException( "withdraw " + amount + " not possible");

Util.sleep(1000); this.balance -= amount;

}

@Override

public int **getBalance**() { return this.balance;

}

private static void **checkAmount**(int amount) { if (amount <= 0)

throw new IllegalArgumentException( "illegal amount : " + amount);

}

}

Die Klasse ist – was ihre Verwendung in einem Singe-Threaded-Kontext angeht – offensichtlich wasserdicht.

In einer Multi-Threaded-Umgebung aber kann folgender Fall eintreten: Ein Thread A betritt die withdraw-Methode. Die Prüfung:

if (this.getBalance() < amount)

verlaufe negativ (m.a.W.: es wurde festgestellt, dass der Betrag abgehoben werden kann). Dann wird in einem weiteren Thread B dieselbe Methode auf dasselbe Objekt aufgerufen. Auch hier verlaufe die Prüfung negativ (also "positiv"!). Beide Threads haben nun die Prüfung durchlaufen. Für beide Threads wird dann anschließend die folgende Zeile ausgeführt:

this.balance -= amount;

Und zwar: ohne noch einmal zu prüfen! Es ist leicht einzusehen, dass eine solche Konstellation zu einer negativen balance führen kann – also die Invariante der Klasse verletzen kann.

Um dieses Problem demonstrieren zu können, kann die Auszahlungs-Methode durch einen Thread.sleep erweitert werden (der Bankangestellte geht in den Tresorraum, um das Geld zu besorgen!):

public void **withdraw**(int amount) { if (amount < 0)

throw new IllegalArgumentException( "illegal amount: " + amount);

if (this.getBalance() < amount)

throw new IllegalArgumentException( "withdraw " + amount + " not possible");

Util.sleep(1000); this.balance -= amount;

}

Dann kann leicht gezeigt werden, dass die Invariante der Klasse verletzt werden kann, wenn zwei Threads "gleichzeitig" (also z.B. mit einer Verzögerung von 500 Millisekunden) diese Auszahlungsmethode ausführen.

M.a.W.: der folgende Abschnitt ist "kritisch":

if (this.getBalance() < amount)

throw new IllegalArgumentException( "withdraw " + amount + " not possible");

Util.sleep(1000); this.balance -= amount;

Wie können wir das Problem demonstrieren?:

Der folgenden demo-Methode wird ein Account übergeben. Sie zahlt auf das Konto zunächst einmal den Betrag von 5000 ein. Dann startet sie zwei Threads – innerhalb des ersten Threads wird die Methode withdraw1, im zweiten Thread die Methode withdraw2 aufgerufen. withdraw1 lässt sich von dem Konto den Betrag 4000 auszahlen, withdraw2 lässt sich vom selben Konto den Betrag 3000 auszahlen.

static void **demo**(Account account) { Log.logMethodCall(account.getClass().getSimpleName()); account.deposit(5000);

Util.startAndJoin( 500,

() -> withdraw1(account), () -> withdraw2(account));

System.out.println(account.getBalance());

}

static void **withdraw1**(Account account) { try {

account.withdraw(4000);

}

catch (final Exception e) { System.out.println(e);

}

}

static void **withdraw2**(Account account) { try {

account.withdraw(3000);

}

catch (final Exception e) { System.out.println(e);

}

}

Diese demo-Methode wird nun mit einem UnsafeAccount aufgerufen:

demo(new UnsafeAccount(4711));

Die Ausgaben:

-2000

Die Garantie, dass balance niemals negativ wird, konnte also offenbar nicht erfüllt werden. In einem Multithreading-Kontext ist die Klasse UnsafeAccount also unsicher.

Das Problem kann wie folgt gelöst werden:

Ein Thread, der diese Anweisungsfolge ausführen will, muss sich einen Schlüssel besorgen. Er kann diese Anweisungsfolge erst dann betreten, wenn er sich einen solchen Schlüssel besorgt hat. Für diese Anweisungsfolge darf es nur einen einzigen Schlüssel geben. Will nun ein zweiter Thread dieselbe Anweisungsfolge betreten, muss er sich ebenfalls den Schlüssel besorgen – aber dieser Schlüssel ist nun bereits vergeben. Der Thread muss warten – solange warten, bis der erste Thread die Anweisungsfolge ausgeführt hat und den Schlüssel wieder freigibt.

Woher den Schlüssel nehmen? Jedes Objekt kann in Java als Schlüssel fungieren. Also kann auch ein Account-Objekt als Schlüssel fungieren – also auch dasjenige Account- Objekt, auf welches ein Thread die withdraw-Methode aufruft (this).

Ein kritischer Abschnitt, zu dessen Ausführung ein Schlüssel benötigt wird, wird eingeleitet mit dem Schlüsselwort synchronized. Es folgt die Referenz auf das Schlüssel-Objekt, welches zur Ausführung des Abschnitts erforderlich ist. Dann folgt der eigentliche kritische Abschnitt – entweder eine einzelne Anweisung oder aber ein Block von Anweisungen. Am Ende dieses Blocks (bzw. nach Ausführung der Anweisung) gibt der den Block oder die Anweisung ausführende Thread den Schlüssel wieder frei. (Intern besitzt jedes Objekt ein Flag, welches anzeigt, ob dieses Objekt im Augenblick

als Schlüssel in Benutzung ist – das Setzen und das Zurücksetzen dieses Flags sind atomische Aktionen.)

Wir können also folgende SafeAccount1-Klasse schreiben:

public class **SafeAccount1** implements Account {

// ... @Override

public void **deposit**(int amount) { checkAmount(amount); synchronized (this) {

Util.sleep(1000); this.balance += amount;

}

}

@Override

public void **withdraw**(int amount) { checkAmount(amount); synchronized (this) {

if (this.getBalance() < amount)

throw new IllegalArgumentException( "withdraw " + amount + " not possible");

Util.sleep(1000); this.balance -= amount;

}

}

@Override

public int **getBalance**() { synchronized (this) {

return this.balance;

}

}

// ...

}

Angenommen, einem Thread gelingt es, den kritischen Abschnitt in withdraw zu betreten: dann kann ein anderer Thread weder diese withdraw-Methode noch die deposit-Methode auf dasselbe Account-Objekt betreten. Er wird solange angehalten, bis der erste Thread die withdraw-Methode (deren kritischen Abschnitt) verlässt. Auf dasselbe Konto kann also immer nur ein einziger Thread gleichzeitig einzahlen oder auszahlen.

Das Ergebnis wird jetzt darin bestehen, dass der zweite Thread, der die withdraw- Methode des Account-Objekts betritt, den Bestand nicht verändert und eine Exception werfen wird.

Der demo-Aufruf:

demo(new SafeAccount1(4711));

Die Ausgaben:

java.lang.IllegalArgumentException: withdraw 4000 not possible 1000

Die Klasse SafeAccount1 verwendet this als "Schlüssel-Lieferant". Wir können aber auch jedes andere Objekt als solchen Lieferanten nutzen. In der folgenden Klasse SafeAccount2 erzeugen wir eine Object und binden es an ein Attribut namens lock. Und genau dieses Objekt nutzen wird dann als Synchronisations-Objekt:

public class **SafeAccount2** implements Account {

// ...

private final Object **lock** = new Object();

// ... @Override

public void **deposit**(int amount) { checkAmount(amount); synchronized (this.lock) {

Util.sleep(1000); this.balance += amount;

}

}

@Override

public void **withdraw**(int amount) { checkAmount(amount); synchronized (this.lock) {

if (this.getBalance() < amount)

throw new IllegalArgumentException( "withdraw " + amount + " not possible");

Util.sleep(1000); this.balance -= amount;

}

}

@Override

public int **getBalance**() { synchronized (this.lock) {

return this.balance;

}

}

// ...

}

Der Aufruf:

demo(new SafeAccount2(4711));

führt auch hier zu "korrekten" Resultaten:

java.lang.IllegalArgumentException: withdraw 4000 not possible 1000

Ein abschließender Hinweis:

Sofern alle Anweisungen einer Methode kritisch sind und this als Synchronisations- Objekt genutzt werden soll, kann die gesamte Methode als synchronized spezifiziert werden:

synchronized public void **someMethod**() {

...

}

Dies ist identisch mit

public void **someMethod**() { synchronized (this) {

...

}

}

Wir sollten aber darauf achten, dass die Synchonisations-Blöcke möglichst kein sind. Die formale Überprüfung von Parametern (checkAmount(amount)) kann vor dem synchronized-Block ausgeführt werden.

* 1. **Synchronized – Lock Splitting**

Wir können auch "feingranulare" Sperren benutzten.

Wir implementieren eine unsichere und eine sichere Variante eines Dictionaries. Ein Dictionary enthält Key-Value-Paare und erlaubt den schlüsselbasierten Zugriff auf seine Einträge. Beide Klassen implementieren folgendes Interface:

public interface **Dictionary**<K, V> {

public abstract boolean **put**(K key, V value); public abstract V **get**(K key);

}

Hier zunächst ein Objektdiagramm:

0

: Dictionary

: ArrayList



: ArrayList

: Entry

: Entry

: Entry

: K

: V

: K

: V

: K

: V

1

2



: ArrayList

: Entry

: Entry

: K

: V

: K

: V

9



: ArrayList

: Entry

: K

: V

Ein Dictionary hat einen Array von ArrayLists. Jeder der ArrayLists können

Entry-Objekte hinzugefügt werden.

Um ein Entry einzufügen, wird ein Index berechnet: hashCode des Keys des Entry- Objekts modulo Größe des ArrayList-Arrays. An dieser Stelle befindet sich die ArrayList, zu welcher das neue Entry-Objekt eingetragen werden muss. Existiert in dieser ArrayList bereits ein Entry, dessen Schlüssel gleich dem Schlüssel des einzufügenden Entry-Objekts ist, wird false geliefert. Ansonsten wird das Entry- Objekt der ArrayList hinzugefügt und true geliefert.

Aufgrund dieser Erläuterungen sollte auch die Funktionsweise der get-Methode klar sein.

Hier die unsichere Variante:

public class **UnsafeDictionary**<K, V> implements Dictionary<K, V> { static class **Entry**<K, V> {

final K **key**; final V **value**;

public **Entry**(K key, V value) { this.key = key;

this.value = value;

}

}

private final List<Entry<K, V>>[] **array** = (List<Entry<K, V>>[]) new ArrayList[10];

{

Arrays.setAll(array, i -> new ArrayList<Entry<K,V>>());

}

@Override

public boolean **put**(K key, V value) {

int index = key.hashCode() % array.length; V existingValue = this.get(key);

if (existingValue != null) return false;

Util.sleep(100);

System.out.println("adding: " + key + " => " + value); array[index].add(new Entry<K, V>(key, value));

return true;

}

@Override

public V **get**(K key) {

int index = key.hashCode() % array.length; for (Entry<K, V> entry : array[index]) {

if (entry.key.equals(key)) return entry.value;

}

return null;

}

}

Wir verwenden folgende demo-Methode:

static void **demo**(Dictionary<String, String> words) { Util.startAndJoin(50, () -> put1(words), () -> put2(words)); System.out.println(words.get("one"));

}

static void **put1**(Dictionary<String,String> words) { if (! words.put("one", "eins"))

System.out.println("put1: one exits");

}

static void **put2**(Dictionary<String,String> words) { if (! words.put("one", "EINS"))

System.out.println("put2: one exits");

}

demo startet zwei Threads, von denen der erste put1 ausführt und der zweite put2. Beide Threads versuchen einen Eintrag mit dem Schlüssel "one" einzufügen – der erste mit dem Wert "eins", der zweite mit dem Wert "EINS".

Wir rufen nun demo mit einem UnsafeDictionary auf:

demo(new UnsafeDictionary<String, String>());

Die Ausgabe zeigt, dass zwei Entry-Objekte hinzugefügt werden – beide mit dem Schlüssen "one" und den Werten "eins" resp. "EINS":

adding: one => eins adding: one => EINS eins

Der zweite Eintrag aber hätte nicht passieren dürfen.

Natürlich könnten wir nun beide Methode z.B. mit this synchronisieren:

public synchronized boolean **put**(K key, V value) { ... } public V **get**(K key) { ... }

Bei näherem Hinsehen reicht es hier aber völlig aus, die jeweiligen (schreibenden oder lesenden) Zugriff auf die aktuell betroffene ArrayList zu synchronisieren.

Wir können also einen feingranulare Lock-Mechanismus benutzen:

public class **SafeDictionary**<K, V> implements Dictionary<K, V> {

// ...

@Override

public boolean **put**(K key, V value) {

int index = key.hashCode() % array.length; synchronized (this.array[index]) {

V existingValue = this.get(key); if (existingValue != null)

return false; Util.sleep(100);

System.out.println("adding: " + key + " => " + value); array[index].add(new Entry<K, V>(key, value));

return true;

}

}

@Override

public V **get**(K key) {

int index = key.hashCode() % array.length; synchronized (this.array[index]) {

for (Entry<K, V> entry : array[index]) { if (entry.key.equals(key))

return entry.value;

}

return null;

}

}

}

Wir die demo-Methode nun mit einem SafeDictionary aufgerufen:

demo(new SafeDictionary<String, String>());

so erhalten wir ein korrektes Resultat – es wird nur "one" => "eins" eingefügt:

adding: one => eins put2: one exits eins

Es ist klar, dass die Implementierung, die nur die aktuelle ArrayList sperrt, performanter ist eine Variante, die jeweils das Dictonary-Objekt selbst als Sperre nutzt.

* 1. **Synchronized – static Lock**

Möchten wir statische Methoden synchronisieren, benötigen wir statische Sperren. Die statische println-Methode der folgenden Klasse ist nicht threadsafe:

package **appl**;

// ...

public class **UnsafeConsole** {

public static void **println**(String arg) { for (int i = 0; i < arg.length(); i++) {

System.out.print(arg.charAt(i)); Util.sleep(1);

}

System.out.println();

}

}

Wir rufen die obige println-Methoden in zwei Threads auf, die gleichzeitig gestartet werden:

static void **demoUnsafeConsole**() { Util.startAndJoin(0,

() -> UnsafeConsole.println("hello world"), () -> UnsafeConsole.println("HELLO WORLD"));

}

Die Ausgaben sind nicht so recht lesbar:

HheElLlLoO wWoOrRlLdD

Hier eine sichere Variante der Console-Klasse:

package **appl**;

// ...

public class **SafeConsole** {

private static Object **lock** = new Object(); public static void **println**(String arg) {

synchronized (lock) {

for (int i = 0; i < arg.length(); i++) { System.out.print(arg.charAt(i)); Util.sleep(1);

}

System.out.println();

}

}

}

Die Klasse benutzt ein statisches Lock-Objekt.

Statt eine eigene lock-Variable hätten wir als Lock-Objekt auch SafeConsole.class benutzen können (also das Class-Objekt der SafeConsole-Klasse). Oder wir hätten die komplette Methode synchronisieren können:

public synchronized static void **println**(String arg) {

// ...

}

Die demo-Methode:

static void **demoSafeConsole**() { Util.startAndJoin(0,

() -> SafeConsole.println("hello world"), () -> SafeConsole.println("HELLO WORLD"));

}

Die Ausgaben sind nun wohlgeordnet:

hello world HELLO WORLD

* 1. **Synchronized – Deadlock**

Natürlich kann man mit diesem synchronized-Mechanismus auch deadlocks implementieren – sogar relativ einfach.

Foo definiert zwei Methoden: alpha und beta. Die alpha-Methode ist mit alphaLock synchronisiert, die beta-Methode mit betaLock. Die alpha-Methode ruft beta auf, die beta–Methode ruft alpha auf:

public class **Foo** {

private final Object **alphaLock** = new Object(); private final Object **betaLock** = new Object();

public void **alpha**() {

synchronized (this.alphaLock) { System.out.println(" --> Foo.alpha()"); Util.sleep(1000);

System.out.println(" trying to call beta()"); this.beta();

System.out.println(" <-- Foo.alpha()");

}

}

public void **beta**() {

synchronized (this.betaLock) { System.out.println(" --> Foo.beta()"); Util.sleep(1000);

System.out.println(" trying to call alpha()"); this.alpha();

System.out.println(" <-- Foo.beta()");

}

}

}

In der demo-Methode werden zwei Threads gestartet. In den ersten Thread wird alpha

aufgerufen, im zweiten Thread beta:

static void **demo**() {

System.out.println("this program will run forever..."); final Foo foo = new Foo();

Util.startAndJoin(

500,

() -> foo.alpha(),

() -> foo.beta());

}

Die Ausgaben:

this program will run forever...

--> Foo.alpha()

--> Foo.beta()

trying to call beta() trying to call alpha()

Weder alpha kehrt zurück, noch beta. Das Programm terminiert nicht.

* 1. **Volatile**

Wird einem gewöhnlichen Attribut ein neuer Wert zugewiesen, so landet dieser Wert zunächst einmal im Cache des schreibenden Threads – er ist somit für andere Threads nicht sichtbar. Denn auch ein Lesezugiff erfolgt natürlich über den Cache des lesenden Threads:

Anders bei Attributen, die als volatile gekennzeichnet sind.

Lese- und Schreibzugriffe auf solche Attribute sind zunächst einmal atomar – und das gilt auch dann, wenn die Attribute vom Typ long oder double sind. Bei Referenztypen sind natürlich die Zugriffe auf die Referenzvariablen atomar – nicht aber der Zugriff auf die Objekte, auf welche die Variablen verweisen.

Wird nun lesend auf eine volatile-Variablen zugegriffen, wird ihr Inhalt zunächst auf dem Hauptspeicher geladen: es findet ein "refresh" statt. Wenn schreibend zugegriffen wird, wird der neue Wert unmittelbar im Hauptspeicher abgelegt: es passiert ein "flush". Das System verhält sich dann so, als gäbe es keine Thread-eigenen Caches. Änderungen, die ein Thread am Zustand einer volatile-Variablen vorgenommen hat, sind für alle anderen Threads auch sichtbar.

Thread-local Cache

value : 43

Thread-local Cache

value : 42

write value

"flush"

read value "refresh"

Hauptspeicher

value : 42

Wir eine volatile-Variable via new gesetzt, ist garantiert, dass die eigentliche Zuweisung an die Variable (und dann der "flush") erst dann geschieht, wenn alle Anweisungen des Konstruktors ausgeführt wurden – wenn also das erzeugte Objekt in einem konsistenten Zustand ist.

Der "refresh" und der "flush" sind natürlich mit Kosten verbunden.

Wir demonstrieren das Verhalten von volatile anhand von zwei Klassen, die beide das Interface Place implementieren:

public interface **Place** {

public abstract int **getValue**();

public abstract void **setValue**(int value);

}

Die erste Implementierung:

public class **NonVolatilePlace** implements Place { private int **value** = 0;

@Override

public int **getValue**() { return this.value;

}

@Override

public void **setValue**(int value) { this.value = value;

}

}

Weder getValue noch setValue sind synchronisiert. Das Attribut value ist ohne(!) volatile definiert.

Wir verwenden folgende demo-Methode:

static void **demo**(Place place) {

final int N = Integer.MAX\_VALUE - 1; Util.startAndJoin(0,

() -> set(place, N),

() -> get(place, N));

}

static void **set**(Place place, int n) { for (int i = 0; i <= n; i++)

place.setValue(i); System.out.println("Thread 1 terminates");

}

static void **get**(Place place, int n) { while (place.getValue() != n)

;

System.out.println("Thread 2 terminates");

}

Im ersten Thread wird set gestartet. Diese Methode ruft N mal die setValue-Methode auf den an demo übergebenen Place auf - und übergibt jeweils i. Beim letzten Aufruf der setValue-Methode wird also N übergeben.

Die get-Methode ruft innerhalb eine while-Schleife die getValue-Methode auf – und terminiert dann, wenn der zurückgeliefert Wert gleich N ist.

Ginge alles "mit rechten Dingen" zu, müsste nicht nur der erste, sondern auch der zweite Thread terminieren.

Rufen wir nun die demo-Methode wie folgt auf:

demo(new NonVolatilePlace());

Dann ist das Resultat etwas merkwürdig: der erste Thread terminiert natürlich immer; der zweite aber nur hin und wieder…

Wir bauen eine zweite Place-Variante:

public class **VolatilePlace** implements Place { private volatile int **value** = 0;

@Override

public int **getValue**() { return this.value;

}

@Override

public void **setValue**(int value) { this.value = value;

}

}

Das value-Attribut ist nun mit volatile gekennzeichnet. Beim folgenden Aufruf:

demo(new VolatilePlace());

kehrt nun auch der zweite Thread immer zurück. Der zweite Thread "sieht" also genau denjenigen Wert der value-Variablen, die auch der erste Thread sieht. Jedes Mal dann, wenn setValue aufgerufen wird, wird der geänderte value-Wert in den Hauptspeicher zurückgeschrieben ("flush"); und jedes Mal, wenn getValue aufgerufen wird, wird der Wert aus dem Hauptspeicher gelesen und der Thread-eigene Cache somit aktualisiert ("refresch").

Natürlich könnte man auf volatile auch verzichten – aber dann müssten die beiden Methoden der Place-Klasse synchronisiert sein (auch bei synchronized findet beim Eintritt in den kritischen Abschnitt ein refresh und beim Austritt ein flush statt):

public class **SynchronizedPlace** implements Place { private int **value** = 0;

@Override

public synchronized int **getValue**() { return this.value;

}

@Override

public synchronized void **setValue**(int value) { this.value = value;

}

}

volatile ist allerdings wesentlich performanter als synchronized. Aber synchronized

ist natürlich allgemeiner verwendbar als volatile.

* 1. **wait / notify**

Zunächst ein Objektdiagramm:

a Producer-Thread a Consumer-Thread

: Place

get

put

some Product

Ein Steinbrucharbeiter produziert rohe Steine – Steine, die für die weitere Verarbeitung zu Grabsteinen gedacht sind. Ein Steinmetz benötigt solche rohen Steine, um Grabsteine zu erstellen. Der Steinbrucharbeiter und der Steinmetz sind durch einen Fluss getrennt, über den eine schmale und baufällige Brücke führt. Diese Brücke wird als "Zwischenlager" für die vom Steinbrucharbeiter produzierten und vom Steinmetz konsumierten rohen Steine benutzt. Dieses Zwischenlager kann aber immer nur maximal einen einzigen Stein aufnehmen.

Steinbrucharbeiter und Steinmetz können mit unterschiedlicher Geschwindigkeit arbeiten. Mal ist der eine schneller, mal der andere. Es kann also sein, dass der Steinmetz darauf warten muss, bis der Steinbrucharbeiter einen neuen Stein produziert hat – und umgekehrt. Wenn der Steinmetz also z.B. auf einen weiteren Stein wartet, macht er ein kleinen "Nickerchen". Der Steinbrucharbeiter wird dann irgendwann seinen nächsten Stein produziert haben und ihn auf die Brücke legen. Dann muss er "pfeifen" – um den schlafenden Steinmetz aufzuwecken.

Und umgekehrt: Der Steinbrucharbeiter hat bereits einen weiteren Stein produziert – ohne dass der Steinmetz bereits den letzten Stein abgeholt hat. Dann wird der Steinbrucharbeiter ein Nickerchen einlegen. Der Steinmetz wird dann irgendwann den noch auf der Brücke befindlichen Stein abholen – und pfeifen müssen, damit der Steinbrucharbeiter aufwacht und seinen neue produzierten Stein auf der Brücke ablegen kann (und dann mit der Produktion des nächsten Steins fortzufahren).

Im folgenden Programm wird die Brücke durch ein Objekt vom Typ Place repräsentiert; und ein Stein durch ein Objekt vom Typ String; und jeder der beiden Arbeiter durch einen eigenen Thread.

Die Klasse Place sieht wie folgt aus (die genaueren Einzelheiten der put- und get- Methoden werden später dargestellt):

public class **Place** {

private String **product** = null;

public void **put** (String product) { ... } public String **get** () { ... }

}

Sofern product den Wert null hat, ist der Platz leer; zeigt product auf ein String- Objekt, so ist er besetzt.

