

# Java – Neuerungen seit Java 9

## Skript

OOA	OOD	OOP
OOA	OOP	OOD
OOD	OOA	OOP
OOD	OOP	OOA
OOP	OOA	OOD
OOP	OOD	OOA

Johannes Nowak

e-mail: [johannes.nowak@t-online.de](mailto:johannes.nowak@t-online.de)

Oktober 2017

Juli 2018

März 2019

Juli 2019

März 2020

# Inhalt

<b>1</b>	<b>Einleitung</b>	<b>6</b>
<b>2</b>	<b>Probleme mit dem "alten" Java</b>	<b>12</b>
2.1	CLASSPATH-Reihenfolge	13
2.2	Unzureichende Kapselung	17
2.3	Probleme mit Reflection	19
2.4	Zur Laufzeit fehlende Jars	21
<b>3</b>	<b>Das Java Platform Module System (JPMS)</b>	<b>23</b>
3.1	Exports / Requires	29
3.2	Beispiel: Service-Interfaces und –Implementierungen	33
3.3	Opens	35
3.4	Beispiel: Deep Reflection	39
3.5	Direkte / Indirekte Nutzung von Modulen	42
3.6	Automatic Modules	46
3.7	Verbot Modul-übergreifender Pakete	48
3.8	Exports To	50
3.9	Requires Transitive	52
3.10	Aggregat-Module	55
3.11	Das ServiceLoader-Konzept	57
3.12	Provides und Uses	60
3.13	Mixing	63
3.14	Add Exports	73
3.15	Add Opens	76
3.16	Add Modules	79
<b>4</b>	<b>Reflection</b>	<b>82</b>
4.1	Class.newInstance deprecated	83
4.2	Die Klassen Module und ModuleDescriptor	86
4.3	Die Klasse ModuleLayer	91
<b>5</b>	<b>Spracherweiterungen</b>	<b>94</b>

---

5.1	Diamond-Operator	95
5.2	Private Interface-Methoden	100
5.3	Erweiterung des resource-try	101
5.4	Vargs	103
5.5	Underscore	106
<b>6</b>	<b>Erweiterungen der Standardbibliothek</b>	<b>107</b>
6.1	Initialisierung von Collections	108
6.2	Process	111
6.3	Stream	115
6.4	Optional	118
6.5	StackWalker	120
6.6	Vereinheitlichtes Logging	129
6.7	CompletableFuture	134
6.8	Kompakte Strings	143
6.9	Arrays	147
6.10	Objects	152
6.11	Deprecated	155
6.12	Cleaner	157
<b>7</b>	<b>Flow</b>	<b>161</b>
7.1	Ein einfacher Subscriber	166
7.2	Ein StdSubscriber für's Testen	172
7.3	Requests	178
7.4	SubmissionPublisher - Details	181
7.5	Ein PeriodicPublisher	184
7.6	Processors	186
7.7	Beispiel: Datenbank	190
7.8	Beispiel: Swing-Diagramme	194
7.9	Beispiel: Swing-Diagramme Client-Server	197
<b>8</b>	<b>Tools</b>	<b>199</b>
8.1	jlink	200
8.2	jdeps	203

---

8.3	jshell	205
8.4	Weitere Werkzeuge	207
<b>9</b>	<b>Java 10</b>	<b>208</b>
9.1	Local Variable Type Inference	209
9.2	Collections und Collectors	213
9.3	Optional	216
9.4	Runtime.Version	218
<b>10</b>	<b>Java 11</b>	<b>219</b>
10.1	Ausführen von Single-File Sourcecode	220
10.2	Benutzung von var in Lambda-Parametern	221
10.3	Erweiterungen der String-Klasse	222
10.4	Erweiterungen der Files-Klasse	224
10.5	Erweiterungen der Optional-Klasse	226
10.6	Erweiterungen des Predicate-Interfaces	227
10.7	Die Klasse HttpClient	228
10.8	WebSockets mit dem HttpClient	232
<b>11</b>	<b>Java 12</b>	<b>236</b>
11.1	Switch	237
11.2	Strings	241
<b>12</b>	<b>Java 13</b>	<b>242</b>
12.1	Text Blocks	243
12.2	Neue String-Methoden	246
<b>13</b>	<b>Anhang: Build mit Maven</b>	<b>247</b>
<b>14</b>	<b>Literatur</b>	<b>252</b>

# 1 Einleitung

## Inhalte

- Im Kapitel 2 werden einige der Nachteile beschrieben, die mit dem Java-`CLASSPATH` zusammenhängen – Nachteile, die das im nächsten Kapitel beschriebene Modul-Konzept aus der Welt schafft.
- Kapitel 3 beschreibt eben dieses neue Modul-Konzept. Neben Klassen und Packages gibt es nur sog. Modules. Ein Modul ist eine Zusammenfassung von Packages unter einem Namen. Module müssen ihre Export- und Import-Beziehungen explizit angeben. Module werden sich insbesondere bei großen Anwendungen als nützliches Strukturierungsmittel erweisen.
- Kapitel 4 beschreibt die Neuerungen im Reflection-API. Hier geht's insbesondere um die neuen Klassen `ModuleLayer`, `Module` und `ModuleDescriptor`.
- Kapitel 5 beschreibt einige Spracherweiterungen u.a. den besseren Resource-Try und private Interface-Methoden.
- Kapitel 6 beschreibt die Erweiterungen der Standardbibliothek: Factory-Methoden zur Erzeugung von Collections, Erweiterungen der `Process`-Klasse, die neue Klasse `StackWalker`, Erweiterungen von `Streams` und `Optionals`, vereinheitlichtes Logging und "kompakte Strings".
- Im Kapitel 7 geht's um das neue `Flow`-API. Diese Interfaces werden die Grundlage für "reaktive Programmierung" mit Java bilden. Dieses Programmier-Paradigma basiert dem Publisher-Subscriber-Konzept – wobei alle beteiligten Instanzen asynchron arbeiten.
- Im Kapitel 8 schließlich geht's um einige neue resp. erweiterte Tools – u.a. um `jlink` – ein Tool, welches ein kompaktes Laufzeit-Image erzeugen kann.
- Im Kapitel 9 geht's um die Erweiterungen von Java 10 (insbesondere um "local variable type inference").
- Im Kapitel 10 geht's um die Erweiterungen von Java 11 (insbesondere um die neue `HttpClient`-Klasse).
- Im Kapitel 11 geht's um die Erweiterungen von Java 12 (insbesondere um die erweiterte `switch`-Anweisung).
- Im Kapitel 12 geht's um die Erweiterungen von Java 13

- Im Anhang wird gezeigt, wie Maven für den Bau modularer jars verwendet werden kann.

## Beispiele zur Modularisierung

In einem weiteren Workspace existieren einige weitere Beispiele zum Thema Modularisierung. Diese Projekte enthalten nicht-modularisierte Anwendungen, die zu modularisierten Anwendungen umgebaut werden.

Es handelt sich um Beispiele zu folgenden Themen:

- Ein CSV-Mapper
- Eine Spring-Anwendung
- Eine JPA-Anwendung
- Eine JSON-Parser

## ant-Skripte

Die Projekte des Eclipse-Workspace lassen sich allesamt mittels ant-Skripts bauen (und natürlich "automatisch" auch mit Eclipse-Mitteln). Die ant-Skripte haben den Vorteil, dass aus ihnen genau ersichtlich ist, mit welchen (neuen) Parametern `java`, `jar` und `java` aufgerufen werden müssen.

Alle ant-Skripte beziehen sich auf ein Basis-Skript: auf die `build.xml` des `shared-Projekts`:

```
<project>

  <property name="workspace" value="${basedir}/.." />
  <property name="shared" value="${workspace}/shared" />
  <property name="dependencies"
value="${workspace}/dependencies" />

  <macrodef name="build">
    <attribute name="name"/>
    <element name="prepare" optional="true"/>
    <element name="paths" optional="true"/>
    <sequential>
      <delete dir="${basedir}-@{name}/tmp"
failonerror="false" />
      <mkdir dir="${basedir}-@{name}/tmp" />
      <prepare/>
      <javac
srcdir="${basedir}-@{name}/src"
```

```
        destdir="${basedir}-${name}/tmp"
        nowarn="true" debug="true" failonerror="true"
        includeantruntime="false">
    <paths/>
</javac>
<jar
    basedir="${basedir}-${name}/tmp"
    filesetmanifest="merge"
    destfile="${basedir}/build/${name}.jar" />
    <delete dir="${basedir}-${name}/tmp"
failonerror="false" />
    </sequential>
</macrodef>

</project>
```

Im Folgenden soll die Funktionsweise des in dieser Datei definierten Ant-Macros (build) erläutert werden.

Viele Anwendungen des Workspaces bestehen i.d.R. aus drei Projekten:

```
<basis-name>
<basis-name>-appl
<basis-name>-mod
```

Dabei ist `<basis-name>` z.B. `x0203-problems-reflection`.

Das `mod`-Projekt enthält Klassen, die im `appl`-Projekt genutzt werden. Das `appl`-Projekt enthält die Startklasse.

Jede Anwendung enthält eine eigene `build.xml`, welche die oben vorgestellte `build.xml`-Datei des `shared`-Projekts importiert. Diese `build.xml` liegt im `<basis-name>`-Projekt (im Folgenden als Basis-Projekt bezeichnet).

Jede `build.xml`-Datei führt folgende Schritte aus:

Die Klassen des `mod`-Projekts werden kompiliert; die Resultate der Kompilierung werden in einem eigenen `tmp`-Verzeichnis des `mod`-Projekts abgelegt. Alle `class`-Dateien dieses `tmp`-Verzeichnisses werden dann zu einer `jar` zusammengeschnürt, welche im `build`-Verzeichnis des Basis-Projekts abgelegt wird. Der Name der `jar`-Datei ist `mod.jar` (enthält also die Endung des Projektnamens).

Dann werden die Klassen des `appl`-Projekts kompiliert – wobei in den `classpath` (resp. dem `Module-Path` – siehe hierzu später) die `mod.jar`-Datei aufgenommen wird. Die Klassen des `appl`-Projekts werden in das `tmp`-Verzeichnis dieses Projekts kompiliert



und zu einer `jar`-Datei namens `appl.jar` zusammengebunden. Auch diese wird im `build`-Verzeichnis des Basisprojekts abgelegt.

Dann wird die Anwendung gestartet. Der bei der Ausführung benutzte `classpath` enthält die beiden `jar`-Dateien (`mod.jar` und `appl.jar`).

Nach erfolgter Kompilierung sehen die Verzeichnisse dann etwa wie folgt aus (wobei das `tmp`-Verzeichnis nach Ausführung von `javac` und `jar` wieder gelöscht wird):

**x0203-problems-reflection**

```
build
  appl.jar
  mod.jar
build.xml
```

**x0203-problems-reflection-appl**

```
src
  jj
    appl
      Application.java
  jj
    domain
      Book.java
tmp
  jj
    appl
      Application.class
  jj
    domain
      Book.class
```

**x0203-problems-reflection-mod**

```
src
  jj
    reflection
      Mapper.java
tmp
  jj
    reflection
      Mapper.class
```

Die von uns verwendeten Package-Namen beginnen jeweils mit `jj` – z.B. `jj.appl` oder `jj.domain`. Dem Autor ist natürlich bekannt, dass "reale" Paket-Namen etwas anders ausschauen... (`jj` ist eine Abkürzung, die für den Autor eine persönliche Bedeutung hat, die er aber nicht verraten will.)

Hier die `build.xml`, mittels derer die oben genannten Schritte ausgeführt werden können:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE project>

<project default="run">

    <import file="../../shared/build.xml" />

    <target name="build-mod">
        <build name="mod">
            <paths>
            </paths>
        </build>
    </target>

    <target name="build-appl" depends="build-mod">
        <build name="appl">
            <paths>
                <classpath location="${basedir}/build/mod.jar" />
            </paths>
        </build>
    </target>

    <target name="run" depends="build-appl">
        <java classname="jj.appl.Application" fork="true">
            <classpath location="${basedir}/build/mod.jar" />
            <classpath location="${basedir}/build/appl.jar" />
        </java>
    </target>

</project>
```

Aus dem an das `build`-Macro übergebenen Namen (`mod` resp. `appl`) wird der Name des entsprechenden Projektes abgeleitet, dessen Klassen kompiliert werden – und der Name der erzeugten `jar`-Datei.

(Die Verwendung des `build`-Macros führt zu sehr übersichtlichen `build`-Dateien. Es funktioniert natürlich nur bei solchen Anwendungen, deren Projektstruktur der oben dargestellten beispielhaften Struktur ähnelt.)

Der Aufruf des `run`-Targets bewirkt dasselbe wie die Ausführung der folgenden Console-Kommandos (wobei `${basedir}` für das Wurzel-Verzeichnis (`x...-jpms-reflection`) steht:

```
mkdir ${basedir}-mod/tmp
mkdir ${basedir}-appl/tmp

javac -d ${basedir}-mod/tmp
    ${basedir}-mod/src/module-info.java
    ${basedir}-mod/src/jj/reflection/Mapper.java

jar --create --file ${basedir}/build/mod.jar
    -C ${basedir}-mod/tmp .

javac --module-path
    ${basedir}/build/mod.jar;${shared}/build/util.jar
    -d ${basedir}-appl/tmp
    ${basedir}-appl/src/module-info.java
    ${basedir}-appl/src/jj/appl/Application.java
    ${basedir}-appl/src/jj/domain/Book.java

jar --create --file ${basedir}/build/appl.jar
    -C ${basedir}-appl/tmp .

java --module-path ${basedir}/build/appl.jar;
    ${basedir}/build/mod.jar;${shared}/build/util.jar
    --module jj.appl/jj.appl.Application
```

(Sieht hierzun das `"raw"`-Target des o.g. Projekts.)

## Das shared-Projekt

Das `shared`-Projekt enthält einige Klassen, die in den anderen Projekten des Workspaces immer wieder verwendet werden. Diese Klassen können mittels des Ant-Scripts `buildUtil.xml` übersetzt und in einer (modularen) `jar`-Datei namens `util.jar` (im `build`-Verzeichnis enthalten) zusammengefasst werden.

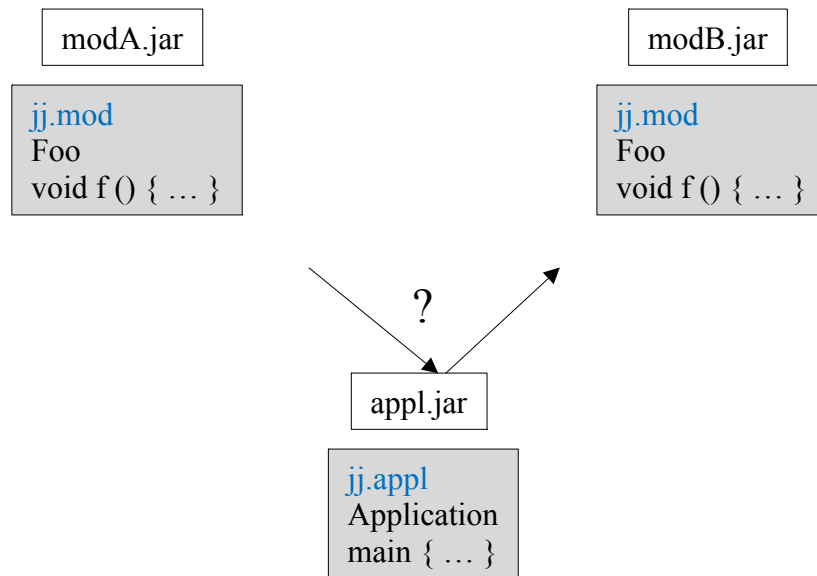


## 2 Probleme mit dem "alten" Java

Das in Java 9 eingeführte Modul-Konzept ist u.a. eine Antwort auf die Probleme, die mit dem Classpath und den jar-Dateien des "alten" Java verbunden waren. Im Folgenden werden deshalb zunächst einige dieser Probleme erörtert:

- Der `CLASSPATH` kann "doppelte" Elemente enthalten
- Elemente ein und desselben Packages können über mehrere jar-Dateien verteilt sein.
- Wir können die Sichtbarkeit von Klassen nicht befriedigend einschränken.
- Mittels Reflection wir auch ohne Einschränkung auf private Elemente von Klassen zugreifen (es sei denn, dies wird von einem `SecurityManager` unterbunden).
- Das Fehlen einer erforderlichen jar-Datei im `CLASSPATH` macht sich bei der Ausführung von Programmen möglicherweise erst spät bemerkbar.

## 2.1 CLASSPATH-Reihenfolge



Angenommen, es existieren zwei `jar`-Dateien, welche jeweils eine Klasse `Foo` mit einer Methode `f` (parameterlos und `void`) enthalten. Beide Klassen liegen im Paket `jj.mod`.

### modA.jar

```
package jj.mod;

public class Foo {
    public void f() {
        System.out.println("set on fire");
    }
}
```

### modB.jar

```
package jj.mod;

public class Foo {
    public void f() {
        System.out.println("blow out fire");
    }
}
```

In der `main`-Methode einer `Application` wird nun die (ja: welche?) Klasse `Foo` genutzt:

```
package jj.appl;

public class Application {
    public static void main(String[] args) {
        new pack.Foo().f();
    }
}
```

Die jar-Dateien werden wie folgt gebaut:

```
<target name="build-modA">
    <build name="modA"/>
</target>

<target name="build-modB">
    <build name="modB"/>
</target>

<target name="build-appl" depends="build-modA, build-modB">
    <build name="appl">
        <paths>
            <classpath location="${basedir}/build/modA.jar" /
>
            <classpath location="${basedir}/build/modB.jar" /
>
        </paths>
    </build>
</target>
```

Wir starten die Application:

```
<target name="run" depends="build-appl">
    <java classname="jj.appl.Application" fork="true">
        <classpath location="${basedir}/build/modA.jar" />
        <classpath location="${basedir}/build/modB.jar" />
        <classpath location="${basedir}/build/appl.jar" />
    </java>
</target>
```

Die Ausgabe:

set on fire

Und wir starten die Application: erneut – diesmal aber mit einem etwas anderen classpath (man beachte die Reihenfolge der Einbindung von modA.jar und modB.jar):

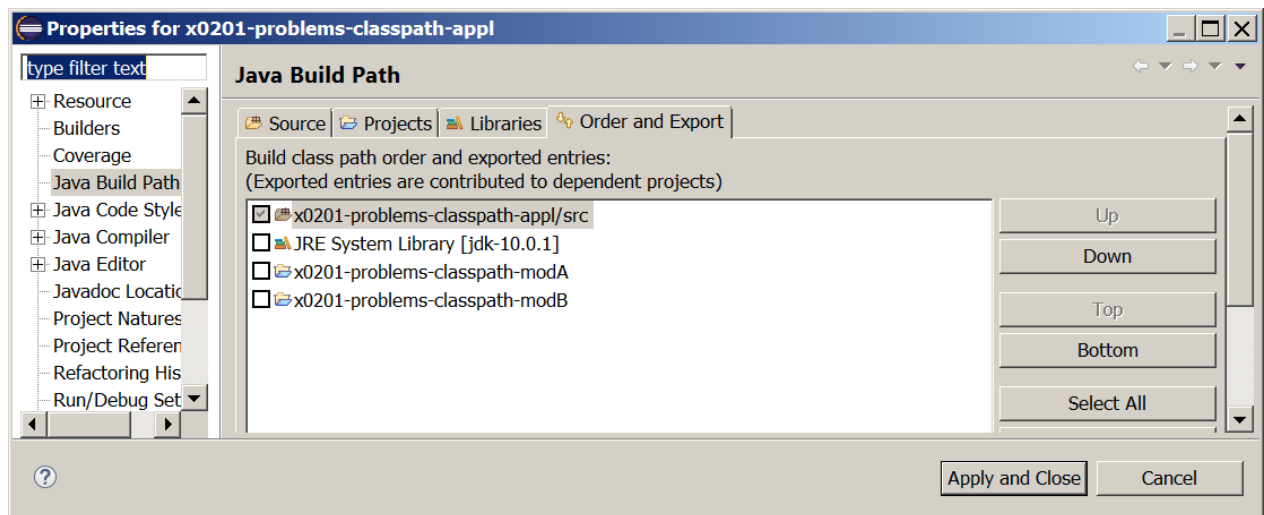
```
<target name="run" depends="build-appl">
  <java classname="jj.appl.Application" fork="true">
    <classpath location="${basedir}/build/modB.jar" />
    <classpath location="${basedir}/build/modA.jar" />
    <classpath location="${basedir}/build/appl.jar" />
  </java>
</target>
```

Die Ausgabe:

```
blow out fire
```

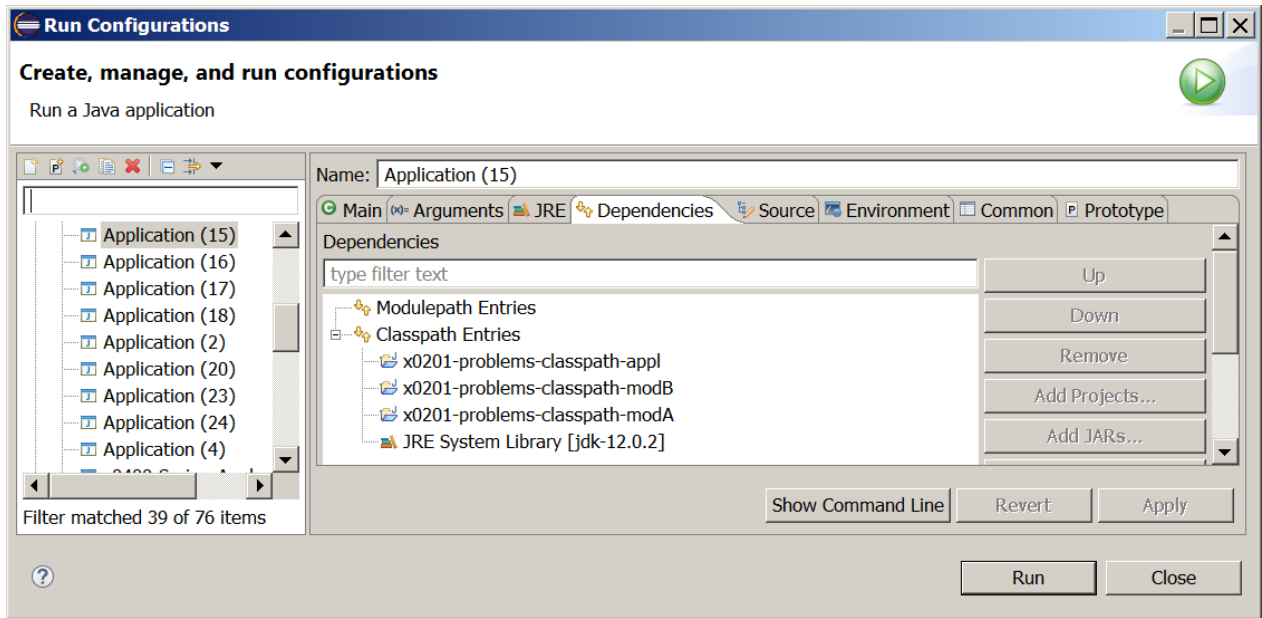
Der Compiler und die virtuelle Maschine prüfen nicht, ob im CLASSPATH Elemente (hier: jars) angegeben sind, welche Klassen mit demselben voll qualifizierten Namen enthalten. Stattdessen wird einfach das erste passende Element genutzt. Es kommt daher auf die Reihenfolge an, in welcher die Elemente im CLASSPATH enthalten sind.

Nebenbei: In Eclipse wird die Reihenfolge der CLASSPATH-Einträge wie folgt festgelegt:



Die Run-Configuration sieht wie folgt aus:





Ein weiteres Problem: Klassen ein und desselben packages können auf mehrere jars (resp. Verzeichnisse) verteilt sein. Eine Anwendung könnte somit etwa folgende Klassen nutzen:

### modC.jar