Die put-Methode legt ein Produkt (einen String) auf den Platz (muss dabei aber eventuell warten, bis der Platz frei wird); die get-Methode holt ein Produkt vom Platz ab (und muss dabei eventuell warten, bis der Platz voll ist) und leert dabei den Platz. Nach Ausführung von put ist product also ungleich null, nach Ausführung von get ist product gleich null.

Sowohl die get- als auch die put-Methode wird also in bestimmten Situationen eine Zeitlang blockieren – in anderen Situationen aber auch sofort zurückkehren können.

Die put-Methode wird vom produzierenden Thread, die get-Methode vom konsumierenden Thread aufgerufen werden.

Sowohl der Producer als auch der Consumer müssen den (gemeinsam genutzten) Platz kennen:

Das Bild zeigt, dass der Platz im Augenblick belegt ist. Die Aufruf der put-Methode würde also im Augenblick blockieren, der Aufruf von get sofort mit dem String zurückkehren.

Ein Producer produziert sechs Produkte – das letzte Produkt ist ein leerer String. Dieses letzte "Spezialprodukt" zeigt dem Consumer an, dass der Producer Feierabend gemacht hat und der Consumer also seinerseits ebenfalls Feierabend machen kann. Der Producer produziert seine Produkte in einer durchschnittlichen Zeit von 5 Sekunden:

Hier die komplette Place-Klasse:

public class **Place** {

private String **product** = null; public void **put**(String product) {

System.out.println(">> put"); synchronized (this) {

while (this.product != null) { System.out.println("WAITING..."); try {

this.wait();

}

catch (final InterruptedException e) { throw new RuntimeException(e);

}

}

assert this.product == null; this.product = product; this.notifyAll();

}

System.out.println("<< put");

}

public String **get**() { System.out.println("\t\t>> get"); String product = null; synchronized (this) {

while (this.product == null) { System.out.println("\t\tWAITING..."); Util.wait(this);

}

assert this.product != null; product = this.product; this.product = null; this.notifyAll();

}

System.out.println("\t\t<< get"); return product;

}

}

Beide Methoden – sowohl die put- als auch die get-Methode – arbeiten in einem kritischen Abschnitt, der durch das Place-Objekt gesperrt wird.

Die put-Methode testet, ob der Platz leer ist. Wenn der Platz besetzt ist, ruft put die Object-Methode wait auf. Diese wartet solange, bis der Platz leer ist (dies wird sofort genauer erläutert werden). Wenn sie aufwacht, ist der Platz garantiert leer. Das an put übergebene product kann dann abgelegt werden. Dann ruft put die Object-Methode notifyAll auf – diese signalisiert einem evtl. per wait wartenden Konsumenten, dass das Ereignis, auf das er wartet (ein neues Produkt) eingetreten ist.

Die get-Methode ist invers zu put. Auch get muss evtl. per wait warten. Wenn wait zurückkehrt (weil in put die Methode notifyAll aufgerufen wurde), ist der Platz garantiert belegt. Das Produkt kann also entnommen und als Resultat von get zurückgeliefert werden – nicht ohne allerdings zuvor notifyAll aufzurufen, um einen evtl. per wait wartenden Produzenten aufzuwecken.

Das "Nickerchen" passiert in wait – das "Pfeifen" ist der Aufruf von notify. Was passiert genau beim wait?

Angenommen, dem Produzenten ist es gelungen, in den kritischen Abschnitt von put einzutreten. Angenommen weiter, er stellt dort fest, dass der Platz noch belegt ist. Er ruft wait auf. Bei diesem Aufruf gibt der Produzenten-Thread seine Sperre wieder frei – damit später der Konsumenten-Thread den kritischen Abschnitt von get betreten kann (der je mit demselben Objekt gesperrt wird der kritische Abschnitt von put).

Sofern dann irgendwann der Konsument das Produkt abholt und notifyAll aufruft, wird der Produzenten-Thread aufgeweckt. Er wird nun versuchen, an derselben Stelle, an der er per wait schlafen gelegt wurde, weiterzurechnen. Dazu benötigt er aber wieder den Schlüssel für den kritischen put-Abschnitt. Und den bekommt er zurück, sobald der Konsument den kritischen Abschnitt von get schließlich verlässt.

this.notifyAll bedeutet also, dass alle, die auf den this-Schlüssel per this.wait warten, wieder aufwachen (wobei dann die meisten sich unmittelbar wieder schlafen legen müssen – daher die while-Schleife um wait).

Sowohl wait als auch notifyAll dürfen also nur innerhalb eines kritischen Abschnitts verwendet werden (ansonsten würde eine Ausnahme geworfen).

Die demo-Methode startet zwei Threads – der erste führt die produce-Methode aus, der zweite die consume-Methode. Man beachte, dass die produce-Methode zuletzt einen "Feierabend"-String fabriziert ("") – damit auch die consume-Methode terminieren kann:

static void **demo**() {

final Place place = new Place(); Util.startAndJoin(

1000,

() -> produce(place),

() -> consume(place));

}

static void **produce**(Place place) { for (int i = 0; i < 5; i++) {

final int time = (int) (Math.random() \* 1000); System.out.println("producing \"" + i + "\" (" + time + ") ..."); Util.sleep(time);

System.out.println("produced!"); place.put("" + i);

}

place.put("");

}

static void **consume**(Place place) { final int time = 500;

while (true) {

final String product = place.get(); if (product.length() == 0)

break; System.out.println("\t\tconsuming \"" +

product + "\" (" + time + ") ..."); Util.sleep(time); System.out.println("\t\tconsumed!");

}

}

Mögliche Ausgaben:

producing "0" (319) ... produced!

>> put

<< put

producing "1" (770) ...

>> get

<< get

consuming "0" (500) ... produced!

>> put

<< put

producing "2" (128) ... produced!

>> put WAITING...

consumed!

>> get

<< get

consuming "1" (500) ...

<< put

producing "3" (747) ... consumed!

>> get

<< get

consuming "2" (500) ... produced!

>> put

<< put

producing "4" (322) ... consumed!

>> get

<< get

consuming "3" (500) ... produced!

>> put

<< put

>> put WAITING...

consumed!

>> get

<< get

<< put

consuming "4" (500) ... consumed!

>> get

<< get

Hinweis: statt while / notifiyAll zu benutzen, könnten wir auch if / notify nutzen. Dann wären wir allerdings auf eine 1:1-Beziehung zwischen Produzenten und Konstumenten festgelegt.

Außerdem könnte wait auch dann zurückkehren, wenn notify überhaupt nicht aufgerufen wird ("spurious wakeup").

* 1. **BlockingQueue**

Ein Objektdiagramm:

: Queue max = 3 list

: LinkedList

oveFirst

rem

add

eue

enqu

queue

: String

: String

: String

Im letzten Programm waren der Produzent und der Konsument sehr eng gekoppelt – es gab nur einen einzigen Platz für den Austausch der Steine.

Wäre die Brücke nicht so baufällig und etwas länger gewesen, hätten also mehrere Steine auf der Brücke Platz gefunden, wäre die Kopplung zwischen den beiden Partner lockerer gewesen. Der Steinbrucharbeiter hätte z.B. maximal 10 Steine auf die Brücke legen können – ohne auf den Steinmetz warten zu müssen. Nur wenn dieser äußerst langsam konsumiert, muss der Steinbrucharbeiter evtl. warten. Er würde dann also wesentlich weniger häufig warten müssen als in der alten Lösung.

Dasselbe gilt natürlich umgekehrt: Liegen erst einmal mehrere Steine auf der Brücke, kann sich der Steinbrucharbeiter Zeit lassen. Der Steinmetz wird trotzdem Steine abholen und zu Grabsteinen weiterverarbeiten können. Irgendwann ist natürlich auch dann Schluss: wenn sich auf der Brücke kein einziger Stein mehr befindet. Dann muss der Steinmetz warten.

Die hier beschriebene etwas längere Brücke wird im folgenden Programm durch ein Queue-Objekt repräsentiert. Ein Queue-Objekt wird also den Platz des alten Place- Objekts einnehmen.

Wir beginnen mit einem Interface:

public interface **Queue**<T> {

public abstract void **enqueue**(T product);

public abstract T **dequeue**(); public abstract boolean **isFull**(); public abstract boolean **isEmpty**();

}

Das Interface wird von der Klasse Queue implementiert:

public class **BlockingQueue**<T> implements Queue<T> {

private final LinkedList<T> **list** = new LinkedList<T>(); private final int **max**;

public **BlockingQueue**(int max) { this.max = max;

}

@Override

public void **enqueue**(T product) { synchronized (this.list) {

while (this.isFull()) Util.wait(this.list);

this.list.add(product); this.list.notifyAll();

}

}

@Override

public T **dequeue**() { synchronized (this.list) {

while (this.isEmpty()) Util.wait(this.list);

final T product = this.list.removeFirst(); this.list.notifyAll();

return product;

}

}

@Override

public boolean **isFull**() { synchronized (this.list) {

return this.list.size() == this.max;

}

}

@Override

public boolean **isEmpty**() { synchronized (this.list) {

return this.list.size() == 0;

}

}

}

Eine Queue hat eine begrenzte Kapazität (ansonsten brauchte der Producer niemals zu warten!): max. Die Queue-Klasse wird implementiert unter Zuhilfenahme der Klasse LinkedList.

Eine Queue ist ein FIFO-Speicher, in welchen mittels enqueue ein Produkt eingehängt werden kann (wobei enqueue evtl. auf den Konsumenten warten muss, bis wieder Platz in der Queue ist. Per dequeue kann ein Produkt entnommen werden (das "älteste"). Evtl. muss natürlich auch dequeue warten (auf den Produzenten).

Eine LinkedList hat zwei Zeiger: der erste zeigt auf den ersten, der zweite auf den letzten Listenknoten. Die Konten selbst sind vorwärts und rückwärts verkettet. Jeder Knoten enthält einen Verweis auf ein Object (hier: auf einen String).

Die queue- und die deqeue-Operationen sind somit sehr effizient: in der LinkedList brauchen nur einige Zeiger "umgebogen" zu werden. Die queue-Methode wird auf die add-Methode von LinkedList, die deqeue-Methode auf die Methode removeFirst abgebildet.

Das folgende Bild zeigt eine Queue, in welcher augenblicklich drei Produkte eingehängt sind (welche also im Augenblick maximal gefüllt ist):

Hier eine Demo-Anwendung:

static void **demo**() {

final Queue<String> queue = new BlockingQueue<>(3); Util.startAndJoin(

1000,

() -> consume(queue),

() -> produce(queue));

}

static void **produce**(Queue<String> queue) { for (int i = 0; i < 5; i++) {

final int sleeptime = (int) (Math.random() \* 1000); Util.sleep(sleeptime);

System.out.println(i + " >"); queue.enqueue(String.valueOf(i));

}

queue.enqueue("");

}

static void **consume**(Queue<String> queue) { while (true) {

final String product = queue.dequeue(); System.out.println("\t> " + product); if (product.length() == 0)

break; Util.sleep(1000);

}

}

Mögliche Ausgaben:

0 >

* 0

1 >

2 >

* 1

3 >

* 2

4 >

* 3
* 4

>

* 1. **ThreadPool mit Runnables**

Ein Objekt-Diagramm:

calls



*Runnable*

*Runnable*

: BlockingQueue<Runnable>

calls

: ThreadPool

: PooledThread

un ()

: PooledThread

un ()

r

r

ecute()

ex

dequeue()

queue()

en

run ()

run ()

*Runnable*

run ()

Ein ThreadPool besitzt eine konfigurierbare Anzahl von PooledThreads (einer nicht statischen Memberklasse von ThreadPool). Und er besitzt eine BlockingQueue, in welcher – mittels seiner execute-Methode - Runnables eingehängt werden können.

PooledThread ist von Thread abgeleitet. Die run-Methode von PooledThread läuft in einer "Endlosschleife". Zu Beginn jedes Schleifendurchlaufs versucht der Thread, sich ein Runnable aus der Queue zu besorgen – via Queue.dequeue. Liefert dequeue ein solches Runnable, wird es in der run-Methode ausgeführt (via Aufruf von run).

Alle PooledThreads konkurrieren also um die Ausführung von Runnables.

Auf den ThreadPool kann shutdown aufgerufen werden. Beim Aufruf dieser Methode werden in der Queue so viele null-Referenzen eingehängt, wie es PooledThreads gibt. Erkennt nun die run-Methode von PooledThread, dass von der Queue eine null- Referenz geliefert wurde, wird die Methode verlassen, womit der Thread terminiert. Somit kann also auf sichere Weise der ThreadPool heruntergefahren werden.

Der Quellcode der Klasse PooledThread:

public class **ThreadPool** {

private final Queue<Runnable> **queue**;

private final List<Thread> **threads** = new ArrayList<Thread>(); private volatile boolean **shutdown** = false;

private class **PooledThread** extends Thread { @Override

public void **run**() { while(true) {

final Runnable runnable = ThreadPool.this.queue.dequeue();

if (runnable == null) break;

runnable.run();

}

}

}

public **ThreadPool**(Queue<Runnable> queue, int size) { Objects.requireNonNull(queue);

if (size < 1)

throw new IllegalArgumentException( "illegal size: " + size);

this.queue = queue;

for (int i = 0; i < size; i++) {

final Thread t = new PooledThread(); t.start();

this.threads.add(t);

}

}

public void **execute**(Runnable runnable) { Objects.requireNonNull(runnable); if (this.shutdown)

throw new RuntimeException("pool was shutdown"); this.queue.enqueue(runnable);

}

synchronized public void **shutdown**() { if (this.shutdown)

throw new RuntimeException("pool was already shutdown"); this.shutdown = true;

for (int i = 0; i < this.threads.size(); i++) this.queue.enqueue(null);

}

}

Und eine kleine demo-Methode:

static void **demo**() {

final ThreadPool pool = new ThreadPool( new BlockingQueue<Runnable>(3), 4);

pool.execute(() -> consumeTime("", 5000)); pool.execute(() -> consumeTime("\t", 2500)); pool.execute(() -> consumeTime("\t\t", 3500)); pool.execute(() -> consumeTime("\t\t\t", 6000));

pool.shutdown();

}

static void **consumeTime**(String indent, int sleepTime) { final long id = Thread.currentThread().getId();

System.out.println(indent + " >> [" + id + "] " + sleepTime); Util.sleep(sleepTime);

System.out.println(indent + " << [" + id + "] " + sleepTime);

}

Mögliche Ausgaben:

>> [12] 5000

>> [14] 6000

>> [15] 3500

>> [13] 2500

<< [13] 2500

<< [15] 3500

<< [12] 5000

<< [14] 6000

Man erkennt, dass hier alle Runnables parallel ausgeführt wurden.

* 1. **Callables und Futures**

In Java 5 wurde ein neues java.util.concurrent-Package eingeführt. Es enthält zahlreiche Interfaces und Klassen, welche die Konstruktion von Multithreading- Anwendungen erleichtern.

Insbesondere wurden dort zwei Interfaces eingeführt: Callable und Future.

Im folgenden zeigen wir, wie diese beiden Interfaces genutzt werden können. Dabei werden wir allerding nicht die "Original-Interfaces" nutzen, sondern eine selbstdefinierte Variante dieser Interfaces.

Das Interface Callable ist eine Abwandlung des Runnable-Interfaces. Die run-Methode von Runnable liefert void, die call-Methode von Callable<T> liefert T:

// a similar interface is specified in java.util.concurrent...

@FunctionalInterface

public interface **Callable**<T> { public T **call**();

}

Ein Objekt, dessen Klassen das Interface Future implementiert, repräsentiert ein Resultat einer Berechnung, welches erst "in Zukunft" vorliegen wird:

// a similar interface is specified in java.util.concurrent...

public interface **Future**<T> { public abstract T **get**();

public abstract boolean **isDone**();

}

Mit der get-Methode kann das Resultat ermittelt werden. Dabei kann sein, dass get blockiert – solange nämlich, bis eben das Resultat vorliegt. isDone kehrt sofort zurück – und liefert true, wenn das Resultat bereits vorliegt, ansonsten false.

Wie kann Future implementiert werden? Ein Schaubild:

*Future*

: FutureImpl

value isDone

ne

isDo

get

set

Thread 1

Thread 2

Thread 3

Ein FutureImpl<T> enthält zwei Instanzvariablen: value und done. Die done-Variablen wird zunächst mit false, die value-Variable mit null initialisiert.

Die set-Mehtode ist nicht(!) Teil des Interfaces – es handelt sich um eine Methode, die im Prinzip nur für denjenigen sichtbar sein soll, der das Ergebnis produziert. Der set- Methode wird eben dieses Resultat übergeben. Sie implementiert einen kritischen Abschnitt, in welchem das Resultat an value zugewiesen und isDone auf true gesetzt wird. Dann wird notifyAll aufgerufen.

Die get-Methode betritt ebenfalls einen kritischen Abschnitt (der mitselben Lock synchronisiert ist wie die set-Methode). Sie ruft wait auf. Diese Methode wird dann zurückkehren, wenn das Ergebnis gesetzt wurde. Und liefert dann eben dieses Ergebnis zurück.

Die isDone-Methode ist ebenfalls mit demselben Lock wie get und set synchronisiert und liefert – ohne warten zu müssen - den Wert des value-Attributs zurück.

class **FutureImpl**<T> implements Future<T> {

private final Object **lock** = new Object(); private T **value**;

private boolean **done** = false;

void **set**(T value) { synchronized (this.lock) {

this.value = value; this.done = true; this.lock.notifyAll();

}

}

@Override public T **get**() {

synchronized (this.lock) { while (! this.done)

Util.wait(this.lock); return this.value;

}

}

@Override

public boolean **isDone**() { synchronized (this.lock) {

return this.done;

}

}

}

Warum ist get mit while (und nicht mit if) implementiert? Weil man auch dann aufwachen kann, wenn niemand notify oder notifyAll aufgerufen hat ("spurious wakeup").

Nun verbinden wird die beiden Themen: Callable und Future.

Der statischen call-Methode der Klasse Caller wird ein Callable übergeben. Sie erzeugt ein FutureImpl. Dann wird ein Thread gestartet, der in der run-Methode des ihm übergebenen Runnables die call-Methode des Callables ausführt. Nachdem der Thread gestartet wurde, kehrt call unmittelbar zurück und liefert das zuvor erzeugte FutureImpl zurück (dessen isDone-Variable noch false enthält).

Die Ausführung der call-Methode des Callables kann natürlich dauern. Kehrt diese call-Methode zurück, wird das von ihr gelieferte Resultat via set an das Future-Objekt zugewiesen. In set findet ein notifyAll statt, so das diejenigen, die auf das Resultat via get warten, aufgeweckt werden und das Resultat geliefert bekommen:

public class **Caller** {

public static <T> Future<T> **call**(final Callable<T> callable) { final FutureImpl<T> future = new FutureImpl<T>();

new Thread(() -> {

T result = callable.call(); future.set(result);

}).start(); return future;

}

}

In der folgenden demo-Methode wird ein Callable mit einer call-Methode erzeugt, dessen Ausführung 1000 ms dauert. Sie liefert als Resultat 42 zurück.

Dieses Callable wird die Caller.call-Methode übergeben, welche ein Future zurückliefert. Mittels einer "busy-loop" wartet dann der Hauptthread auf die Fertigstellung des Resultats – via isDone. Liefert isDone den Wert true zurück, kann das Resultat via get abgeholt werden.

(In diesem Falle würde get natürlich auch nicht blockieren – get würde nur dann blockieren, wenn die busy-loop entfernt würde.)

static void **demo**() {

final Callable<Integer> callable = () -> { System.out.println(">> call"); Util.sleep(1000); System.out.println("<< call");

return 42;

};

final Future<Integer> future = Caller.call(callable);

while (!future.isDone()) { Util.sleep(300);

System.out.println("waiting for future...");

}

final int result = future.get(); System.out.println("get returns " + result);

}

Die Ausgaben:

>> call

waiting for future... waiting for future... waiting for future...

<< call

waiting for future... get returns 42

* 1. **ThreadPool mit Callables**

Im vorletzten Abschnitt konstruierten wir einen ThreadPool, der Runnables ausführen kann. Mittels einer execute-Methode konnten Runnables in eine Queue eingefügt werden. Im ThreadPool liefen mehrere Threads, die sich konkurrierend um die Ausführung dieser Runnables bemühten. Die excute-Methode hatte folgenden Kopf:

public void **execute**(Runnable runnable) ...

Der folgende ThreadPool benutzt die beiden im letzten Abschnitt eingeführten Interfaces (Callable und Future) und hat eine execute-Methode, der Callables übergeben werden können:

public <T> Future<T> **execute**(Callable<T> callable) ...

Die execute-Methode fügt das Callable wiederum in einer Queue ein und kehrt sofort zurück – liefert dabei ein Future-Objekt zurück. Mittels der get-Methode dieses Objekts kann dann das Resultat des Callables ermittelt werden (wobei get möglicherweise blockiert – solange, bis eben dieses von der call-Methode des Callables berechnete Ergebnis vorliegt).

Weiterhin sollen aber auch Runnables in dem ThreadPool ausgeführt werden. Zu diesem Zweck existiert eine überladene execute-Methode:

public Future<Void> **execute**(Runnable runnable) ...

Diese excute-Methode liefert ein Future<Void> zurück – also ein Future-Objekt, mittels dessen wir zwar auf die Terminierung des Runnables warten können, uns aber keinerlei Resultat abholen können (ein Runnable liefert nun einmal kein Resultat).

Das an diese exeute-Methode übergebene Runnable wird dabei einfach in ein Callable<Void> "eingewickelt" – in ein Callable, dessen call-Methode einfach die run-Methode des Runnables ausführt.

Hier der vollständige Quellcode der neuen ThreadPool-Klasse:

public class **ThreadPool** {

private static class **CallableWithFuture**<T> implements Callable<T> { private final Callable<T> **callable**;

private final FutureImpl<T> **future**;

public **CallableWithFuture**(

Callable<T> callable, FutureImpl<T> future) { this.callable = callable;

this.future = future;

}

public void **execute**() {

final T result = this.call(); this.future.set(result);

}

@Override

public T **call**() {

return this.callable.call();

}

}

private final Queue<Callable<?>> **queue**;

private final List<Thread> **threads** = new ArrayList<Thread>(); private volatile boolean **shutdown** = false;

private class **PooledThread** extends Thread { @Override

public void run() { while(true) {

final CallableWithFuture<?> callable = (CallableWithFuture<?>)

ThreadPool.this.queue.dequeue(); if (callable == null)

break; callable.execute();

}

}

}

public **ThreadPool**(Queue<Callable<?>> queue, int size) { Objects.requireNonNull(queue);

if (size < 1)

throw new IllegalArgumentException("illegal size: " + size); this.queue = queue;

for (int i = 0; i < size; i++) {

final Thread t = new PooledThread(); t.start();

this.threads.add(t);

}

}

public <T> Future<T> **execute**(Callable<T> callable) { Objects.requireNonNull(callable);

if (this.shutdown)

throw new RuntimeException("pool was shutdown"); final FutureImpl<T> future = new FutureImpl<>();

this.queue.enqueue(new CallableWithFuture<T>(callable, future)); return future;

}

public Future<Void> **execute**(Runnable runnable) { Objects.requireNonNull(runnable);

return this.execute(() -> { runnable.run();

return null;

});

}

synchronized public void **shutdown**() { if (this.shutdown)

throw new RuntimeException("pool was already shutdown"); this.shutdown = true;

for (int i = 0; i < this.threads.size(); i++) this.queue.enqueue(null);

}

}

Das genaue Studium dieser Klasse sei dem Leser / der Lesering überlassen. Hier eine kleine demo-Anwendung und ihre Ausgaben:

static void **demo**() {

final ThreadPool pool = new ThreadPool( new BlockingQueue<Callable<?>>(3), 5);

final Future<Integer> f1 = pool.execute( () -> produceValue("", 3500, 42));

final Future<Integer> f2 = pool.execute( () -> produceValue("\t", 6000, 77));

pool.shutdown();

System.out.println("===> f1 = " + f1.get()); System.out.println("===> f2 = " + f2.get());

}

static int **produceValue**(

String indent, int sleepTime, int valueToProduce) { final long id = Thread.currentThread().getId(); System.out.println(indent +

" >> [" + id + "] " + sleepTime + " " + valueToProduce); Util.sleep(sleepTime);

System.out.println(indent +

" << [" + id + "] " + sleepTime + " " + valueToProduce); return valueToProduce;

}

Die Ausgaben:

>> [12] 3500 42

>> [13] 6000 77

<< [12] 3500 42

===> f1 = 42

<< [13] 6000 77

===> f2 = 77

Hinweis: im java.util.concurrent-Package sind solche Thread-Pools bereits enthalten. Natürlich sollten also diese Standard-Pools benutzt werden. (Die her vorgestellten Pools sollen nur eine mögliche Funktionsweise solcher Pools demonstrieren.)

* 1. **ExceptionHandler**

Was passiert, wenn die run-Methode eines Threads (oder die run-Methode eines mit einem Thread assoziierten Runnables) eine RuntimeEsxception wirft (eine checked Exception kann ja bekanntlich nicht geworfen werfen – RuntimeExceptions aber sehr wohl)?

Der Thread terminiert – und wir bekommen einen Stack-Trace zu sehen. Wie aber können wir die Exception außerhalb des sie werfenden Threads auffangen?

Wir können entweder einen "globalen" UncaughtExceptionHandler registrieren oder aber einen mit dem jeweiligen Thread assoziierten Handler registrieren.

Wurde bei einem Thread ein spezieller Handler registriert, so wird beim Werfen einer RuntimeException eben dieser Handler aufgerufen. Wurde kein spezieller Handler registriert, wird der globale Handler aufgerufen (der "Default-Handler").

Ein globale Handler wird mittels einer statischen Methode registriert – mittels der Thread-Methode setDefaultUncaughtExceptionHandler. Soll für einen Thread ein spezifischer Handler registriert werden, können wir die nicht-statische Thread-Methode setUncaughtExceptionHandler nutzen.

Die folgende demo-Methode zeigt beide Varianten:

static void **demo**() {

Thread.setDefaultUncaughtExceptionHandler( (Thread t, Throwable e) ->

System.out.println("default uncaught: " + e));

final Thread t1 = new Thread(() -> { Util.sleep(1000);

System.out.println("t1 throws a RuntimeException..."); throw new RuntimeException("Exception from t1");

});

final Thread t2 = new Thread(() -> { Util.sleep(2000);

System.out.println("t2 throws a RuntimeException..."); throw new RuntimeException("Exception from t2");

});

t2.setUncaughtExceptionHandler( (Thread t, Throwable e) ->

System.out.println("uncaught: " + e));

t1.start();

t2.start();

}

Die Ausgaben:

t1 throws a RuntimeException...

default uncaught: java.lang.RuntimeException: Exception from t1 t2 throws a RuntimeException...

uncaught: java.lang.RuntimeException: Exception from t2

Die obige Anwendung startet zwei Threads. Im Kontext beider Threads wird eine RuntimeException geworfen. Die im Thread t1 geworfene Exception landet beim Default-Handler, die im Thread t2 geworfene Exception bei demjenigen Handler, der eigens für diesen Thread bereitgestellt wurde.

* 1. **ThreadLocal**

Ein ThreadLocal<T>-Objekt ist eine Registratur, in welcher ein Thread ein Objekt vom Typ T registrieren kann und dieses Objekt später dann wieder abholen kann. Mittels ThreadLocals können somit Objekte an einen Thread gebunden werden (Thread lokaler Speicher).

Ein ThreadLocal ist im Prinzip eine Map, in welcher Threads als Schüssel fungieren – und somit also jeder Thread mit einem Objekt assoziiert werden kann. Wir können z.B. Threads mit Benutzern assoziieren (die Benutzer: ein "Generalist" und ein "Philosoph"):

: ThreadLocal<String>

"Philosoph"

Thread 2

"Generalist"

Thread 1

Auf ein ThreadLocal<T>-Objekt können folgende Methoden aufgerufen werden:

* + - Mittels set(T object) wird ein Eintrag zum ThreadLocal hinzugefügt. Dabei wird nur das Objekt übergeben – nicht aber der Thread. Dieser wird in der set- Methode automatisch berechnet: via Thread.currentThread().
    - Mittels der get()-Methode kann dasjenige Objekt ermittelt werden, welches mit dem aktuellen Thread (Thread.currentThread()) assoziiert ist. get ist parameterlos und liefert ein T zurück.
    - Mittels der parameterlosen remove()-Methode schließlich kann der Eintrag für den aktuellen Thread entfernt werden.

Ein Objekt der folgenden Tracer-Klasse enthält einen ThreadLocal, der Threads auf User abbildet. Mittels der setUser-Methode wird der an diese Methode übergebene User mit dem aktuellen Thread verbunden; mittels der removeUser-Methode wird der Eintrag für den aktuellen Thread entfernt. Die trace-Methode kann den mit dem aktuellen Thread verbundenen User ermitteln und ausgeben:

public class **Tracer** {

private final ThreadLocal<String> **users** = new ThreadLocal<String>();

public void **setUser**(String user) { this.users.set(user);

}

public void **removeUser**() { this.users.remove();

}

public void **trace**(String message) {

final String user = this.users.get(); System.out.println(user + " : " + message);

}

}

In der folgenden demo-Methoden werden ein Tracer-Objekt erzeugt und zwei Threads gestartet, welche die Methoden run1 resp. run2 ausführen. Beiden Methoden teilen sich denselben Tracer.

Jeder der beiden run-Methoden registriert den aktuellen User, gibt via Tracer.trace eine Nachrichten aus und deregistriert dann den User. Der trace-Methode wir nur die auszugebende Nachricht übergeben – nicht aber der User. Trotzdem kann nun die trace-Methode auch den User ausgeben:

static void **demo**() {

final Tracer tracer = new Tracer(); Util.startAndJoin(

500,

() -> run1(tracer),

() -> run2(tracer));

}

static void **run1**(Tracer tracer) { tracer.setUser("Generalist"); Util.sleep(1000); tracer.trace("Schiller gelesen"); Util.sleep(1000); tracer.trace("PI berechnet"); Util.sleep(1000);

tracer.trace("wait/notify verstanden"); tracer.removeUser();

}

static void **run2**(Tracer tracer) { tracer.setUser("\tPhilosoph"); Util.sleep(1000); tracer.trace("Kant verstanden"); Util.sleep(1000); tracer.trace("Hegel verstanden"); tracer.removeUser();

}

Die Ausgaben:

Generalist : Schiller gelesen Philosoph : Kant verstanden

Generalist : PI berechnet Philosoph : Hegel verstanden

Generalist : wait/notify verstanden

Ein ThreadLocal-Objekt kann also verwendet werden, um Daten transparent an einen Thread zu binden und entlang einer Aufruf-Kette (von "oben" nach "unten") mitzunehmen – Daten, die ansonsten über den Parametermechanismus entlang dieser Aufruf-Kette weitergeleitet werden müssten.

Typische Verwendungen:

U.a. werden ThreadLocal-Objekt genutzt, um Datenbankverbindungen (Connections) an Threads zu binden. Ein Connection-Objekt ist nicht thread-safe – für jeden Thread muss also eine eigene Connection erzeugt werden. Die Erzeugung passiert auf "Controller"-Ebene (z.B. mittels eines Proxies); die Connection wird aber erst auf einer sehr tiefen Ebene (der "DAO"-Ebene) benötigt. Anstatt die Connection von der "Controller"-Ebene über die "Service"-Ebene zum "DAO"-Ebene via Parameter herunterzureichen, kann diese in ein ThreadLocal eingetragen werden. Die an die Service- und DAO-Ebene zu übergebenden Parameter können sich also auf fachliche Daten beschränken.

Wird z.B. JPA genutzt, können die benötigten EntityManager mittels desselben Mechanismus von oben nach unten weitergereicht werden.

* 1. **States**

Ein Thread kann folgende Zustände haben (als State-Enum in der Klasse Thread

definiert)

NEW, RUNNABLE, WAITING, TIMED\_WAITING, BLOCKING, TERMINATED

* + - NEW: Der Thread ist zwar erzeugt, aber die run-Methode ist noch nicht gestartet.
    - RUNNABLE: Der Thread rechnet oder schreibt oder liest
    - WAITING: Es wurde u.a. eine der folgenden Methoden aufgerufen: join() wait().
    - TIMED\_WAITING: es wurde eine überladene WAITING-Methode aufgerufen – z.B.:

join(timeout), wait(timeout).

* + - BLOCKED: der Thread wartet darauf, einen synchronized-Block betreten zu können.
    - TERMINATED: der Thread ist beendet.

package **appl**;

// ...

public class **Application** {

public static void **main**(String[] args) { final int MILLIS = 2000;

Thread thread = new Thread(() -> {

sleep(MILLIS); calculate(MILLIS); read(MILLIS); wait(MILLIS); waitMillis(MILLIS); block(MILLIS);

});

thread.start();

while (thread.getState() != Thread.State.TERMINATED) { System.out.println(thread.getState()); Util.sleep(MILLIS / 4);

}

}

static void **sleep**(int millis) { Util.sleep(millis);

}

static double **calculate**(int millis) { double sum = 0;

long start = System.currentTimeMillis();

while (System.currentTimeMillis() - start < millis) { sum += Math.sqrt(2);

}

return sum;

}

static void **read**(int millis) {

long start = System.currentTimeMillis();

while (System.currentTimeMillis() - start < millis) { try(Socket socket = new Socket("localhost", 8000)) {

System.out.println(socket);

}

catch (IOException e) {

}

}

}

static void **wait**(int millis) {

final Object lock = new Object(); Thread notifier = new Thread(() -> {

Util.sleep(millis); synchronized (lock) {

lock.notify();

}

});

notifier.setDaemon(true); notifier.start(); synchronized (lock) {

Try.execute(() -> lock.wait());

}

}

static void **waitMillis**(int millis) { final Object lock = new Object();

long start = System.currentTimeMillis();

while (System.currentTimeMillis() - start < millis) { synchronized (lock) {

Try.execute(() -> lock.wait(1000));

}

}

}

static void **block**(int millis) {

final Object lock = new Object(); Thread notifier = new Thread(() -> {

synchronized (lock) { Util.sleep(millis); lock.notify();

}

});

notifier.setDaemon(true); notifier.start(); Util.sleep(100); synchronized (lock) {

}

}

}

Die Ausgaben:

RUNNABLE TIMED\_WAITING TIMED\_WAITING TIMED\_WAITING RUNNABLE RUNNABLE RUNNABLE RUNNABLE RUNNABLE RUNNABLE RUNNABLE RUNNABLE RUNNABLE WAITING WAITING WAITING WAITING TIMED\_WAITING TIMED\_WAITING TIMED\_WAITING TIMED\_WAITING BLOCKED BLOCKED BLOCKED BLOCKED

* 1. **Enumerate**

Mittels der statischen Methode Thread.enumerate können die aktuell laufenden Threads ermittelt werden. Der Methode wird ein Array mit Referenzen vom Typ Thread übergeben (dieser Array fungiert als Output-Parameter); die Methode liefert die Anzahl der laufenden Threads zurück.

Hier eine Demo-Anwendung:

package **appl**; import jn.util.Try;

public class **Application** {

public static void **main**(String[] args) { final Thread first = new Thread(

() -> Try.execute(() -> Thread.sleep(1000))); first.setName("first");

first.start();

final Thread second = new Thread(

() -> Try.execute(() -> Thread.sleep(1000))); second.setName("second");

second.start();

final Thread[] threads = new Thread[64]; final int n = Thread.enumerate(threads); for (int i = 0; i < n; i++) {

final Thread thread = threads[i]; System.out.println(thread.getId() + " " + thread.getName());

}

}

}

Die Ausgaben:

1 main

1. first
2. second

**17 Das concurrent-Package**

Im folgenden werden einige wichtige Klassen des Pakets java.util.concurrent

beschrieben.

Was Nebenläufigkeit angeht, begnügte sich Java vor 1.5 mit dem Interface Runnable, der Klasse Thread und einigen wenigen Synchronisations-Mechanismen: mit volatile, synchronized und den Object-Methoden wait, notify und notifyAll. Es handelte sich hier also um ein sehr elegantes, flexibel nutzbares und schlankes Konzept.

Concurrent Programming hieß daher aber immer auch, dass man eigene Klassen für wiederkehrende Aufgaben schreiben musste – Queue-Klassen für den Austausch von Produkten, Klassen zum Poolen von Threads etc.

Das concurrent-Paket definiert nun eine Reihe von Standardklassen, die für solche Aufgaben verwendet werden können. Vieles von dem, was wir bislang selbst implementieren mussten, kommt also in Form fertiger Bibliotheks-Klassen daher.

### Übersicht

* Atomic-Klassen
* Thread-sichere Collections
* ReentrantLocks
* Conditions.
* Queues.
* Thread-Pools (Executors)
* Fork-Join
* CompletableFutures
  1. **Atomics**

Die Lese- und Schreib-Zugriffe auf Variablen primitiven Typs (außer long und double) sind atomar; ebenso atomar sind Lese- und Schreib-Zugriffe auf Referenzvariablen.

Zuweilen aber benötigt man eine Operation auf solche, die aus mehreren Aktionen besteht. Natürlich könnte man auch hier synchronized oder Locks verwenden. Manchmal aber umfassen die Operationen nur wenige, immer wieder benötigte Schritte.

Man möchte z.B. eine int-Variable namens num inkrementieren. Die einfachste Variante:

num++;

Hinter dieser einen Quellcode-Zeile verbergen sich drei Aktionen:

* + - lesen des num-Werts (laden des num-Werts in ein Register)
    - inkrementieren des Register-Werts
    - schreiben des Ergebnissen in die num-Variable (übertragen des Registerwerts in die Variable).

Da diese Anweisungsfolge unterbrechbar ist, müssten wir synchronisieren – im einfachsten Fall mit synchronized:

synchronized { num++;

}

Synchronisation ist bekanntlich teuer. Prozessoren bieten heute Mittel an, mittels derer solche Operationen billiger zu haben sind. Und seit Java-5 können diese Hardware- Mittel nun auch genutzt werden.

Java-5 führt eine Reihe von Klassen ein, deren Methoden atomar ausgeführt werden:

AtomicBoolean

AtomicInteter, AtomicIntergerArrray, AtomicIntegerFieldUpdater AtomicLong, AtomicLongArrray, AtomicLongFieldUpdater AtomicReference, AtomicReferenceArrray, AtomicReferenceFieldUpdater AtomicMarkableReference

AtomicStampedReference

Die Klasse AtomicInteger wird im folgenden näher erläutert.

static void **demoIncrementAndGet**() { AtomicInteger i = new AtomicInteger(42);

int j = i.incrementAndGet(); System.out.println(j); // 43

}

static void **demoGetAndIncrement**() { AtomicInteger i = new AtomicInteger(42); int j = i.getAndIncrement(); System.out.println(j); // 42

}

static void **demoAddAndGet**() {

AtomicInteger i = new AtomicInteger(42); int j = i.addAndGet(77); System.out.println(j); // 119

}

static void **demoGet**() {

AtomicInteger i = new AtomicInteger(42); int j = i.get();

System.out.println(j); // 42

}

static void **demoSet**() {

AtomicInteger i = new AtomicInteger(); i.set(42); System.out.println(i.get()); // 42

}

* 1. **ConcurrentHashMap**

Die Methoden der Klasse HashMap sind bekanntlich nicht synchronisiert. Wir können leicht zeigen, dass eine HashMap korrumpiert wird, wenn mehrere Threads gleichzeitig schreibende Methoden aufrufen.

Wir könnten einer HashMap ein Synchronisations-Proxy voranstellen – ein solches Proxy erhalten wir von der Collections-Methode synchronizedMap. Dieses Proxy nutzt als Lock die HashMap selbst – und dies ist nicht allzu performant. Besser ist die Benutzung der Klasse ConcurrentHashMap.

package **appl**;

// ...

public class **Application** {

public static void **main**(String[] args) throws Exception { demoMapPerformance(

new HashMap<Integer, String>()); demoMapPerformance(

Collections.synchronizedMap(new HashMap<Integer, String>())); demoMapPerformance(

new ConcurrentHashMap<Integer, String>());

}

static void **demoMapPerformance**(final Map<Integer, String> map) { final int N = 10000; PerformanceRunner.run(map.getClass().getName(), 1000, () -> {

final Thread t1 = new Thread(() -> { for (int i = 0; i < N; i++)

map.put(i, String.valueOf(i));

});

final Thread t2 = new Thread(() -> { for (int i = 0; i < N; i++)

map.put(i, String.valueOf(i));

});

}

});

t1.start();

t2.start();

Try.execute(() -> t1.join());

Try.execute(() -> t2.join());

|  |  |
| --- | --- |
| } |  |
| Die Ausgaben sprechen für sich: |
| java.util.HashMap : | 1248 |
| java.util.Collections$SynchronizedMap : | 1732 |
| java.util.concurrent.ConcurrentHashMap : | 1082 |

* 1. **CopyOnWriteArrayList**

Eine CopyOnWriteArrayList kopiert bei jedem Aufruf der add-Methode den internen Elemente-Array – in einen neu allokierten Array, der exakt um ein Element größer ist als der alte.

Der (etwas angepasste) Quellcode der add-Methode:

public boolean **add**(E e) { synchronized (lock) {

int len = this.elements.length;

Object[] newElements = Arrays.copyOf(this.elements, len + 1); this.elements[len] = e;

return true;

}

}

Das ist natürlich zeitaufwendig – insbesondere dann, wenn die Liste bereits eine große Anzahl von Elementen beinhalten.

Im folgenden zeigen wir, was passiert, wenn mehrere Threads beteiligt sind. Der eine Thread schreibt (add), der andere liest (mittels eines Iterators):

static void **demoReadWrite**(final List<Integer> list) throws Exception { Log.logMethodCall(list.getClass().getName());

for (int i = 0; i < 3; i++) list.add(i);

new Thread(() -> {

Iterator<Integer> iter = list.iterator(); try {

while (iter.hasNext()) {

Try.execute(() -> Thread.sleep(500)); System.out.println(iter.next());

}

}

catch (ConcurrentModificationException ex) { System.out.println(ex);

}

}).start();

for (int i = 0; i < 3; i++) { Thread.sleep(500); list.add(i);

}

}

(Statt des expliziten Iterators hätten wir natürlich auch den "impliziten" benutzten können, der von der for-each-Schleife verwendet wird.)

Wir rufen diese Methode mit einer ArrayList auf:

demoReadWrite(new ArrayList<Integer>());

Die (möglichen) Ausgaben:

java.util.ConcurrentModificationException

Dann rufen wir diese Methode mit einer CopyOnWriteArrayList auf:

demoReadWrite(new CopyOnWriteArrayList<>());

Die Ausgaben:

0

1

2

Die ArrayList wirft eine Exception; der Iterator, den die CopyOnWriteArrayList geliefert hat, operiert genau auf dem Array, der zum Zeitpunkt der Erzeugung existierte (und der ja bei jedem add wieder durch einen anderen ersetzt wird).

Betrachten wir zuletzt die Performance:

static void **demoListPerformance**(final List<Integer> list) { final int N = 100;

PerformanceRunner.run(list.getClass().getName(), 100, () -> { final Thread t1 = new Thread(() -> {

for (int i = 0; i < N; i++) list.add(i);

});

final Thread t2 = new Thread(() -> { for (int i = 0; i < N; i++)

list.add(i);

});

t1.start();

t2.start();

Try.execute(() -> t1.join());

Try.execute(() -> t2.join());

});

System.out.println("Number of Elements = " + list.size());

}

Wir fügen in zwei Threads jeweils 100 \* 100 Elements zu der List hinzu. Wir rufen die Methode mit einer ArrayList auf. Die (möglichen) Ausgaben:

java.util.ArrayList : 77

Number of Elements = 19796

Und rufen sie dann mit einer CopyOnWriterArrayList auf. Die Ausgaben:

java.util.concurrent.CopyOnWriteArrayList : 242 Number of Elements = 20000

Die ArrayList ist natürlich viel flotter als die CopyOnWriteArrayList – funktioniert aber leider nicht korrekt. Die CopyOnWriteArrayList funktioniert – ist aber etwa um den Faktor 3 langsamer als die ArrayList.

Die CopyOnWriteArrayList sollte man nur bei kleinen Listen verwenden – z.B. für die Registrierung von Listenern. Solche Registrierungen sind "selten" – aber sehr häufig werden wir über eine solche Listener-Liste iterieren wollen.

* 1. **ReentrantLock**

Ein ReentrantLock kann wie folgt erzeugt werden:

ReentrantLock lock = new ReentrantLock(true);

(Der Parameter des Konstruktor entscheidet über die "Fairnis" des Locks.) Ein solcher Lock kann wie folgt genutzt werden:

this.lock.lock(); try {

// do critical work...

}

finally {

this.lock.unlock();

}

Neben der lock-Methode existieren noch tryLock-Methoden – siehe die JavaDoc.

Ein ReentrantLock kann überall dort eingesetzt werden, wo im "alten" Java ein synchronized-Block verwendet werden musste. Im Gegensatz zu synchronized müssen wird einen ReentrantLock explizit freigeben.

Das bekannte Account-Beispiel – nun aber statt mit synchronized mittels eines

ReentrantLocks implementiert:

package **appl**;

import java.util.concurrent.locks.ReentrantLock; import jn.util.threading.Util;

public class **Account** {

private final int **number**; private int **balance**;

private final ReentrantLock **lock** = new ReentrantLock(true); public **Account**(int number) {

this.number = number;

}

public int **getNumber**() { return this.number;

}

public void **deposit**(int amount) { checkAmount(amount); this.lock.lock();

Log.tlog(

"withdraw - holdCount = " + this.lock.getHoldCount()); try {

Util.sleep(1000); this.balance += amount;

}

finally {

this.lock.unlock();

}

}

public void **withdraw**(int amount) { checkAmount(amount); this.lock.lock();

Log.tlog(

"withdraw - holdCount = " + this.lock.getHoldCount());

try {

if (this.getBalance() < amount)

throw new IllegalArgumentException( "withdraw not possible: " + amount);

Util.sleep(1000); this.balance -= amount;

}

finally {

this.lock.unlock();

}

}

public int **getBalance**() { this.lock.lock(); Log.tlog(

"getBalance - holdCount = " + this.lock.getHoldCount()); try {

return this.balance;

}

finally {

this.lock.unlock();

}

}

private static void **checkAmount**(int amount) { if (amount <= 0)

throw new IllegalArgumentException("illegal amount : " + amount);

}

}

getHoldCount liefert die Anzahl der "holds" zurück, die der aktuelle Thread auf das Lock-Objekt besitzt. Ein ReentrantLock enthält einen Zähler, der bei jedem lock inkrementiert wird und beim unlock jeweils dekrementiert wird. Ein und derselbe Thread kann also mehrfach hintereinander lock (oder eine der zusätzlichen lock-Varianten) aufrufen. Er sollte dann natürlich ebenso häufig die unlock-Methode aufrufen. getHoldCount liefert genau den Wert dieses Zählers zurück. (Ein ReentrantLock heißt eben deshalb so, weil ein Thread einen solchen Lock mehrfach in Besitz nehmen kann.)