```
package jj.mod;

public class Alpha {
    public void f() {
        new Beta().g();
    }
}
```

### modD.jar

```
package jj.mod;

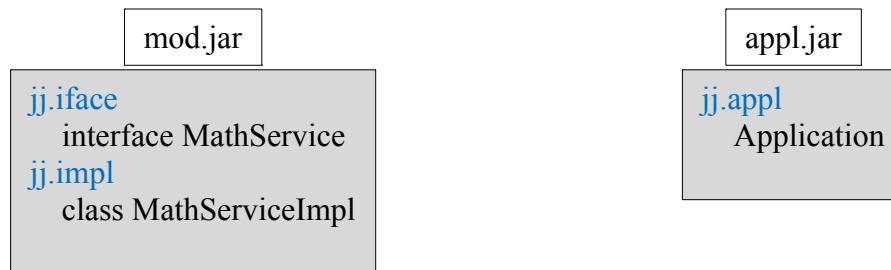
public class Beta {
    void g() { ... } // default-Sichtbarkeit
}
```

Alpha (aus modC.jar) könnte auf die mit der Default-Sichtbarkeit ausgestatten Elemente der in der modD.jar-Datei enthaltenen Klasse Beta zugreifen – denn die Package-Namen beider Klassen sind identisch.

Schön ist das nicht – und kann leicht zu Verwirrungen führen...



## 2.2 Unzureichende Kapselung



Die Datei `mod.jar` enthalte die `class`-Dateien der folgenden Interfaces / Klassen:

### `mod.jar`

```
package jj.iface;

import impl.MathServiceImpl;

public interface MathService {

    public abstract int sum(int x, int y);
    public abstract int diff(int x, int y);

    public final MathService instance = new MathServiceImpl();
}
```

```
package jj.impl;

import iface.MathService;

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;
    }
}
```

Damit `MathServiceImpl` in `MathService` genutzt werden kann, muss die `MathServiceImpl`-Klasse `public` sein (weil sie in einem anderen Paket angesiedelt ist als `MathService`). Angenommen, nur `MathService` soll von Fremden genutzt werden können – `MathServiceImpl` soll nur von `MathService` genutzt werden. Diese unterschiedliche Sichtbarkeit kann leider nicht spezifiziert werden.

Mit dieser `jar` im `CLASSPATH` kann die folgende Klasse übersetzt werden (sie nutzt sowohl `MathService` als auch `MathServiceImpl`):

### **appl.jar**

```
package jj.appl;

import jj.iface.MathService;
import jj.impl.MathServiceImpl;

public class Application {
    public static void main(String[] args) {
        MathService mathService = new MathServiceImpl();
        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));
    }
}
```

Natürlich könnten wir sowohl das Interface als auch die Implementierung in einem einzigen Package unterbringen – und die `MathServiceImpl`-Klasse mit der Package-Sichtbarkeit ausstatten. Dann könnte die Anwendung tatsächlich nur das Interface nutzen. Ein solches Paket, das sowohl die Spezifikation als auch die Implementierung beinhaltet, ist aber wenig wünschenswert...

Und die Sachlage ändert sich auch dann nicht, wenn wir eine `MathServiceFactory` einführen...

## 2.3 Probleme mit Reflection



Das `mod`-Projekt enthält einen einfacher Reflection-basierten `Mapper`:

### mod.jar

```
package jj.reflection;

import java.lang.reflect.Field;
import java.util.HashMap;
import java.util.Map;

public class Mapper {
    public static Map<String, Object> map(Object obj) {
        final Map<String, Object> map = new HashMap<>();
        try {
            for (final Field field :
obj.getClass().getDeclaredFields()) {
                field.setAccessible(true);
                final Object value = field.get(obj);
                map.put(field.getName(), value);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return map;
    }
}
```

Man beachte den Aufruf von `setAccessible` – der `Mapper` muss auf die Attribute von Objekten zugreifen können, die i.d.R. `private` sind.

Im `appl`-Projekt sind die Startklasse und eine Klasse `Book` definiert

### appl.jar

```
package jj.domain;

public class Book {

    private String isbn;
    private String title;
    private int year;
    private String author;

    public Book(String isbn, String title, int year, String
author) { ... }

    // getter und setter...

    @Override
    public String toString() {
        return this.getClass().getSimpleName() +
            " [" + isbn + ", " + title + ", " + year + ", " +
author + "];"
    }
}
```

```
package jj.appl;

import jj.domain.Book;
import jj.reflection.Mapper;

public class Application {

    public static void main(String[] args) {
        Book book = new Book("1111", "Pascal", 1970, "N. Wirth");
        Mapper.map(book).forEach((name, value) ->
            System.out.println(name + " = " + value));
    }
    private void pri() { }
}
```

Der Reflection-basierte Zugriff auf die privaten Attribute einer Klasse ist also jederzeit möglich (es sei denn, wir registrieren für die Anwendung einen `SecurityManager`).

Es wäre schön, wenn wir genau spezifizieren könnten, welche Pakete für einen solchen Reflection-basierten Zugriff auf private Elemente ihrer Klassen offen stehen.

(Der Reflection-basierte Zugriff auf private Elemente wird auch als Deep Reflection bezeichnet.)



## 2.4 Zur Laufzeit fehlende Jars



Auch ein System, was korrekt gebaut wurde, kann zur Laufzeit Probleme bereiten. Eine `jar`-Datei, die zur Übersetzungszeit vorlag, fehlt möglicherweise zur Laufzeit. Dann wird der Versuch, ein Element dieser nicht vorhandenen `jar`-Datei anzusprechen, natürlich zu einem Fehler führen.

Dieser Fehler wird allerdings möglicherweise erst dann erkannt, wenn die Woche bereits eine Woche lang lief (wenn nämlich erst dann dieser problematische Zugriff stattfindet).

### mod.jar

```
package jj.mod;

public class Foo { }
```

### appl.jar

```
package jj.appl;

import jj.mod.Foo;

public class Application {
    public static void main(String[] args) {
        System.out.println("main starts...");
        new Foo();
    }
}
```

Wir erzeugen die beiden `jar`-Dateien (wobei zur Erzeugung von `appl.jar` natürlich `mod.jar` bereits existieren muss):

```
<target name="build-mod">
    <build name="mod" />
```



```
</target>

<target name="build-appl" depends="build-mod">
  <build name="appl">
    <paths>
      <classpath location="${basedir}/build/mod.jar" />
    </paths>
  </build>
</target>
```

Dann starten wir die Anwendung – vergessen dabei aber, `mod.jar` in den Classpath aufzunehmen:

```
<target name="run" depends="build-appl">
  <java classname="jj.appl.Application" fork="true">
    <classpath location="${basedir}/build/appl.jar" />
  </java>
</target>
```

Die Ausgaben:

```
main starts...
Exception in thread "main" java.lang.NoClassDefFoundError: jj/mod/Foo
    at jj.appl.Application.main(Application.java:8)
```

Man beachte die Ausgabe von `main starts.`

Schöner wäre es, wenn ein solches System überhaupt erst nicht starten würde...

## 3 Das Java Platform Module System (JPMS)

Zu Beginn ein Zitat von Joshua Bloch (eine kleine Warnung?):

*It is too early to say whether modules will achieve widespread use outside of the JDK itself. In the meantime, it seems best to avoid them unless you have a compelling need.*

### Module und modulare jars

Eine `jar`-Datei war bislang eine "wahllose" Zusammenfassung von `class`-Dateien und Ressourcen. Klassen konnten im Prinzip willkürlich zu `jar`-Dateien zusammengefasst werden – mit der einzigen Einschränkung, dass keine bidirektionale Abhängigkeit solcher Dateien erlaubt war. Die "Physik" solcher `jar`-Dateien hatte keine "logische" Grundlage.

Java-9 führt nun sog. modulare `jar`-Dateien ein. Jede (physische) modulare `jar` repräsentiert genau ein (logisches) "Modul".

### module-info

Ein Modul wird beschrieben in einer eigenen `module-info.java`-Datei. Die Datei hat eine eigene Syntax – mit Schlüsselwörtern wie `module`, `exports`, `requires` etc. Diese Datei wird vom Java-Compiler kompiliert – das Resultat ist eine `module-info.class`-Datei. Diese muss im Wurzelverzeichnis der `jar`-Datei angesiedelt sein. Ein Modul ist somit selbstbeschreibend. Solche Module heißen "Named Application Modules".

In der `module-info` wird zunächst ein logischer Name für das Modul vereinbart. Dieser Name unterscheidet sich i.d.R. vom Namen der `jar`-Datei. Es sollten reverse-domain-pattern-Namen verwendet werden (wie auch bei packages). Man kann z.B. den Namen des "Wurzel-Pakets" verwenden. Aber im Prinzip kann ein beliebiger Name verwendet werden.

Via `exports` können in der `module-info` diejenigen in der `jar`-Datei enthaltenen Packages angegeben werden, die von anderen Modulen genutzt werden können.

Via `requires` kann ein Modul angeben, von welchen anderen Modulen es abhängig ist. (Man beachte: via `exports` werden Packages(!) "exportiert", via `requires` werden Module(!) "importiert".

Neben diesen fundamentalen Modul-Abhängigkeiten können weitere Eigenschaften beschrieben werden. Welche Pakete eines Moduls sollen z.B. der Deep Reflection (Reflection-basierter Zugriff auf private Elemente) zugänglich sein? Etc.

## modulepath

Der alte `classpath` wird in rein modularen Systemen durch den neuen "`modulepath`" ersetzt. Der `modulepath` wird (wie auch der alte `classpath`) sowohl vom Compiler als auch von der VM genutzt.

Im Gegensatz zum `classpath` spielt die Reihenfolge im `modulepath` keine Rolle. Zudem ist sichergestellt, dass ein Modul nur einmal im `modulepath` enthalten ist.

Es ist ausgeschlossen, dass Klassen ein und desselben Packages auf verschiedene Module verteilt sind.

Bei Start einer Anwendung kann sichergestellt werden, dass für alle Typen, die in der Anwendung angesprochen werden, die entsprechende Module tatsächlich existieren.

## Automatic Modules

Neben den Named Application Modules (die in einer `module-info` beschrieben sind), können auch "alte" `jar`-Dateien als Module verwendet werden.

Sie exportieren implizit alles und können alle anderen Module nutzen

Für ein automatische Modul wird automatisch ein Name generiert: Heißt die `jar`-Datei z.B. `x-y-z-1.0.0.jar`, dann wird daraus der Modulname `x.y.z` abgeleitet - und das Modul kann dann auch unter diesem Namen angesprochen werden. (Allerdings kann ein solcher Name auch in der `manifest.mf` definiert werden.)

## Das namenlose Modul

`modulepath` und `classpath` können koexistieren. Über den `modulepath` werden Module herangezogen (Named Application Modules oder Automatic Modules). Über den `classpath` können weitere `jars` herangezogen werden.

Alle `jars` (egal, ob modular oder nicht), die im `classpath` liegen, bilden zusammen das namenlose Modul. Es exportiert implizit alles und kann alle andere Module nutzen

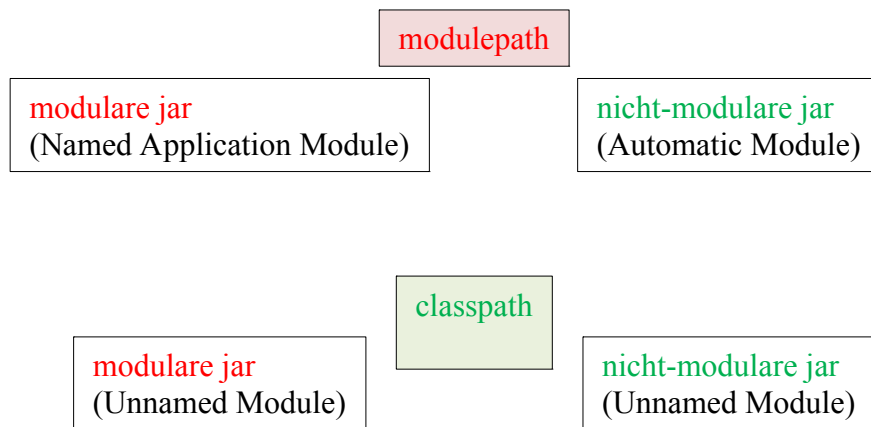
## Named Application Module – Automatic Module – Unnamed Module

Eine Übersicht:

Sofern eine modulare jar (eine jar mit einer `module-info`) im `modulepath` liegt, fungiert sie als "Named Application Module".

Sofern eine nicht-modulare jar (eine jar ohne `module-info`) im `modulepath` liegt, fungiert sie als "Automatic Module".

Sofern eine jar (egal, ob modular oder nicht) über den `classpath` herangezogen wird, ist sie Teil des "Unnamed Module".



## Platform-Module

Auch das JDK (die `rt.jar`) selbst ist aufgesplittet worden in Module: `java.base`, `java.xml` etc. Von `java.base` sind implizit alle anderen Module abhängig.

## Ziele

Zunächst einmal ging's Oracle darum, die Standardbibliotheken vernünftig zu modularisieren. Eine Anwendung sollte nicht mehr von großen `rt.jar` abhängig sein, sondern nurmehr von denjenigen Modulen, deren Typen sie auch tatsächlich nutzt. es sollten schlanke Runtime-Images erzeugt und ausgeliefert werden können.

Die Konfiguration von Systemen sollte zuverlässiger werden (die im letzten Kapitel beschrieben `classpath`-Probleme sollten vermieden werden).

Es sollte ein zusätzliches, höheres Kapselung-Konzept eingeführt werden – die Kapselung via Klassen und via Packages sollte ergänzt werden durch eine Kapselung auf Modul-Ebene.

## Unterschiede zu OSGi

Auch das bewährte OSGi-Framework implementiert ein Modul-Konzept – aber auf ganz andere Weise als Java-9.

Module werden dort als Bundles bezeichnet. Die Export-Import-Beziehungen werden dort in der `MANIFEST.MF` einer jar beschrieben. Im Unterschied zu Java-9 werden (i.d.R.) Packages importiert – und nicht Module.

Die modulare Kapselung wird in OSGi erst zur Laufzeit garantiert – bei Java-9 wird sie bereits bei der Kompilierung (und natürlich auch zur Laufzeit von der VM) garantiert. Die Kapselung wird in OSGi durch den Classloading-Mechanismus sichergestellt – jedes Modul hat seinen eigenen `ClassLoader`. In Java-9 wird für alle Module derselbe `ClassLoader` verwendet.

## Inhalte

- Im ersten Abschnitt wird gezeigt, wie ein Modul definiert wird und wie in der `module-info` Export- und Import-Beziehungen festgelegt werden können (`export`, `require`).
- Im Abschnitt 2 wird eine einfache praktische Nutzenanwendung von `export` und `requires` vorgestellt.
- Der Abschnitt 3 zeigt, wie Pakete mittels `opens` für Deep-Reflection geöffnet werden können.
- Im Abschnitt 4 wird eine kleine praktischen Nutzenanwendung von `opens` gezeigt.
- Im Abschnitt 5 geht's um direkte und indirekte Abhängigkeiten – und um den Unterschied zwischen dem Kompilations- und dem Runtime-Module-Path..
- Im Abschnitt 6 geht's um "Automatische Module".
- Der Abschnitt 7 zeigt, dass die Klassen eines Packages nun nicht mehr über mehrere Module verstreut werden können.
- Abschnitt 8 zeigt, wie ein Modul ein Paket an ganz bestimmte andere (und nur an diese anderen) exportieren kann (`exports to`).
- Im Abschnitt 9 wird gezeigt, wie ein von einem Modul A gefordertes Modul B in einem das Modul A nutzenden dritten Modul C genutzt werden kann, ohne es (das Modul B) noch einmal anfordern zu müssen (`requires transitive`).
- Im Abschnitt 10 geht's um Aggregations-Module. Solche Module verlangen eine Reihe weiterer Module – die dann allesamt via `requires transitive` anderen Modulen in kompakter Form (in der Form des Aggregats) zur Verfügung stehen.
- Im Abschnitt 11 wird das `ServiceLoader`-Konzept vorgestellt – wobei hier "alte" Mechanismen verwendet werden.
- Im Abschnitt 12 geht's um neuen Mechanismen des `ServiceLoaderS` (`provides with`).
- Im Abschnitt 13 geht's darum, wie "alte" Komponenten und "neue" Module koexistieren können: wie modulare und nicht-modulare Elementen gemischt werden können – und um die Benutzung von `classpath` und `modulepath`.
- In den letzten 4 Abschnitten stellen wird Mechanismen vor, die von dem Modulkonzept zur Verfügung gestellt werden, um eben dieses Modulkonzept

teilweise zu "umgehen": `--add-exports`, `--add-opens`, `--add-modules`.  
(Es existieren noch weitere Mittel zur "Feinjustierung" wie `--add-reads` und `-patch-modules`, die hier aber nicht weiter vorgestellt werden...)

Im folgenden verwenden wir u.a. folgende Utility-Klasse (implementiert im `shared-Projekt`):

```
package jj.util.trycatch;

public class TryCatch {
    @FunctionalInterface
    public interface Action {
        public abstract void execute() throws Throwable;
    }
    public static void run(Action action) {
        try {
            action.execute();
        }
        catch(RuntimeException e) {
            throw e;
        }
        catch(Throwable t) {
            t.printStackTrace(System.out);
        }
    }
}
```

Der statischen `run`-Methode der `TryCatch`-Klasse kann eine `Action` übergeben werden, deren `execute`-Methode eine checked Exception werfen darf. Diese wird von `run` in eine `RuntimeException` eingebunden, die dann weitergeworfen wird. Wir ersparen uns somit den lästigen `try...catch...`

Ein Beispiel:

```
TryCatch.run(() -> Thread.sleep(1000));
```

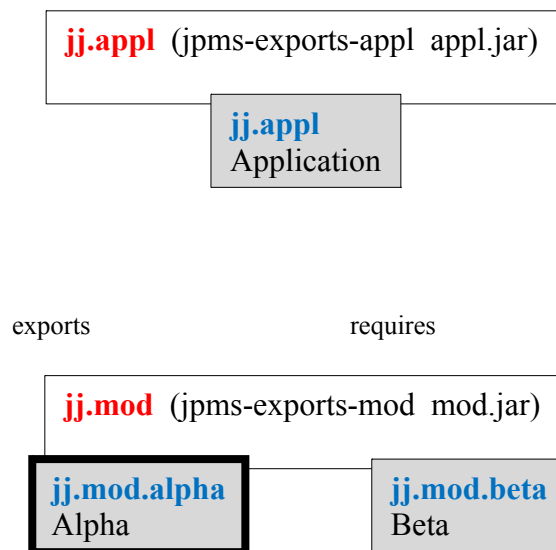
Die von `Thread.sleep` möglicherweise geworfene `InterruptedException` (eine checked-Exception) wird in eine `RuntimeException` eingebunden und als solche weitergeworfen.

## 3.1 Exports / Requires

Ein Modul `jj.mod` enthält zwei Pakete: `jj.mod.alpha` und `jj.mod.beta`. Ein Client soll nur die Typen des Paket `jj.mod.alpha` nutzen können. In der `module`-Datei wird daher nur dieses Paket mittels `exports` zur Benutzung freigegeben.

Die `module`-Datei des Clients muss via `requires` das Modul `jj.mod` zur Benutzung anfordern.

Man beachte: `exports` bezieht sich auf ein Paket – `requires` auf ein Modul.



### mod.jar

`jj.mod` exportiert `jj.mod.alpha`:

```
module jj.mod {
    exports jj.mod.alpha;
}
```

Das `jj.mod.beta`-Paket enthält eine öffentliche Klasse `Beta` (auf welche alle anderen Klassen des `jj.mod`-Moduls zugreifen können). `Beta` hat eine öffentliche und eine private Methode (in anderen Paketen des `jj.mod`-Moduls kann dann natürlich nur die öffentliche Methode genutzt werden):

```
package jj.mod.beta;
```



```
public class Beta {  
    public static void pub() { }  
    private static void pri() { }  
}
```

`jj.mod.alpha.Alpha` kann die öffentliche `Beta`-Methode nutzen (aber natürlich nicht die private):

```
package jj.mod.alpha;  
  
import jj.mod.beta.Beta;  
  
public class Alpha {  
    public static void pub() {  
        Beta.pub();  
        // Beta.pri(); // illegal  
    }  
    private static void pri() { }  
}
```

## appl.jar

`jj.appl` fordert `jj.mod` zur Benutzung an (und zusätzlich auch das Helper-Modell `jj.util` – welches im `shared`-Projekt implementiert ist und die `TryCatch`-Klasse enthält):

```
module jj.appl {  
    requires jj.mod;  
    requires jj.util;  
}
```

Eine Klasse des `jj.appl`-Moduls kann nun die Typen des Pakets `jj.mod.alpha` nutzen – nicht aber die Typen von `jj.mod.beta`:

```
package jj.appl;  
  
import jj.util.log.Log;  
import jj.mod.alpha.Alpha;  
// import jj.mod.beta.Beta; // illegal  
  
public class Application {  
    // Aufruf der folgenden demo-Methoden...  
}
```

Im `jj.appl`-Modul kann nun die Klasse `Alpha` und ihre öffentliche Methode (`pub`) genutzt werden (aber aber natürlich nicht die private `pri`-Methode).

Die `pub`-Methode kann auch via Reflection aufgerufen werden (auch diese Möglichkeit setzt den `requires`-Eintrag `voarus`). Die private `pri`-Methode kann via Reflection allerdings nicht(!) aufgerufen werden:

```
static void demoAlpha() {  
  
    Alpha.pub();  
  
    // Alpha.pri(); // illegal  
  
    TryCatch.run(() -> Class.forName("jj.mod.alpha.Alpha")  
        .getMethod("pub").invoke(null));  
  
    TryCatch.run(() -> {  
        Method m = Class.forName("jj.mod.alpha.Alpha")  
            .getDeclaredMethod("pri");  
        m.setAccessible(true); //  
InaccessibleObjectException  
        m.invoke(null);  
    });  
}
```

Wir unterscheiden im Folgenden "Public-Reflection" von "Deep-Reflection". Mittels Public-Reflection kann nur auf nicht-private Elemente zugegriffen werden; mittels Deep-Reflection können wir auch auf private Elemente zugreifen (vorher muss natürlich `setAccessible` aufgerufen werden!).

Die Klasse `jj.mod.beta.Beta` kann in `jj.appl` als Bezeichner nicht verwendet werden. Das `Class`-Objekt und die `Method`-Objekte dieser Klasse können via Reflection ermittelt werden – aber mit diesem `Class`-Objekt kann man nichts Gescheites anstellen (also z.B. nicht die `Beta`-eigene öffentliche `pub`-Methode aufrufen):

```
static void demoBeta() {  
    // Beta.pub() // illegal  
  
    TryCatch.run(() -> {  
        Method m = Class.forName("jj.mod.beta.Beta")  
            .getDeclaredMethod("pub");  
        System.out.println(m); // ok  
        m.invoke(null); // IllegalAccessException (missing  
exports...)  
    });  
}
```

Hier die Targets der `build.xml`:

```
<target name="build-mod">
    <build name="mod">
        <paths>
        </paths>
    </build>
</target>

<target name="build-appl" depends="build-mod">
    <build name="appl">
        <paths>
            <modulepath location="${shared}/build/util.jar" />
        >
            <modulepath location="${basedir}/build/mod.jar" />
        >
        </paths>
    </build>
</target>

<target name="run" depends="build-appl">
    <java module="jj.appl" classname="jj.appl.Application"
fork="true">
        <modulepath location="${shared}/build/util.jar" />
        <modulepath location="${basedir}/build/mod.jar" />
        <modulepath location="${basedir}/build/appl.jar" />
    </java>
</target>
```

Wie kann das System mit Console-Befehlen gebaut und gestartet werden?

Sie hierzu das `"raw"`-Target der `build.xml`. Dieses Target benutzt ausschließlich `<exec>`-Statements – Statements also, aus denen genau hervorgeht, welche Tools mit welchen Parametern aufgerufen werden.

Hier die einzelnen Statements:

```
mkdir ${basedir}-mod/tmp
mkdir ${basedir}-appl/tmp

javac -d ${basedir}-mod/tmp
```

```
    ${basedir}-mod/src/module-info.java
    ${basedir}-mod/src/jj/mod/alpha/Alpha.java
    ${basedir}-mod/src/jj/mod/beta/Beta.java

jar --create -file ${basedir}/build/mod.jar
    -C ${basedir}-mod/tmp .

javac --module-path
    ${basedir}/build/mod.jar;${shared}/build/util.jar
    -d ${basedir}-appl/tmp
    ${basedir}-appl/src/module-info.java
    ${basedir}-appl/src/jj/appl/Application.java

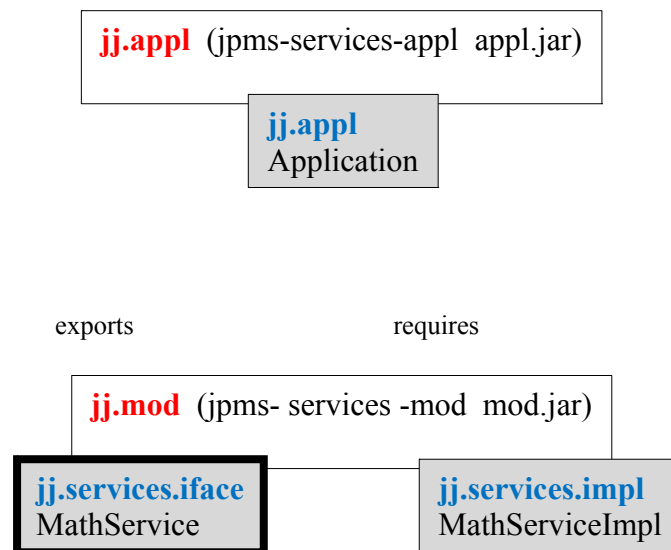
jar --create --file ${basedir}/build/appl.jar
    -C ${basedir}-appl/tmp .

java --module-path ${basedir}/build/appl.jar;
    ${basedir}/build/mod.jar;${shared}/build/util.jar
    --module jj.appl/jj.appl.Application
```

## 3.2 Beispiel: Service-Interfaces und –Implementierungen

Ein "praktisches" Beispiel:

Ein Modul enthält sowohl das API (ein Interface) als auch eine Implementierung. Interface und Implementierung existieren in verschiedenen Paketen (`jj.services.iface` resp. `jj.services.impl`). Ein Client soll nur das Interface benutzen können:



### mod.jar

```
module jj.services {
    exports jj.services.iface;
}
```

```
package jj.services.iface;

import jj.services.impl.MathServiceImpl;

public interface MathService {

    public abstract int sum(int x, int y);
    public abstract int diff(int x, int y);

    public final static MathService instance = new
    MathServiceImpl();
}
```

```
}
```

```
package jj.services.impl;  
  
import jj.services.iface.MathService;  
  
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;  
    }  
}
```

## appl.jar

```
module jj.appl {  
    requires jj.util;  
    requires jj.services;  
}
```

```
package jj.appl;  
  
import jj.services.iface.MathService;  
import jj.util.log.Log;  
  
public class Application {  
  
    public static void main(String[] args) {  
        // ...  
    }  
  
    static void demoServices() {  
        MathService mathService = MathService.instance;  
        System.out.println(mathService.sum(40, 2));  
        System.out.println(mathService.diff(80, 3));  
    }  
}
```

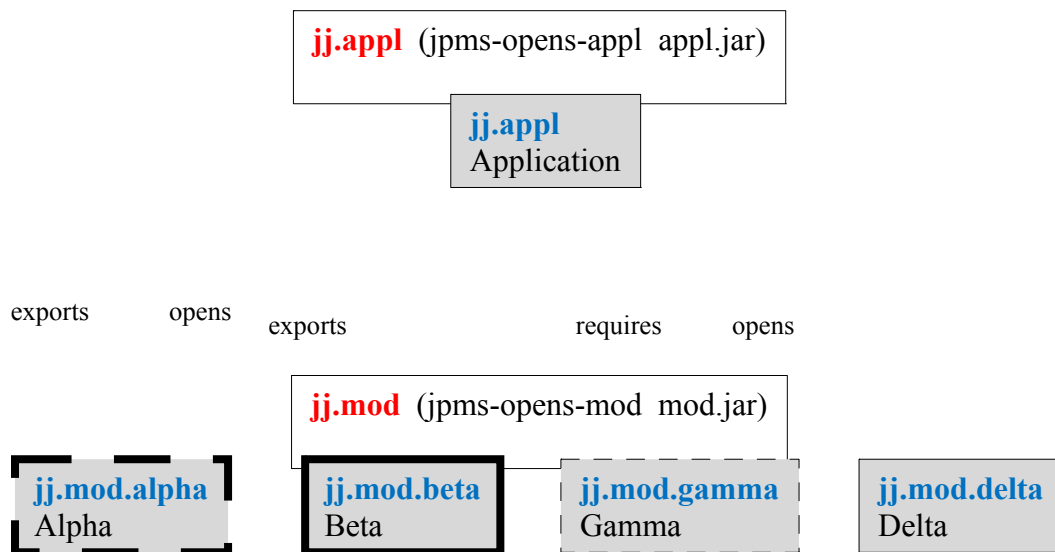


### 3.3 Opens

Mittels `opens` kann ein Modul für den Zugriff via Deep Reflection geöffnet werden (also für den Reflection-basierten Zugriff auf private Elemente).

Das folgende `jj.mod`-Modul enthält vier Pakete. Das erste Paket wird sowohl exportiert (via `exports`) und für Reflection geöffnet (via `opens`). Das zweite Paket wird nur exportiert. Das dritte Paket wird zwar für Reflection geöffnet, aber nicht exportiert. Und das letzte Paket wird weder geöffnet noch exportiert.

Das Modul `jj.appl` fordert `jj.mod` zur Benutzung an (via `requires`):



#### mod.jar

```
module jj.mod {  
  
    exports jj.mod.alpha;  
    opens jj.mod.alpha;  
  
    exports jj.mod.beta;  
  
    opens jj.mod.gamma;  
}
```



Alle Klassen dieses Moduls (Alpha, Beta, Gamma und Delta) enthalten jeweils eine öffentliche Methode `pub` und eine private Methode `pri`:

```
package jj.mod.alpha;  
  
public class Alpha {  
    public static void pub() { }  
    private static void pri() { }  
}
```

```
package jj.mod.beta;  
  
public class Beta {  
    public static void pub() { }  
    private static void pri() { }  
}
```

```
package jj.mod.gamma;  
  
public class Gamma {  
    public static void pub() { }  
    private static void pri() { }  
}
```

```
package jj.mod.delta;  
  
public class Delta {  
    public static void pub() { }  
    private static void pri() { }  
}
```

Das `jj.appl`-Modul fordert `jj.mod` zur Benutzung an:

### **appl.jar**

```
module jj.appl {  
    requires jj.mod;  
}
```

```
package jj.appl;  
  
import java.lang.reflect.Method;  
  
import jj.mod.alpha.Alpha;
```

```
import jj.mod.beta.Beta;
// import jj.mod.delta.Gamma; // illegal
import jj.util.trycatch.TryCatch;
import jj.util.log.Log;

public class Application {
    // Aufruf der folgenden demo-Methoden...
}
```

Eine Klasse von `jj.appl` kann zunächst einmal die Klasse `jj.mod.alpha.Alpha` nutzen – und deren öffentliche Methode aufrufen. Via Reflection kann natürlich die `pub`-Methode ermittelt und aufgerufen werden – da `jj.mod` aber `jj.mod.alpha.Alpha` auch geöffnet hat, kann auch die private `pri`-Methode via Reflection genutzt werden:

```
static void demoAlpha() {
    Alpha.pub();
    // alpha.pri(); // illegal
    TryCatch.run(() -> {
        final Method m =
Alpha.class.getDeclaredMethod("pub");
        m.invoke(null);
    });
    TryCatch.run(() -> {
        final Method m =
Alpha.class.getDeclaredMethod("pri");
        m.setAccessible(true);
        m.invoke(null);
    });
}
```

Die Klasse `jj.mod.beta.Beta` kann "normal" genutzt werden; auch der Reflection-basierte Zugriff auf öffentliche Elemente ist möglich. Deep Reflection aber funktioniert nicht - weil nämlich `jj.mod` das Paket `jj.mod.beta` nur exportiert, nicht aber geöffnet hat:

```
static void demoBeta() {
    Beta.pub();
    // Beta.pri(); // illegal
    TryCatch.run(() -> {
        final Method m = Beta.class.getDeclaredMethod("pub");
        m.invoke(null);
    });
    TryCatch.run(() -> {
        final Method m = Beta.class.getDeclaredMethod("pri");
        m.setAccessible(true); // throw an
IllegalAccessException
    });
}
```

```
        m.invoke(null);
    });
}
```

Da das Paket `jj.mod.gamma` von `jj.mod` nicht exportiert wurde, ist der normale Zugriff auf `Gamma` nicht erlaubt. Da `jj.mod.gamma` aber geöffnet wurde, ist sowohl der Reflection- als auch der Deep-Reflection-basierte Zugriff auf `Gamma` möglich:

```
static void demoGamma() {
    // Gamma.pub(); // illegal
    TryCatch.run(() -> {
        final Class<?> cls =
Class.forName("jj.mod.gamma.Gamma");
        TryCatch.run(() -> {
            final Method m = cls.getDeclaredMethod("pub");
            m.invoke(null);
        });
        TryCatch.run(() -> {
            final Method m = cls.getDeclaredMethod("pri");
            m.setAccessible(true);
            m.invoke(null);
        });
    });
}
```

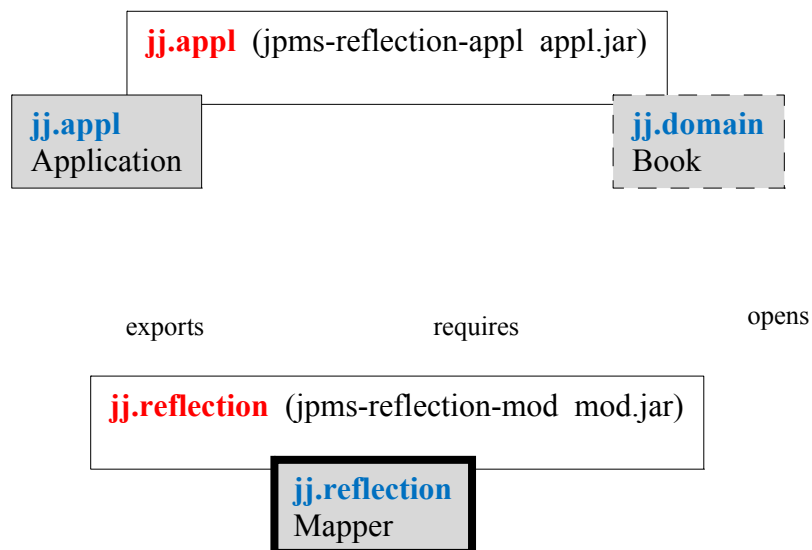
Auf `jj.mod.delta.Delta` kann im `jj.appl`-Modul überhaupt nicht zugegriffen werden (weil `jj.mod.delta` weder exportiert noch geöffnet wurde):

```
static void demoDelta() {
    // Delta.pub(); // illegal
    TryCatch.run(() -> {
        final Class<?> cls =
Class.forName("jj.mod.delta.Delta");
        final Object obj =
cls.getConstructor().newInstance();
    });
}
```

### 3.4 Beispiel: Deep Reflection

Wann wird `opens` praktisch relevant? Viele Frameworks und Bibliotheken nutzen Reflection, um auf die Eigenschaften von Objekten der Anwendung zuzugreifen. JPA (Hibernate) z.B. kann mittels Reflection die Werte von privaten Attributen eines Objekts auslesen bzw. setzen. Auch Spring macht regen Gebrauch von Deep-Reflection.

Im Folgenden Beispiel wird ein allgemein verwendbarer `Mapper` entwickelt, welchem ein Objekt einer Applications-spezifischen Klasse übergeben wird (einer `Book`-Klasse). Der `Mapper` wird die Attribute dieses Objekts via Deep-Reflection auslesen und für jedes Attribut einen Eintrag in der `Map` erzeugen (wobei der Name des Attributs als Schlüssel des Eintrags genutzt wird)



Die `Mapper`-Klasse ist in dem Modul `jj.reflection` implementiert. Das gleichnamig Paket wird exportiert:

#### mod.jar

```
module jj.reflection {  
    exports jj.reflection;  
}
```

Der `map`-Methode des Mappers wird ein `Object` übergeben. Sie iteriert über die `Field`-Objekte der Klasse dieses Objekts. Jedes dieser `Field`-Objekte wird genutzt, um den Wert des von dem `Field`-Objekt beschriebenen Attributs auszulesen (welches i.d.R.

natürlich privat ist). Die somit ermittelten Werte werden dann in die `Map` eingetragen, wobei der Name des Attributs als Schlüssel genutzt wird. Und diese `Map` wird als Resultat zurückgeliefert:

```
package jj.reflection;  
  
import java.lang.reflect.Field;  
import java.util.HashMap;  
import java.util.Map;  
  
public class Mapper {  
    public static Map<String, Object> map(Object obj) {  
        final Map<String, Object> map = new HashMap<>();  
        try {  
            for (final Field field :  
obj.getClass().getDeclaredFields()) {  
                field.setAccessible(true);  
                final Object value = field.get(obj);  
                map.put(field.getName(), value);  
            }  
        }  
        catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
        return map;  
    }  
}
```

Das `jj.appl`-Modul fordert `jj.reflection` an und öffnet `jj.domain` (in `jj.domain` ist die `Book`-Klasse angesiedelt, deren Objekte vom `Mapper` initialisiert werden sollen):

### **appl.jar**

```
module jj.appl {  
    requires jj.reflection;  
    opens jj.domain;  
}
```

Die Klasse `Book` enthält vier private Attribute:

```
package jj.domain;  
  
public class Book {  
    private String isbn;
```

```
private String title;  
private int year;  
private String author;  
  
public Book(String isbn, String title, int year, String  
author) { ... }  
  
// getter, setter...  
  
@Override  
public String toString() {  
    return this.getClass().getSimpleName() +  
        " [" + isbn + ", " + title + ", " + year + ", " +  
author + "];"  
}  
}
```

Die Application erzeugt ein `Book`-Objekt und übergibt es an die `map`-Methode des `Mapper`s. Die von `map` zurückgelieferte `Map` wird ausgegeben:

```
package jj.app1;  
  
import jj.domain.Book;  
import jj.reflection.Mapper;  
  
public class Application {  
  
    public static void main(String[] args) {  
        Book book = new Book("1111", "Pascal", 1970, "N. Wirth");  
        Mapper.map(book).forEach(  
            (name, value) -> System.out.println(name + " = " +  
value));  
    }  
}
```

Die Ausgaben:

```
year = 1970  
author = N. Wirth  
isbn = 1111  
title = Pascal
```

### 3.5 Direkte / Indirekte Nutzung von Modulen

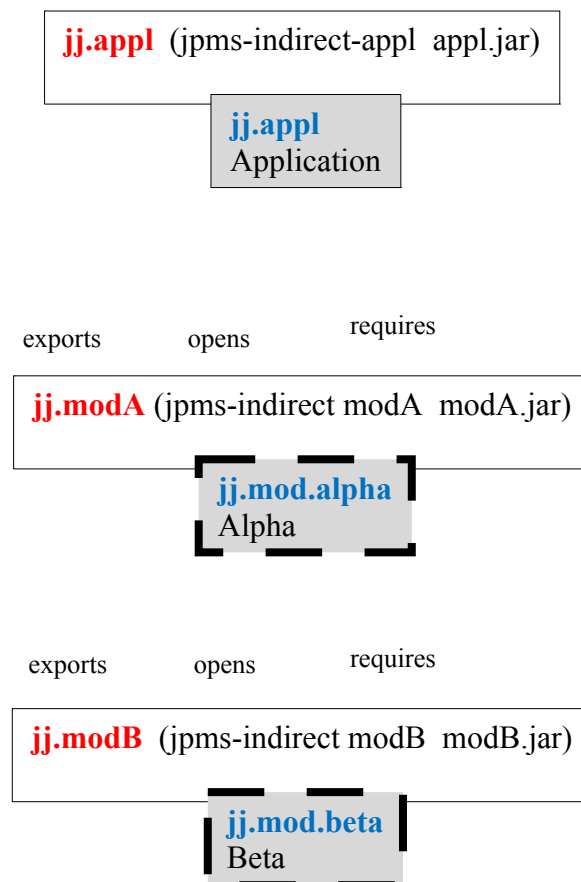
Natürlich kann ein Modul (`jj.appl`) ein anderes Modul (`jj.modA`) nutzen, welches seinerseits wiederum ein weiteres Modul (`jj.modB`) nutzt.

`jj.modB` exportiert und öffnet das Paket `jj.modB.beta`.

`jj.modA` fordert `jj.modB` an – und exportiert und öffnet das Paket `jj.modA.alpha`.

`jj.appl` fordert `jj.modA` an (nicht aber `jj.modB`).

Wer kann worauf und wie zugreifen?



**modB**

```
module jj.modB {  
    exports jj.modB.beta;  
    opens  jj.modB.beta;  
}  
  
package jj.modB.beta;  
  
public class Beta {  
    public static void pub() { }  
    private static void pri() { }  
}
```

**modA**

```
module jj.modA {  
    requires jj.modB;  
    requires jj.util;  
    exports jj.modA.alpha;  
    opens  jj.modA.alpha;  
}
```

Weil `jj.modB` das Paket `jj.modB.beta` exportiert und `jj.modA` das Modul `jj.modB` anfordert, kann `Beta` als Bezeichner verwendet werden (über den die `pub`-Methode aufgerufen werden kann) . Deshalb kann `pub` auch via public-Reflection aufgerufen werden. Und da `jj.modB` dasselbe Paket auch geöffnet hat, ist via Deep-Reflection auch der Zugriff auf die private `pri`-Methode möglich:

```
package jj.modA.alpha;  
  
import java.lang.reflect.Method;  
import jj.modB.beta.Beta;  
import jj.util.trycatch.TryCatch;  
  
public class Alpha {  
    public static void pub() {  
  
        Beta.pub();  
  
        TryCatch.run(() -> {  
            Method method = Class.forName("jj.modB.beta.Beta")  
                                .getDeclaredMethod("pri");  
            method.setAccessible(true);  
            method.invoke(null);  
        });  
    }  
}
```



```
        TryCatch.run(() -> {
            Method method = Class.forName("jj.modB.beta.Beta")
                               .getDeclaredMethod("pri");
            method.setAccessible(true);
            method.invoke(null);
        });
    }
    private static void pri() { }
```

## appl

Das Modul `jj.appl` verlangt nur die Benutzung von `jj.modA`:

```
module jj.appl {
    requires jj.util;
    requires jj.modA;
}
```

```
package jj.appl;

import java.lang.reflect.Method;

import jj.modA.alpha.Alpha;
import jj.util.log.Log;
import jj.util.trycatch.TryCatch;

public class Application {

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

Da `jj.modA` das Paket `jj.modA.alpha` exportiert hat, kann `Alpha` als Bezeichner verwendet werden, über den die `pub`-Methode aufgerufen werden kann. Und deshalb kann diese `pub`-Methode auch via `public-Reflection` aufgerufen werden. Und da `jj.modA` das Paket `jj.modA.alpha` geöffnet hat, können wir via `Deep-Reflection` zudem auch die `pri`-Methode von `Alpha` aufrufen:

```
static void demoAlpha() {

    Alpha.pub();

    TryCatch.run(() -> {
```

```
        final Class<?> cls =
Class.forName("jj.modA.alpha.Alpha");
        final Method m = cls.getDeclaredMethod("pub");
        m.invoke(null);
    });

    TryCatch.run(() -> {
        final Class<?> cls =
Class.forName("jj.modA.alpha.Alpha");
        final Method m = cls.getDeclaredMethod("pri");
        m.setAccessible(true);
        m.invoke(null);
    });
}
```