Die folgende Application ruft in zwei Threads "gleichzeitig" die withdraw-Methode auf ein Account-Objekt auf. Der erste Aufruf gelingt, beim zweiten Aufruf wird eine Exception geworfen:

package **appl**;

import jn.util.threading.Util; public class **Application** {

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

final Account account = new Account(4711); account.deposit(5000);

Util.startAndJoin(

500,

() -> withdraw(account, 4000), () -> withdraw(account, 3000));

System.out.println("Balance = " + account.getBalance());

}

static void **withdraw**(Account account, int amount) { try {

account.withdraw(amount);

}

catch (Exception e) { System.out.println(e);

}

}

}

Die Ausgaben:

[ 1] -- withdraw - holdCount = 1

[12] -- withdraw - holdCount = 1

1. -- getBalance - holdCount = 2
2. -- withdraw - holdCount = 1

[13] -- getBalance - holdCount = 2 java.lang.IllegalArgumentException: withdraw not possible: 3000 [ 1] -- getBalance - holdCount = 1

Balance = 1000

* 1. **ReentrantReadWriteLock**

Ein ReentrantReadLock kann zwei verschiedene Locks liefern – einen ReadLock und einen WriteLock. Besitzt ein Thread einen WriteLock, wird ein anderer Thread, der entweder einen ReadLock oder einen WriteLock anfordert, blockiert. Besitzt ein Thread einen ReadLock, wird ein weiterer Thread, der einen WriteLock anfordert, ebenfalls blockiert. Ein solcher Thread kann aber einen ReadLock erhalten.

Ein WriteLock wird für schreibende, ein ReadLock für lesende Zugriffe verwendet. Die neue Account-Klasse:

package **appl**;

import java.util.concurrent.locks.ReentrantReadWriteLock;

import java.util.concurrent.locks.ReentrantReadWriteLock.ReadLock; import java.util.concurrent.locks.ReentrantReadWriteLock.WriteLock;

import jn.util.threading.Util; public class **Account** {

private final int **number**; private int **balance**;

private final ReentrantReadWriteLock **lock** = new ReentrantReadWriteLock(true);

private final ReadLock **readLock** = this.lock.readLock(); private final WriteLock **writeLock** = this.lock.writeLock();

public **Account**(int number) { this.number = number;

}

public int **getNumber**() { return this.number;

}

public void **deposit**(int amount) { checkAmount(amount); this.writeLock.lock();

try { Util.sleep(1000);

this.balance += amount;

}

finally {

this.writeLock.unlock();

}

}

public void **withdraw**(int amount) { checkAmount(amount); this.writeLock.lock();

try {

if (this.getBalance() < amount)

throw new IllegalArgumentException( "withdraw not possible: " + amount);

Util.sleep(1000); this.balance -= amount;

}

finally {

this.writeLock.unlock();

}

}

public int **getBalance**() { this.readLock.lock(); try {

return this.balance;

}

finally {

this.readLock.unlock();

}

}

private static void **checkAmount**(int amount) { if (amount <= 0)

throw new IllegalArgumentException("illegal amount : " + amount);

}

}

* 1. **Conditions**

Das Interface Lock (und damit auch die Klasse ReentrantLock) besitzt eine Methode, die bislang noch nicht näher erläutert wurde: newCondition. Diese Methode liefert ein Objekt zurück, dessen Klasse das Interface Condition implementiert. Das Interface Condition spezifiziert u.a. folgende Methoden:

public interface **Condition** {

void **await**() throws InterruptedException; void **signal**();

void **signalAll**();

// ...

}

Die Methoden dieses Interfaces erinnern natürlich an die Object-Methoden wait, notify und notifyAll. Diese "alten" Methoden werden also zu einer neuen Klasse zusammengefasst.

Eine Condition ist signalisiert oder nicht signalisiert. Unmittelbar nach seiner Erzeugung ist es signalisiert. Ein await-Aufruf auf eine signalisierte Condition wird sofort zurückkehren. Vor der Rückkehr von await wird die Condition in den nicht- signalisierten Status gesetzt. Der Aufruf von signal setzt die Condition wieder in den signalisierten Status.

Wichtig ist der Hinweis auf die beiden signal-Varianten. Während signal nur einen einzigen Thread aufweckt, der auf die Condition wartet, weckt signalAll alle wartenden Threads auf (nur einer wird natürlich die Condition in Besitz nehmen können – die anderen werden wieder schlafen gelegt). Wenn man also davon ausgehen muss, dass mehr als nur ein Thread auf die Condition warten kann, sollte unbedingt signalAll verwendet werden. (Es gelten hier also dieselben Überlegungen wie bei der Verwendung der Object-Methoden notify und notifyAll.)

Das Beispiel:

Eine eingleisige Bahnstrecke kann zu einer Zeit immer nur von einem Zug befahren werden. Es gibt an beiden Enden einer solchen Strecke ein Signal, welches anzeigt, ob die Strecke frei ist oder nicht. Ein Zug, der die Strecke benutzen möchte, muss warten, bis das Signal grün anzeigt. Wenn der Zug dann die eingleisige Strecke betritt, schalten die Signale auf rot.

Die eingleisige Bahnstrecke kann durch folgende Klasse beschrieben werden:

package **appl**;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.ReentrantLock; public class **SingleTrack** {

private final ReentrantLock **lock** = new ReentrantLock(true); private final Condition **free** = this.lock.newCondition(); private boolean **isFree** = true;

public void **enter**() { this.lock.lock(); try {

while (!this.isFree) this.free.await();

this.isFree = false;

}

catch(final InterruptedException e) { throw new RuntimeException(e);

}

finally {

this.lock.unlock();

}

}

public void **leave**() { this.lock.lock(); try {

this.isFree = true; this.free.signalAll();

}

finally {

this.lock.unlock();

}

}

}

Man beachte, dass die free-Variable die isFree-Variable nicht ersetzen kann! Man benötigt also beide Variablen. (Denn eine Condition kennt natürlich keine spezifische Semantik – zumal es sich auch um eine "komplexe" Semantik handeln könnte.)

Um die Methoden await und signal (bzw. signalAll) aufrufen zu können, muss man zuvor das Lock-Objekt vorgemerkt haben, an welches die entsprechenden Conditions gebunden sind (ebenso wie man bei der Benutzung von wait und notify zuvor das Objekt über einen synchronized-Block hat vormerken müssen).

Sowohl await (und die Varianten von await) als auch signal (und signalAll) sind also nur in einem lock/unlock-Kontext aufrufbar.

Die Application schickt zwei Züge auf den Weg – beide müssen die einspurige Strecke überwinden. Das Verhalten jedes der beiden Züge wird in der run-Methode definiert (die in zwei Threads aufgerufen wird):

package **appl**;

import jn.util.threading.Util; public class **Application** {

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

final SingleTrack track = new SingleTrack(); Util.startAndJoin(

500,

() -> run("", track),

() -> run("\t\t", track));

}

static void **run**(String label, SingleTrack track) { for (int i = 0; i < 2; i++) {

System.out.println(label + "running"); Util.sleep(1000);

}

System.out.println(label + "try to enter..."); track.enter();

System.out.println(label + "entered!"); for (int i = 0; i < 3; i++) {

System.out.println(label + "running on SingleTrack"); Util.sleep(1000);

}

System.out.println(label + "exiting"); track.leave();

for (int i = 0; i < 2; i++) { System.out.println(label + "running"); Util.sleep(1000);

}

}

}

Die Ausgaben:

running

running running

running try to enter... entered!

running on SingleTrack try to enter...

running on SingleTrack running on SingleTrack exiting

running

entered!

running on SingleTrack

running on SingleTrack running

running on SingleTrack exiting

running running

Die "linken" Ausgaben produziert der erste Zug, die "rechten" der zweite.

* 1. **ArrayBlockingQueue**

Eine ArrayBlockingQueue ist eine Queue mit begrenzter Anzahl von Elementen (ein "bounded buffer"), die in einem Array verwaltet werden. Die Größe des Arrays wird dem Konstruktor übergeben. Einmal erzeugt, kann die Größe einer solchen Queue nicht mehr geändert werden. Der interne Array wird als Ringpuffer verwendet.

package **appl**;

import java.util.concurrent.ArrayBlockingQueue; import java.util.concurrent.BlockingQueue; import jn.util.threading.Util;

public class **Application** {

public static void **main**(String[] args) { final ArrayBlockingQueue<String> queue =

new ArrayBlockingQueue<String>(3, true); Util.startAndJoin(

0,

() -> produce(queue),

() -> consume(queue));

}

static void **produce**(ArrayBlockingQueue<String> queue) { try {

for (int i = 0; i < 5; i++) {

final int sleeptime = (int) (Math.random() \* 1000); Util.sleep(sleeptime);

System.out.println(i + " >"); queue.put(String.valueOf(i));

}

queue.put("");

}

catch (final InterruptedException e) { throw new RuntimeException(e);

}

}

static void **consume**(BlockingQueue<String> queue) { try {

while (true) {

final String product = queue.take(); System.out.println("\t> " + product); if (product.length() == 0)

break; Util.sleep(1000);

}

}

catch (final InterruptedException e) { throw new RuntimeException(e);

}

}

}

Die möglichen Ausgaben (die linken Ausgaben stammen vom "Produzenten", die rechten vom "Konsumenten"):

0 >

* 0

1 >

* 1

2 >

* 2

3 >

4 >

* 3
* 4

>

* 1. **Executor mit Runnables**

Executors sind Thread-Pools. Die Klasse Executors enthält eine Vielzahl statischer Factory-Methoden für solche Pools. Im folgenden wird ein Pool erzeugt, der mit einer fixen Anzahl Threads arbeitet.

Die folgende Anwendung stellt Runnables in den Thread-Pool ein – mittels der Methode

execute:

package **appl**;

import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors; import java.util.concurrent.TimeUnit;

import jn.util.Watch;

import jn.util.threading.Util; public class **Application** {

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

final ExecutorService pool = Executors.newFixedThreadPool(4); final Watch watch = new Watch();

pool.execute(() -> consumeTime("", 5000)); pool.execute(() -> consumeTime("\t", 2500)); pool.execute(() -> consumeTime("\t\t", 3500)); pool.execute(() -> consumeTime("\t\t\t", 6000));

pool.shutdown(); try {

while (! pool.awaitTermination(100, TimeUnit.MILLISECONDS))

;

}

catch (final InterruptedException e) { throw new RuntimeException(e);

}

System.out.println("duration = " + watch);

}

static void **consumeTime**(String indent, int sleepTime) { final long id = Thread.currentThread().getId();

System.out.println(indent + " >> [" + id + "] " + sleepTime); Util.sleep(sleepTime);

System.out.println(indent + " << [" + id + "] " + sleepTime);

}

}

Mögliche Ausgaben:

>> [12] 5000

>> [14] 3500

>> [13] 2500

>> [15] 6000

<< [13] 2500

<< [14] 3500

<< [12] 5000

<< [15] 6000

duration = 6019

* 1. **Executor mit Callables**

Ein Executor kann auch Callables ausführen (wobei die submit-Methode, der solche

Callables übergeben werden, jeweils ein Future-Objekt zurückliefert):

package **appl**;

import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors; import java.util.concurrent.Future;

import java.util.concurrent.LinkedBlockingQueue; import java.util.concurrent.ThreadPoolExecutor; import java.util.concurrent.TimeUnit;

import jn.util.Watch;

import jn.util.threading.Util; public class **Application** {

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

final ExecutorService pool = Executors.newFixedThreadPool(4); final Watch watch = new Watch();

final Future<Integer> f1 =

pool.submit(() -> produceValue("", 3500, 42)); final Future<Integer> f2 =

pool.submit(() -> produceValue("\t", 6000, 77));

try {

System.out.println("==> " + f1.get()); System.out.println("==> " + f2.get());

}

catch (final Exception e) {

throw new RuntimeException(e);

}

pool.shutdown(); try {

while (!pool.awaitTermination(100, TimeUnit.MILLISECONDS))

;

}

catch (final InterruptedException e) { throw new RuntimeException(e);

}

System.out.println("duration = " + watch);

}

static Integer **produceValue**(

String indent, int sleepTime, int valueToProduce) {

final long id = Thread.currentThread().getId(); System.out.println(indent + " >> [" + id + "] " +

sleepTime + " " + valueToProduce); Util.sleep(sleepTime);

System.out.println(indent + " << [" + id + "] " + sleepTime + " " + valueToProduce);

return valueToProduce;

}

}

Mögliche Ausgaben:

>> [12] 3500 42

>> [13] 6000 77

<< [12] 3500 42

==> 42

<< [13] 6000 77

==> 77

duration = 6023

* 1. **ForkJoin**

Um z.B. einen großen Array zu sortieren, kann dieser in zwei Hälften aufgesplittet werden. Der erste Thread sortiert die erste Hälfte, der zweite Thread sortiert die zweite. Dann werden beide Ergebnisse zusammen gemischt.

Natürlich kann der Array auch in Viertel aufgeteilt werden und somit die Sortierung vier Thread überlassen werden. Dann müssen drei merge-Aktionen ausgeführt werden: die Ergebnisse vom ersten und zweiten Thread müssen gemischt werden, die Ergebnisse des dritten und vierten und schließlich müssen die beiden Misch-Resultate zu dem End- Resultat gemischt werden.

Hier zunächst eine Anwendung, welche die Summe aller in einem (großen) Array enthaltenen Elemente berechnet.

Wir bauen eine Klasse NumberAddingTask, die von der Standardklasse RecursiveTask

abgeleitet ist – wobei wir die Compute-Methode dieser Klasse überschreiben:

package **appl**;

import java.util.concurrent.RecursiveTask; import jn.util.Log;

import jn.util.threading.Util;

public class **NumberAddingTask** extends RecursiveTask<Integer> { private static final long **serialVersionUID** = 1L;

final int[] **array**; final int **start**; final int **end**;

int **result**;

public **NumberAddingTask**(int[] array) { this(array, 0, array.length);

}

private **NumberAddingTask**(int[] array, int start, int end) { this.array = array;

this.start = start; this.end = end;

}

@Override

protected Integer **compute**() {

if (this.end - this.start <= 10) {

Log.tlog("compute " + this.start + " " + this.end); int sum = 0;

for (int i = this.start; i < this.end; i++) {

sum += this.array[i]; Util.sleep(10);

}

this.result = sum;

}

else {

Log.tlog("split " + this.start + " " + this.end); final int center = (this.end + this.start) / 2; final NumberAddingTask t0 =

new NumberAddingTask(this.array, this.start, center); final NumberAddingTask t1 =

new NumberAddingTask(this.array, center, this.end); RecursiveTask.invokeAll(t0, t1);

this.result = t0.result + t1.result;

}

return this.result;

}

}

Sofern der Array, dessen Elements summiert werden soll, klein ist, wird die Summierung vom aktuellen Thread erledigt.

Anderenfalls wird das Array zweigeteilt: es werden zwei neue NumberAddingTasks erzeugt und an die statische invokeAll-Methode von RecursiveTask übergeben. Diese Methode sorgt dafür, dass die beiden NumberAddingTasks in zwei verschiedenen Threads ausgeführt werden.

Die Resultate werden in dem jeweiligen NumberAddingTask gespeichert, so dass sie nach Rückkehr von invokeAll zu einem neuen Resultat zusammenfasst werden können.

Die Anwendung:

package **appl**;

import java.util.concurrent.ForkJoinPool; import jn.util.Watch;

public class **Application** {

public static void **main**(String[] args) { final int N = 100;

final int[] array = new int[N]; for (int i = 0; i < N; i++)

array[i] = i + 1;

final NumberAddingTask task = new NumberAddingTask(array); final ForkJoinPool pool = new ForkJoinPool(4);

final Watch watch = new Watch(); final int sum = pool.invoke(task);

System.out.println("Duration: " + watch); final int expected = N/2 \* (N + 1);

System.out.println("expeced = " + expected + " actual = " + sum);

}

}

Die Ausgaben:

[12] -- split 0 100

[12] -- split 0 50

[13] -- split 50 100

[12] -- split 0 25

[14] -- split 25 50

[13] -- split 50 75

[14] -- split 25 37

[15] -- split 12 25

1. -- split 0 12

[15] -- compute 12 18

[14] -- compute 25 31

[13] -- split 50 62

1. -- compute 50 56

[12] -- compute 0 6

1. -- compute 6 12

[14] -- compute 31 37

1. -- compute 56 62

[15] -- compute 18 25

[12] -- split 37 50

1. -- compute 37 43

[13] -- split 62 75

1. -- compute 62 68
2. -- compute 43 50

[15] -- split 75 100

[15] -- split 75 87

1. -- compute 75 81
2. -- compute 68 75

[13] -- split 87 100

1. -- compute 87 93

[15] -- compute 81 87

[13] -- compute 93 100 Duration: 374

expeced = 5050 actual = 5050

* 1. **CompletableFuture**

apply(x) x - 1

*Function*

Wir können gerichtete Grafen benutzen, um Aktivitäten zu beschreiben, von denen möglicherweise mehrere parallel ausgeführt werden können. Solche Grafen können beliebig komplex sein. Die Knoten münden i.d.R. in einen einzigen Ende-Knoten, welcher das Resultat der gesamten Berechnung repräsentiert.

Ein Beispiel:

Ausgehend von einem Anfangswert, der gesetzt werden kann, sollen zunächst zwei Resultate berechnet werden: der nächst höhere Wert und der nächst niedrigere Wert. Diese beiden Resultate sollen dann multipliziert werden.

Zunächst ein Schaubild:

: CF

: CF

: CF

complete(10)

*Function*

99  get()

*BiFunction*

apply(x, y) x \* y

apply(x) x + 1

: CF

CF steht für CompletableFuture. Wir erzeugen vier solcher Objekte und verbinden diese. Das oberste Objekte repräsentiert den Anfangswert. Das linke Kind dieses Knotens repräsentiert eine Berechnung, die durch eine Function vorgegeben ist. Auch das rechte Kind erfordert eine Function-Berechnung. Diese beiden Kinder münden in den untersten Knoten, dessen Berechnung eine BiFunction erfordert.

Die beiden mittleren Knoten können parallel ausgeführt werden. Zur Implementierung.

Die beiden mittleren Knoten werden durch die Methode thenApplyAsync erzeugt. Diese Methode wird auf den obersten Knoten aufgerufen (und ihr wird eine Function übergeben). Der unterste Knoten wird durch die Methode thenCombine erzeugt. Diese Methode wird auf den linken Knoten aufgerufen. Ihr wird der rechte Knoten und eine BiFuntion übergeben.

Wenn der Graf aufgebaut ist, wird auf den obersten Knoten die Methode complete aufgerufen – und somit den Berechnungen ihr Anfangswert übergeben. Dann können wir auf das Ergebnis warten – indem wir auf den untersten Knoten die get-Methode aufrufen.

import java.util.concurrent.CompletableFuture; import java.util.concurrent.ExecutionException;

static void **demo1**() {

final CompletableFuture<Integer> f1 = new CompletableFuture<>(); final CompletableFuture<Integer> f2 = f1.thenApplyAsync(x -> x + 1); final CompletableFuture<Integer> f3 = f1.thenApplyAsync(x -> x - 1); final CompletableFuture<Integer> f4 =

f2.thenCombine(f3, (x, y) -> x \* y); f1.complete(10);

try { System.out.println(f4.get());

}

catch (InterruptedException | ExecutionException e) { throw new RuntimeException(e);

}

}

Die Ausgabe: 99

In der folgenden Anwendung sehen wir den Pythagoras bei seiner Arbeit – er berechnet aufgrund von zwei Katheten die Hypotenuse eines rechtwinkligen Dreiecks.

static void **demo2**() {

final CompletableFuture<Double> f1 = new CompletableFuture<>(); final CompletableFuture<Double> f2 = new CompletableFuture<>(); final CompletableFuture<Double> f3 = f1.thenApplyAsync(x -> x \* x); final CompletableFuture<Double> f4 = f2.thenApplyAsync(x -> x \* x); final CompletableFuture<Double> f5 =

f3.thenCombine(f4, (v1, v2) -> v1 + v2);

final CompletableFuture<Double> f6 = f5.thenApply(x -> Math.sqrt(x)); f1.complete(4.0);

f2.complete(3.0); try {

System.out.println(f6.get());

}

catch (InterruptedException | ExecutionException e) { throw new RuntimeException(e);

}

}

*Function*

apply(x) x \* x

: CF

apply(x) x \* x

: CF

*Function*

*Function*

apply(x) sqrt(x)

: CF

Die Ausgabe: 5.0

Hier der duch die obige Anwendung aufgebaute Graf:

: CF

: CF

complete(4.0) complete(3.0)

*BiFunction*

apply(x, y) x + y

: CF

5.0  get()

* 1. **SimpleCompletableFuture**

Im folgenden wird eine sehr einfache Implementierung des CompletableFuture- Konzepts vorgestellt. Das Interface, von dem die Knotenklassen abgeleitet sind, wird hier als Node bezeichnet.

Hier zunächst eine einfache Anwendung:

package **appl**; import util.Node;

public class **Application** {

public static void **main**(String[] args) { final Node<Double> n1 = Node.create(); final Node<Double> n2 = Node.create();

final Node<Double> n3 = n1.thenApply(a -> a \* a); final Node<Double> n4 = n2.thenApply(b -> b \* b);

final Node<Double> n5 = n3.thenCombine(n4, (aq, bq) -> aq + bq); final Node<Double> n6 = n5.thenApply(s -> Math.sqrt(s)); n1.begin(3.0);

n2.begin(4.0);

Double result = n6.get(); System.out.println("Result = " + result);

}

}

Ein Klassendiagramm:

*AbstractNode*

*Node*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | |  | | | |
|  |  | | |  | | |  |
| StartNode | |  | UnaryNode | |  | BinaryNode | |

*Function*

*BiFunction*

Ein Aufruf von Node.create erzeugt einen StartNode.

Auf einen Node kann thenApply aufgerufen werden. Diese Methode erzeugt einen

UnaryNode (dem die an thenApply übergebene Function mitgegeben wird).

Auf einen Node kann thenCombine aufgerufen werden. Diese Methode erzeugt einen

BinaryNode (dem die an thenCombine übergebene BiFunction mitgegeben wird). Das genaue Studium der Implementierung sei dem Leser / der Leserin überlassen…

package **util**;

import java.util.function.BiFunction; import java.util.function.Function;

public interface **Node**<T> {

public static <T> Node<T> **create**() { return new StartNode<T>();

}

public abstract <R> Node<R> **thenApply**(

Function<? super T, ? extends R> function);

public abstract <T1, R> Node<R> **thenCombine**(Node<T1> other, BiFunction<? super T, ? super T1, ? extends R> function);

public abstract void **begin**(T value); public abstract T **get**();

}

package **util**;

import java.util.ArrayList; import java.util.List;

import java.util.concurrent.CountDownLatch; import java.util.concurrent.Executor; import java.util.concurrent.Executors; import java.util.function.BiFunction; import java.util.function.Function;

import java.util.function.Supplier; import jn.util.Try;

abstract class **AbstractNode**<T> implements Node<T> {

protected final List<Node<?>> **nextNodes** = new ArrayList<>();

protected final CountDownLatch **countDownLatch** = new CountDownLatch(1); protected volatile T **result**;

@Override

public final <R> Node<R> **thenApply**(

Function<? super T, ? extends R> function) {

final UnaryNode<R> nextNode = new UnaryNode<>(function); this.nextNodes.add(nextNode);

return nextNode;

}

@Override

public final <T1, R> Node<R> **thenCombine**( Node<T1> other,

BiFunction<? super T, ? super T1, ? extends R> function) { final BinaryNode<R> nextNode =

new BinaryNode<>(this, (AbstractNode<?>) other, function); this.nextNodes.add(nextNode);

((AbstractNode<?>) other).nextNodes.add(nextNode); return nextNode;

}

protected abstract void **ready**(AbstractNode<?> previous); @Override

public final T **get**() {

Try.execute(() -> this.countDownLatch.await()); return this.result;

}

private static Executor **executor** = Executors.newFixedThreadPool(8); protected final void **runAsync**(Supplier<T> resultSupplier) {

executor.execute(() -> {

this.result = resultSupplier.get(); this.nextNodes.forEach(node ->

((AbstractNode<?>) node).ready(this)); this.countDownLatch.countDown();

});

}

@Override

public final void **begin**(T value) { this.result = value; this.runAsync(() -> value);

}

}

package **util**;

final class **StartNode**<T> extends AbstractNode<T> {

**StartNode**() { } @Override

protected void **ready**(AbstractNode<?> previous) {

throw new AssertionError();

}

}

package **util**;

import java.util.function.Function;

final class **UnaryNode**<T> extends AbstractNode<T> { private final Function<?, ? extends T> **function**; **UnaryNode**(Function<?, ? extends T> function) {

this.function = function;

}

@SuppressWarnings({ "unchecked", "rawtypes" }) @Override

protected void **ready**(AbstractNode<?> previous) {

this.runAsync(() -> (T)((Function)function).apply(previous.result));

}

}

package **util**;

import java.util.concurrent.atomic.AtomicInteger; import java.util.function.BiFunction;

final class **BinaryNode**<T> extends AbstractNode<T> {

private final AbstractNode<?> **leftPrevious**; private final AbstractNode<?> **rightPrevious**;

private final AtomicInteger **readyCount** = new AtomicInteger(0); private final BiFunction<?, ?, ? extends T> **function**;

**BinaryNode**(

AbstractNode<?> leftPrevious, AbstractNode<?> rightPrevious, BiFunction<?, ?, ? extends T> function) {

this.leftPrevious = leftPrevious; this.rightPrevious = rightPrevious; this.function = function;

}

@SuppressWarnings({ "unchecked", "rawtypes" }) @Override

protected void **ready**(AbstractNode<?> previous) { if (this.readyCount.incrementAndGet() < 2)

return; this.runAsync(

() -> (T) ((BiFunction) function).apply( this.leftPrevious.result, this.rightPrevious.result)

);

}

}

**18 XML-Parser**

### Übersicht

* Der SAX-Parser
* Der DOM-Parser
* Der JDOM-Parser
* Der PullParser
* JAXB

Dabei werden stets folgende Klassen verwendet:

city

Library

books 0..n

isbn title price

Book

package **appl**; public class **Book** {

private String **isbn**; private String **title**; private double **price**;

public **Book**() { }

public **Book**(String isbn, String title, double price) { ... }

// **getter**, **setter**, **toString**...

}

package **appl**;

// ...

public class **Library** {

private final List<Book> **books** = new ArrayList<Book>(); private final String **city**;

public **Library**(String city) { this.city = city;

}

public String **getCity**() { return this.city;

}

public void **add**(Book book) { this.books.add(book);

}

public List<Book> **getBooks**() {

return Collections.unmodifiableList(this.books);

}

@Override

public String **toString**() {

final StringBuilder buf = new StringBuilder(); buf.append(this.getClass().getName() + " [" + this.city + " {\n"); for (Book book : this.books)

buf.append("\t" + book + "\n"); return buf.append("}]").toString();

}

}

Der Aufbau von XML-Dateien, die eine Library beschreiben, sind durch folgende DTD spezifiziert:

<!-- library.dtd -->

<!ELEMENT **library** (book\*)>

<!ELEMENT **book** (isbn, title, price)>

<!ELEMENT **isbn** (#PCDATA)>

<!ELEMENT **title** (#PCDATA)>

<!ELEMENT **price** (#PCDATA)>

<!ATTLIST **library** city CDATA #REQUIRED>

Hier eine beispielhafte XML-Beschreibung einer Library:

<?xml version='1.0'?>

<!DOCTYPE library SYSTEM "library.dtd">

<library city="**Paderborn**">

<book>

<isbn>**11111**</isbn>

<title>**Pascal**</title>

<price>**10.00**</price>

</book>

<book>

<isbn>**22222**</isbn>

<title>**Modula**</title>

<price>**20.00**</price>

</book>

<book>

<isbn>**33333**</isbn>

<title>**Oberon**</title>

<price>**30.00**</price>

</book>

</library>

Alle im folgenden vorgestellten Beispiele erzeugen aufgrund dieser XML-Datei eine

Library, die drei Books enthält.

Die toString-Methode dieser Library hat folgendes Resultat:

appl.Library [Paderborn { appl.Book [11111, Pascal, 10.0]

appl.Book [22222, Modula, 20.0]

appl.Book [33333, Oberon, 30.0]

}]

* 1. **SAX-Parser**

Ein Objektdiagramm:

produces



: LibraryBuilderHandler

(extends DefaultHandler)

*Handler*

: SAXParser

*InputStream*

rse

pa

racters

cha

dElement

en

rtElement

sta

: Book

"33333"

"Oberon" 30.30

: Book

"22222"

"Modula" 20.20

: Book

"11111"

"Pascal" 10.10

: Library

"Paderborn"

Die Anwendung muss ein Objekt einer Klasse zur Verfügung stellen, welche das Interface Handler implementiert. Es existiert bereits eine Basisklasse DefaultHandler, von welcher diese Klasse abgeleitet werden kann. Das Interface spezifiziert u.a. die Methoden startElement, characters und endElement.

Der parse-Methode des SAXParsers wird neben einem InputStream eben dieses Handler-Objekt übergeben. Die parse-Methode wird immer dann, wenn sie im Eingabestrom ein Start-Element entdeckt, die startElement-Methode aufrufen; immer dann, wenn sie ein Text-Element entdeckt, die Methode characters aufrufen; und immer dann, wenn sie ein Ende-Element entdeckt, die Methode endElement aufrufen.

Die parse-Methode ist der aktive Part; das Handler-Objekt reagiert nur auf die von der

parse-Methode erzeugten "Ereignisse".

Aufgrund einer von parse-Methode erzeugten Ereignisfolge ein komplexes Java-Objekt zu erzeugen, ist nicht immer ganz einfach.

Der Vorteil dieses Parsers aber liegt darin, dass er erstens auch einen Endlos- Datenstrom verarbeiten kann. Zweitens kann der Handler einfach bestimmte Ereignisse

einfach ignorieren – und somit z.B. nur für ausgewählte XML-Elemente entsprechende Java-Objekte erzeugen kann.

Hier die Implementierung des LibraryBilderHandlers:

package **appl**;

import org.xml.sax.Attributes; import org.xml.sax.SAXException;

import org.xml.sax.helpers.DefaultHandler;

public class **LibraryBuilderHandler** extends DefaultHandler { private String **pcdata**;

private Library **library**;

private Book **book**;

public Library **getLibrary**() { return this.library;

}

@Override

public void **startElement**(

String namespaceURI, String localName, String rawName,

Attributes attributes) throws SAXException {

if (rawName.equals("library"))

this.library = new Library(attributes.getValue("city")); else if (rawName.equals("book"))

this.library.add(this.book = new Book());

}

@Override

public void **endElement**(

String namespaceURI, String localName,

String rawName) throws SAXException { if (rawName.equals("isbn"))

this.book.setIsbn(this.pcdata); else if (rawName.equals("title"))

this.book.setTitle(this.pcdata); else if (rawName.equals("price"))

this.book.setPrice(Double.parseDouble(this.pcdata));

}

@Override

public void **characters**(char[] ch, int start, int end) throws SAXException {

this.pcdata = new String(ch, start, end);

}

}

import javax.xml.parsers.SAXParser;

import javax.xml.parsers.SAXParserFactory;

Die Anwendung:

static void **demo**() {

final String filename = "library1.xml"; final String systemId = "file:library.dtd";

final LibraryBuilderHandler handler = new LibraryBuilderHandler();

try(final FileInputStream in = new FileInputStream(filename)) { final SAXParser parser =

SAXParserFactory.newInstance().newSAXParser(); parser.parse(in, handler, systemId);

}

catch (Exception e) {

throw new RuntimeException(e);

}

final Library library = handler.getLibrary(); System.out.println(library);

}

}

* 1. **DOM-Parser**

Ein Objektdiagramm:

produces



: DocumentBuilder

*InputStream*

rse

pa

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | : Document | |  | | |
|  | | | |  | | | |
|  | | | : Element | |  | | |
|  | | | |  | | | |
|  |  | | |  | | |  |
| : Element | |  | : Element | |  | : Element | |

transform

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | : Library  "Paderborn" |  |  |
|  |  |  |
|  |  |  |  |  |
| : Book  "11111"  "Pascal" 10.10 |  | : Book  "22222"  "Modula" 20.20 |  | : Book  "33333"  "Oberon" 30.30 |

Die parse-Methode eines DOM-Parser (DocumentBuilder) erzeugt einen kompletten Baum von "technischen" DOM-Objekten (Document, Element, …). Sie kehrt erst dann zurück, wenn die Eingabe komplett verarbeitet wurde – und liefert dann die Wurzel auf diesen Baum zurück. Dann muss die Anwendung diesen Baum in die von der Anwendung gewünschte Struktur (Library, Books) transformieren.

Der Vorteil dieses Parsers liegt darin, dass er bereits einen kompletten Baum (von technischen Objekten) erzeugt – ohne unser Zutun.

Darin liegt aber auch der Nachteil: der Baum muss i.d.R. transformiert werden; und für einen Endlos-Datenstrom ist dieser Parser denkbar schlecht geeignet.

Für die Transformation ist in der folgenden Anwendung die Methode buildLibrary

zuständig. Sie bekommt ein Document und liefert eine Library:

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory; import org.w3c.dom.Document;

import org.w3c.dom.Element;

import org.w3c.dom.NodeList;

static void **demo**() {

final String filename = "library1.xml"; final String systemId = "file:library.dtd";

try (FileInputStream in = new FileInputStream(filename)) { final DocumentBuilder parser =

DocumentBuilderFactory.newInstance().newDocumentBuilder();

final Document doc = parser.parse(in, systemId); final Library library = buildLibrary(doc); System.out.println(library);

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

private static Library **buildLibrary**(Document doc) { final Element root = doc.getDocumentElement(); final String city = root.getAttribute("city"); final Library library = new Library(city);

final NodeList books = root.getElementsByTagName("book"); for (int i = 0; i < books.getLength(); i++) {

final Element book = (Element) books.item(i);

final String isbn = book.getElementsByTagName("isbn")

.item(0).getTextContent();

final String title = book.getElementsByTagName("title")

.item(0).getTextContent();

final double price = Double.parseDouble( book.getElementsByTagName("price")

.item(0).getTextContent()); library.add(new Book(isbn, title, price));

}

return library;

}

}

* 1. **JDOM-Parser**

Der JDOM-Parser funktioniert ähnlich wie der DOM-Parser. Der Unterschied liegt darin, dass der produzierte Baum nur "relevante" Elemente enthält (z.B. keine Whitespace- Elemente).

import org.jdom.Document; import org.jdom.Element;

import org.jdom.input.SAXBuilder;

static void **demo**() {

final String filename = "library1.xml"; try {

final SAXBuilder builder = new SAXBuilder(); builder.setExpandEntities(true); builder.setIgnoringBoundaryWhitespace(true); final Document doc = builder.build(filename); final Library library = buildLibrary(doc); System.out.println(library);

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

private static Library **buildLibrary**(Document doc) { final Element root = doc.getRootElement(); final Library library =

new Library(root.getAttributeValue("city")); final List<?> books = root.getChildren();

for (int i = 0; i < books.size(); i++) {

final Element bookElement = (Element)books.get(i); final String isbn = bookElement.getChildText("isbn"); final String title = bookElement.getChildText("title"); final double price = Double.parseDouble(

bookElement.getChildText("price"));

final Book book = new Book(isbn, title, price); library.add(book);

}

return library;

}

}

* 1. **Pull-Parser**

Ein Objektdiagramm:



: XMLStreamReader

produces

: Library

"Paderborn"

: Book

"11111"

"Pascal" 10.10

: Book

"22222"

"Modula" 20.20

: Book

"33333"

"Oberon" 30.30

*InputStream*

getAttributeValue (i)

getAttributeName (i)

getAttributeCount ()

tEventType ()

ge

getText ()

xt ()

ne

getName ()

sNext ()

ha

Dem Konstruktor vom XMLStreamReader wird der InputStream übergeben. Dann übernimmt die Anwendung den "aktiven" Part: sie kann die Methoden hasNext und next aufrufen (die ähnlich funktionieren wie ein Iterator) – und somit den Parser auf das jeweils nächste Symbol weitertreiben.

Um die Art des aktuelle Symbol zu ermitteln, kann die Methode getEventType

aufgerufen werden – sie liefert START\_ELEMENT, CHARACTERS oder END\_ELEMENT zurück.

Mittels getName kann der Name des Symbols ermittelt werden. Mittels getText kann ein der Text eines Text-Elements ermittelt werden.

Und mittels der Kombination von getAttributeCount, getAttributeName und getAttributeValue können die Attribute des aktuellen XML-Elements ermittelt werden.

Der Vorteil dieses Parsers liegt darin, dass es die Anwendung selbst ist, die ihn weitertreibt. Der Parse-Algorithmus kann somit aus der Struktur des XML-Dokuments und somit auch der Struktur der zu produzierenden Objekt-Wolke "abgeleitet" werden. Auch mittels eines Pull-Parsers können wir natürlich einen Endlos-Datenstrom verarbeiten – und wir können Elemente, die uns nicht interessieren, einfach übergehen.

Die folgende Anwendung benutzt eine weitere Klasse namens XScanner, welche die Benutzung eines XMLStreamReaders vereinfacht. Das Studium dieser Klasse sei dem Leser / der Leserin überlassen.

Die Klasse XScanner:

package **appl**;

// ...

import javax.xml.stream.XMLStreamReader; import javax.xml.stream.events.XMLEvent;

public class **XScanner** {

public final XMLStreamReader **reader**;

public **XScanner**(XMLStreamReader reader) throws Exception { this.reader = reader;

this.read();

}

public boolean **isStart**(String name) {

if (this.reader.getEventType() != XMLEvent.START\_ELEMENT) return false;

if (! name.equals(this.reader.getName().getLocalPart())) return false;

return true;

}

public Map<String,String> **getAttributes**() { final Map<String,String> map =

new HashMap<String,String>();

for (int i = 0; i < this.reader.getAttributeCount(); i++) map.put(

this.reader.getAttributeName(i).toString(), this.reader.getAttributeValue(i));

return map;

}

public void **start**(String name) throws Exception {

if (this.reader.getEventType() != XMLEvent.START\_ELEMENT) throw new RuntimeException("startElement expected");

if (! name.equals(this.reader.getName().getLocalPart())) throw new RuntimeException("<" + name +

"> expected. found: <" + this.reader.getName() + ">"); this.read();

}

public void **end**(String name) throws Exception {

if (this.reader.getEventType() != XMLEvent.END\_ELEMENT) throw new RuntimeException("endElement expected");

if (! name.equals(this.reader.getName().getLocalPart())) throw new RuntimeException("</" + name +

"> expected. found: </" + this.reader.getName() + ">"); this.read();

}

public String **startTextEnd**(String name) throws Exception { this.start(name);

if (this.reader.getEventType() != XMLEvent.CHARACTERS) throw new RuntimeException("Characters expected");

final String text = this.reader.getText(); this.read();

this.end(name); return text;

}

private int **read**() throws Exception { while (this.reader.hasNext()) {

final int eventType = this.reader.next(); if (eventType == XMLEvent.START\_ELEMENT

|| eventType == XMLEvent.CHARACTERS

|| eventType == XMLEvent.END\_ELEMENT) return eventType;

}

return this.reader.getEventType();

}

}

Die Anwendung:

static void **demo**() {

final String filename = "library1.xml"; XMLStreamReader reader = null;

try {

final XMLInputFactory factory = XMLInputFactory.newInstance();

reader = factory.createXMLStreamReader( new FileInputStream(filename));

final XScanner scanner = new XScanner(reader); assert (scanner.isStart("library"));

final Library library =

new Library(scanner.getAttributes().get("city")); scanner.start("library");

while (scanner.isStart("book")) { scanner.start("book");

String isbn = scanner.startTextEnd("isbn"); String title = scanner.startTextEnd("title"); double price = Double.parseDouble(

scanner.startTextEnd("price")); Book book = new Book(isbn, title, price); library.add(book);

scanner.end("book");

}

scanner.end("library"); System.out.println(library);

}

catch (Exception e) {

throw new RuntimeException(e);

}

finally {

if (reader != null) { try {

reader.close();

}

catch (Exception e) {

}

}

}

}

* 1. **JAXB**

Ein Objektdiagramm:



: JAXBContext

creates

: Unmarshaller

produces

: Library

"Paderborn"

*InputStream*

: Book

"11111"

"Pascal" 10.10

: Book

"22222"

"Modula" 20.20

: Book

"33333"

"Oberon" 30.30

marshal

un

reateUnmarshaller

c

Library.class

Book.class

JAXB bildet ein XML-Dokument automatisch auf eine Datenstruktur mit Objekten benutzerdefinierten Typs ab. Diese Abbildung erfolgt mittels Reflection.

Die Klassen der zu produzierenden Objekte müssen mit entsprechenden JAXB- Annotationen angereichert werden.

Die Klassen werden mit @XmlRootElement annotiert.

@XmlAccessorType legt fest, ob die Initialisierung der Objekte via Aufruf von Properties oder direkt über das Setzen der Attribute erfolgen soll.

Jede Property (resp. jedes Attribut) wird mit @XmlElement oder mit @XmlAttribut

annotiert.

Die Klasse Book:

package **appl**;

import javax.xml.bind.annotation.XmlAccessType; import javax.xml.bind.annotation.XmlAccessorType; import javax.xml.bind.annotation.XmlElement;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement @XmlAccessorType(XmlAccessType.PROPERTY) public class **Book** {

private String **isbn**; private String **title**; private double **price**;

public **Book**() { }

public **Book**(String isbn, String title, double price) { ... }

@XmlElement

public String **getIsbn**() { ... }

public void **setIsbn**(String value) { ... }

@XmlElement

public String **getTitle**() { ... }

public void **setTitle**(String value) { ... }

@XmlElement

public double **getPrice**() { ... }

public void **setPrice**(double value) { ... }

@Override

public String **toString**() { ... }

}

Die Klasse Library:

package **appl**;

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.XmlAttribute; import javax.xml.bind.annotation.XmlElement; import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement @XmlAccessorType(XmlAccessType.PROPERTY) public class **Library** {

private List<Book> **books** = new ArrayList<Book>(); private String **city**;

@XmlElement(name="book") public List<Book> **getBooks**() {

return this.books;

}

public void **setAuthors**(List<Book> books) { this.books = books;

}

@XmlAttribute

//@XmlAttribute(name="city") public String **getCity**() {

return this.city;

}

public void **setCity**(String city) { this.city = city;

}

@Override

public String **toString**() { ... }

}

Die Anwendung:

final String filename = "library1.xml"; final JAXBContext context =

JAXBContext.newInstance(Book.class, Library.class); final Unmarshaller unmarshaller =

context.createUnmarshaller(); final Library library = (Library)

unmarshaller.unmarshal(new FileReader(filename)); System.out.println(library);

**19 RMI (Remote Method Invocation)**

Bei der Diskussion der DynamicProxies wurde bereits eine eigene, einfache Implementierung von Remote Method Invocation entwickelt. Im Folgenden nun wird im das "richtige", offizielle RMI vorgestellt.

### Übersicht

* Grundlagen
* Statusbehaftete Objekte und Factories
* Client-seitige Listener
* Namensdienst eines RMI-Servers
  1. **Start**

Ein Objektdiagramm:

*Math Service*



: $Proxy…

ze (Point p)

si

*Invocation Handler*

: ???

voke

in

"math"

*Math Service*

: MathService Impl

size (Point p)

Namens dienst

: Socket

: Socket

Client Server

Im folgenden geht's zunächst um statuslose Service-Objekte.

Auf dem Server soll ein Service installiert werden, der die "Größe" eines Point-Objekts berechnet (sqrt(sqr(x) + sqr(y)) und zurückliefert.

Hier die serialisierbare Klasse Point:

package **types**;

import java.io.Serializable;

public class **Point** implements Serializable {

private static final long **serialVersionUID** = 1L;

public final int **x**; public final int **y**;

public **Point**(final int x, final int y) { this.x = x;

this.y = y;

}

@Override

public String **toString**() { ... }

}

Das Service-Interface spezifiziert nur eine einzige Methode – die Methode size:

package **services.iface**;

import java.rmi.Remote;

import java.rmi.RemoteException; import types.Point;

public interface **MathService** extends Remote {

public abstract double **size**(Point p) throws RemoteException;

}

Leider ist dieses Interface etwas "unsauber". Es beerbt das Marker-Interface Remote; und alle Mehtoden des Interfaces kündigen eine RemoteException an (dabei handelt es sich um eine checked Exception). Aber ein Interface, welches RMI-tauglich sein soll, muss exakt diese Eigenschaften besitzen.

Die (serverseitige) Implementierung des Interfaces hat mit RemoteExceptions natürlich nichts zu schaffen (hier fehlt die throws-Klausel) – denn eine RemoteException kann natürlich nur die RMI-Infrastruktur geworfen werden:

package **services.impl**;

import services.iface.MathService; import types.Point;

public class **MathServiceImpl** implements MathService { @Override

public double **size**(Point p) {

return Math.sqrt(p.x \* p.x + p.y \* p.y);

}

}

Der Server erzeugt eine RMI-Registry, welche über den Port 1099 erreichbar ist. Sie erzeugt dann ein MathServiceImpl-Objekt und registriert dieses bei der Registry – unter dem Namen "math":

package **server**;

import java.rmi.registry.LocateRegistry; import java.rmi.registry.Registry;

import java.rmi.server.UnicastRemoteObject;

import services.iface.MathService; import services.impl.MathServiceImpl;

public class **Server** {

public static void **main**(String[] args) { try {

System.out.println("Server starts ..."); final Registry registry =

LocateRegistry.createRegistry(1099);

final MathService mathService =

(MathService) UnicastRemoteObject.exportObject( new MathServiceImpl(), 0);

registry.rebind("math", mathService);

System.out.println("Server running ...");

}

catch (final Exception e) { System.out.println(e);

}

}

}

Der Client fordert via Naming.lookup eine "Remote-Referenz" auf den MathService an

* ihm wird natürlich Dynamic Proxy zurückgeliefert. Über dieses Proxy-Objekt können nun die Methoden des MathService-Interfaces aufgerufen werden:

package **client**;

import java.rmi.Naming;

import services.iface.MathService; import types.Point;

public class **Client** {

public static void **main**(String[] args) throws Exception { final String mathURL = "rmi://localhost:1099/math";

final MathService mathService = (MathService) Naming.lookup(mathURL); System.out.println(mathService.size(new Point(3, 4)));

System.out.println(mathService.size(new Point(4, 3)));

}

}

Die Ausgaben: zweimal die 5.0.

* 1. **Factories**

Ein Objektdiagramm:

*Calculator Factory*



: $Proxy…

reate

c

*Invocation Handler*

: ???

voke

in

"calc Factory"

*Calculator Factory*

: Calculator Factory

Impl

create

*Calculator*

: Calculator

Impl

add subtract getValue

*Calculator*

: Calculator

Impl

add

subtract getValue

77

42

Namens dienst

*Calculator*

: $Proxy…

*Invocation Handler*

: ???



add subtract getValue

voke

in

*Calculator*

: $Proxy…

*Invocation Handler*

: ???



add subtract getValue

voke

in

Wir benötigen viele Calculator-Objekte – jeder Client benötigt sein eigenes.