Auf `Beta` kann mit normalen Java-Mitteln nicht zugegriffen werden (weil `jj.modB` nicht angefordert wird). Weil aber `jj.modB` das Paket `jj.modB.beta` geöffnet hat, kann via public-Reflection die `pub`-Methode von `Beta` und via Deep-Reflection die `pri`-Methode von `Beta` aufgerufen werden:

```
static void demoBeta() {

    // Beta.pub(); // illegal

    TryCatch.run(() -> {
        final Class<?> cls =
Class.forName("jj.modB.beta.Beta");
        final Method m = cls.getDeclaredMethod("pub");
        m.invoke(null);
    });

    TryCatch.run(() -> {
        final Class<?> cls =
Class.forName("jj.modB.beta.Beta");
        final Method m = cls.getDeclaredMethod("pri");
        m.setAccessible(true);
        m.invoke(null);
    });
}
```

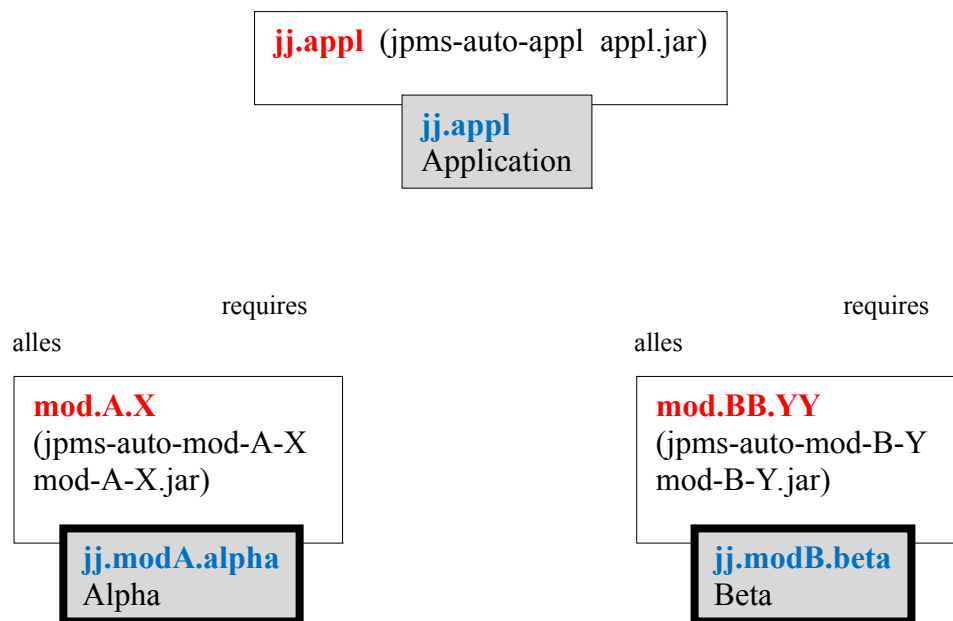
## 3.6 Automatic Modules

Im Folgenden verwenden wir zwei nicht-modulare jar-Datei (also jar-Dateien ohne module-Datei): `mod-A-X.jar` und `mod-B-Y.jar`. Die `mod-B-Y.jar` enthält allerdings eine `manifest.mf` mit folgendem Eintrag:

```
Automatic-Module-Name: mod.BB.YY
```

Beide jar-Dateien werden über den Module-Path herangezogen und vom `jj.appl`-Modul via `requires` angefordert.

Die beiden jar-Dateien werden somit zu automatischen Modulen. Der Name des Moduls ist im ersten Falle `mod.A.X` (wird also vom Namen der jar-Datei abgeleitet) und im zweiten Falle `mod.BB.YY` (wird der `manifest.mf` entnommen).



### mod-A-X

```
package jj.modA.alpha;

public class Alpha {
    public static void pub() { }
    private static void pri() { }
}
```

## mod-BB-YY

```
package jj.modB.beta;

public class Beta {
    public static void pub() { }
    public static void pri() { }
}
```

```
This-is-a-comment: META-INF/MANIFEST.MF
Automatic-Module-Name: mod.BB.YY
```

## appl

Ein Modul, welches die beiden automatischen Module nutzt, kann automatisch alle öffentlichen Elemente dieser Module nutzen:

```
module jj.appl {
    requires mod.A.X;
    // Warning: Name of automatic module 'mod.A.X' is
    unstable,
    // it is derived from the module's file name
    requires mod.BB.YY;
}
```

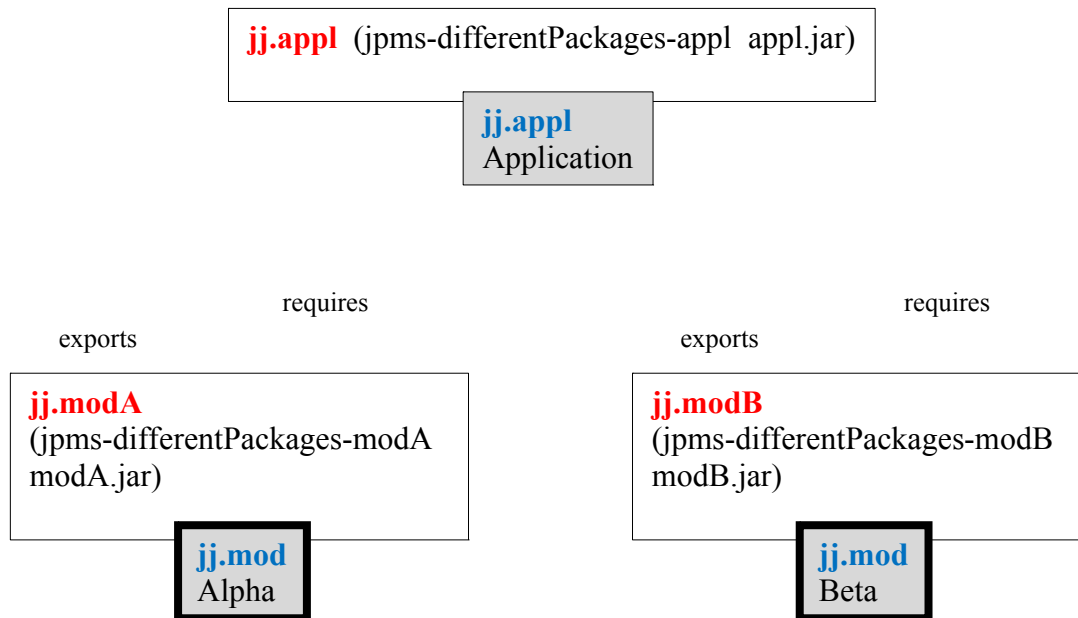
```
package jj.appl;

import jj.modA.alpha.Alpha;
import jj.modB.beta.Beta;

public class Application {
    public static void main(String[] args) {
        Alpha.pub();
        Beta.pub();
    }
}
```

### 3.7 Verbot Modul-übergreifender Pakete

Eine Applikation kann keine zwei Module nutzen, die gleichnamige Pakete enthalten:



Die Module `jj.modA` und `jj.modB` enthalten beide ein Paket namens `jj.mod`:

#### **modA**

```
module jj.modA {
    exports jj.mod;
}

package jj.mod;

public class Alpha { }
```

#### **modB**

```
module jj.modB {
    exports jj.mod;
}
```

```
package jj.mod;  
  
public class Beta { }
```

## **appl**

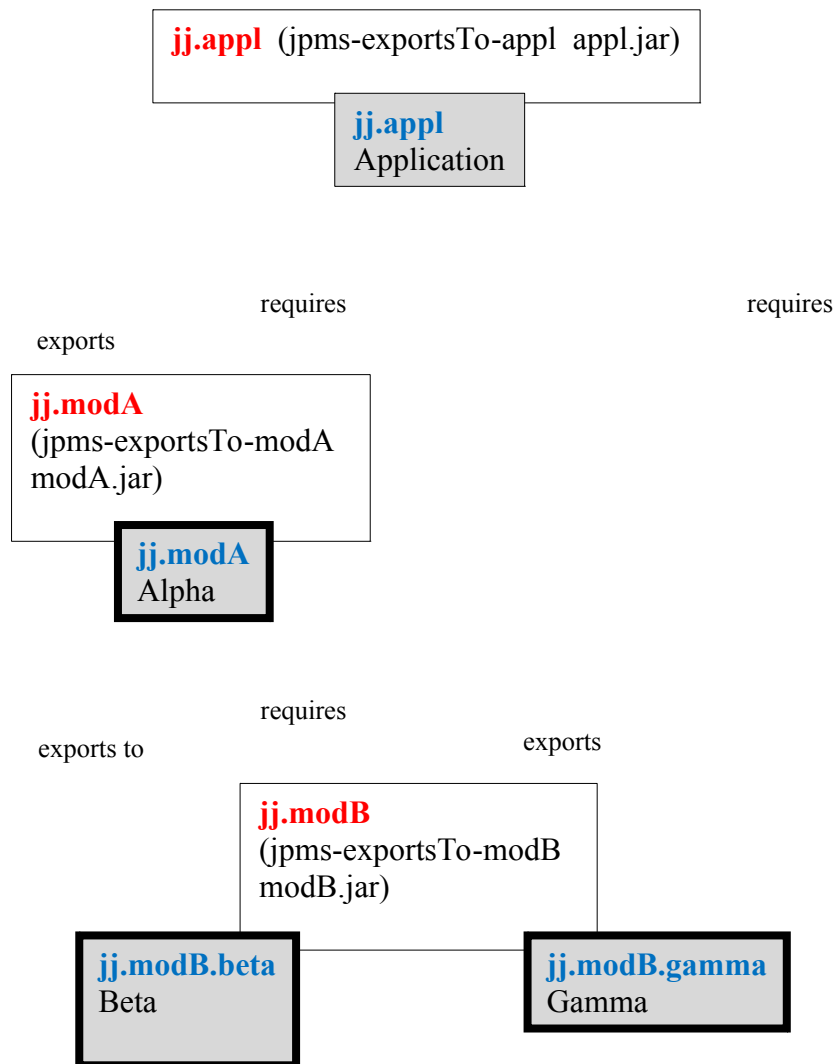
Das Modul `jj.appl` wird nicht übersetzt. Der Compiler beschwert sich beim Versuch, die module-Datei von `jj.appl` zu übersetzen:

```
module jj.appl {  
    requires jj.modA;  
    // The package jj.mod is accessible from more than one  
module:  
    // jj.modA, jj.modB  
    requires jj.modB;  
    // The package jj.mod is accessible from more than one  
module:  
    // jj.modA, jj.modB  
}
```

## 3.8 Exports To

Der Export kann auf bestimmte namentlich angegebenen Klienten eingeschränkt werden.

Das Modul `jj.modB` erlaubt die Benutzung des `jj.modB.beta`-Pakets nur dem Modul `jj.modA` (das Paket `jj.modB.gamma` wird aber zur allgemeinen Benutzung freigegeben):



### modB

```
module jj.modB {
    exports jj.modB.beta to jj.modA;
```

```
    exports jj.modB.gamma;  
}
```

```
package jj.modB.beta;
```

```
public class Beta { }
```

```
package jj.modB.gamma;
```

```
public class Gamma { }
```

## modA

```
module jj.modA {  
    requires jj.modB;  
    exports jj.modA.alpha;  
}
```

```
package jj.modA.alpha;
```

```
import jj.modB.beta.Beta;
```

```
import jj.modB.gamma.Gamma;
```

```
public class Alpha { }
```

## appl

```
module jj.appl {  
    requires jj.modA;  
    requires jj.modB;  
}
```

```
package jj.appl;
```

```
import jj.modA.alpha.Alpha;
```

```
// import jj.modB.beta.Beta; // not accessible
```

```
import jj.modB.gamma.Gamma;
```

```
public class Application {
```

```
    public static void main(String[] args) {  
        Alpha alpha;  
        Gamma gamma;  
    }  
}
```



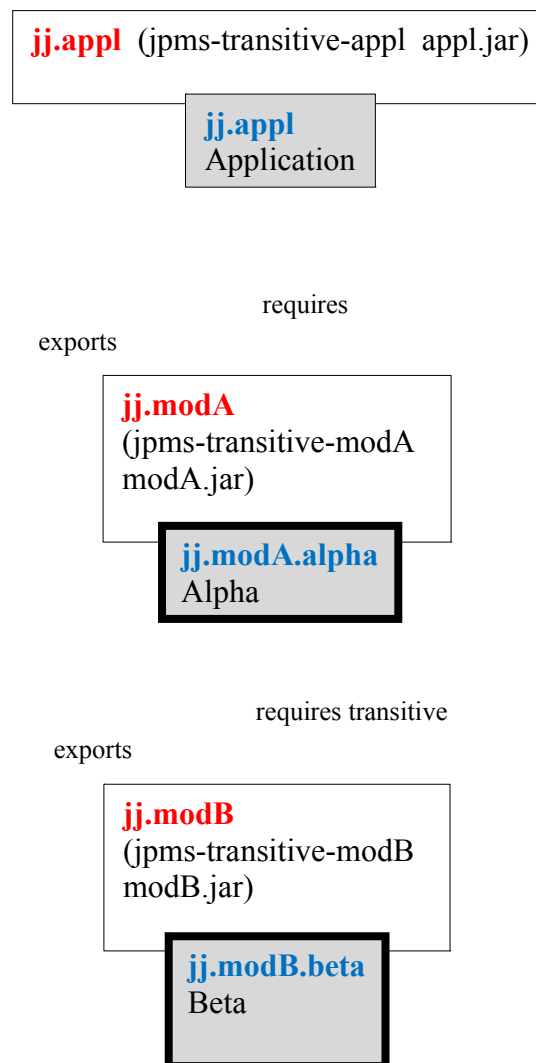


### 3.9 Requires Transitive

Ein Modul kann transitiv angefordert werden – via `requires transitive`.

`jj.modA` fordert `jj.modB` transitiv an. Wenn dann `jj.appl` das Modul `jj.modA` anfordert, kann `jj.appl` zunächst natürlich wieder alles nutzen, was `jj.modA` exportiert – zusätzlich nun aber auch alles, was `jj.modB` exportiert (ohne explizit die Nutzung von `jj.modB` anfordern zu müssen):

Das ist natürlich insbesondere dann interessant, wenn `jj.modA` einen Typ von `jj.modB` in einer öffentlichen Schnittstelle nutzt (einen Typ, der dann natürlich auch dem Client von `jj.modA` bekannt sein sollte).



## modB

```
module jj.modB {  
    exports jj.modB.beta;  
}
```

```
package jj.modB.beta;  
  
public class Beta { }
```

## modA

jj.modA fordert das Modul jj.modB transitiv an:

```
module jj.modA {  
    requires transitive jj.modB;  
    exports jj.modA.alpha;  
}
```

Die Alpha-Klasse besitzt eine pub-Methode, die ein Beta liefert (und um diese Methode nutzen zu können, benötigt ein Aufrufer von pub natürlich auch den Zugriff auf die Klasse Beta).

```
package jj.modA.alpha;  
  
import jj.modB.beta.Beta;  
  
public class Alpha {  
    public Beta pub() {  
        return new Beta();  
    }  
}
```

## appl

Das jj.appl-Modul fordert (explizit) nur jj.modA – bekommt aber automatisch auch Zugriff auf die Typen von jj.modB:

```
module jj.appl {  
    requires jj.modA;  
}
```

```
package jj.appl;
```

```
import jj.modA.alpha.Alpha;
import jj.modB.beta.Beta;

public class Application {

    public static void main(String[] args) {
        new Alpha();
        Beta beta = new Alpha().pub();
        new Beta();
    }
}
```

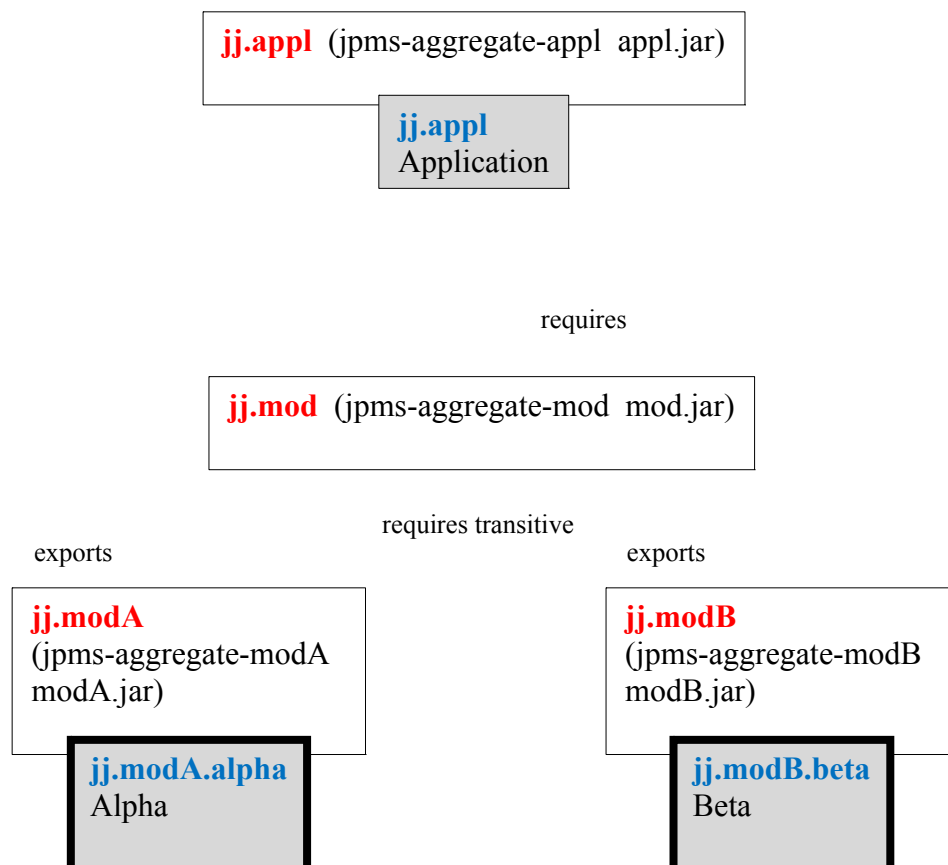
### 3.10 Aggregat-Module

Angenommen, viele Anwendungen benötigen stets das gleiche Set von Modulen. Jede dieser Anwendungen könnte in ihrer `module`-Datei all diese benötigten Module anfordern. Alle Anwendungen hätten also dieselbe umfangreiche `requires`-Liste.

Solche Duplikationen können mit dem Konzept der Aggregat-Module vermieden werden.

Ein Aggregations-Modul ist ein leeres Modul. Es enthält nur eine `module`-Datei, in welcher benötigte Module via `requires transitive` aufgezählt sind.

Das `jj.mod`-Modul der folgenden Anwendung ist ein solches Aggregations-Modul. Es fordert die Module `jj.modA` und `jj.modB` transitiv an:



## modA

```
module jj.modA {  
    exports jj.modA.alpha;  
}
```

```
package jj.modA.alpha;  
  
public class Alpha { }
```

## modB

```
module jj.modB {  
    exports jj.modB.beta;  
}
```

```
package jj.modB.beta;  
  
public class Beta { }
```

## mod

```
module jj.mod {  
    requires transitive jj.modA;  
    requires transitive jj.modB;  
}
```

## appl

Das `jj.appl`-Modul muss nun nurmehr `jj.mod` anfordern, um alle Typen in den von `jj.modA` und `jj.modB` exportierten Paketen nutzen zu können:

```
module jj.appl {  
    requires jj.mod;  
}
```

```
package jj.appl;  
  
import jj.modA.alpha.Alpha;  
import jj.modB.beta.Beta;  
  
public class Application {  
    public static void main(String[] args) {  
        Alpha alpha;  
        Beta beta;  
    }  
}
```

```
    }  
}
```

## 3.11 Das ServiceLoader-Konzept

Das neue Modul-System vereinfacht die Nutzung des `ServiceLoader`-Konzepts.

Da dieses Konzept relativ unbekannt ist, wird im Folgenden zunächst gezeigt, wie der `ServiceLoader` bislang (also ohne Modul-Unterstützung) genutzt werden konnte.

Wir benötigen Operatoren (Objekte, deren Klassen jeweils eine mathematische binäre Operation implementieren). Die eigentliche Anwendung soll diese Objekte nutzen können, ohne aber die Implementierungs-Klassen kennen zu müssen.