Das folgende Interfaces spezifiziert die Funktionalität eines Calculators:

package **objects.iface**;

// ...

public interface **Calculator** extends Remote {

public abstract void **add**(int v) throws RemoteException; public abstract void **subtract**(int v) throws RemoteException; public abstract int **getValue**() throws RemoteException;

}

Die Implementierungsklasse definiert ein Attribut value, welches von add und subtract manipuliert und dessen Wert von getValue zurückgeliefert wird. Ein CalculatorImpl ist also statusbehaftet:

package **objects.impl**;

// ...

public class **CalculatorImpl** implements Calculator { private int **value**;

@Override

public void **add**(int value) { this.value += value;

}

@Override

public void **subtract**(int value) { this.value -= value;

}

@Override

public int **getValue**() { return this.value;

}

}

Das folgende Remote-Interface spezifiziert eine CalculatorFactory. Der Aufruf von create muss einen Calculator zurückliefern:

package **factories.iface**;

// ...

public interface **CalculatorFactory** extends Remote {

public abstract Calculator **create**() throws RemoteException;

}

Hier die Implementierungsklasse:

package **factories.impl**;

// ...

public class **CalculatorFactoryImpl** implements CalculatorFactory { @Override

public Calculator **create**() throws RemoteException {

return (Calculator) UnicastRemoteObject.exportObject( new CalculatorImpl(), 0);

}

}

Man beachte, dass das erzeugte CalculatorImpl-Objekt bei RMI zum "Export" vorgesehen werden muss – via exportObject.

Der Server erzeugt und registriert nur die CalculatorFactory:

package **server**;

// ...

public class **Server** {

public static void **main**(String[] args) { try {

System.out.println("Server starts ...");

final Registry registry = LocateRegistry.createRegistry(1099);

final CalculatorFactory calculatorFactory = (CalculatorFactory) UnicastRemoteObject.exportObject(

new CalculatorFactoryImpl(), 0); registry.rebind("calcFactory", calculatorFactory);

System.out.println("Server running ...");

}

catch (final Exception e) { System.out.println(e);

}

}

}

Der Client besorgt sich eine Remote-Referenz auf die CalculatorFactory. Dann kann er beliebig häufig deren create-Methode aufrufen und bekommt stets einen neuen Calculator (natürlich ein Proxy – denn die CalculatorImpl-Objekte liegen ja auf dem Server):

package **client**;

// ...

public class **Client** {

public static void **main**(String[] args) { try {

final String calcFactoryURL = "rmi://localhost:1099/calcFactory";

final CalculatorFactory calculatorFactory = (CalculatorFactory) Naming.lookup(calcFactoryURL);

final Calculator calculator1 = calculatorFactory.create(); final Calculator calculator2 = calculatorFactory.create(); calculator1.add(44);

calculator2.add(80); calculator1.subtract(2); calculator2.subtract(3);

System.out.println(calculator1.getValue()); System.out.println(calculator2.getValue());

}

catch (Exception e) { System.out.println(e);

}

}

}

* 1. **Listeners**

Ein Objektdiagramm:

Namens dienst

*Calculator Factory*



: $Proxy…

*Invocation Handler*

: ???

*Calculator Factory*

: Calculator Factory

Impl

"calc Factory"

create

voke

in

reate

c

*Calculator*

: $Proxy…

*Invocation Handler*

: ???



*Calculator*

: Calculator

Impl

add subtract getValue

voke

in

tValue

ge

tract

sub

d

ad

42

addCalculator Listener

removeCalculator Listener

*Invocation Handler*

*Calculator Listener*

<Lambda>

update

: ???

*Calculator Listener*

: $Proxy…

update

e

invok

Der Client möchte von den Server-seitigen Berechnungen benachrichtigt werden.

Man beachte im obigen Objektdiagramm, dass nun auch auf der Server-Seite ein Dynamic-Proxy eingerichtet wird – über welches der Server den Client-seitigen CalculatorListener ansprechen kann.

Hier das Interface, über welches die Benachrichtigungen erfolgen sollen:

package **listeners.iface**;

// ...

public interface **CalculatorListener** extends Remote {

abstract public void **update**(Calculator calculator) throws RemoteException;

}

Das Calculator-Interface wird um zwei Methoden erweitert: addCalculatorListener

und removeCalculatorListener:

package **objects.iface**;

// ...

public interface **Calculator** extends Remote {

public abstract void **add**(int v) throws RemoteException; public abstract void **subtract**(int v) throws RemoteException; public abstract int **getValue**() throws RemoteException;

public abstract void **addCalculatorListener**( CalculatorListener l) throws RemoteException;

public abstract void **removeCalculatorListener**( CalculatorListener l) throws RemoteException;

}

Ein CalculatorImpl hat nun zusätzlich eine Liste, in der CalculatorListener eingetragen werden können – und eine Methode fireUpdate, die alle registrierten CalculatorListener benachrichtigt. Diese fireUpdate-Methode wir in der add- und der subtract-Methode aufgerufen:

package **objects.impl**;

// ...

public class **CalculatorImpl** implements Calculator {

private int **value**;

private final List<CalculatorListener> **listeners** = new ArrayList<>();

@Override

public void **add**(int value) { this.value += value; this.fireUpdate();

}

@Override

public void **subtract**(int value) { this.value -= value; this.fireUpdate();

}

@Override

public int **getValue**() { return this.value;

}

public void **addCalculatorListener**(CalculatorListener l) { this.listeners.add(l);

}

public void **removeCalculatorListener**(CalculatorListener l) { this.listeners.remove(l);

}

private void **fireUpdate**() {

for (final CalculatorListener listener : this.listeners) { try {

listener.update(this);

}

catch (final Exception e) { System.out.println(e);

}

}

}

}

Das Interface CalculatorFactory und die Klasse CalculatorFactoryImpl haben wir unverändert aus dem letzten Projekt übernommen. Dasselbe gilt auch für die Klasse Server.

Hier der neue Client:

package **client**;

// ...

public class **Client** {

public static void **main**(String[] args) { try {

final String calcFactoryURL = "rmi://localhost:1099/calcFactory";

final CalculatorFactory calculatorFactory = (CalculatorFactory) Naming.lookup(calcFactoryURL);

final Calculator calculator = calculatorFactory.create();

final CalculatorListener listener =

c -> System.out.println("===> " + c.getValue());

calculator.addCalculatorListener((CalculatorListener) UnicastRemoteObject.exportObject(listener, 0));

calculator.add(44); calculator.subtract(2); System.out.println(calculator.getValue());

calculator.removeCalculatorListener(listener); UnicastRemoteObject.unexportObject(listener, true);

}

catch (Exception e) { System.out.println(e);

}

}

}

Die Zeilen

final CalculatorListener listener =

c -> System.out.println("===> " + c.getValue());

erzeugen eine Client-seitigen CalculatorListener. Dieser wird dann beim Calculator registriert:

calculator.addCalculatorListener((CalculatorListener) UnicastRemoteObject.exportObject(listener, 0));

Man beachte, dass der listener zunächst an exportObject übergeben werden muss. Die Ausgaben:

===> 44

===> 42

42

* 1. **Naming**

Ein Objektdiagramm:

*MathService*

: MathService Impl

*TextService*

: TextService Impl

oLower

d

oUpper

t

iff

d

um

s

: Namensdienst

"text"

"math"



*MathService*

: $Proxy…

*Invocation Handler*

: $$$

nvoke

i

ff

di

um

s

Wie können wir ermitteln, welche Services im Server laufen?



*TextService*

: $Proxy…

*Invocation Handler*

: $$$

nvoke

i

Lower

do

Upper

do

Wir bauen zwei Services – einen MathService und einen TextService:

package **services.iface**;

// ...

public interface **MathService** extends Remote {

public abstract int **sum**(int x, int y) throws RemoteException; public abstract int **diff**(int x, int y) throws RemoteException;

}

package **services.impl**;

// ...

public class **MathServiceImpl** implements MathService { @Override

public int **sum**(int x, int y) { return x + y; }

@Override

public int **diff**(int x, int y) { return x - y; }

}

package **services.iface**;

// ...

public interface **TextService** extends Remote {

public abstract String **toUpper**(String s) throws RemoteException; public abstract String **toLower**(String s) throws RemoteException;

}

package **services.impl**;

// ...

public class **TextServiceImpl** implements TextService { @Override

public String **toUpper**(String s) { return s.toUpperCase(); } @Override

public String **toLower**(String s) { return s.toLowerCase(); }

}

Beide Services werden beim Server registriert:

package **server**;

// ...

public class **Server** {

public static void **main**(String[] args) { try {

System.out.println("Server starts ...");

final Registry registry = LocateRegistry.createRegistry(1099); final MathService mathService =

(MathService) UnicastRemoteObject.exportObject(

new MathServiceImpl(), 0); registry.rebind("math", mathService);

final TextService textService =

(TextService) UnicastRemoteObject.exportObject( new TextServiceImpl(), 0);

registry.rebind("text", textService);

System.out.println("Server running ...");

}

catch (final Exception e) { System.out.println(e);

}

}

}

Der folgende Client listet alle verfügbaren Services auf:

package **client**;

// ...

public class **Client** {

public static void **main**(String[] args) { try {

final String url = "rmi://localhost:1099";

final List<String> names = Arrays.asList(Naming.list(url)); for (int i = 0; i < names.size(); i++)

System.out.println(names.get(i));

}

catch (Exception e) { System.out.println(e);

}

}

}

Die Ausgaben:

//localhost:1099/math

//localhost:1099/text

**20 Class Loading**

Wie werden Klassen geladen? Wie können Klassen zur Laufzeit dynamisch nachgeladen werden? Wie können mehrere Klassen in einer VM geladen werden, die denselben voll qualifizierten Klassennamen haben (wie können also innerhalb eines Servers z.B. mehrere "Anwendungen" laufen, welche Klassen mit denselben Namen besitzen? Wie kann Hot-Deployment implementiert werden?

Solche und ähnliche Fragen sollen im folgenden diskutiert werden.

### Übersicht

* Bootstrap-, Platform- und System-Classloader.
* Bau eigener Classloader-Klassen
* URLClassLoader
* Sichtbarkeit von Klassen
* Ein kleines Plugin-Konzept
* Hot-Deployment
  1. **Grundlagen**

Klassen werden von Classloadern geladen. Genau wie jedes Objekt seine Klasse kennt (Object.getClass()), kenn jedes Class-Objekt seinen ClassLoader:

Class<?> cls = ...

ClassLoader loader = cls.getClassLoader();

Und genauso wenig, wie eine Klasse ihre Objekte kennt, kennt ein Classloader die von ihm geladenen Klassen (es sei denn, man baut einen eigenen ClassLoader).

: Foo

: Bar

: Class

"Bar"

: ClassLoader

: Class

"Foo"

Eine Klasse ist in der VM zur Laufzeit eindeutig durch das Triple <Classloader, package- name, class-name> beschrieben. In ein und derselben VM können also Klassen laufen, die denselben voll qualifizierten Klassennamen haben – sofern sie von unterschiedlichen Classloadern geladen wurden. (Dies wird später noch genauer untersucht werden).

Beim Starten der virtuellen Maschine werden drei Classloader genutzt (die in einer Vater-Kind-Beziehung stehen):

<Bootstrap-Classloader>

parent == null

ClassLoader.getPlatformClassLoader()

<Platform-Classloader>

parent

ClassLoader.getSytemClassLoader()

<System-Classloader>

Der Platform- und der System-Classloader sind Objekte von Klassen, die von der Basisklasse ClassLoader abgeleitet sind. Sie können mittels des Aufrufs der statischen ClassLoader-Methoden getPlatformClassLoader und getSystemClassLoader ermittelt werden.

Der Platform-Classloader ist für das Laden aller Systemklassen verantwortlich. Der System-Classloader lädt alle Klassen, die über den Classpath resp. den Modulepath erreichbar sind. Der Vater des System-Classloaders ist der Platform-Classloader.

Der Platform-Classloader hat ebenfalls einen Vater: den Bootstrap-Classloader. Dieser aber ist eigentlich gar kein "richtiger" Classloader (also keine von der Java-Klasse ClassLoader abgeleiteter Classloader) – und insofern ist die Parent-Referenz des Platform-Classloader auch null.

Für die folgenden Demo-Methoden binden wir den Platform-Classloader und den System-Classloader an zwei Variablen:

final static ClassLoader **platformClassLoader** = ClassLoader.getPlatformClassLoader();

final static ClassLoader **systemClassLoader** = ClassLoader.getSystemClassLoader();

Wir geben die Namen der Classloader-Klassen aus und demonstrieren die Vater- Beziehungen der Classloader:

static void **demoBasics**() { System.out.println(

platformClassLoader.getClass().getName()); System.out.println(

systemClassLoader.getClass().getName()); System.out.println(

platformClassLoader.getParent() == null); System.out.println(

systemClassLoader.getParent() == platformClassLoader);

}

Die Ausgaben:

jdk.internal.loader.ClassLoaders$PlatformClassLoader jdk.internal.loader.ClassLoaders$AppClassLoader

true true

Die folgende Methode zeigt die Bedeutung der Methode Class.forName. Sie benutzt denjenigen Classloader, mittels dessen auch die aufrufende Klasse (hier: Application) geladen wurde. Wir besorgen uns dann direkt denjenigen Classloader, mit dem die Klasse Application geladen wurde – und lassen die Klasse java.awt.Point sowohl

via Class.forName als auch mittels des Application-Classloaders ermitteln. Beide Aufrufe liefern dasselbe Class-Objekt zurück:

static void **demoForName**() throws Exception { Class<?> cls1 = Class.forName("java.awt.Point");

Class<?> cls2 = Application.class.getClassLoader()

.loadClass("java.awt.Point"); System.out.println(cls1 == cls2);

}

Die Ausgabe: true

Wann wird eine Klasse geladen? Wann wird sie initialisiert? Das sind zwei verschiedene Dinge. Nach dem Laden gibt's eine Referenz auf das Class-Objekt dieser Klasse (es ist also ein Class-Objekt erzeugt worden). Erst während der Initialisierung der Klasse aber werden deren statische Blöcke ausgeführt und die statischen Variablen initialisiert (sofern vorhanden).

Sei folgende Klasse gegeben:

package **foo**;

public class **Foo** {

**static** {

System.out.println("Foo: static block...");

}

public static void **work**() { System.out.println("Foo: working...");

}

}

Die Klasse hat einen statischen Block und eine statische work-Mehode.

Wird die Klasse über ihr Class-Literal angesprochen (Foo.class), so wird die Klasse geladen, aber noch nicht initialisiert.

Wird die Klasse via Class.forName(name, initialize, loader) geladen, wird – sofern initialze den Wert false hat – die Klasse ebenfalls noch nicht initialisiert.

Nur dann, wenn diese Methode mit dem initialize-Wert true aufgerufen wird, wird die Klasse initialisiert. Dasselbe passiert auch beim Aufruf der Class.forName- Methode, der nur der Klassenname übergeben wird.

Eine Klasse, die zwar geladen, aber noch nicht initialisiert ist, wird natürlich spätestens dann initialisiert, wenn sie instanziiert wird resp. ein statisches Element der Klasse angesprochen wird.

Hier eine Demo-Methode:

static void **demoLoadAndInitialize**() throws Exception {

{

System.out.println("----- 1 ");

Class<?> cls = Foo.class; System.out.println(cls.getName());

}

{

System.out.println("----- 2 ");

Class<?> cls = Class.forName(

"foo.Foo", false, systemClassLoader); System.out.println(cls.getName());

}

{

System.out.println("----- 3 ");

Class<?> cls = Class.forName("foo.Foo"); System.out.println(cls.getName());

}

}

Die Ausgaben:

1 foo.Foo

2 foo.Foo

3

Foo: static block... foo.Foo

Von welchen Classloaden werden Array-Klassen geladen?

static void **demoSpecials**() { System.out.println(

int.class.getClassLoader() == null); System.out.println(

int[].class.getClassLoader() == int.class.getClassLoader()); System.out.println(

Connection[].class.getClassLoader() == Connection.class.getClassLoader());

}

Die Ausgaben: true true true

Eine Klasse Foo[] wird also von demselben Classloader geladen wie die Klasse Foo

selbst.

Die Klassen ClassLoader, String und JFrame z.B. werden vom Bootstrap-Classloader geladen:

static void **demoBootstrapClassLoader**() { System.out.println(ClassLoader.class.getClassLoader() == null); System.out.println(String.class.getClassLoader() == null); System.out.println(JFrame.class.getClassLoader() == null);

}

Die Ausgaben: true true true

Die Klassen Application, Foo und Log (Klassen, die also über den Classpath erreichbar sind) werden vom System-Classloader geladen:

static void **demoSystemClassLoader**() { System.out.println(

Application.class.getClassLoader() == systemClassLoader); System.out.println(

Foo.class.getClassLoader() == systemClassLoader); System.out.println(

Log.class.getClassLoader() == systemClassLoader);

}

Die Ausgaben: true true true

Klassen wie z.B. Connection werden über den Platform-Classloader geladen (Java- Klassen, die aber nicht zum "Kern" von Java gehören):

static void **demoPlatformClassLoader**() { System.out.println(

Connection.class.getClassLoader() == platformClassLoader);

}

Die Ausgaben: true

Eine Klasse, die vom Platform-Classloader geladen wurde, kann auch vom System- Classloader ermittelt werden (ein Kind-Classloader kennt auch alle Klassen seines Vater-Classloaders). Eine Klasse aber, die von dem System-Classloader geladen würde, ist dem Platform-Classloader nicht bekannt (ein Vater kennt nicht die Klassen, die ein Kind-Classloader geladen hat):

static void **demoLoadingClasses**() throws Exception { System.out.println(

platformClassLoader

.loadClass("java.sql.Connection").getClassLoader() == platformClassLoader);

System.out.println( systemClassLoader

.loadClass("java.sql.Connection").getClassLoader() == platformClassLoader);

System.out.println( systemClassLoader

.loadClass("appl.Application").getClassLoader() == systemClassLoader);

try { platformClassLoader.loadClass("appl.Application");

}

catch(ClassNotFoundException e) { System.out.println("expected: " + e);

}

}

Die Ausgaben:

true true true

expected: java.lang.ClassNotFoundException: appl.Application

Via ClassLoader.getDefinedPackage() können die Packages derjenigen Klassen ermittelt werden, die der entsprechende Classloader geladen hat:

static void **demoSystemClassLoaderGetDefinedPackages**() { new Foo();

for(Package p : systemClassLoader.getDefinedPackages()) { System.out.println(p);

}

}

Die Ausgaben:

package jn.util package appl package foo

Im folgenden benutzen wir einige Klassen, die vom Platform-Classloader geladen wurden. Damit sie überhaupt geladen werden, stellen wir einfach einige unsinnige Dinge mit ihnen an. Schließlich ermitteln wir dann alle Pakete, die vom Platform-Classloader geladen wurden:

static void **demoPlatformClassLoaderGetDefinedPackages**() { try {

DriverManager.getConnection("");

}

catch(Throwable e) {

// do nothing...

}

for(Package p : platformClassLoader.getDefinedPackages()) { System.out.println(p);

}

}

Die Ausgaben:

package java.sql

* 1. **Eigene ClassLoader**

Wir können eigene Classloader-Klassen schreiben, die direkt von der Basisklasse

ClassLoader abgeleitet sind.

Wir definieren zunächst eine kleine Hilfsklasse, mittels derer alle Bestandteile einer Klasse ausgegeben werden können: der Classloader die Klasse und der voll qualifizierte Klassenname:

package **jn.util**;

public class **ClassName** {

public static String **of**(Class<?> cls) {

return cls.getClassLoader().getName() + "//" + cls.getName();

}

public static String **of**(Object obj) { return of(obj.getClass());

}

}

Das Projekt besteht aus drei Teilprojekten: OwnClassLoaders, OwnClassLoaders-A und OwnClassLoaders-B – dabei sind die letzten beiden Projekte vom ersten Projekt abhängig:

**OwnClassLoaders**

appl.Application *iface.MathService* util.SimpleClassLoader

depends depends

**OwnClassLoaders-A**

impl.MathServiceImpl

**OwnClassLoaders-B**

impl.MathServiceImpl

Im Classpath der A- und B-Projekte ist das oberste Projekt enthalten. Das oberste Projekt kann in seinem eigenen Classpath also nicht(!) die A- und B-Projekte enthalten (das wäre ein illegaler Zirkel).

Im Projekt OwnClassLoaders ist u.a. ein Interface iface.MathService vereinbart – ein Interface, welches von zwei gleichnamigen Klassen in den A- und B-Projekten implementiert ist (daher die Abhängigkeit der A- und B-Projekte von dem obersten Projekt):

**// OwnClassLoader**

package **iface**;

public interface **MathService** {

public abstract int **sum**(int x, int y); public abstract int **diff**(int x, int y);

}

**// OwnClassLoader-A**

package **impl**;

import iface.MathService;

public class **MathServiceImpl** implements MathService {

**static** {

System.out.println("A : MathServiceImpl - static block");

}

public int **sum**(int x, int y) { return x + y; } public int **diff**(int x, int y) { return x - y; }

}

**// OwnClassLoader-B**

package **impl**;

import iface.MathService;

public class **MathServiceImpl** implements MathService {

**static** {

System.out.println("B : MathServiceImpl - static block");

}

public int **sum**(int x, int y) { return x + y; } public int **diff**(int x, int y) { return x - y; }

}

Im demoBasics zeigen wir zunächst, dass im obersten Projekt das Interface iface.MathService natürlich ansprechbar ist – nicht aber die impl.FooImpl- Klasse(n), weil sie nicht im Classpath des obersten Projekts liegen. Wir zeigen weiterhin, dass die impl.FooImpl-Klassen(n) natürlich auch nicht via Reflection ansprechbar sind. Wie könnte eine der impl-Klasse auch angesprochen werden – gibt es doch zwei verschiedene(!) impl.FooImpl-Klassen.

**// OwnClassLoader**

static void **demoBasics**() { iface.MathService mathService = null;

// impl.FooImpl mathServiceImpl = null; // not in CLASSPATH try {

Class<?> cls = Class.forName("iface.MathService"); System.out.println("found: " + cls.getName());

}

catch (ClassNotFoundException e) { System.out.println("unexpected: " + e);

}

try {

Class<?> cls = Class.forName("impl.MathServiceImpl"); System.out.println("found: " + cls.getName());

}

catch (ClassNotFoundException e) { System.out.println("expected: " + e);

}

}

Die Ausgaben:

found: iface.MathService

expected: java.lang.ClassNotFoundException: impl.MathServiceImpl

Wir bauen uns nun eine eigene SimpleClassLoader-Klasse (im obersten Projekt):

**// OwnClassLoader**

package **util**;

import java.io.FileInputStream; import java.io.IOException;

public class **SimpleClassLoader** extends ClassLoader { private final String **dir**;

public **SimpleClassLoader**(String name, String dir, ClassLoader parent) { super(name, parent);

this.dir = dir;

}

public **SimpleClassLoader**(String name, String dir) { this(name, dir, ClassLoader.getSystemClassLoader());

}

@Override

protected Class<?> **findClass**(String name) throws ClassNotFoundException {

final String filename =

this.dir + "/" + name.replace('.', '/') + ".class";

try (final FileInputStream fis = new FileInputStream(filename)) { final byte data[] = fis.readAllBytes();

final Class<?> cls =

this.defineClass(name, data, 0, data.length); return cls;

}

catch (IOException e) {

throw new ClassNotFoundException(e.toString());

}

}

}

Die Klasse ist direkt abgeleitet von der Standardklasse ClassLoader.

Dem ersten Konstruktor der Klasse wird ein Name (der Name des Classloaders), der Name eines bin-Verzeichnisse (dessen Klassen von dem Classloader geladen werden sollen) und derjenige Classloader übergeben, der als Vater des SimpleClassLoaders fungieren soll. Dabei wird der Name der Classloaders und Vater-Classloader an den Konstruktor der Basisklasse weitergereicht; der Name des bin-Verzeichnisses wird dem dir-Attribut zugewiesen.

Der zweite Konstruktor wird auf den ersten abgebildet, wobei als Vater-Classloader der System-Classloader weitergereicht wird.

Eine eigene Classloader-Klasse muss dann die Methode findClass implementieren. Diese Methode wird der voll qualifizierte Name einer Klasse übergeben und muss für diese Klasse ein Class-Objekt erzeugen und dieses zurückliefern.

findClass leitet aus dem Namen der Klasse den Pfad derjenigen class-Datei ab, in welcher die Klasse definiert ist. Der Inhalt dieser class-Datei wird dann in einen byte- Array eigelesen und dieser schließlich zusammen mit dem Namen an die von ClassLoader geerbte defineClass-Methode übergeben. Diese erzeugt für die Klasse das Class-Objekt und liefert dieses zurück.

Im obersten Projekt ist auch die eigene SimpleClassLoader-Klasse implementiert. Bevor aber diese vorgestellt wird, wird zunächst einmal eine einfache Anwendung demonstriert (die ebenfalls im obersten Projekt hinterlegt ist), die noch nichts mit dem neuen Classloader zu tun hat:

Um die Funktionalität des SimpleClassLoaders zu verdeutlichen, berechnen wir zunächst zwei File-Variablen – Variablen, welche die beiden bin-Verzeichnisse des A- und des B-Projekts repräsentieren:

**// OwnClassLoader**

final static File **userDir** = new File(System.getProperty("user.dir")); final static File **binA** = new File(userDir.getPath() + "-A/bin"); final static File **binB** = new File(userDir.getPath() + "-B/bin");

userDir zeigt auf das alle Arbeitsverzeichnis (auf das oberste Projekt); binA zeigt auf das bin-Verzeichnis des A-Projekts, binB auf das bin-Verzeichnis des B-Projekts.

Nun die Demo:

**// OwnClassLoader**

static void **demoSimpleClassLoader**() { try {

final ClassLoader classLoader1 =

new SimpleClassLoader("A1", binA.getPath()); final ClassLoader classLoader2 =

new SimpleClassLoader("A2", binA.getPath());

final ClassLoader classLoader3 =

new SimpleClassLoader("B", binB.getPath());

final Class<?> cls1 = classLoader1.loadClass("impl.MathServiceImpl");

final Class<?> cls2 = classLoader2.loadClass("impl.MathServiceImpl");

final Class<?> cls3 = classLoader3.loadClass("impl.MathServiceImpl");

System.out.println(cls1 == cls2); System.out.println(cls2 == cls3); System.out.println(cls1 == cls3);

final MathService mathService1 =

(MathService) cls1.getConstructor().newInstance(); final MathService mathService2 =

(MathService) cls2.getConstructor().newInstance(); final MathService mathService3 =

(MathService) cls3.getConstructor().newInstance();

System.out.println(ClassName.of(mathService1)); System.out.println(ClassName.of(mathService2)); System.out.println(ClassName.of(mathService3));

}

catch (Exception e) { System.out.println("unexpected: " + e);

}

}

In der obigen Demo erzeugen wird drei SimpleClassLoader mit den Namen "A1", "A2" und "B". Die ersten beiden können Klassen des A-Projekts laden (binA.getPath()), der letzte kann Klassen des B-Projekts laden (binB.getPath()).

Dann fordern wir alle drei SimpleClassLoader auf, die Klasse "impl.MathServiceImpl" zu laden. Wir erhalten drei Class-Objekte zurück.

Obwohl alle drei Class-Objekte dieselbe "textuelle" Klasse repräsentieren, sind alle drei Class-Objekte verschieden. Wir haben also drei verschiedene Klassen geladen. Die Ausgabe:

false false false

Die drei Klassen sind geladen worden, sind aber noch initialisiert. Wir rufen daher auf alle drei Class-Objekte die Methode getConstructor().newInstance() auf – und initialisieren somit die jeweilige Klasse und lassen von jeder Klasse ein Objekt erzeugen. Und geben schließlich diese drei Objekte aus.

Die Ausgaben:

1. : MathServiceImpl - static block A : MathServiceImpl - static block B : MathServiceImpl - static block A1//impl.MathServiceImpl A2//impl.MathServiceImpl B//impl.MathServiceImpl

Wir können statt eines SimpleClassLoader auch einen VerboseClassLoader bauen – der bei seiner Arbeit etwas geschwätzig ist und dabei noch die Methode loadClass der Basisklasse ClassLoader überschreibt:

**// OwnClassLoader**

package **util**;

import java.io.FileInputStream;

public class **VerboseClassLoader** extends ClassLoader { private final String **dir**;

public **VerboseClassLoader**(String name, String dir, ClassLoader parent) { super(name, parent);

this.dir = dir;

}

public **VerboseClassLoader**(String name, String dir) { this(name, dir, ClassLoader.getSystemClassLoader());

}

@Override

protected Class<?> **loadClass**(String name, boolean resolve) throws ClassNotFoundException {

Logger.enter("loadClass(" + name + ", " + resolve + ")"); Class<?> cls = this.findLoadedClass(name);

if (cls != null) {

Logger.log("Class " + cls.getName() + " already loaded"); Logger.exit();

return cls;

}

try {

//cls = this.getParent().loadClass(name); cls = this.findSystemClass(name);

Logger.log("System class found: " + cls.getName()); Logger.exit();

return cls;

}

catch (Exception e) {

// go on!

}

final byte data[]; final String filename =

this.dir + "/" + name.replace('.', '/') + ".class";

try (final FileInputStream fis = new FileInputStream(filename)) { data = fis.readAllBytes();

}

catch (Exception e) {

throw new ClassNotFoundException(e.toString());

}

Logger.enter("defineClass");

cls = this.defineClass(name, data, 0, data.length); Logger.exit();

if (resolve)

this.resolveClass(cls); Logger.log(

"class '" + cls.getName() + "' loaded from '" + filename + "'"); Logger.exit();

return cls;

}

}

Diese Klasse überschreibt die loadClass-Methode der Basisklasse. Hierin wird deutlich, dass ein Classloader zunächst stets seinen Vater-Classloader beauftragt, die Klasse zu laden (oder die bereits geladene Klasse zurückzuliefern). Nur dann, wenn der Vater die Klasse nicht liefern kann, wird der VerboseClassLoader seinerseits versuchen, die Klasse zu laden. (Das genaue Studium sei dem Leser / der Leserin überlassen.)

Eine Anwendung eines VerboseClassLoaders:

**// OwnClassLoader**

static void **demoVerboseClassLoader**() { try {

final ClassLoader classLoader =

new VerboseClassLoader("B", binB.getPath());

Class<?> cls = classLoader.loadClass("impl.MathServiceImpl"); MathService mathService =

(MathService)cls.getConstructor().newInstance(); System.out.println(ClassName.of(mathService));

}

catch (Exception e) { System.out.println("unexpected: " + e);

}

}

Es sei dem Leser / der Leserin überlassen, die Ausgaben der obigen Demo-Methode genauer zu studieren:

loadClass(impl.MathServiceImpl, false) { defineClass {

loadClass(iface.MathService, false) { System class found: iface.MathService

}

loadClass(java.lang.Object, false) { System class found: java.lang.Object

}

}

class 'impl.MathServiceImpl' loaded from

'C:\Users\Kohl\workshop\java\java-extensions2-new\projects\ x1802-ClassLoader-OwnClassLoaders-B\bin/impl/ MathServiceImpl.class'

}

loadClass(java.lang.System, false) { System class found: java.lang.System

}

loadClass(java.io.PrintStream, false) { System class found: java.io.PrintStream

}

1. : MathServiceImpl - static block B//impl.MathServiceImpl
   1. **URLClassLoader**

Im folgenden demonstrieren wir die Verwendung der (instanziierbaren) Standardklasse URLClassLoader. Dabei verwenden wir dasselbe Beispiel wie im letzten Abschnitt (MathService etc.).

Auch dieses Projekt enthält drei Teilprojekte:

**URLClassLoader**

appl.Application

*iface.MathService*

depends depends

**URLClassLoader-A a.jar**

impl.MathServiceImpl

**URLClassLoader-B b.jar**

impl.MathServiceImpl

Die Klassen des A- resp. B-Projekts sind nun aber zusätzlich in einer jar-Datei hinterlegt (a.jar, b.jar). Beide Projekte enthalten eine build.xml (eine ant-Datei), mittels derer diese jar-Dateien gebaut werden können.

Wir werden zeigen, dass ein URLClassLoader die Klassen sowohl aus einem Verzeichnis als auch aus einer jar-Datei laden kann.

Die Klasse URLClassLoader hat folgenden Konstruktor:

public **URLClassLoader**(String name, URL[] urls, ClassLoader parent)

Die URLs enthalten die Verzeichnisse resp. die jar-Dateien, in denen der Classloader nach zu ladenden Klassen suchen soll.

Wir benutzen zwei statische Hilfsmethoden, mittels derer wir die Erzeugung von URLClassLoadern vereinfachen können. Statt eines URL-Arrays werden den Methoden einfache ein Verzeichnisname resp. der Pfad einer einzigen jar-Datei übergeben wird:

public class **ClassLoaderUtil** {

public static URLClassLoader **createURLClassLoader**( String name,

File file,

ClassLoader parent) throws Exception { return new URLClassLoader(

name,

new URL[] { file.toURI().toURL() }, parent);

}

public static URLClassLoader **createURLClassLoader**( String name,

File file) throws Exception { return createURLClassLoader(

name, file,

ClassLoader.getSystemClassLoader());

}

}

Weiterhin nutzen wir die folgende Methode loadAndUseClassen, um den Umgang mit einem URLClassLoader zu demonstrieren. Der Methode werden zwei File-Referenzen übergeben, die entweder auf die beiden A- und B-Projekte oder auf die a.jar und b.jar zeigen können:

**// URLClassLoader**

private static void **loadAndUseClasses**(

final File fileA, final File fileB) throws Exception {

try (final URLClassLoader classLoaderA =

ClassLoaderUtil.createURLClassLoader ("a", fileA); final URLClassLoader classLoaderB =

ClassLoaderUtil.createURLClassLoader ("b", fileB)) { final Class<?> clsA =

classLoaderA.loadClass("impl.MathServiceImpl"); final Class<?> clsB =

classLoaderB.loadClass("impl.MathServiceImpl"); System.out.println(clsA == clsB);

final MathService mathServiceA =

(MathService) clsA.getConstructor().newInstance(); final MathService mathServiceB =

(MathService) clsB.getConstructor().newInstance(); System.out.println(mathServiceA.sum(40, 2));

System.out.println(mathServiceB.diff(80, 3));

}

}

Ein URLClassLoader ist Closeable – daher die beiden Resource-try-Blöcke. Der erste URLClassLoader wird für das bin-Verzeichnis des A-Projekt resp. die a.jar zuständig sein, der zweite für das B-Projekt resp. für die b.jar.

Wir lassen beide Classloader die Klassen impl.MathServiceImpl laden – und vergleichen dann die Referenzen auf die beiden von loadClass erhaltenen Class- Objekte (um zu zeigen, dass es sich um verschiedene(!) Class-Objekte handelt).

Schließlich erzeugen via getConstructor().newInstance() jeweils ein Objekt dieser Klassen – und binden sie an Variablen des Interface-Typs MathService (mathServiceA und mathServiceB). Und können dann mit diesen MathServices rechnen (sum, diff).

Wir rufen dann diese Methode zunächst mit File-Parametern auf, welche die bin- Verzeichnisse der A- und B-Projekte repräsentieren:

**// URLClassLoader**

static void **demoDirectories**() throws Exception { Log.logMethodCall();

final File userDir = new File(System.getProperty("user.dir")); final File binA = new File(userDir.getPath() + "-A/bin"); final File binB = new File(userDir.getPath() + "-B/bin"); loadAndUseClasses(binA, binB);

}

Die Ausgaben:

false

A : MathServiceImpl - static block B : MathServiceImpl - static block 42

77

Dieselbe loadAndUseClasses-Methode rufen wir schließlich mit File-Referenzen auf, welche die a.jar- und b.jar-Dateien repräsentieren:

**// URLClassLoader**

static void **demoJars**() throws Exception { Log.logMethodCall();

final File userDir = new File(System.getProperty("user.dir")); final File jarA = new File(userDir.getPath() + "-A/a.jar"); final File jarB = new File(userDir.getPath() + "-B/b.jar"); loadAndUseClasses(jarA, jarB);

}

Die Ausgaben sind die gleichen wie bei demoDirectories.

* 1. **Sichtbarkeit**

Im folgenden wird untersucht, welche Klassen über welche ClassLoader dynamisch sichtbar sind.

Wir bauen vier Projekte, die statisch komplett unabhängig voneinander sind. Im Hauptprojekt wird eine Klasse p.Demo definiert; in den A-, B- und C-Projekten jeweils eine Klasse p.Alpha, p.Beta und p.Gamma. Alle Klassen sind Runnable.

Die Klassen des Hauptprojekts werden über den System-ClassLoader geladen. Die Klasse des A-Projekts wird mit einem URLClassLoader geladen werden, dessen Vater der System-ClassLoader ist. Die Klasse des B-Projekts wird mit einem URLClassLoader geladen werden, dessen Vater der URLClassLoader von A ist. Und die Klasse des C- Projekts wird mittels eines URLClassLoaders geladen werden, dessen Vater wieder der System-ClassLoader ist.

Das folgende Schaubild zeigt die Vater-Kind-Beziehung der beteiligten ClassLoader. Und es zeigt, welcher ClassLoader für das Laden der Klassen der vier Verzeichnisse verantwortlich sein werden:

**Visibility** appl.Application p.Demo

System ClassLoader

**Visibility-A**

p.Alpha

URLClassLoader a

URLClassLoader c

**Visibility-C**

p.Gamma

**Visibility-B**

p.Beta

URLClassLoader b

In der main-Methode der Application-Klasse erzeugen wir die drei benötigten

URLClassLoader und laden mit deren Hilfe die Klassen der A-, B- und C-Projekte.

Wir erzeugen dann vier Objekte: ein Demo-, ein Alpha-, ein Beta- und ein Gamma-Objekt - und rufen auf alle vier Objekte die jeweilige run-Methode auf. In diesen run-Methoden werden wir zeigen, welche Klassen über den ClassLoader der entsprechenden Klasse sichtbar sind.

**// Visibility**

package **appl**;

// ...

public class **Application** {

public static void **main**(String[] args) throws Exception {

final File userDir = new File(System.getProperty("user.dir")); final File binA = new File(userDir.getPath() + "-A/bin"); final File binB = new File(userDir.getPath() + "-B/bin"); final File binC = new File(userDir.getPath() + "-C/bin");

final ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();

final URLClassLoader classLoaderA = ClassLoaderUtil

.createURLClassLoader("a", binA, systemClassLoader); final URLClassLoader classLoaderB = ClassLoaderUtil

.createURLClassLoader("b", binB, classLoaderA); final URLClassLoader classLoaderC = ClassLoaderUtil

.createURLClassLoader("c", binC, systemClassLoader);

final Class<?> clsDemo = systemClassLoader.loadClass("p.Demo"); final Class<?> clsAlpha = classLoaderA.loadClass("p.Alpha"); final Class<?> clsBeta = classLoaderB.loadClass("p.Beta"); final Class<?> clsGamma = classLoaderC.loadClass("p.Gamma");

final Runnable demo =

(Runnable) clsDemo.getConstructor().newInstance(); final Runnable alpha =

(Runnable) clsAlpha.getConstructor().newInstance(); final Runnable beta =

(Runnable) clsBeta.getConstructor().newInstance(); final Runnable gamma =

(Runnable) clsGamma.getConstructor().newInstance();

demo.run();

alpha.run();

beta.run();

gamma.run();

System.out.println("Done!");

}

}

Die Anwendung terminiert normal.

(Der Einfachheit halber haben wir bei der Erzeugung der URLClassLoader auf den Resource-try verzichtet.)

In den folgenden Klassen benutzen wir neben der bereits bekannten execute-Methode eine weitere Methode der Try-Klasse (im shared-Projekt enthalten):

public static void **willThrow**(

Class<? extends Exception> exceptionClass, Supplier<?> supplier) {

try { supplier.get();

throw new RuntimeException(

"expected Exception: " + exceptionClass);

}

catch(Exception e) {

if (! exceptionClass.isAssignableFrom(e.getClass())) throw new RuntimeException("expected: " +

exceptionClass.getName () + "but was: " + e);

}

}

Die run-Methode der Demo-Klasse zeigt, dass über den ClassLoader dieser Klasse nur die Klasse p.Demo selbst sichtbar sind – p.Alpha, p.Beta und p.Gamma sind unsichtbar:

**// Visibility**

package **p**;

// ...

public class Demo implements Runnable {

@Override

public void **run**() {

final ClassLoader loader = this.getClass().getClassLoader(); Try.execute(() -> loader.loadClass("p.Demo")); Try.willThrow(ClassNotFoundException.class,

() -> loader.loadClass("p.Alpha")); Try.willThrow(ClassNotFoundException.class,

() -> loader.loadClass("p.Beta")); Try.willThrow(ClassNotFoundException.class,

() -> loader.loadClass("p.Gamma"));

}

}

Die run-Methode der Alpha-Klasse zeigt, dass über den ClassLoader dieser Klasse nur die Klassen p.Demo und p.Alpha sichtbar sind (und nur diese könnten also in der Alpha-Klasse genutzt werden) – p.Beta und p.Gamma sind unsichtbar:

**// Visibility-A**

package **p**;

// ...

public class **Alpha** implements Runnable { @Override

public void **run**() {

final ClassLoader loader = this.getClass().getClassLoader(); Try.execute(() -> loader.loadClass("p.Demo")); Try.execute(() -> loader.loadClass("p.Alpha")); Try.willThrow(ClassNotFoundException.class,

() -> loader.loadClass("p.Beta")); Try.willThrow(ClassNotFoundException.class,

() -> loader.loadClass("p.Gamma"));

}

}

Die run-Methode der Beta-Klasse zeigt, dass über den ClassLoader dieser Klasse die Klassen p.Demo, p.Alpha und p.Beta sichtbar sind (p.Gamma aber nicht):

**// Visibility-B**

package **p**;

// ...

public class **Beta** implements Runnable { @Override

public void **run**() {

final ClassLoader loader = this.getClass().getClassLoader(); Try.execute(() -> loader.loadClass("p.Demo")); Try.execute(() -> loader.loadClass("p.Alpha")); Try.execute(() -> loader.loadClass("p.Beta")); Try.willThrow(ClassNotFoundException.class,

() -> loader.loadClass("p.Gamma"));

}

}

Die run-Methode der Gamma-Klasse zeigt schließlich, dass über den ClassLoader dieser Klasse die Klassen nur p.Demo und p.Gamma sichtbar sind (aber weder p.Alpha noch p.Beta):

**// Visibility-C**

package **p**;

// ...

public class **Gamma** implements Runnable { @Override

public void **run**() {

final ClassLoader loader = this.getClass().getClassLoader(); Try.execute(() -> loader.loadClass("p.Demo")); Try.willThrow(ClassNotFoundException.class,

() -> loader.loadClass("p.Alpha")); Try.willThrow(ClassNotFoundException.class,

() -> loader.loadClass("p.Beta")); Try.execute(() -> loader.loadClass("p.Gamma"));

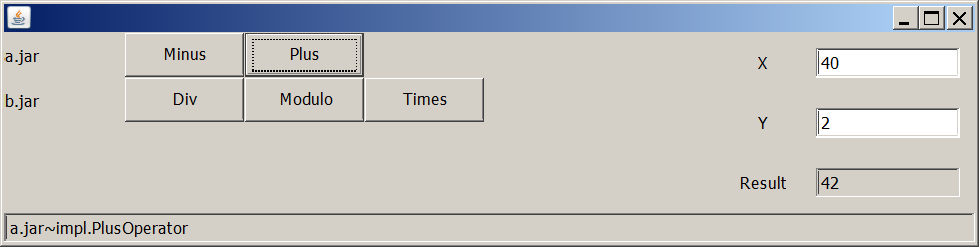
}

}

* 1. **Plugins**

Die folgende Anwendung zeigt eine "realistische" Anwendung von URLClassLoadern.

Die Anwendung könnte sich wie folgt präsentieren (sie könnte sich aber auch ganz anders präsentieren…):



Die eigentliche Anwendung ist ein Rahmen ohne eigene Funktionalität. Die eigentliche Funktionalität wird durch "Plugins" bereitgestellt, die in diesen Rahmen eingebunden werden können.

Plugins sind jar-Datei, die in einem Deploy-Verzeichnis hinterlegt werden. Jedes Plugin enthält Klassen, die ein bestimmtes Interface implementieren – das Interface NamedOperator. Dabei muss es natürlich erlaubt sein, dass mehrere Plugins Klassen mit demselben voll qualifizierten Klassennamen enthalten. Für die Klassen jedes Plugins wird also ein eigener Classloader verantwortlich sein.

Der Rahmen sorgt dafür, dass für jedes im Deploy-Verzeichnis enthaltene Plugin auf der Oberfläche eine Zeile eingerichtet eingerichtet wird – eine Zeile mit dem Namen der jar-Datei und eine Reihe von Buttons. Mittels der Betätigung einer dieser Buttons kann die Funktionalität einer Klasse ausgeführt werden, die in der jeweiligen jar-Datei enthalten ist.

Das Projekt enthält auch hier wieder drei Teil-Projekte. Im oberste Projekt ist der Rahmen implementiert. Dieses Projekt enthält auch das Deploy-Verzeichnis, in welchem nach die Plugins gesucht wird.

Das A- und das B-Projekt implementiert jeweils ein Plugin. Das Plugin des A-Projekts stellt zwei Klassen zur Verfügung, die NamedOperator implementieren (daher die beiden Buttons "Minus" und "Sum". Das B-Projekt stellt drei Klassen zur Verfügung – daher die Buttons "Div", "Modulo" und "Times". Das Interface ist im obersten Projekt angesiedelt (der Rahmen ist natürlich von diesem Interface abhängig) – daher sind die A- und B-Projekte wieder vom obersten Projekt abhängig.

Hier eine Übersicht zu den Projekten:

**Plugins**

*api.NamedOperator* appl.Application appl.Loader gui.Frame

gui.OperatorsListPanel gui.OperatorsMapPanel gui.InputPanel

depends depends

**Plugins-A a.jar**

impl.MinusOperator impl.PlusOperator

**Plugins-B b.jar**

impl.DivOperator impl.ModuloOperator impl.TimesOperator

Und hier das im obersten Projekt enthaltene Interface:

**// Plugins**

package **api**;

public interface **NamedOperator** { public abstract String **name**();

public abstract int **apply**(int x, int y);

}

Das oberste Projekt enthält einige Klassen, die für den Aufbau der GUI genutzt werden

* hier aber nicht näher vorgestellt werden:
  + Ein Frame repräsentiert die gesamte GUI.
  + Ein OperatorsListPanel enthält jeweils eine Zeile (die Buttons eines Plugins)
  + Ein OpereratorsMapPanel enthält für jedes Plugin ein OperatorsListPanel.
  + Ein InputPanel enthält die Ein- und Ausgabefelder.

Im folgenden wird nur die Klasse Loader genauer vorgestellt. Diese Klasse ist für das Auffinden der Plugins und für die Instanziierung der in diesen Plugins enthaltenen Klassen verantwortlich.

Die load-Methode der Klasse Loader enthält eine Map, welche die Namen von jar- Dateien (a.jar, b.jar) abbildet auf Listen von NamedOperators. Eine solche Liste wird für jede in der entsprechenden jar-Datei hinterlegten Klassen ein Objekt enthalten.