Wir benötigen also zunächst ein Interface (im Paket `jj.operators.iface`):

**mod**

```
package jj.operators.iface;

public interface Operator {
    public abstract String name();
    public abstract int apply(int x, int y);
}
```

Im `jj.operators`-Paket werden zwei Implementierungen dieses Interfaces bereitgestellt:

**operators**

```
package jj.operators;

import jj.operators.iface.Operator;

public class MinusOperator implements Operator {
    @Override
    public String name() {
        return "minus";
    }
    @Override
    public int apply(int x, int y) {
        return x - y;
    }
}
```

```
package jj.operators;

import jj.operators.iface.Operator;
```



```
public class PlusOperator implements Operator {  
    @Override  
    public String name() {  
        return "plus";  
    }  
    @Override  
    public int apply(int x, int y) {  
        return x + y;  
    }  
}
```

Das Projekt (und damit die jar-Datei) enthält ein `META-INF`-Verzeichnis, welches seinerseits ein Verzeichnis namens `services` enthält. Dieses enthält eine Datei mit dem voll-qualifizierten Namen des Interfaces:

```
META-INF/services/jj.operators.iface.Operator
```

Und diese Datei schließlich enthält die voll-qualifizierten Namen der Implementierungsklassen:

```
jj.operators.PlusOperator  
jj.operators.MinusOperator
```

## appl

Der Client ruft die statische Methode `load` der `ServiceLoader`-Klasse auf und übergibt dieser die Klasse des Interfaces (in Form einer `Class`-Referenz). Die Methode untersucht alle `META-INF/services`-Verzeichnisse und liefert schließlich eine Instanz der `ServiceLoader`-Klasse zurück, welche für jede der so gefundenen Implementierungsklassen ein Objekt der Klasse enthält. Die von `load` erzeugten `Operator`-Objekte können dann aus dem `ServiceLoader`-Objekt ausgelesen werden und z.B. in eine `Map` eingetragen werden:

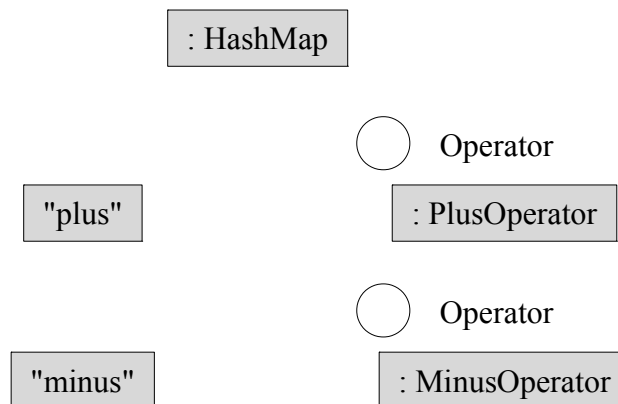
```
package jj.appl;  
  
import java.util.HashMap;  
import java.util.Map;  
import java.util.ServiceLoader;  
  
import jj.operators.iface.Operator;  
  
public class Application {  
    public static void main(String[] args) {
```

```
final ServiceLoader<Operator> loader =  
    ServiceLoader.load(Operator.class);  
  
final Map<String, Operator> operators = new HashMap<>();  
loader.forEach(op -> operators.put(op.name(), op));  
  
operators.forEach((name, op) -> {  
    System.out.println(name + ": " + op.apply(40, 2));  
});  
}
```

Die Ausgaben:

```
minus: 38  
plus: 42
```

Hier die erzeugt Map:



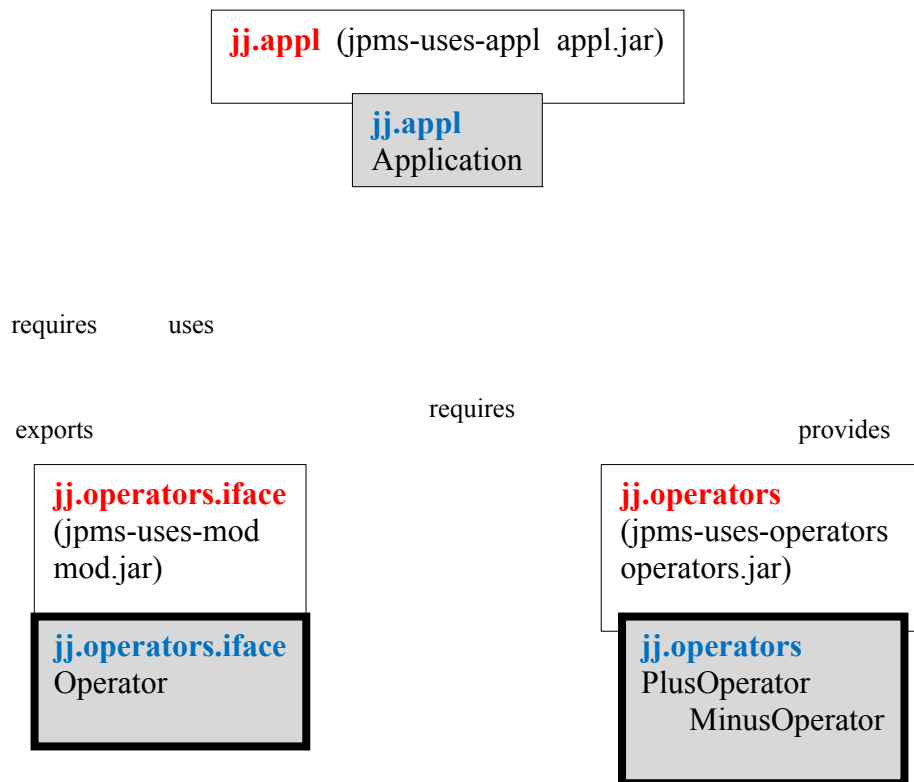
## 3.12 Provides und Uses

In Java 9 kann das `META-INF/services`-Verzeichnisse mit der darin enthaltenen Datei entfallen.

Diese Datei wird ersetzt durch einen Eintrag in der `module`-Datei derjenigen `jar`, welche die Implementierungsklasse enthält: `provides`.

Das Modul `jj.operators.iface` exportiert das Interfaces. Das Interface wird sowohl von `jj.appl` als auch von `jj.operators` genutzt. Das Modul `jj.operators` enthält die Implementierungsklassen. In seiner `module`-Datei wird mittels eines `provides-with`-Eintrags hinterlegt, welche Klassen das Interface implementieren:

```
provides jj.operators.iface.Operator
    with MinusOperator, PlusOperator;
```



## mod

```
module jj.operators.iface {  
    exports jj.operators.iface;  
}
```

```
package jj.operators.iface;  
  
public interface Operator {  
    public abstract String name();  
    public abstract int apply(int x, int y);  
}
```

## operators

```
import jj.operators.MinusOperator;  
import jj.operators.PlusOperator;  
  
module jj.operators {  
    requires jj.operators.iface;  
    provides jj.operators.iface.Operator with MinusOperator,  
PlusOperator;  
}
```

```
package jj.operators;  
  
import jj.operators.iface.Operator;  
  
public class MinusOperator implements Operator {  
    // ...  
}
```

```
package jj.operators;  
  
import jj.operators.iface.Operator;  
  
public class PlusOperator implements Operator {  
    // ...  
}
```

## appl

```
module jj.appl {  
    requires jj.operators.iface;
```

```
    uses jj.operators.iface.Operator;  
}
```

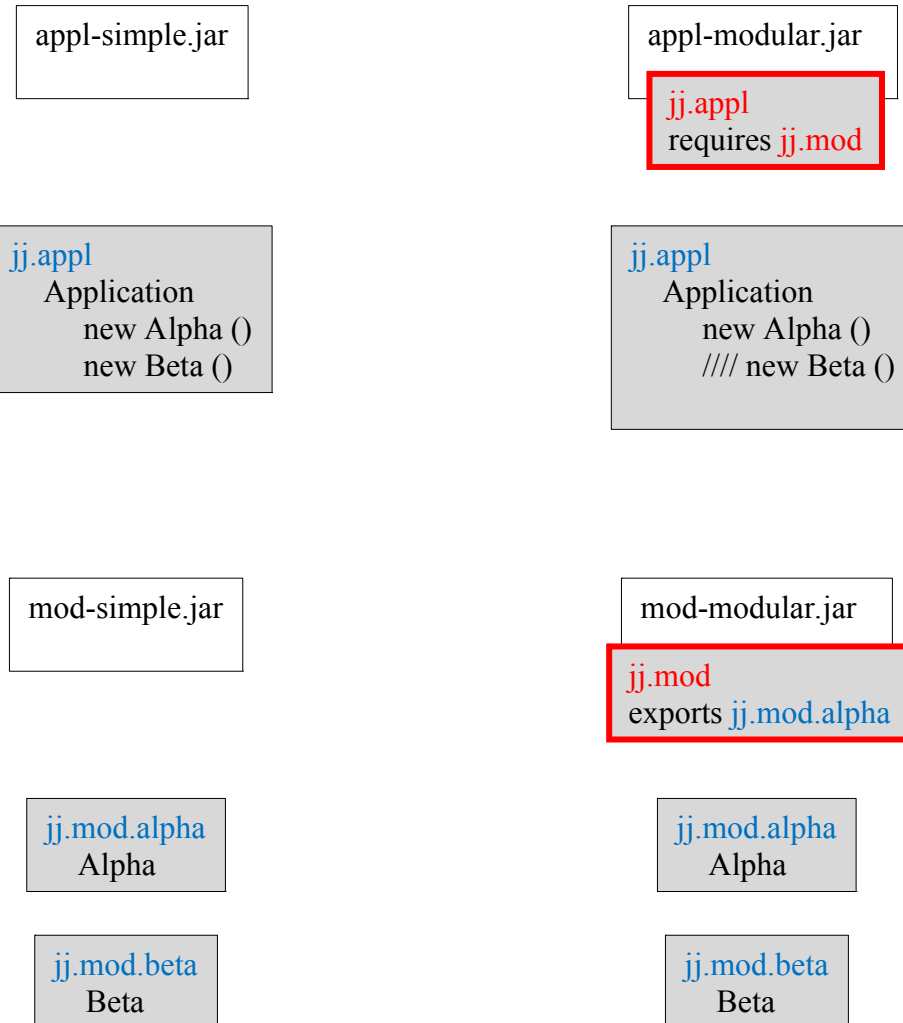
An der Benutzung des `ServiceLoaders` hat sich nichts geändert:

```
package jj.appl;  
  
import java.util.HashMap;  
import java.util.Map;  
import java.util.ServiceLoader;  
  
import jj.operators.iface.Operator;  
  
public class Application {  
    public static void main(String[] args) {  
        final ServiceLoader<Operator> loader =  
            ServiceLoader.load(Operator.class);  
  
        final Map<String, Operator> operators = new HashMap<>();  
        loader.forEach(op -> operators.put(op.name(), op));  
  
        operators.forEach((name, op) -> {  
            System.out.println(name + ": " + op.apply(40, 2));  
        });  
    }  
}
```

### 3.13 Mixing

Wie können neue (modulare) Anwendungen alte (nicht-modulare) jars nutzen? Und wie können alte Anwendungen neue jars nutzen?

Hier zunächst eine Übersicht:



Wir erzeugen zwei "Server"-jars: `mod-simple.jar` und `mod-modular.jar`. Beide enthalten die folgenden Klassen:

```
package jj.mod.alpha;  
  
public class Alpha {  
    Beta beta;  
}
```

```
package jj.mod.beta;  
  
public class Beta {  
}
```

Die `mod-modular.jar` enthält folgende `module-info`:

```
module jj.mod {  
    exports jj.mod.alpha;  
}
```

Und wir erzeugen zwei "Client"-jars: `appl-simple.jar` und `appl-modular.jar`.

Die `appl-simple.jar` enthält die folgende Klasse:

```
package jj.appl;  
  
import jj.mod.alpha.Alpha;  
import jj.mod.beta.Beta;  
  
public class Application {  
    public static void main(String[] args) {  
        System.out.println(new Alpha());  
        System.out.println(new Beta());  
    }  
}
```

Die `appl-modular.jar` enthält eine Klasse `Application`, die nur die Klasse `jj.mod.alpha.Alpha` nutzt:

```
package jj.appl;  
  
import jj.mod.alpha.Alpha;  
///// import jj.mod.beta.Beta;  
  
public class Application {
```

```
public static void main(String[] args) {  
    System.out.println(new Alpha());  
    ////// System.out.println(new Beta());  
}  
}
```

Die `appl-modular.jar` enthält folgende `module-info`:

```
module jj.appl {  
    requires jj.mod;  
}
```

Wir müssen nun unterscheiden: Wie können die jars zur Kompilationszeit genutzt werden und wie können sie zur Laufzeit genutzt werden.

Wir betrachten zunächst die Kompilation.

Die jar-Dateien `mod-simple.jar` und `mod-modular.jar` sind einfach zu erstellen (weil sie keinerlei weitere Abhängigkeiten aufweisen):

```
<target name="1" description="build simple.mod">  
    <build name="simple-mod" />  
</target>
```

```
<target name="2" description="build modular.mod">  
    <build name="modular-mod" />  
</target>
```

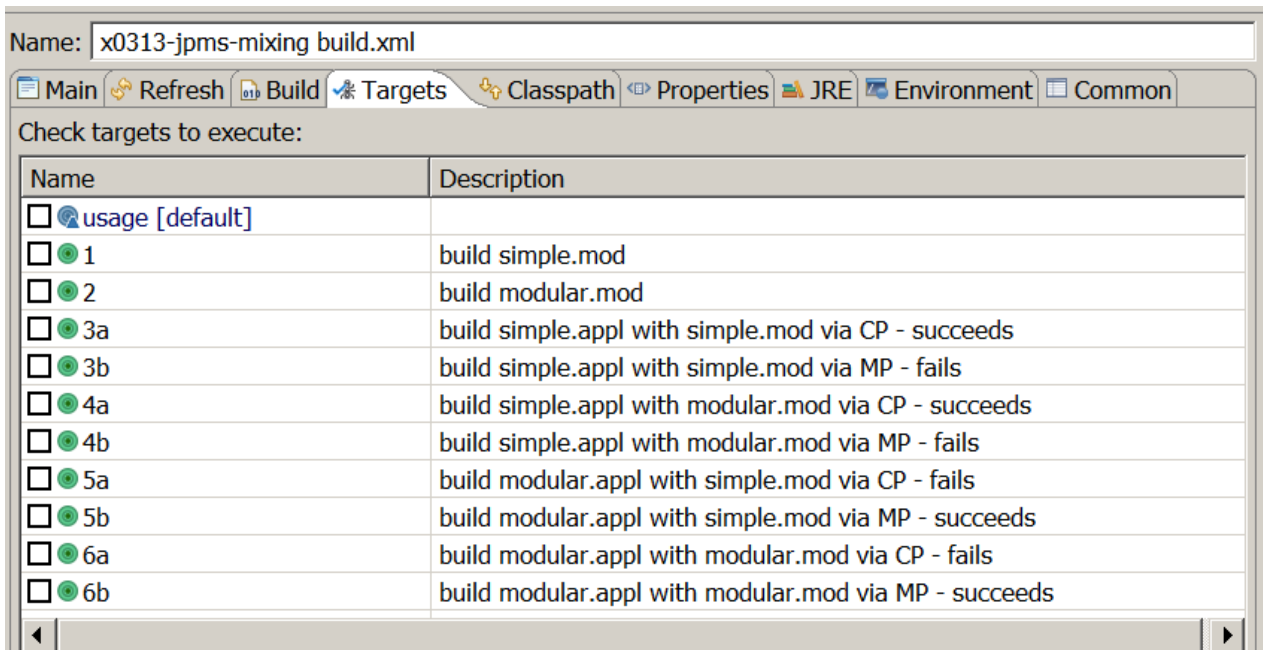
Die zu erzeugenden `appl-simple.jar`- und `appl-modular.jar`-Dateien aber sind abhängig von den `mod`-Dateien. Dann stellt sich die Frage, wie welche der `mod`-Dateien für die Erstellung der `appl`-Dateien genutzt werden können.

Es gibt theoretisch 8 Möglichkeiten, von denen aber nicht alle auch funktionieren.

Die `build.xml` des Wurzelverzeichnisses enthält für jede dieser theoretisch möglichen Varianten ein eigenes Target. Es sollte auch stets nur eines dieser Targets ausgeführt werden (Run As Ant Build...). Bei jedem Target wird angegeben, ob das Build erfolgreich ausgeführt wurde oder aber scheiterte.

Hier die Anzeige der Targets der Datei `build.xml`:





Hier der Quellcode der build.xml:

```
<project>

  <import file="../../shared/build.xml" />

  <target name="1" description=
    "build simple.mod">
    <build name="simple-mod">
      <paths>
      </paths>
    </build>
  </target>

  <target name="2" description=
    "build modular.mod">
    <build name="modular-mod">
      <paths>
      </paths>
    </build>
  </target>

  <target name="3a" description=
    "build simple.appl with simple.mod via CP -
succeeds">
    <build name="simple-appl">
```

```
        <paths>
            <classpath location="${basedir}/build/simple-
mod.jar" />
        </paths>
    </build>
</target>

<target name="3b" description=
    "build simple.appl with simple.mod via MP - fails">
    <build name="simple-appl">
        <paths>
            <modulepath location="${basedir}/build/simple-
mod.jar" />
        </paths>
    </build>
</target>

<target name="4a" description=
    "build simple.appl with modular.mod via CP -
succeeds">
    <build name="simple-appl">
        <paths>
            <classpath location="${basedir}/build/modular-
mod.jar" />
        </paths>
    </build>
</target>

<target name="4b" description=
    "build simple.appl with modular.mod via MP - fails">
    <build name="simple-appl">
        <paths>
            <modulepath location="${basedir}/build/modular-
mod.jar" />
        </paths>
    </build>
</target>

<target name="5a" description=
    "build modular.appl with simple.mod via CP - fails">
    <build name="modular-appl">
        <paths>
            <classpath location="${basedir}/build/simple-
mod.jar" />
        </paths>
    </build>
```

```

</target>

<target name="5b" description=
    "build modular.appl with simple.mod via MP -
succeeds">
    <!-- change module-info of modular-appl -> automatic
module name -->
    <build name="modular-appl">
        <paths>
            <modulepath location="${basedir}/build/simple-
mod.jar" />
        </paths>
    </build>
</target>

<target name="6a" description=
    "build modular.appl with modular.mod via CP - fails">
    <build name="modular-appl">
        <paths>
            <classpath location="${basedir}/build/modular-
mod.jar" />
        </paths>
    </build>
</target>

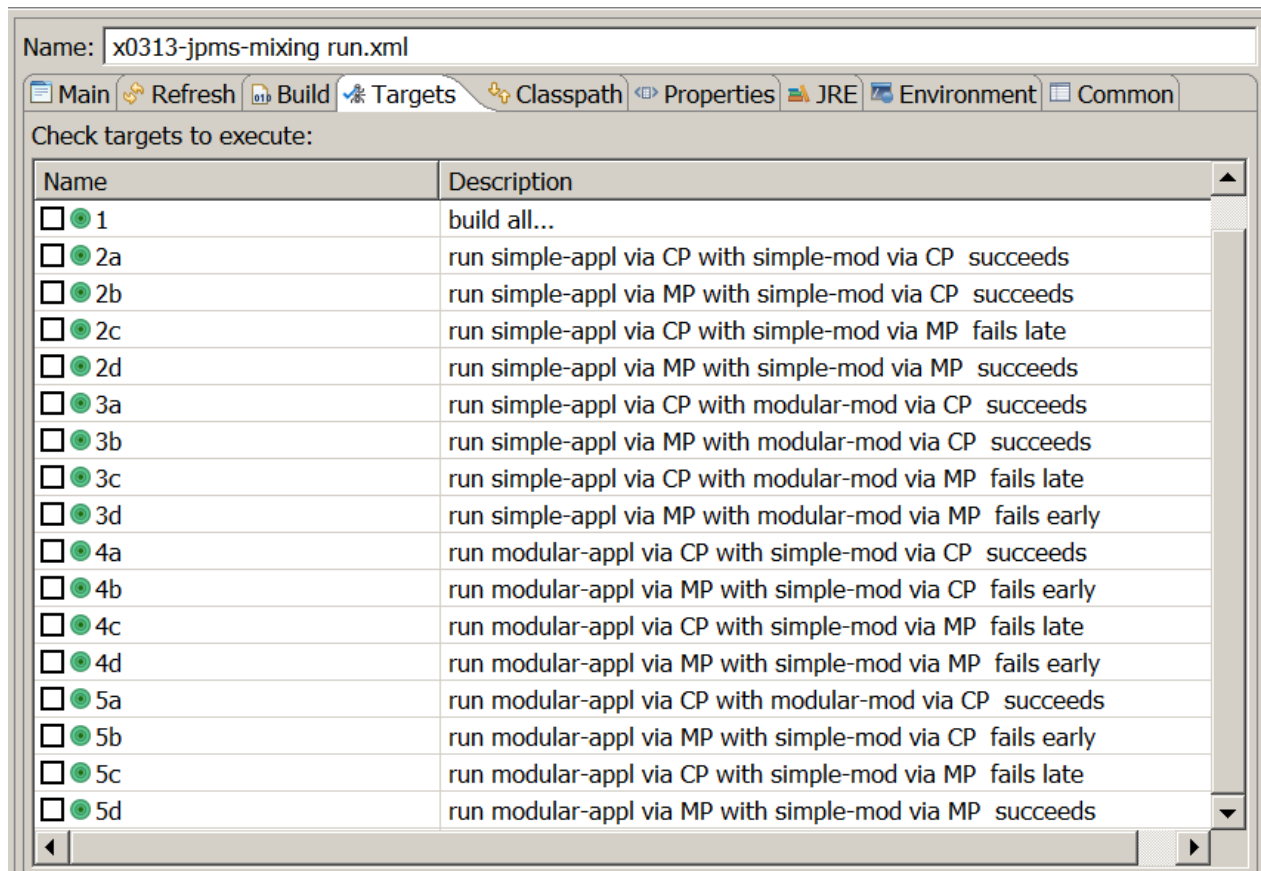
<target name="6b" description=
    "build modular.appl with modular.mod via MP -
succeeds">
    <build name="modular-appl">
        <paths>
            <modulepath location="${basedir}/build/modular-
mod.jar" />
        </paths>
    </build>
</target>

</project>

```

Wie verhält sich die Sache zur Laufzeit? Hier sind theoretisch 16 verschiedenen Varianten möglich.