Map<String, List<NamedOperator>> operatorsMap = new TreeMap<>();

: TreeMap

"b.jar"

"a.jar"

*Named*

*Operator*

*Named Operator*

*Named Operator*

: Plus Operator

: Minus Operator

: ArrayList

*Named Operator*

*Named Operator*

: Times Operator

: Modulo Operator

: Div Operator

: ArrayList

Hier die in den A- und B-Projekten enthaltenen Implementierungsklassen:

**// Plugins-A**

package **impl**;

import api.NamedOperator;

public class **MinusOperator** implements NamedOperator { public String **name**() { return "Minus"; }

public int **apply**(int x, int y) { return x - y; }

}

**// Plugins-A**

package **impl**;

// ...

public class **PlusOperator** implements NamedOperator { ... }

**// Plugins-B**

package **impl**;

// ...

public class **DivOperator** implements NamedOperator { ... }

**// Plugins-B**

package **impl**;

// ...

public class **ModuloOperator** implements NamedOperator { ... }

**// Plugins-C**

package **impl**;

// ...

public class **TimesOperator** implements NamedOperator { ... }

Die jar-Dateien, die in den A- und B-Projekten gebaut werden, werden kopiert in das

"deploy"-Verzeichnis des Hauptprojekts. Hier die Loader-Klasse:

**// Plugins**

package **appl**;

// ...

public class **Loader** {

private final File **deployDir**;

private final Consumer<Map<String, List<NamedOperator>>> **consumer**; private final Map<String, List<NamedOperator>> **operatorsMap** =

new TreeMap<>();

public **Loader**(File deployDir, Consumer<Map<String, List<NamedOperator>>> consumer) {

this.deployDir = deployDir; this.consumer = consumer;

}

public void **load**() {

for (final File jar : this.deployDir.listFiles( f -> f.getName().endsWith(".jar"))) {

final List<NamedOperator> list = createOperatorsList(jar); operatorsMap.put(jar.getName(), list);

}

this.consumer.accept(this.operatorsMap);

}

private List<NamedOperator> **createOperatorsList**(final File file) { try (final URLClassLoader classLoader = new URLClassLoader(

file.getName(),

new URL[] { file.toURI().toURL() }, ClassLoader.getSystemClassLoader())) {

try (final JarFile jar = new JarFile(file)) { return jar.stream()

.filter(entry ->

!entry.isDirectory())

.filter(entry -> entry.getName().endsWith(".class"))

.map(entry ->

classnameOf(entry.getName()))

.map(name ->

Try.execute(() -> classLoader.loadClass(name)))

.filter(cls ->

NamedOperator.class.isAssignableFrom(cls))

.map(cls ->

Try.execute(() -> (NamedOperator) cls.getConstructor().newInstance()))

.collect(Collectors.toList());

}

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

private static String **classnameOf**(String filename) { return filename.substring(0, filename.length() –

".class".length()).replace('/', '.');

}

}

Dem Konstruktor wird das Deploy-Verzeichnis und ein Consumer übergeben, dessen accept-Methode die aufgebaute Map übergeben werden wird. Dieser Consumer wird die Oberfläche sein, welche die Map präsentieren wird.

Die load-Methode iteriert über alle im Deploy-Verzeichnis enthaltenen jar-Dateien. Für jede dieser jar-Dateien wird die Methode createOpreatorsList aufgerufen. Diese liefert eine Liste zurück, die für jede der in der jeweiligen jar-Datei enthaltenen Klasse ein Objekt eben dieser Klasse enthält. Diese Liste wird unter dem Namen der jar-Datei in eine Map eingetragen, welche dann schließlich an den Consumer übergeben wird (an die Oberfläche, welche diese Map dann präsentiert.

Der Methode createOperatorsList wird eine File-Referenz auf eine jar-Datei übergeben. Aufgrund dieser Referenz wird ein URLClassLoader erzeugt. Alle Einträge der jar-Datei werden analysiert. Handelt es sich um eine class-Datei, wird versucht, diese zu laden. Handelt es sich dabei um eine Klasse, die das Interface NamedOperator implementiert, wird die Klasse instanziiert. Und die Instanz wird der von createOperatorList zurückzuliefernden Liste hinzugefügt.

Die obige Loader-Klasse verwendet zusätzlich folgende Try-Klasse.

package **jn.util**; public class **Try** {

@FunctionalInterface

static public interface **Supplier**<T> { public T **get**() throws Exception;

}

@FunctionalInterface

static public interface **Runnable** { public void **run**() throws Exception;

}

public static <T> T **execute**(Supplier<T> supplier) { try {

return supplier.get();

}

catch(Exception e) {

throw new RuntimeException(e);

}

}

public static void **execute**(Runnable runnable) { execute(() -> {

runnable.run(); return null;

});

}

// ...

}

Hier die main-Methode:

**// Plugins**

package **appl**;

import java.io.File;

import javax.swing.UIManager; import gui.Frame;

public class **Application** {

public static void **main**(String[] args) throws Exception { final Frame frame = new Frame(4, 5);

final File userDir = new File(System.getProperty("user.dir")); final File deployDir = new File(userDir.getPath() + "/deploy"); final Loader loader = new Loader(deployDir,

map -> frame.showOperatorsMap(map)); loader.load();

}

}

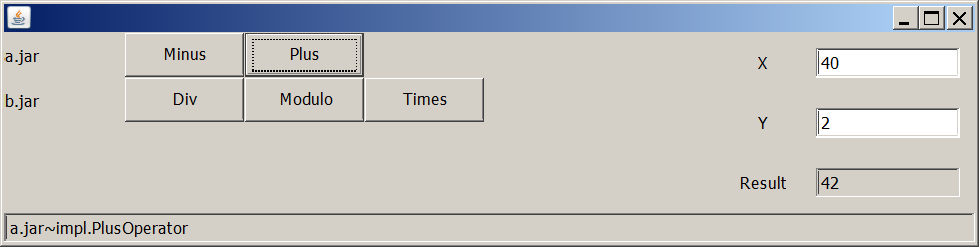
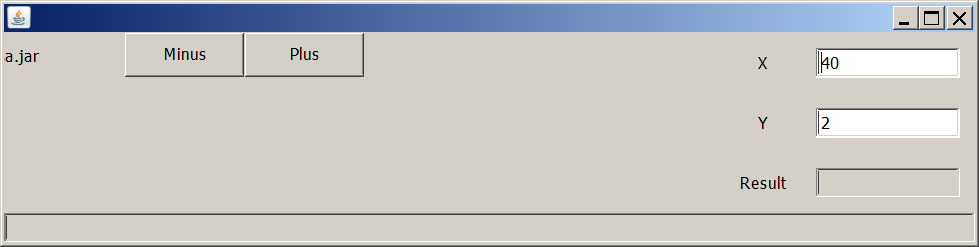
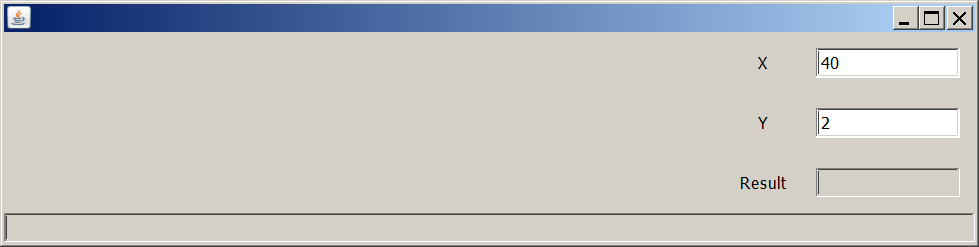
Die accept-Methode des Lambda-Objekt, das an den Konstruktor von Loader

übergeben wird, delegiert an die showOperatorsMap der Frame-Objekts.

* 1. **Hot Deployment**

In letzten Anwendung mussten alle jar-Dateien bereits zum Zeitpunkt des Startens im Deploy-Verzeichnis enthalten sein. Dieses wurde nur ein einziges Mal (beim Startup) gescannt.

In der folgenden Anwendung können jar-Dateien zur Laufzeit ins Deploy-Verzeichnis kopiert werden. Dieses Verzeichnis wird von einem WatchService überwacht. Immer dann, wenn eine jar-Datei hinzugefügt oder eine solche Datei entfernt wird, wird die operatorsMap neu aufgebaut und neu präsentiert – über den erneuten Aufruf des an den Loader übergebene Consumers).



Das oberste Projekt enthält eine Klasse WatchThread, welche einen neuen Thread einrichtet, in welchem das Deploy-Verzeichnis mit Hilfe eines WatchService überwacht wird:

**// HotDeployment**

package **util**;

import java.nio.file.Path;

import java.nio.file.StandardWatchEventKinds; import java.nio.file.WatchKey;

import java.nio.file.WatchService; import java.util.Objects;

import java.util.function.Consumer; public class **WatcherThread** {

private final Path **path**;

private final Consumer<Path> **onCreate**; private final Consumer<Path> **onDelete**;

public **WatcherThread**(

Path path, Consumer<Path> onCreate,

Consumer<Path> onDelete) {

this.path = Objects.requireNonNull(path);

this.onCreate = Objects.requireNonNullElse(onCreate, e -> { }); this.onDelete = Objects.requireNonNullElse(onDelete, e -> { });

}

public void **watch**() { try {

final WatchService watchService = path.getFileSystem().newWatchService();

path.register(watchService, StandardWatchEventKinds.ENTRY\_CREATE, StandardWatchEventKinds.ENTRY\_DELETE);

final Thread thread = new Thread(() -> doWatch(watchService)); thread.setDaemon(true);

thread.start();

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

private void **doWatch**(WatchService watchService) { while (true) {

try {

final WatchKey key = watchService.take(); key.pollEvents().forEach(event -> {

if (event.kind().name().equals("ENTRY\_CREATE")) this.onCreate.accept((Path) event.context());

else if (event.kind().name().equals("ENTRY\_DELETE")) this.onDelete.accept((Path) event.context());

});

key.reset();

}

catch (Exception e) {

// ignore...

}

}

}

}

Die Loader-Klasse erzeugt einen WatcherThread und wird durch zwei neue Methoden erweitert, welche auf das Hinzufügen resp. Entfernen einer jar-Datei reagieren – indem sie einen Eintrag zur operatorsMap hinzufügen resp. einen solchen Eintrag entfernen – und dann jeweils erneut den für die Präsentation zuständigen Consumer aufrufen:

**// HotDeployment**

package **appl**;

// ...

public class **Loader** {

private final File **deployDir**;

private final Consumer<Map<String, List<NamedOperator>>> **consumer**; private final Map<String, List<NamedOperator>> **operatorsMap** =

new TreeMap<>();

public **Loader**(File deployDir, Consumer<Map<String, List<NamedOperator>>> consumer) {

this.deployDir = deployDir; this.consumer = consumer;

final Path path = Paths.get(deployDir.getAbsolutePath());

new WatcherThread(path, this::onCreate, this::onDelete).watch();

}

public void **load**() { ... }

private List<NamedOperator> **createOperatorsList**(final File file) { ... } private static String **classnameOf**(String filename) { ... }

private void **onCreate**(Path path) {

final String jarName = path.toString();

final File file = new File(this.deployDir + "/" + path);

final List<NamedOperator> operatorsList = createOperatorsList(file); this.operatorsMap.put(jarName, operatorsList); this.consumer.accept(this.operatorsMap);

}

private void **onDelete**(Path path) { this.operatorsMap.remove(path.toString()); this.consumer.accept(this.operatorsMap);

}

}

* 1. **ContextClassLoader**

Das A-Projekt enthält einen kleinen Mapper, dessen map-Methode ein Objekt einer bestimmten Klasse erzeugt, dieses via Reflection mit den Werten einer der Methode übergebenen Map initialisiert und dieses Objekt schließlich zurückliefert. Die Klasse Mapper wird mittels eines eigenen URLClassLoaders geladen.

Das B-Projekt enthält eine Klasse Book. Auch diese Klasse wird mittels eines eigenen URLClassLoaders geladen. Der Mapper wird ein Objekt der Klasse Book erzeugen und initialisieren.

Das Problem: Der Mapper weiß nicht, welcher ClassLoader die Book-Klasse geladen hat.

Die Lösung: Einem Thread kann via setContextClassLoader ein ClassLoader zugeordnet werden – die Methode getContextClassLoader liefert dann diesen zugewiesenen ClassLoader zurück. (Es handelt sich hier, wenn man so will, um einen "speziellen" ThreadLocal.)

**ContextClassLoader**

appl.Application

**ContextClassLoader-A**

util.Mapper

**ContextClassLoader-B**

domain.Book

Der map-Methode des Mappers wird der Name der zu instanziierenden Klasse übergeben. Sie ermittelt den ContextClassLoader des aktuellen Threads – und benutzt diesen, um die Klasse zu ggf. zu laden (wenn sie nicht bereits zuvor geladen wurde) und somit eine Referenz auf das Class-Objekt dieser Klasse zu erlangen:

**// ContextClassLoader-A**

package **util**;

// ...

public class **Mapper** {

public Object **map**(String className, Map<String, String> map) { try {

// final Class<?> cls = Class.forName(className); // throws... final Class<?> cls = Thread.currentThread()

.getContextClassLoader().loadClass(className);

final Object obj = cls.getConstructor().newInstance(); for (Field field : cls.getFields())

field.set(obj, map.get(field.getName())); return obj;

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

}

Hinweis: Würde – wie oben in auskommentierter Form gezeigt – die Methode

Class.forName aufgerufen, würde eine ClassNotFoundException geworfen. Die Book-Klasse ist trivial:

**// ContextClassLoader-B**

package **domain**; public class **Book** {

public String **isbn**; public String **title**;

@Override

public String **toString**() { ... }

}

Die main-Methode der Application-Klasse erzeugt zwei URLClassLoader – einen für die Klasse Mapper und einen zweiten für die Klasse Book. Mittels dieser ClassLoader werden die beiden Klassen geladen und somit die Class-Objekte für diese Klassen erzeugt.

Der aktuelle Thread wird dann mit demjenigen ClassLoader verbunden, der die Book- Klasse geladen hat. Dann wird ein Mapper erzeugt und auf diesen via Reflection die map-Methode aufgerufen, welcher der Name der zu instanziierenden Klasse übergeben wird – und eine Map, aufgrund derer das erzeugte Objekt zu initialisieren ist:

**// ContextClassLoader**

package **appl**;

// ...

public class **Application** {

public static void **main**(String[] args) throws Exception {

final File userDir = new File(System.getProperty("user.dir")); final File binA = new File(userDir.getPath() + "-A/bin"); final File binB = new File(userDir.getPath() + "-B/bin");

try (final URLClassLoader classLoaderA =

ClassLoaderUtil.createURLClassLoader("a", binA); final URLClassLoader classLoaderB =

ClassLoaderUtil.createURLClassLoader ("b", binB)) {

final Class<?> clsA = classLoaderA.loadClass("util.Mapper"); final Class<?> clsB = classLoaderB.loadClass("domain.Book");

final ClassLoader oldClassLoader = Thread.currentThread().getContextClassLoader();

Thread.currentThread().setContextClassLoader( clsB.getClassLoader());

final Map<String, String> map = new HashMap<>(); map.put("isbn", "1111");

map.put("title", "Pascal");

final Object mapper = clsA.getConstructor().newInstance(); final Object book =

clsA.getMethod("map", String.class, Map.class)

.invoke(mapper, "domain.Book", map); Thread.currentThread().setContextClassLoader(oldClassLoader);

System.out.println(book);

}

}

}

Die Ausgabe:

Book [1111, Pascal]

* 1. **Deserialisierung**

Die Deserialisierungs-Klasse ObjectInputStream hat eine überschreibbare Methode

resolveClass:

protected Class<?> **resolveClass**(ObjectStreamClass cls)

Was hat es damit auf sich?

**Deserialisation**

appl.Application

**Deserialisation-A**

domain.Book domain.BookSupplier

**Deserialisation-B**

domain.Book domain.BookConsumer

Angenommen, wir haben zwei Projekte (A und B), deren Klassen jeweils von einem eigenen ClassLoader geladen werden. Beide Projekten enthalten eine Book-Klasse. Die beiden Book-Klassen sind strukturgleich. Da sie aber von verschiedenen ClassLoadern geladen werden, handelt es sich natürlich um verschiedene(!) Klassen.

Das A-Projekt enthält zusätzlich eine Klasse BookSupplier, deren get-Methode ein Book zurückliefert. Das B-Projekt enthält eine "inverse" Klasse BookConsumer, deren accept-Methode ein Book ausgibt. (BookSupplier implementiert das Interface Supplier, die Klasse BookConsumer implementiert das Interface Consumer.)

Wie kann ein Book, das der BookSupplier liefert, an den BookConsumer übergeben werden? (Der BookSupplier liefert ja ein Book zurück, dessen Klasse zur Klasse des vom BookConsumer verlangten Books nicht kompatibel ist.)

Eine Lösung besteht darin, das vom BookSupplier gelieferte Book zu serialisieren und dem BookConsumer dann das Resultat einer Deserialisierung zu übergeben – wobei wir dann aber bei der Deserialisierung eingreifen müssen. Wir müssen resolveClass überschreiben.

An resolveClass wird ein ObjectStreamClass-Objekt übergeben, welches den Namen der Klasse des zu deserialisierenden Objekts enthält. Diesen Namen können wir

nutzen, um diejenige Klasse zu ermitteln, die vom ClassLoader des BookConsumer

geladen wurde.

Hier die Klassen des A-Projekts:

**// Deserialisation-A**

package **domain**;

import java.io.Serializable;

public class **Book** implements Serializable { public String **isbn**;

public String **title**;

public **Book**(String isbn, String title) { this.isbn = isbn;

this.title = title;

}

@Override

public String **toString**() { ... }

}

**// Deserialisation-A**

package **domain**;

import java.util.function.Supplier;

public class **BookSupplier** implements Supplier<Book> { @Override

public Book **get**() {

return new Book("1111", "Pascal");

}

}

Und hier die Klassen des B-Projekts:

**// Deserialisation-B**

package **domain**;

import java.io.Serializable;

public class **Book** implements Serializable { public String **isbn**;

public String **title**;

public **Book**(String isbn, String title) { this.isbn = isbn;

this.title = title;

}

@Override

public String **toString**() { ... }

}

**// Deserialisation-B**

package **domain**;

import java.util.function.Consumer;

public class **BookConsumer** implements Consumer<Book> { @Override

public void **accept**(Book book) {

System.out.println(book);

}

}

Die main-Methode erzeugt zwei ClassLoader: der erste ist für das Laden der A- Klassen, der zweite für das Laden der B-Klassen zuständig. Wir laden die Klasse BookSupplier und BookConsumer und erzeugen jeweils eine Instanz dieser beiden Klassen (die wir dann über die Interfaces Supplier resp. Consumer ansprechen können).

Dann lassen wir uns von dem BookSupplier ein Book liefern (von dem wir nur wissen, dass es irgendein Object ist). Würden wir dieses Book an den BookConsumer übergeben, würden wir uns eine ClassCastException einfangen.

Also serialisieren wir das Book und stellen bei der Deserialisierung sicher, dass das von readObject gelieferte Book das Objekt einer Klasse ist, die von B-ClassLoader geladen wurde (indem wir resolveClass geeignet überschreiben):

**// Deserialisation**

package **appl**;

// ...

public class **Application** {

public static void **main**(String[] args) throws Exception {

final File userDir = new File(System.getProperty("user.dir"));

final File binA = new File(userDir.getPath() + "-A/bin"); final File binB = new File(userDir.getPath() + "-B/bin");

try (final URLClassLoader classLoaderA =

ClassLoaderUtil.createURLClassLoader ("a", binA); final URLClassLoader classLoaderB =

ClassLoaderUtil.createURLClassLoader ("b", binB)) {

final Class<?> clsSupplier = classLoaderA.loadClass("domain.BookSupplier");

final Class<?> clsConsumer = classLoaderB.loadClass("domain.BookConsumer");

final Supplier supplier =

(Supplier) clsSupplier.getConstructor().newInstance(); final Consumer consumer =

(Consumer) clsConsumer.getConstructor().newInstance();

final Object bookA = supplier.get(); System.out.println("bookA : " + ClassName.of(bookA));

// consumer.accept(bookA);

// a//domain.Book cannot be cast to b//domain.Book

final ByteArrayOutputStream aos = new ByteArrayOutputStream(); final ObjectOutputStream oos = new ObjectOutputStream(aos); oos.writeObject(bookA);

oos.close();

final ByteArrayInputStream ais =

new ByteArrayInputStream(aos.toByteArray());

final ObjectInputStream ois = new **ObjectInputStream**(ais) { @Override

protected Class<?> **resolveClass**(ObjectStreamClass cls)

throws IOException, ClassNotFoundException { return clsConsumer.getClassLoader()

.loadClass(cls.getName());

}

};

final Object bookB = ois.readObject(); System.out.println("bookB : " + ClassName.of(bookB)); consumer.accept(bookB);

}

}

}

Die Ausgaben:

bookA : a//domain.Book bookB : b//domain.Book Book [1111, Pascal]

* 1. **DynamicProxy**

Der Methode Proxy.newProxyInstance muss als erster Parameter bekanntlich ein ClassLoader übergeben werden (wir haben in den Beispielen zum Thema DynamicProxy stets den System-ClassLoader übergeben). Was hat es mit diesem ClassLoader auf sich?

In einem A-Projekt ist sowohl das Interface MathService als auch die Implementierungs-Klasse MathServiceImpl enthalten. Das Hauptprojekt enthält neben der Application-Klasse eine Klasse TraceHandler (ein InvocationHandler) . Das Hauptprojekt hat keinerlei Abhängigkeiten vom A-Projekt – kennt also weder das Interface noch die Implementierungsklasse.

**DynamicProxy**

appl.Application appl.TraceHandler

**DynamicProxy-A**

iface.MathService impl.MathServiceImpl

Wie kann die Application ein DynamicProxy erzeugen, welches das Interface MathService implementiert und als InvocationHandler einen TraceHandler benutzt, der an ein MathServiceImpl-Objekt delegiert?

Hier die Elemente des A-Projekts:

**// DynamicProxy-A**

package **iface**;

public interface **MathService** {

public abstract int **sum**(int x, int y); public abstract int **diff**(int x, int y);

}

**// DynamicProxy-A**

package **impl;**

import iface.**MathService**;

public class **MathServiceImpl** implements MathService { public int **sum**(int x, int y) { return x + y; } public int **diff**(int x, int y) { return x - y; }

}

Die Application:

**// DynamicProxy**

package **appl**;

// ...

public class **Application** {

public static void **main**(String[] args) throws Exception {

final File userDir = new File(System.getProperty("user.dir")); final File binA = new File(userDir.getPath() + "-A/bin");

try (final URLClassLoader classLoader =

ClassLoaderUtil.createURLClassLoader ("a", binA)) {

final Class<?> clsMathService = classLoader.loadClass("iface.MathService");

final Class<?> clsMathServiceImpl = classLoader.loadClass("impl.MathServiceImpl");

final Object mathServiceImpl = clsMathServiceImpl.getConstructor().newInstance();

final Object mathService = Proxy.newProxyInstance( classLoader,

new Class<?>[] { clsMathService }, new TraceHandler(mathServiceImpl));

System.out.println(ClassName.of(mathService));

mathService.getClass().getMethod(

"sum", int.class, int.class).invoke(mathService, 40, 2); mathService.getClass().getMethod(

"diff", int.class, int.class).invoke(mathService, 80, 3);

}

}

}

Man beachte, dass an newProxyInstance genau derjenige ClassLoader übergeben werden muss, mittels dessen die Klasse das zu implementierende Interfaces geladen wurde.

Als TraceHandler wird die im DynamicProxy-Kapitel entwickelte Klasse verwendet. Die Ausgaben:

a//com.sun.proxy.$Proxy0

>> sum[40, 2]

<< sum[40, 2] --> 42

>> diff[80, 3]

<< diff[80, 3] --> 77

Wie man sieht, ist die generierte Proxy-Klasse von demjenigen ClassLoader geladen worden, der auch das Interface MathService geladen hat (der a-Loader).

**21 Literatur**

Joshua Block: Effective Java Joshua Block: Java-Puzzles Bruce Eckel: Thinking in Java Horstmann: Java

Robert C. Martin: Clean Code Martin Fowler: Refactoring Gamma et al: Design Pattern

Kent Beck: Test Driven Development

Doug Lea: Concurrent Programming in Java