Die Target-Übersicht der `run.xml` zeigt diese möglichen Varianten – und zeigt an, welche der denkbaren Varianten tatsächlich möglich sind und welche nicht:



Hier der Quellcode der `run.xml`:

```
<project>

  <import file="../../shared/build.xml" />

  <target name="1" description=
    "build all...">
    <build name="simple-mod">
      <paths>
      </paths>
    </build>
    <build name="modular-mod">
      <paths>
      </paths>
    </build>
    <build name="simple-appl">
      <paths>
        <classpath location="${basedir}/build/simple-
mod.jar" />
      </paths>
    </build>
  </target>
</project>
```

```

        </build>
        <build name="modular-appl">
            <paths>
                <modulepath location="${basedir}/build/modular-
mod.jar" />
            </paths>
        </build>
    </target>

    <!-- ===== simple-appl - simple-mod
===== -->

    <target name="2a" description=
        "run simple-appl via CP with simple-mod via CP
succeeds">
        <java classname="jj.appl.Application" fork="true">
            <classpath location="${basedir}/build/simple-
appl.jar" />
            <classpath location="${basedir}/build/simple-mod.jar"
/>
        </java>
    </target>

    <target name="2b" description=
        "run simple-appl via MP with simple-mod via CP
succeeds">
        <!-- automatic module name -->
        <java module="simple.appl"
            classname="jj.appl.Application" fork="true">
            <modulepath location="${basedir}/build/simple-
appl.jar" />
            <classpath location="${basedir}/build/simple-mod.jar"
/>
        </java>
    </target>

    <target name="2c" description=
        "run simple-appl via CP with simple-mod via MP fails
late">
        <java classname="jj.appl.Application" fork="true">
            <classpath location="${basedir}/build/simple-
appl.jar" />
            <modulepath location="${basedir}/build/simple-
mod.jar" />
        </java>
    </target>

```

```
<target name="2d" description=
    "run simple-appl via MP with simple-mod via MP
succeeds">
    <!-- automatic module name -->
    <java module="simple.appl"
        classname="jj.appl.Application" fork="true">
        <modulepath location="${basedir}/build/simple-
appl.jar" />
        <modulepath location="${basedir}/build/simple-
mod.jar" />
    </java>
</target>

<!-- ===== simple-appl - modular-mod
===== -->

<target name="3a" description=
    "run simple-appl via CP with modular-mod via CP
succeeds">
    <java classname="jj.appl.Application" fork="true">
        <classpath location="${basedir}/build/simple-
appl.jar" />
        <classpath location="${basedir}/build/modular-
mod.jar" />
    </java>
</target>

<target name="3b" description=
    "run simple-appl via MP with modular-mod via CP
succeeds">
    <!-- automatic module name -->
    <java module="simple.appl"
        classname="jj.appl.Application" fork="true">
        <modulepath location="${basedir}/build/simple-
appl.jar" />
        <classpath location="${basedir}/build/modular-
mod.jar" />
    </java>
</target>

<target name="3c" description=
    "run simple-appl via CP with modular-mod via MP
fails late">
    <java classname="jj.appl.Application" fork="true">
```

```

        <classpath location="${basedir}/build/simple-
appl.jar" />
        <modulepath location="${basedir}/build/modular-
mod.jar" />
    </java>
</target>

    <target name="3d" description=
        "run simple-appl via MP with modular-mod via MP
fails early">
        <!-- automatic module name -->
        <java module="simple.appl"
            classname="jj.appl.Application" fork="true">
            <modulepath location="${basedir}/build/simple-
appl.jar" />
            <modulepath location="${basedir}/build/modular-
mod.jar" />
        </java>
    </target>

    <!-- ===== modular-appl - simple-mod
===== -->

    <target name="4a" description=
        "run modular-appl via CP with simple-mod via CP
succeeds">
        <java classname="jj.appl.Application" fork="true">
            <classpath location="${basedir}/build/modular-
appl.jar" />
            <classpath location="${basedir}/build/simple-mod.jar"
/>
        </java>
    </target>

    <target name="4b" description=
        "run modular-appl via MP with simple-mod via CP
fails early">
        <!-- named module name -->
        <java module="jj.appl"
            classname="jj.appl.Application" fork="true">
            <modulepath location="${basedir}/build/modular-
appl.jar" />
            <classpath location="${basedir}/build/simple-mod.jar"
/>
        </java>
    </target>

```

```
<target name="4c" description=
    "run modular-appl via CP with simple-mod via MP
fails late">
    <java classname="jj.appl.Application" fork="true">
        <classpath location="${basedir}/build/modular-
appl.jar" />
        <modulepath location="${basedir}/build/simple-
mod.jar" />
    </java>
</target>

<target name="4d" description=
    "run modular-appl via MP with simple-mod via MP
fails early">
    <!-- named module name -->
    <java module="jj.appl"
        classname="jj.appl.Application" fork="true">
        <modulepath location="${basedir}/build/modular-
appl.jar" />
        <modulepath location="${basedir}/build/simple-
mod.jar" />
    </java>
</target>

<!-- ===== modular-appl - modular-mod
===== -->

<target name="5a" description=
    "run modular-appl via CP with modular-mod via CP
succeeds">
    <java classname="jj.appl.Application" fork="true">
        <classpath location="${basedir}/build/modular-
appl.jar" />
        <classpath location="${basedir}/build/modular-
mod.jar" />
    </java>
</target>

<target name="5b" description=
    "run modular-appl via MP with simple-mod via CP
fails early">
    <!-- named module name -->
    <java module="jj.appl"
        classname="jj.appl.Application" fork="true">
```



```
        <modulepath location="${basedir}/build/modular-
appl.jar" />
        <classpath location="${basedir}/build/modular-
mod.jar" />
    </java>
</target>

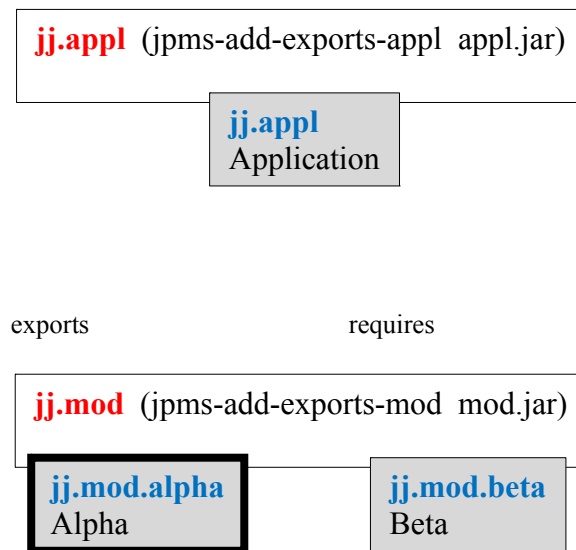
    <target name="5c" description=
        "run modular-appl via CP with simple-mod via MP
fails late">
        <java classname="jj.appl.Application" fork="true">
            <classpath location="${basedir}/build/modular-
appl.jar" />
            <modulepath location="${basedir}/build/modular-
mod.jar" />
        </java>
    </target>

    <target name="5d" description=
        "run modular-appl via MP with simple-mod via MP
succeeds">
        <!-- named module name -->
        <java module="jj.appl"
            classname="jj.appl.Application" fork="true">
            <modulepath location="${basedir}/build/modular-
appl.jar" />
            <modulepath location="${basedir}/build/modular-
mod.jar" />
        </java>
    </target>
</project>
```

### 3.14 Add Exports

Wie kann ein Modul auf die Klasse eines anderen Moduls zugreifen, wenn dieses nicht in der module-info des diese Klasse enthaltenen Moduls via `exports` für eine solche Nutzung vorgesehen wurde?

Im Folgenden möchte die `Application` die Klasse `Beta` zugreifen. Das Package `jj.mod` exportiert in seiner module-Info aber nur `Alpha`:



#### mod

```
module jj.mod {  
    exports jj.mod.alpha;  
}
```

```
package jj.mod.alpha;  
  
public class Alpha { }
```

```
package jj.mod.beta;  
  
public class Beta { }
```

#### appl

```
module jj.appl {  
    requires jj.util;  
    requires jj.mod;  
}
```

```
package jj.appl;  
  
import jj.mod.alpha.Alpha;  
import jj.mod.beta.Beta;  
import jj.util.trycatch.TryCatch;  
  
public class Application {  
  
    public static void main(String[] args) {  
        Alpha alpha = new Alpha();  
        Beta beta = new Beta();  
        TryCatch.run(() -> {  
            final Class<?> cls =  
Class.forName("jj.mod.beta.Beta");  
            final Object obj =  
cls.getConstructor().newInstance();  
            System.out.println(obj);  
        });  
    }  
}
```

### build.xml

```
<?xml version="1.0" encoding="iso-8859-1"?>  
<!DOCTYPE project>  
  
<project default="run">  
  
    <import file="../../shared/build.xml" />  
  
    <target name="build-mod">  
        <build name="mod">  
            <paths>  
            </paths>  
        </build>  
    </target>  
  
    <target name="build-appl" depends="build-mod">  
        <build name="appl">  
            <paths>
```

```
        <compilerarg
value="--add-exports=jj.mod/jj.mod.beta=jj.appl"/>
        <modulepath location="${shared}/build/util.jar" /
>
        <modulepath location="${basedir}/build/mod.jar" /
>
        </paths>
    </build>
</target>

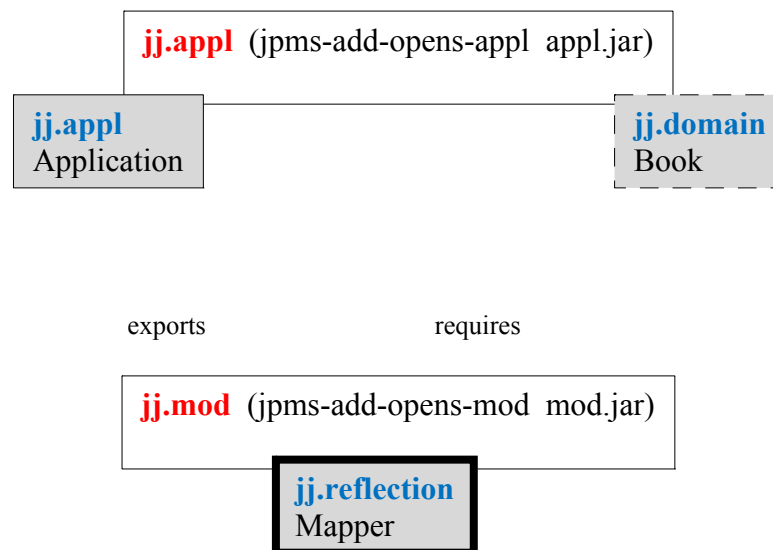
    <target name="run" depends="build-appl">
        <java module="jj.appl" classname="jj.appl.Application"
fork="true">
            <jvmarg value="--add-exports" />
            <jvmarg value="jj.mod/jj.mod.beta=jj.appl" />
            <modulepath location="${shared}/build/util.jar" />
            <modulepath location="${basedir}/build/appl.jar" />
            <modulepath location="${basedir}/build/mod.jar" />
        </java>
    </target>
</project>
```

Dem Compiler und der virtuellen Maschine wird jeweils ein `--add-exports`-Argument übergeben.

### 3.15 Add Opens

Wie kann ein Modul via Reflection auf die Klasse eines anderen Moduls zugreifen, wenn dieses nicht in der module-info des diese Klasse enthaltenen Moduls via `opens` für einen solchen Zugriff geöffnet wurde?

Im Folgenden möchte der `Mapper` via Reflection auf die Elemente von `Book` zugreifen. Das Package `jj.domain` ist aber in der module-Info von `jj.appl` nicht geöffnet



#### mod.jar

```
module jj.reflection {
    exports jj.reflection;
}
```

```
package jj.reflection;

import java.lang.reflect.Field;
import java.util.HashMap;
import java.util.Map;

public class Mapper {
    public static Map<String, Object> map(Object obj) {
        // ...
    }
}
```

```
}  
}
```

## appl.jar

```
module jj.appl {  
    requires jj.util;  
    requires jj.reflection;  
    // does not open domain...  
}
```

```
package jj.domain;
```

```
public class Book {  
    // ...  
}
```

```
package jj.appl;
```

```
import jj.domain.Book;  
import jj.reflection.Mapper;  
import jj.util.log.Log;
```

```
public class Application {  
  
    public static void main(String[] args) {  
        Book book = new Book("1111", "Pascal", 1970, "N. Wirth");  
        Mapper.map(book).forEach(  
            (name, value) -> System.out.println(name + " = " +  
value));    }  
}
```

## build.xml

```
<?xml version="1.0" encoding="iso-8859-1"?>  
<!DOCTYPE project>  
  
<project default="run">  
  
    <import file="../../shared/build.xml" />  
  
    <target name="build-mod">  
        <build name="mod">  
            <paths>
```

```
        </paths>
      </build>
    </target>

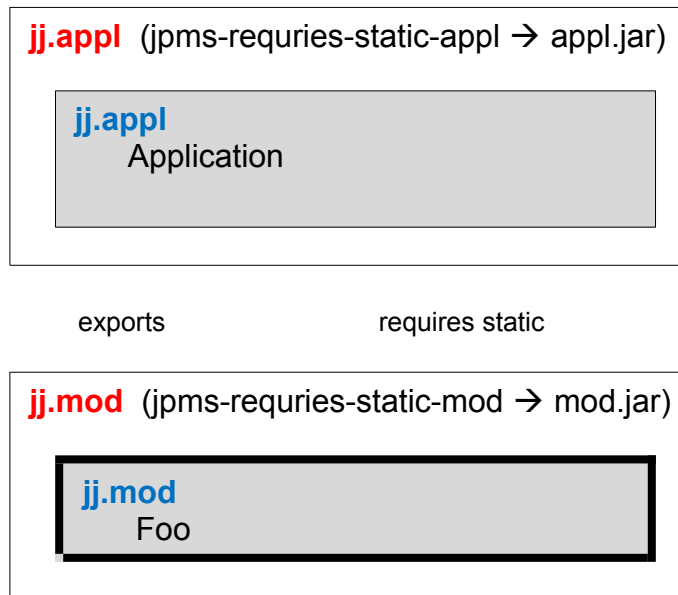
    <target name="build-appl" depends="build-mod">
      <build name="appl">
        <paths>
          <modulepath location="${shared}/build/util.jar" /
>
          <modulepath location="${basedir}/build/mod.jar" /
>
        </paths>
      </build>
    </target>

    <target name="run" depends="build-appl">
      <java module="jj.appl" classname="jj.appl.Application"
fork="true">
        <jvmarg value="--add-opens" />
        <jvmarg value="jj.appl/jj.domain=jj.reflection" />
        <modulepath location="${shared}/build/util.jar" />
        <modulepath location="${basedir}/build/mod.jar" />
        <modulepath location="${basedir}/build/appl.jar" />
      </java>
    </target>
  </project>
```

Dem Compiler und der virtuellen Maschine wird jeweils ein `--add-opens`-Argument übergeben.

## 3.16 Add Modules

Zur Compilations-Zeit muss `jj.mod` vorhanden sein – aber zur Laufzeit nicht unbedingt. Ob es vorhanden ist oder nicht (ob es also zur Laufzeit genutzt werden kann oder nicht) soll über die `java`-Parameter (über die JVM-Parameter) einstellbar sein.



### mod

```
module jj.mod {  
    exports jj.mod;  
}
```

```
package jj.mod;  
  
public class Foo { }
```

### appl

```
module jj.appl {  
    requires jj.util;  
    requires static jj.mod;  
}
```



Zur Compilationszeit muss `jj.mod` existieren, zur Laufzeit wird es aber nicht (!) automatisch herangezogen. Damit es zur Laufzeit herangezogen werden kann, muss es via `add-modules` "hinzugelinkt" werden...

```
package jj.appl;  
  
import java.util.Optional;  
  
import jj.mod.Foo;  
import jj.util.log.Log;  
  
public class Application {  
  
    public static void main(String[] args) {  
        // ...  
    }  
}
```

```
static void demoTryCatch() {  
    Log.logMethodCall();  
    try {  
        Foo foo = new Foo();  
        System.out.println("Done");  
    }  
    catch (NoClassDefFoundError e) {  
        System.out.println(e);  
    }  
}
```

```
static void demoFind() {  
    Log.logMethodCall();  
    Optional<Module> module =  
ModuleLayer.boot().findModule("jj.mod");  
    System.out.println(module);  
}
```

## build.xml

```
<?xml version="1.0" encoding="iso-8859-1"?>  
<!DOCTYPE project>  
  
<project default="run">  
  
    <import file="../../shared/build.xml" />
```

```
<target name="build-mod">
    <build name="mod">
        <paths>
        </paths>
    </build>
</target>

<target name="build-appl" depends="build-mod">
    <build name="appl">
        <paths>
            <modulepath location="${shared}/build/util.jar" /
>
            <modulepath location="${basedir}/build/mod.jar" /
>
        </paths>
    </build>
</target>

<target name="run" depends="build-appl">
    <java module="jj.appl" classname="jj.appl.Application"
fork="true">
        <modulepath location="${shared}/build/util.jar" />
        <modulepath location="${basedir}/build/mod.jar" />
        <modulepath location="${basedir}/build/appl.jar" />
        <jvmarg value="--add-modules" />
        <jvmarg value="jj.mod" />
    </java>
</target>

</project>
```

Existiert das `add-modules`-Argument, erzeugt die `main`-Methode folgende Ausgaben:

```
Done
[java] Optional[module jj.mod]
```

Fehlt das `add-modules`-Argument, erzeugt die `main`-Methode folgende Ausgaben:

```
java.lang.NoClassDefFoundError: jj/mod/Foo
Optional.empty
```

Vorsicht: Im Module-Path muss aber `jj.mod` so oder so vorhanden sein!!! (Aber auch dann, wenn dieser Eintrag vorhanden ist, wird eben `jj.mod` nicht automatisch herangezogen – dies geschieht erst via `add-modules`...

(In Eclipse wird das `add-modules`-Argument offensichtlich automatisch gesetzt...)



## 4 Reflection

Klassen werden von `Class`-Objekten repräsentiert, Packages von `Package`-Objekten.

Module werden repräsentiert von `Module`- und von `ModuleDescriptor`-Objekten.

Reflection ist also erweitert worden.

### Inhalte

- Im ersten Abschnitt wird gezeigt, dass (und warum) `Class.newInstance` nun deprecated ist.
- Im zweiten Abschnitt demonstrieren wir dann die Klassen `Module` und `ModuleDescription`.
- Im dritten Abschnitt demonstrieren wir die Klasse `ModuleLayer`.

## 4.1 Class.newInstance deprecated

Die `Class`-Methode `newInstance()` ist deprecated. Der Grund liegt im Umgang mit Exceptions, die im Konstruktor der zu instanziierten Klasse geworfen werden.

Sei folgende Klasse gegeben:

```
package jj.appl;

public class Foo {

    public static boolean doThrow = false;

    public Foo() {
        if (doThrow)
            throw new RuntimeException("water in drive a:");
    }

    public void f() {
        System.out.println(this.getClass().getSimpleName() +
            ".f()");
    }
}
```

Über das statische Attribut kann eingestellt werden, ob der Konstruktor eine Ausnahme wirft oder nicht.

Wir instanziierten die Klasse via `Class.newInstance()` – wobei wir den `Foo`-Konstruktor eine `RuntimeException` werfen lassen:

```
static void demoNewInstanceException() {
    try {
        Foo.doThrow = true;
        final Class<?> cls = Class.forName("jj.appl.Foo");
        final Foo foo = (Foo) cls.newInstance();
        foo.f();
    }
    catch (final RuntimeException e) {
        System.out.println(e);
    }
    catch (final Exception e) {
        System.out.println(e);
    }
}
```

Die Ausgabe:

```
java.lang.RuntimeException: water in drive a:
```

Die vom Konstruktor geworfene `Exception` wird von `newInstance` weitergeworfen – die dem Aufrufer von `newInstance` zugestellte `Exception` ist also genau diejenige `Exception`, die vom Konstruktor der zu instanziierten Klasse geworfen wird.

Anders sieht die Sache bei der Verwendung eines `Constructor`-Objekts aus:

```
static void demoGetConstructorException() {
    try {
        Foo.doThrow = true;
        final Class<?> cls = Class.forName("jj.appl.Foo");
        final Foo foo = (Foo)
cls.getConstructor().newInstance();
        foo.f();
    }
    catch (final InvocationTargetException e) {
        System.out.println(e);
        System.out.println(e.getTargetException());
    }
    catch (final Exception e) {
        System.out.println(e);
    }
}
```

Die Ausgaben:

```
java.lang.reflect.InvocationTargetException
java.lang.RuntimeException: water in drive a:
```

Die `newInstance`-Methode der `Constructor`-Klasse wickelt die vom `Foo`-Konstruktor geworfene `RuntimeException` in eine `InvocationTargetException` ein – und wird in eben dieser Form dem Aufrufer zugestellt. Dieser kann dann die "eingewickelte" `Exception` mittels `getTargetException` ermitteln.

Auch die `invoke`-Methode der `Method`-Klasse wickelt die von einer Zielmethode geworfene `Exception` in eine `InvocationTargetException` ein.

Dieses "Einwickeln" einer `Exception` in eine `InvocationTargetException` ist also der Standard. Von diesem Standard weicht `Class.newInstance()` ab – und eben deshalb ist diese Methode nun deprecated.

`Class.newInstance` hat noch einen weiteren schwerwiegenden Nachteil.

Gegeben sei eine Klasse, deren Konstruktor ggf. eine checked-Exception wirft:

```
package jj.appl;

import java.io.IOException;

public class Bar {
    public Bar() throws IOException {
        throw new IOException("water in drive a:");
    }
}
```

Wir ermitteln das Class-Objekt, welches die Klasse `Bar` beschreibe:

```
final Class<?> cls = Class.forName("jj.appl.Bar");
```

Und erzeugen anschließend mittels `Class.newInstance` eine Instanz dieser Klasse – wobei wir für jede von dieser Methode geworfene checked-Exception einen eigenen catch-Zweig vorsehen:

```
try {
    final Bar bar = (Bar) cls.newInstance();
}
catch (InstantiationException e) {
    e.printStackTrace();
}
catch (IllegalAccessException e) {
    e.printStackTrace();
}
catch (IllegalArgumentException e) {
    e.printStackTrace();
}
catch (SecurityException e) {
    e.printStackTrace();
}
// illegal:
// catch (IOException e) {
//     e.printStackTrace();
// }
```

Für die vom `Bar`-Konstruktor geworfene `IOException` kann leider kein `catch`-Zweig eingebaut werden – denn eine solche Exception wird von `Class.newInstance` natürlich nicht deklariert. (Wir könnten allenfalls eine `catch`-Zweig einbauen, der jede Exception fängt: `catch(Exception e) .`)

Der Compiler lässt es im obigen Falle also nicht zu, eine checked-Exception in einem spezifischen `catch` zu behandeln!



## 4.2 Die Klassen Module und ModuleDescriptor

Das Reflection-API wurde erweitert um die Klassen `Module` (im Package `java.lang`) und `ModuleDescriptor` (im Package `java.lang.module`).

Sei z.B. folgendes Demo-Modul gegeben:

### Das Modul `jj.mod`:

```
@jj.util.annotations.Author(name = "Nowak")
module jj.mod {
    requires jj.util;
    exports jj.mod.pub to jj.appl;
    opens  jj.mod.pri;
}
```

```
package jj.mod.pub;

public class Foo { }
```

```
package jj.mod.pri;

public class Bar { }
```

Und folgende Applikation:

### Das Modul `jj.appl`:

```
module jj.appl {
    requires java.sql;
    requires java.xml;
    requires jj.mod;
    requires jj.util;
}
```

```
package jj.appl;

import java.lang.Module;
import java.lang.annotation.Annotation;
import java.lang.module.ModuleDescriptor;
import java.util.Set;

import jj.mod.pub.Foo;
import jj.util.log.Log;
```

```
public class Application {  
    // ...  
}
```

Die `Application`-Klasse definiert einige `demo`-Methoden, die im Folgenden erläutert werden.

Mittels der Class-Methode `getModule()` kann dasjenige `Module`-Objekt ermittelt werden, dem die vom `Class`-Objekt beschriebene Klasse zugeordnet ist. Die `Module`-Methode `getName()` liefert den Namen des Moduls zurück und `isNamed()` liefert `true`, sofern es sich um ein benanntes Applikations-Modul handelt:

```
static void demoGetModule() throws Exception {  
    final Module m1 = Foo.class.getModule();  
    final Module m2 =  
Class.forName("jj.mod.pri.Bar").getModule();  
    System.out.println(m1 == m2);  
    System.out.println(m1.getName());  
    System.out.println(m1.isNamed());  
}
```

Die Ausgaben:

```
true  
jj.mod  
true
```

Welchem Modul gehören die primitiven Typen an?:

```
static void demoGetModulePrimitives () {  
    System.out.println(int.class.getModule().getName());  
    System.out.println(void.class.getModule().getName());  
}
```

Die Ausgaben:

```
java.base  
java.base
```

Welchem Modul gehören Typen der Standardbibliothek an?:

```
static void demoGetModuleSystemClasses () {  
    System.out.println(String.class.getModule().getName());  
    System.out.println(  
        java.sql.Connection.class.getModule().getName());  
}
```

```
        System.out.println(  
java.xml.stream.XMLEventReader.class.getModule().getName());  
    }
```

Die Ausgaben:

```
java.base  
java.sql  
java.xml
```

Welchem Modul gehören Array-Typen an? Demjenigen Modul, dem der Elementtyp des entsprechenden Array-Typs angehört.

```
static void demoGetModuleArrays() {  
    System.out.println(int[].class.getModule().getName());  
    System.out.println(Foo[].class.getModule().getName());  
}
```

Die Ausgaben:

```
java.base  
jj.mod
```

Mittels der Module-Methode `getAnnotations` können die Annotationen ausgelesen werden, die mit `@RetentionType.RUNTIME` annotiert sind:

```
static void demoGetAnnotations() {  
    final Annotation[] annotations =  
        Foo[].class.getModule().getAnnotations();  
    for(final Annotation annotation : annotations)  
        System.out.println(annotation);  
}
```

Die Ausgabe:

```
@jj.util.annotations.Author(name="Nowak")
```

Die hier verwendete `@Author`-Annotation ist im Modul `jj.util` wie folgt definiert:

```
package jj.util.annotations;  
// ...  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.MODULE)  
public @interface Author {  
    public abstract String name();  
}
```

```
}
```

Das Package `jj.util.annotations` wird natürlich von `jj.util` exportiert:

```
module jj.util {  
    // ...  
    exports jj.util.annotations;  
}
```

Mittels der Module-Methode `canRead(Module m)` kann ermittelt werden, ob das aktuelle Modul das Modul `m` benutzen kann. `appl` kann `mod` benutzen, `mod` aber nicht `appl`:

```
static void demoCanRead() {  
    final Module mod = Foo.class.getModule();  
    final Module appl = Application.class.getModule();  
    System.out.println(appl.canRead(mod));  
    System.out.println(mod.canRead(appl));  
}
```

Die Ausgaben:

```
true  
false
```

Mittels der Module-Methode `getPackages()` können die Packages (in Form von Strings!) ermittelt werden, die zu einem Modul gehören. Mittels der Methoden `isExported(p)` resp. `isOpen(p)` kann ermittelt werden, ob das Package mit dem Namen `p` exportiert wird resp. für Reflection geöffnet ist. Mit `isExported(p, m)` resp. mit `isOpen(p, m)` kann ermittelt werden, ob das Package `p` für das Modul `m` exportiert wird resp. für den Reflection-Zugriff für `m` geöffnet ist:

```
static void demoGetPackages() {  
    final Module mod = Foo.class.getModule();  
    final Module appl = Application.class.getModule();  
    final Set<String> packages = mod.getPackages();  
    packages.forEach(p -> {  
        System.out.println(p);  
        System.out.println("\t" + mod.isExported(p));  
        System.out.println("\t" + mod.isExported(p, appl));  
        System.out.println("\t" + mod.isOpen(p));  
        System.out.println("\t" + mod.isOpen(p, appl));  
    });  
}
```

Die Ausgaben:

```
jj.mod.pri
  true
  true
  true
  true
jj.mod.pub
  false
  true
  false
  false
```

Mittels der `Module`-Methode `getDescriptor` kann der `ModuleDescriptor` ermittelt werden. Dieser Deskriptor enthält detailliertere Informationen zum entsprechenden Modul:

```
static void demoGetDescriptor() {
    final ModuleDescriptor descr =
        Foo.class.getModule().getDescriptor();
    System.out.println(descr.name());

    System.out.println("exports:");
    final Set<ModuleDescriptor.Exports> exports =
descr.exports();
    exports.forEach(export -> System.out.println("\t" +
        export.source() + " => " + export.targets()));

    System.out.println("opens:");
    final Set<ModuleDescriptor.Opens> opens = descr.opens();
    opens.forEach(open -> System.out.println("\t" +
        open.source() + " => " + open.targets()));
}
```

Die Ausgaben:

```
jj.mod
exports:
  jj.mod.pub => [jj.appl]
opens:
  jj.mod.pri => []
```

Leider gibt's keine statische `Module`-Methode, mittels derer sich alle an einem System beteiligten Module deren `Module`-Objekte ermitteln lassen. Und es existiert auch keine `Module`-Methode, mittels derer sich aufgrund des Namens eines Modul das entsprechende `Module`-Objekt ermitteln lässt (analog zu `Class.forName`)

## 4.3 Die Klasse ModuleLayer

**appl**

```
module jj.appl{  
    requires jj.util;  
    requires java.sql;  
}
```

```
package jj.appl;  
  
import java.lang.module.Configuration;  
import java.lang.module.ModuleReader;  
import java.lang.module.ModuleReference;  
import java.lang.module.ResolvedModule;  
import java.util.Optional;  
import java.util.stream.Stream;  
  
import jj.util.log.Log;  
  
public class Application {  
    public static void main(String[] args) throws Exception {  
        demoModuleLayer();  
        demoModuleReader("jj.demo");  
    }  
    // ...  
}
```

```
    private static void printModule(ResolvedModule m) {  
        System.out.println(m.name());  
        m.reads().forEach(r -> System.out.println("\t" +  
r.name()));  
        System.out.println("\t=> " + m.reference());  
    }
```

```
    static void demoModuleLayer() {  
        Log.logMethodCall();  
        final ModuleLayer l = ModuleLayer.boot();  
        System.out.println("Modules");  
        l.modules().forEach(System.out::println);  
        final Configuration c = l.configuration();  
        System.out.println("Configuration.Modules");  
        c.modules().forEach(m -> printModule(m));  
    }
```

```
Modules
module jdk.crypto.cryptoki
module jdk.jfr
module java.smartcardio
module jdk.internal.vm.ci
module java.sql
module java.security.sasl
module java.compiler
module jdk.management.jfr
module java.prefs
...
Configuration.Modules
java.sql
    java.xml
    java.logging
    java.base
    => [module java.sql, location=jrt:/java.sql]
jj.appl
    java.sql
    java.xml
    java.base
    java.logging
    jj.util
    => [module jj.appl,
location=file:///C:/Users/Nowak/jn/seminar/java/java-9/projects/x0403-
reflection-ModuleLayer/bin/]
java.security.sasl
    java.logging
    java.base
    => [module java.security.sasl, location=jrt:/java.security.sasl]
...
```

```
static void demoModuleReader(String moduleName) throws
Exception {
    Log.logMethodCall();
    System.out.println("Inspecting " + moduleName);
    final Optional<ResolvedModule> optionalResolvedModule =
ModuleLayer.boot().configuration().findModule(moduleName);
    if (! optionalResolvedModule.isPresent())
        return;
    final ResolvedModule resolvedModule =
        optionalResolvedModule.get();
    final ModuleReference moduleReference =
resolvedModule.reference();
    printModule(resolvedModule);

    try (ModuleReader reader = moduleReference.open()) {
        final Stream<String> stream = reader.list();
```

```
        stream.forEach(System.out::println);  
    }  
}
```

Inspecting jj.appl

jj.appl

java.sql

java.xml

java.base

java.logging

jj.util

=> [module jj.appl,

location=file:///C:/Users/Nowak/jn/seminar/java/java-9/projects/x0403-  
reflection-ModuleLayer/bin/]

jj/

jj/appl/

jj/appl/Application.class

module-info.class



## 5 Spracherweiterungen

Die Spracherweiterungen von Java 9 halten sich in übersichtlichen Grenzen.

### Inhalte

- Der in Java 7 eingeführte Diamond-Operator kann nun auch bei der Implementierung anonymer Klassen verwendet werden.
- Interface können nun auch private Methoden besitzen (sowohl statische als auch nicht-statische)
- Der in Java 7 eingeführte resource-try kann nun auch für bereits zuvor initialisierte Resource-Variablen verwendet werden.
- Die `@SafeVararg`-Annotation kann nun auch bei privaten Methoden verwendet werden.
- Der Underscore ist als Bezeichner nicht mehr erlaubt.

## 5.1 Diamond-Operator

Der Diamond-Operator konnte und kann bekanntlich wie folgt verwendet werden:

```
List<Map<Integer, String>> list = new ArrayList<>();
```

Bei der Definition anonymer Klassen konnte er bislang aber nicht verwendet werden:

```
List<Map<Integer, String>> list = new ArrayList<>() {  
    // ...  
};
```

Mit Java 9 ist er nun auch bei solchen Definitionen erlaubt.

By the way: Im strengen Sinne gibt's überhaupt keinen Diamond-"Operator". Dieser "Operator" setzt sich aus zwei Operatoren zusammen: dem <- und dem >-Operator. Wir könnten also auch schreiben: `new Callable< >() {...}`.

Im Folgenden werden wir näher untersuchen, wie der Compiler bei anonymen Klassen mit einem Diamond die tatsächlichen Parametertypen ermittelt.

Wir verwenden dabei folgende Hilfsmethode:

```
static void printGenericType(Object obj) {  
    try {  
        Class<?> cls = obj.getClass();  
        ParameterizedType pt = (ParameterizedType)  
            cls.getGenericInterfaces()[0];  
        Type[] typeArgs = pt.getActualTypeArguments();  
        System.out.print(  
            cls.getInterfaces()[0].getSimpleName() + "<";  
        for (int i = 0; i < typeArgs.length; i++) {  
            if (i > 0)  
                System.out.print(", ");  
            System.out.print(((Class<?>  
>)typeArgs[i]).getSimpleName());  
        }  
        System.out.println(">");  
    }  
    catch (Exception e) {  
        System.err.println(e);  
    }  
}
```

Das Interface `java.util.function.Function` konnte im alten Java z.B. wie folgt implementiert werden:

```
static void demoFunctionOld() {  
    final Function<String, Integer> func =  
        new Function<String, Integer>() {  
        @Override  
        public Integer apply(String s) {  
            printGenericType(this);  
            return s.length();  
        }  
    };  
    System.out.println(func.apply("Hello"));  
}
```

`printGenericType` gibt – das ist nicht überraschend – die folgende Zeile aus:

```
Function<String, Integer>
```

Java 9 erlaubt folgende Verkürzung:

```
static void demoFunctionNew() {  
    final Function<String, Integer> func = new Function<>() {  
        @Override  
        public Integer apply(String s) {  
            printGenericType(this);  
            return s.length();  
        }  
    };  
    System.out.println(func.apply("Hello"));  
}
```

Auch hier gibt `printGenericType` folgende Zeile aus:

```
Function<String, Integer>
```

Der Compiler hat für die generierte Klasse also exakt diejenigen Typen übernommen, mittels derer auch die `Function`-Variable `func` deklariert ist.

Für die folgenden Beispiele verwenden wir eine kleine Klassenhierarchie:

```
class Drink { }  
class Wine extends Drink { }  
class RedWine extends Wine { }
```

Wir konsumieren Getränke:

```
static void demoConsumer1() {  
    Consumer<Wine> c = new Consumer<>() {  
        @Override  
        public void accept(Wine w) {  
            printGenericType(this);  
            System.out.println(w);  
        }  
    };  
    c.accept(new RedWine());  
}
```

Der Typ der generierten anonymen Klasse wird direkt aufgrund des Typs der Ziel-Variable `c` ermittelt: `Consumer<Wine>`.

Im Folgenden Beispiel ist die Zielvariable etwas anders definiert – und die anonyme Klasse wird mit einem expliziten Typ-Parameter definiert:

```
static void demoConsumer2() {  
    Consumer<? super Wine> c = new Consumer<Drink>() {  
        @Override  
        public void accept(Drink w) {  
            printGenericType(this);  
            System.out.println(w);  
        }  
    };  
    c.accept(new RedWine());  
}
```

An die Variable `c` kann sowohl ein `Consumer<Wine>` als auch ein `Consumer<Drink>` zugewiesen werden.

Was passiert nun, wenn statt des expliziten Typ-Parameters der Diamond verwendet wird?:

```
static void demoConsumer3() {  
    Consumer<? super Wine> c = new Consumer<>() {  
        @Override  
        public void accept(Wine w) {  
            printGenericType(this);  
            System.out.println(w);  
        }  
    };  
    c.accept(new RedWine());  
}
```

Der Compiler ermittelt hier denjenigen Parameter-Typ, der am besten zu `<? super Wine>` passt: nämlich `Wine`. Die `accept`-Methode muss daher mit `Wine` parametrisiert sein.

Wir betrachten das inverse Interface: `Supplier`.

```
static void demoSupplier1() {
    Supplier<Wine> s = new Supplier<>() {
        @Override
        public Wine get() {
            printGenericType(this);
            return new RedWine();
        }
    };
    Wine w = s.get();
    System.out.println(w);
}
```

Die generierte anonyme Klasse ist vom Typ `Supplier<Wine>`.

Zur Definition der Zielvariablen verwenden wir nun `? extends Wine`:

```
static void demoSupplier2() {
    Supplier<? extends Wine> s = new Supplier<RedWine>() {
        @Override
        public RedWine get() {
            printGenericType(this);
            return new RedWine();
        }
    };
    Wine w = s.get();
    System.out.println(w);
}
```

An die Variable `s` kann sowohl ein `Supplier<Wine>` als auch ein `Supplier<RedWine>` zugewiesen werden.

Was passiert, wenn statt des expliziten Typ-Parameters der Diamond verwendet wird?:

```
static void demoSupplier3() {
    Supplier<? extends Wine> s = new Supplier<>() {
        @Override
        public Wine get() {
            printGenericType(this);
            return new Wine();
        }
    }
}
```

```
};  
Wine w = s.get();  
System.out.println(w);  
}
```

Auch hier wird der beste Typ ermittelt, der zu `? extends Wine` passt: nämlich `Wine`. Die `get`-Methode muss daher `Wine` liefern.

Die `get`-Methode könnte aber auch "mehr" als `Wine` liefern – z.B. `RedWine` (weil eine überschreibende Methode einen spezifischeren Wert liefern darf als die überschriebene).

Typ-Parameter können auch aus dem verlangten Return-Typ einer Methode ermittelt werden:

```
class Range implements Iterable<Integer> {  
  
    public final int first;  
    public final int last;  
  
    public Range(int first, int last) {  
        this.first = first;  
        this.last = last;  
    }  
  
    @Override  
    public Iterator<Integer> iterator() {  
        return new Iterator<>() {  
            int current = first;  
            @Override  
            public boolean hasNext() {  
                return current <= last;  
            }  
            @Override  
            public Integer next() {  
                if (! hasNext())  
                    throw new NoSuchElementException();  
                return current++;  
            }  
        };  
    }  
}
```

Da `iterator` einen `Iterator<Integer>` liefern muss, muss der Typ-Parameter der anonymen, das Interface `Iterator` implementierenden Klasse `Integer` sein.

Eine kleine Anwendung:

```
static void demoRange() {  
    Range r = new Range(10, 12);  
    for(Integer e : r) {  
        System.out.println(e);  
    }  
}
```

Die Ausgaben:

10  
11  
12

## 5.2 Private Interface-Methoden

Seit Java 8 können in einem Interface öffentliche statische Methoden und öffentliche `default`-Methoden implementiert sein. Java 9 erlaubt nun auch die Implementierung sowohl privater statischer Methoden als auch privater Instanz-Methoden (man beachte, dass bei der Implementierung privater Instanz-Methoden das `default`-Schlüsselwort nicht(!) benutzt wird – anders also als bei der Implementierung öffentlicher Instanz-Methoden).

Im Folgenden ein Interface mit einer öffentlichen `default`-Methode, eine privaten Instanz-Methode, einer privaten statischen Methode und einer (öffentlichen) abstrakten Methode:

```
package jj.appl;

public interface Foo {
    public default void f() {
        this.g();
        g(this);
    }
    private void g() {
        this.h();
    }
    private static void g(Foo foo) {
        foo.h();
    }
    public abstract void h();
}
```

Das Interface kann etwa wie folgt implementiert werden:

```
package jj.appl;

public class Bar implements Foo {
    @Override
    public void h() {
        System.out.println("hello world");
    }
}
```

Natürlich kann in der `h`-Methode von `Bar` keine der beiden privaten Interface-Methoden aufgerufen werden.

Interface-Methoden können nun zwar `private` sein, nicht aber `protected`.





## 5.3 Erweiterung des resource-try

Im "alten" resource-try musste die Resource-Variable in der `try`-Überschrift definiert und initialisiert werden:

```
static void demo1() throws Exception {
    try (FileInputStream in = new FileInputStream(FILENAME))
    {
        final int first = in.read();
        System.out.println((char) first);
    }
    catch (final IOException e) {
        System.out.println(e);
    }
}
```

Im nun erweiterten resource-try kann auch eine bereits zuvor definierte und initialisierte Variable verwendet werden:

```
static void demo2() throws Exception {
    FileInputStream in = new FileInputStream(FILENAME);
    try (in) {
        final int first = in.read();
        System.out.println((char) first);
    }
    catch (final IOException e) {
        System.out.println(e);
    }
}
```

Welche Konsequenzen hat diese Erweiterung?

Angenommen, einer `copy`-Methode werden zwei Streams übergeben: ein `InputStream` und ein `OutputStream`. Beide Streams sollen in der `copy`-Methode geschlossen werden. Im "alten" Java mussten wir die Methode wie folgt implementieren:

```
private static void copy(InputStream in, OutputStream out) {
    try (InputStream i = in; OutputStream o = out) {
        int b;
        while ((b = i.read()) != -1) {
            o.write(b);
        }
    }
    catch (final IOException e) {
        throw new RuntimeException(e);
    }
}
```

```
    }  
}
```

Wir mussten also zwei neue Variablen definieren und initialisieren: `i` und `o` (wir mussten also zusätzliche "künstliche" Namen einführen).

Mit Java 9 lässt sich die Methode wie folgt reformulieren:

```
private static void copy(InputStream in, OutputStream out) {  
    try (in; out) {  
        int b;  
        while ((b = in.read()) != -1) {  
            out.write(b);  
        }  
    }  
    catch (final IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

Wir können in der `try`-Überschrift also einfach bereits initialisierte Variablen oder Parameter verwenden (hier: `in` und `out`). Allerdings müssen diese Variablen (resp. Parameter) als `final` oder effektiv `final` sein.

Beide obigen `copy`-Methoden könnten z.B. wie folgt genutzt werden:

```
static void demoCopy() throws Exception {  
    OutputStream out = new ByteArrayOutputStream();  
    copy(new FileInputStream(FILENAME), out);  
    System.out.println(out);  
}
```

## 5.4 Vargs

Bei der Benutzung von Vargs-Parametern können Fälle auftreten, in denen der Compiler nicht sicherstellen kann, dass diese Vargs korrekt genutzt werden. Der Compiler quittiert solche Benutzungen mit einer Warnung: *Type safety: Potential heap pollution via varargs parameter*. Dann kann die Annotation `@SafeVarargs` benutzt werden, solche Warnungen zu unterdrücken.

Die `@SafeVarargs`-Annotation konnte bislang nur bei statischen Methoden und Methoden, die als `final` deklariert sind, verwendet werden.

Neu in Java 9: `@SafeVarargs` kann nun auch für private Methoden verwendet werden. Sie kann nun also für alle Methoden verwendet werden, die nicht überschreibbar sind.

```
@SafeVarargs
public static void alpha(List<String>... args) { }

//@SafeVarargs // not allowed
public void beta(List<String>... args) { } // compiler-
warning

@SafeVarargs
public final void gamma(List<String>... args) { }

@SafeVarargs // Java 9
private void delta(List<String>... args) { }
```

Wir holen hier etwas weiter aus, um die Hintergründe von `@SafeVargs` zu klären.

Betrachten wir folgende Methoden:

```
public static List<String>[] nonVargs(List<String>[] args) {
    // ...
    return args;
}
```

```
public static List<String>[] vargs(List<String>... args) {
    // ...
    return args;
}
```

Um die erste Methode aufzurufen, müssen wir einen expliziten Array mit String-Listen erzeugen. Ein erster Versuch:

```
nonVargs(new List<String>[] {  
    Arrays.asList("rot", "gruen", "blau"),  
    Arrays.asList("red", "green", "blue")  
});
```

Leider funktioniert das nicht. Der Compiler beschwert sich: *Error: generic array creation*. Wie können also nur eine "rohe" Liste übergeben:

```
nonVargs(new List[] {  
    Arrays.asList("rot", "gruen", "blau"),  
    Arrays.asList("red", "green", "blue")  
});
```

Der Compiler ist immer noch nicht so recht zufrieden – lässt es aber bei einer Warnung bewenden.

Auch der zweiten Methode – `vargs` - können wir einen expliziten Array übergeben:

```
vargs(new List[] {  
    Arrays.asList("rot", "gruen", "blau"),  
    Arrays.asList("red", "green", "blue")  
});
```

Um die zweite Methode aufzurufen, können allerdings auch scheinbar beliebig viele String-Listen übergeben werden:

```
vargs(  
    Arrays.asList("rot", "gruen", "blau"),  
    Arrays.asList("red", "green", "blue")  
);
```

Hinter unserem Rücken fasst der Compiler diese Listen zu einem einzigen Array zusammen – der `vargs`-Methode werden also nicht "beliebige viele" Parameter übergeben, sondern nur ein einziger: ein Array – ein Array wiederum vom Typ `List[]`.

Da die generischen Informationen zur Kompilationszeit bekanntlich "verdampfen", sind die auch die formalen Parameter beider Methoden zur Laufzeit einfach vom Typ `List[]`.

Wir implementieren nun beide Methode nach demselben Schema:

```
public static List<String>[] nonVarargs(List<String>[] args)  
{  
    List[] array = args;  
    List<Integer> list = Arrays.asList(10, 20, 30);  
    array[0] = list;  
}
```

```
        return args;
    }
```

```
public static List<String>[] varargs(List<String>... args) {
    List[] array = args;
    List<Integer> list = Arrays.asList(10, 20, 30);
    array[0] = list;
    return args;
}
```

Und rufen `nonVarargs` wie folgt auf:

```
List<String>[] l1 = nonVarargs(new List[] {
    Arrays.asList("rot", "gruen", "blau"),
    Arrays.asList("red", "green", "blue")
});
l1.forEach(System.out::println);
```

Im Kontext des `forEach`-Aufrufs wird eine `ClassCastException` geworfen.

Dasselbe passiert bei folgendem Aufruf von `varargs`:

```
List<String>[] l2 = varargs(new List[] {
    Arrays.asList("rot", "gruen", "blau"),
    Arrays.asList("red", "green", "blue")
});
l2.forEach(System.out::println);
```

Und die Exception wird natürlich auch dann geworfen, wenn an `varargs` scheinbar beliebig viele Parameter übergeben werden:

```
List<String>[] l3 = varargs(
    Arrays.asList("rot", "gruen", "blau"),
    Arrays.asList("red", "green", "blue")
);
l3.forEach(System.out::println);
```

Auch die letzte Form der Parameterübergabe ist also problematisch – daher generiert der Compiler eine Warnung: *Type safety: A generic array of List<String> is created for a varargs parameter.* Wenn die `varargs`-Methode allerdings korrekt implementiert wäre, wäre diese Warnung überflüssig.

Wenn wir nun die `varargs`-Methode mit `@SafeVarargs` annotieren, versprechen wir, den übergebenen Array korrekt zu behandeln – mit der Konsequenz, dass der Compiler beim Aufruf dieser Methode keinerlei Warnungen mehr erzeugt.

Handelt es sich bei einer solchen Methode allerdings um eine öffentliche Methode, die überschrieben werden kann, können wir natürlich gar nichts versprechen...

## 5.5 Underscore

Der Underscore konnte bislang als Bezeichner verwendet werden:

```
int _ = 42;
```

Java 9 verbietet diese Benutzung. Der Unterstrich ist nun ein reserviertes Wort.

Er wird wahrscheinlich in der nächsten Java-Version als Name für unbenutzte Parameter verwendet werden können – etwa für unbenutzte Parameter von Lambda-Ausdrücken (wie z.B. in Scala).

Man wird in (ferner?) Zukunft dann wahrscheinlich folgendes Zeilen schreiben können:

```
// heute noch verboten:  
  
    BiConsumer<String,String> c = (x, _) ->  
System.out.println(x);  
    c.accept("Brot", "Wein");
```



## 6 Erweiterungen der Standardbibliothek

Erweiterungen der Standardbibliothek betreffen die `Collections`, die `Process`-Klasse, die `Streams`, die `Optionals` und die `CompletableFutureS`.

Neu ist die Klassen `StackWalker` und ein vereinheitlichtes Logging.

Und schließlich sind `Strings` etwas kompakter geworden.

### Inhalte

- Java 9 führt neue statische Factory-Methoden für `Collections` ein – in den Interfaces `List`, `Set` und `Map`.
- Das bislang äußerst rudimentäre `Process`-API ist um weitere Interfaces (`ProcessHandle`, `ProcessHandle.Info`) und um neue Methoden der `Process`-Klasse erweitert worden.
- Im `Stream`-Interface wurden vier neue Methoden aufgenommen – zwei `Intermediate`-Operationen und zwei `Factory`-Methoden
- Die `Optional`-Klasse wurde erweitert – u.a. um eine Methode, die einen `Stream` liefert (entweder einen leeren `Stream` oder einen `Stream`, der genau ein einziges Element enthält).
- Java 9 führt eine neue Klasse `StackWalker` ein, die es erlaubt, effizient Informationen über den aktuellen Stack zu ermitteln.
- Ein neues `Logger`-API ermöglicht vereinheitlichtes Logging. Auch die VM selbst benutzt dieses API.
- Das `CompletableFuture`-Interface aus dem `concurrent`-Paket ist um einige Methoden erweitert worden (Timeout- und Failure-Handling).
- Java 9 führt "kompakte Strings" ein. Besteht ein String ausschließlich aus 1-Byte-Zeichen, so wird jedes dieser Zeichen in einem `byte` gespeichert (bislang wurde jeweils ein `char` benötigt)

## 6.1 Initialisierung von Collections

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

Das `List`-Interface enthält nun 11 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)
```

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)
```

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> Set<K,V> of()  
static <K,V> Set<K,V> of(K k1, V v1)  
static <K,V> Set<K,V> of(K k1, V v1, K k2, V v2)  
// ...  
static <K,V> Set<K,V> of(K k1, V v1, K k2, V v2, ... K k10, V v10)
```

(Nun wird vielleicht verständlich, warum bei den `of`-Methoden von `List` und `Set` keine Varargs verwendet wurden...)

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 nun 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
```



```
private static void printProcessHandleInfo(ProcessHandle.Info
info) {
    S.o.p("ProcessHandle.Info");
    S.o.p("\tcommand          = " + info.command());
    S.o.p("\tcommandLine       = " + info.commandLine());
    S.o.p("\targuments          = " + info.arguments());
    S.o.p("\tstartInstant         = " + info.startInstant());
    S.o.p("\ttotalCpuDuration      = " + info.totalCpuDuration());
    S.o.p("\tuser                  = " + info.user());
}
```

Die folgende Methode ermittelt den `ProcessHandle` zum aktuellen Prozess und gibt die Informationen zu diesem `ProcessHandle` und dessen `ProcessHandle.Info` aus:

```
static void demoProcessHandle() {
    ProcessHandle handle = ProcessHandle.current();
    printProcessHandle(handle);
    ProcessHandle.Info info = handle.info();
    printProcessHandleInfo(info);
}
```

Die Ausgaben:

```
ProcessHandle
  pid          = 4772
  isAlive       = true
  parent        = Optional[8624]
  children      = []
  descendants    = []
  supports...   = false
ProcessHandle.Info
  command       = Optional[C:\...\Java\jdk-9\bin\java.exe]
  commandLine   = Optional.empty
  arguments     = Optional.empty
  startInstant  = Optional[2017-10-22T07:24:37.521Z]
  totalCpuDuration = Optional[PT0.468003S]
  user          = Optional[Nowak-PC\Nowak]
```

Die folgende Methode gibt die alle Prozesse aus, deren Ausführungskommando irgendetwas mit "java" oder "eclipse" zu tun haben:

```
static void demoAllProcesses() throws Exception {
    Stream<ProcessHandle> handles =
ProcessHandle.allProcesses();
    handles
        .filter(h -> filterJavaAndEclipse(h))
```

```
        .forEach(h -> printProcessHandle(h));  
    }
```

```
    private static boolean filterJavaAndEclipse(ProcessHandle  
handle) {  
        ProcessHandle.Info info = handle.info();  
        Optional<String> command = info.command();  
        if (!command.isPresent())  
            return false;  
        return command.get().contains("java") ||  
            command.get().contains("eclipse");  
    }
```

### Die Ausgaben:

```
ProcessHandle  
  pid          = 7428  
  isAlive       = true  
  parent       = Optional[5520]  
  children     = [8624]  
  descendants   = [8624, 4772]  
  supports...  = false  
ProcessHandle  
  pid          = 8624  
  isAlive       = true  
  parent       = Optional[7428]  
  children     = [4772]  
  descendants   = [4772]  
  supports...  = false  
ProcessHandle  
  pid          = 4772  
  isAlive       = true  
  parent       = Optional[8624]  
  children     = []  
  descendants   = []  
  supports...  = false
```

Die folgende Methode startet den Windows-Taschenrechner und gibt den erzeugten Process **aus**:

```
    static void demoExec() throws Exception {  
        // Runtime.getRuntime().exec(new String[]  
{ "calc.exe" });  
        Process process = new ProcessBuilder("calc.exe").start();  
        ProcessHandle.Info info = process.info();  
        printProcessHandleInfo(info);  
        Optional<ProcessHandle> handle =  
ProcessHandle.of(process.pid());
```



```
        if (handle.isPresent())  
            printProcessHandle(handle.get());  
    }
```

Wir starten den Taschenrechner und warten auf dessen Terminierung:

```
static void demoExecWaitFor() throws Exception {  
    Process process = new ProcessBuilder("calc.exe").start();  
    System.out.println(process);  
    System.out.println("waiting...");  
    int result = process.waitFor();  
    System.out.println("Finished: " + result);  
}
```

Wir starten den Taschenrechner erneut und warten erneut auf dessen Terminierung – diesmal aber mittels eines `CompletableFutureS`:

```
static void demoExecFuture() throws Exception {  
    Process process = new ProcessBuilder("calc.exe").start();  
    System.out.println(process);  
    final CompletableFuture<Process> future =  
process.onExit();  
    System.out.println("waiting...");  
    Process p = future.get();  
    System.out.println(p);  
    System.out.println("Finished");  
}
```

Wir starten den Taschenrechner und terminieren ihn programmtechnisch:

```
static void demoDestroy() throws Exception {  
    Process process = new ProcessBuilder("calc.exe").start();  
    Optional<ProcessHandle> handle =  
ProcessHandle.of(process.pid());  
    Thread.sleep(2000);  
    if (handle.isPresent()) {  
        boolean done = handle.get().destroy();  
        System.out.println("Destroyed: " + done);  
    }  
}
```

## 6.3 Stream

Das in Java 8 eingeführte `Stream`-Interface ist u.a. um folgende zwei Methoden erweitert worden:

```
default Stream<T> takeWhile(Predicate<? super T> predicate)
default Stream<T> dropWhile(Predicate<? super T> predicate)
```

Bei beiden Methoden handelt es sich um intermediate Operationen.

`dropWhile` "überliest" alle Elemente der Eingabe, bis das aktuelle Element einer bestimmten Bedingung genügt:

```
static void demoDropWhile() {
    Stream.of(10, 11, 12, 13, 14, 15)
        .dropWhile(v -> v < 13)
        .forEach(System.out::println);
}
```

Die Ausgaben:

```
13
14
15
```

`takeWhile` reicht alle Elemente an die nächste Stream-Station weiter, bis das aktuelle Element einer bestimmten Bedingung genügt:

```
static void demoTakeWhile() {
    Stream.of(10, 11, 12, 13, 14, 15)
        .takeWhile(v -> v < 12)
        .forEach(System.out::println);
}
```

Die Ausgaben:

```
10
11
```

Natürlich können `dropWhile` und `takeWhile` kombiniert werden:

```
static void demoDropWhileTakeWhile() {
    Stream.of(10, 11, 12, 13, 14, 15)
        .dropWhile(v -> v < 12)
        .takeWhile(v -> v < 15)
```

```
        .forEach(System.out::println);  
    }
```

Die Ausgaben:

```
12  
13  
14
```

Im Folgenden Beispiel geht's darum, den Inhalt des Bodies eines HTML-Dokuments auszugeben:

```
static void demoHtml() {  
    Stream.of(  
        "<html>",  
        "<head>",  
        "    <title>Foo</title>",  
        "</head>",  
        "<body>",  
        "    <h1>Foo</h1>",  
        "    <p>Bar</p>",  
        "</body>",  
        "</html>")  
        .dropWhile(s -> !s.equals("<body>"))  
        .skip(1)  
        .takeWhile(s -> !s.equals("</body>"))  
        .forEach(System.out::println);  
}
```

Die Ausgaben:

```
<h1>Foo</h1>  
<p>Bar</p>
```

Das Stream-Interface enthält nun zusätzlich auch zwei statische Factory-Methoden:

```
static <T> Stream<T> ofNullable(T t)
```

```
static <T> Stream<T> iterate(T seed,  
    Predicate<? super T> hasNext, UnaryOperator<T> next)
```

`ofNullable` kann wie folgt genutzt werden:

```
static void demoOfNullable() {  
    Stream<String> s1 = Stream.ofNullable((String)null);  
    s1.forEach(System.out::println);  
    Stream<String> s2 = Stream.ofNullable("Hello");  
}
```

```
s2.forEach(System.out::println);  
}
```

Der erste Aufruf von `forEach` produziert eine leere Ausgabe; der zweite Aufruf produziert "Hello".

Mittels der neuen `iterate`-Methode kann die "alte" for-Schleife simuliert werden:

```
static void demoIterate() {  
    Stream<Integer> s = Stream.iterate(5, i -> i < 10, i -> i  
+ 2);  
    s.forEach(System.out::println);  
}
```

Die Ausgaben:

```
5  
7  
9
```

## 6.4 Optional

Die in Java 8 eingeführte Klasse `Optional` ist um drei Methoden erweitert worden:

```
public Stream<T> stream()
public Optional<T> or(Supplier<? extends Optional<? extends T>>
supplier)
public void ifPresentOrElse(Consumer<? super T> action, Runnable
emptyAction)
```

Zunächst zur `stream`-Methode.

Im Falle, dass das `Optional` einen Wert hat, liefert die `stream`-Methode einen `Stream` mit genau diesem einen Wert zurück; hat das `Optional` keinen Wert, so wird ein leerer `Stream` geliefert.

Ein Beispiel:

```
static void demoStream1() {
    Optional<String> s1 = Optional.empty();
    s1.stream().forEach(System.out::println);

    Optional<String> s2 = Optional.of("Hello");
    s2.stream().forEach(System.out::println);
}
```

Nur der zweite `forEach`-Aufruf produziert eine Ausgabe: `Hello`.

Ein weiteres Beispiel:

```
static void demoStream2() {
    Stream.of(Optional.of("red"), Optional.empty(),
Optional.of("blue"))
        .flatMap(opt -> opt.stream())
        .forEach(str -> System.out.println(str));
    // oder einfacher:
    Stream.of(Optional.of("red"), Optional.empty(),
Optional.of("blue"))
        .flatMap(Optional::stream)
        .forEach(System.out::println);
}
```

Die Ausgaben:

```
red
blue
red
blue
```

Die Methode kann also auch dazu benutzt werden, um einen `Stream` mit optionalen Elementen in eine `Stream` zu transformieren, der nurmehr die Werte der gefüllten `Optional`-Objekte liefert.

Zur neuen `or`-Methode. Diese Methode kann benutzt werden, um im Falle eines leeren `Optional` lazy ein anderes `Optional` zu erzeugen und zurückzuliefern.

Ein Beispiel:

```
static void demoOr() {
    Optional<String> str1 = Optional.ofNullable((String)null)
        .or(() -> Optional.of("Hello"));
    System.out.println(str1);

    Optional<String> str2 = Optional.ofNullable("World")
        .or(() -> Optional.of("Hello"));
    System.out.println(str2);
}
```

Die Ausgaben:

```
Optional[Hello]
Optional[World]
```

Und schließlich zur Methode `isPresentOrElse`.

An `isPresentOrElse` wird ein `Consumer` und ein `Runnable` übergeben. Enthält das `Optional` einen Wert, so wird der `Consumer` aufgerufen (mit eben diesem Wert); ansonsten wird das `Runnable` aufgerufen.

Ein Beispiel:

```
static void demoIfPresentOrElse() {
    Optional<String> o1 = Optional.of("Hello");
    o1.ifPresentOrElse(
        str -> System.out.println(str),
        () -> System.out.println("not present"));

    Optional<String> o2 = Optional.empty();
    o2.ifPresentOrElse(
        str -> System.out.println(str),
```

```
        () -> System.out.println("not present"));
    }
```

Die Ausgaben:

```
Hello
not present
```

## 6.5 StackWalker

Java 9 führt eine neue Klasse `StackWalker` und eine weitere innere Klasse `StackFrame` ein, die es erlauben, effizient die Frames des aktuellen Stacks zu ermitteln. Damit werden die alte Methode `Thread.getStackTrace` und der Typ `StackTraceElement` obsolet.

`StackWalker` ist im `java.lang`-Paket enthalten:

```
import java.lang.StackWalker;
import java.lang.StackWalker.StackFrame;
```

Eine überladene statische `getInstance`-Methode dient als Factory für `StackWalker`:

```
static StackWalker getInstance()
static StackWalker getInstance(StackWalker.Option option)
static StackWalker getInstance(Set<StackWalker.Option> options)
```

Die `forEach`-Methode übergibt jeden `StackFrame` an einen `Consumer`, der beim Aufruf, von `forEach` übergeben wird:

```
void forEach(Consumer<? super StackFrame> action)
```

Der `walk`-Methode wird eine `Function` übergeben, dessen `apply`-Methode ein `Stream` übergeben wird, aus dem alle `StackFrames` ausgelesen werden können. Diese `Function` kann die von dem `Stream` gelieferten `StackFrames` auslesen und z.B. in einer Liste sammeln, welche dann als Resultat zurückgegeben wird. Und eben dieses Resultat der `Function` wird dann auch von `walk` zurückgegeben.

```
<T> T walk(Function<? super Stream<StackFrame>, ? extends T>
function)
```

Die `demo`-Methoden benutzen die Klassen `Alpha`, `Beta` und `Gamma` – um jeweils eine Hierarchie von Methodenaufrufen aufzubauen (also den Stack, der dann jeweils analysiert wird). Am Ende der Aufrufhierarchie wird dann ein `Runnable` gestartet, welches diese Analyse vornimmt).

Die statische `alpha`-Methode von `Alpha` ruft die statische `beta`-Methode der `Beta`-Klasse auf – wobei das an `alpha` übergebene `Runnable` an `beta` weitergereicht wird:

```
class Alpha {
    static void alpha(Runnable runnable) {
        Beta.beta(runnable);
    }
}
```



```
}
```

Die `beta`-Methode von `Beta` ruft die `gamma`-Methode von `Gamma` auf – und reicht das ihr übergebene `Runnable` an die aufgerufene `gamma`-Methode weiter:

```
class Beta {  
    static void beta(Runnable runnable) {  
        Gamma.gamma(runnable);  
    }  
}
```

Die `gamma`-Methode von `Gamma` schließlich führt das ihr übergebene `Runnable` aus:

```
class Gamma {  
    static void gamma(Runnable runnable) {  
        runnable.run();  
    }  
}
```

Die `main`-Methode ruft zwei `demo`-Methoden auf:

```
public static void main(String[] args) {  
    demoForEach();  
    demoWalk();  
}
```

In der ersten `demo`-Methode wird ein `Runnable` erzeugt, dessen `run`-Methode einen `StackWalker` erzeugt und auf diesen die `forEach`-Methode aufruft – wobei die `accept`-Methode des an `forEach` übergebenen `Consumer`s den jeweiligen `StackFrame` ausgibt. Dieses `Runnable` wird an die `Alpha.alpha`-Methode übergeben:

```
static void demoForEach() {  
    Alpha.alpha(() -> {  
        StackWalker walker = StackWalker.getInstance();  
        walker.forEach((StackFrame f) ->  
System.out.println(f));  
    });  
}
```

Zum Zeitpunkt des Aufrufs der `StackWalker.forEach`-Methode liegen u.a. die `main`-Methode, die `demoForEach`-Methode und die `alpha`, `beta` und `gamma`-Methoden auf dem Stack (und noch eine weitere vom Compiler generierte "Lambda"-Methode:

```
jj.appl.Application.lambda$demoForEach$1 (Application.java:53)  
jj.appl.Gamma.gamma (Application.java:30)  
jj.appl.Beta.beta (Application.java:24)
```

```
jj.appl.Alpha.alpha(Application.java:18)
jj.appl.Application.demoForEach(Application.java:51)
jj.appl.Application.main(Application.java:37)
```

Die zweite `demo`-Methode demonstriert die Benutzung von `StackWalker.walk`. Sie übergibt an `walk` eine Function, welche die vom Stream gelieferten `StackFrames` in einer `List` sammelt und diese zurückliefert:

```
static void demoWalk() {
    Alpha.alpha(() -> {
        StackWalker walker = StackWalker.getInstance();
        List<StackFrame> stack = walker.walk(
            (Stream<StackFrame> s) ->
                s.collect(Collectors.toList()));
        for (StackFrame f : stack) {
            System.out.println(f);
        }
    });
}
```

Die Ausgaben sehen ähnlich aus wie bei der ersten `demo`-Methode:

```
jj.appl.Application.lambda$demoWalk$3(Application.java:61)
jj.appl.Gamma.gamma(Application.java:30)
jj.appl.Beta.beta(Application.java:24)
jj.appl.Alpha.alpha(Application.java:18)
jj.appl.Application.demoWalk(Application.java:59)
jj.appl.Application.main(Application.java:38)
```

Die folgende `demo`-Methode übergibt an `walk` eine Function, die genau denjenigen Stream liefert, der ihr übergeben wurde: `s -> s`. Die `walk`-Methode liefert dann natürlich genau diesen Stream zurück. Der von `walk` gelieferte Stream wird dann anschließend (also außerhalb(!) des Kontextes von `walk`) verarbeitet. Der Compiler ist zufrieden – aber zur Laufzeit wird eine `IllegalStateException` geworfen:

```
static void demoWalkIllegal() {
    try {
        Alpha.alpha(() -> {
            StackWalker walker = StackWalker.getInstance();
            Stream<StackFrame> stream = walker.walk(s -> s);
            stream.forEach(f -> System.out.println(f));
        });
    }
    catch (Exception e) {
        System.out.println(e);
    }
}
```

```
}
```

```
java.lang.IllegalStateException:  
    This stack stream is not valid for walking.
```

Der Leser / die Leserin überlege sich den Grund dieser Exception!

Der `StackWalker.getInstance`-Methode kann die Option `RETAIN_CLASS_REFERENCE` übergeben. Dann enthält jeder gelieferte `StackFrame` zusätzlich die Class-Referenz der Klasse der aktuellen Methode.

```
static void demoWithClasses() {  
    Alpha.alpha() -> {  
        StackWalker walker = StackWalker.getInstance(  
            StackWalker.Option.RETAIN_CLASS_REFERENCE);  
        walker.forEach(f -> {  
            System.out.println(f);  
            System.out.println("\t" + f.getDeclaringClass());  
        });  
    };  
};  
}
```

Die Ausgaben:

```
jj.appl.Application.lambda$demoWithClasses$8 (Application.java:86)  
    class jj.appl.Application  
jj.appl.Gamma.gamma (Application.java:30)  
    class jj.appl.Gamma  
jj.appl.Beta.beta (Application.java:24)  
    class jj.appl.Beta  
jj.appl.Alpha.alpha (Application.java:18)  
    class jj.appl.Alpha  
jj.appl.Application.demoWithClasses (Application.java:84)  
    class jj.appl.Application  
jj.appl.Application.main (Application.java:40)  
    class jj.appl.Application
```

An `StackWalker.getInstance` kann auch ein Set von Options übergeben werden (auf die genaue Bedeutung dieser Options gehen wir hier nicht näher ein):

```
static void demoOptions() {  
    Alpha.alpha() -> {  
        StackWalker walker = StackWalker.getInstance(  
Set.of(StackWalker.Option.RETAIN_CLASS_REFERENCE,  
StackWalker.Option.SHOW_HIDDEN_FRAMES,
```

```
StackWalker.Option.SHOW_REFLECT_FRAMES));
    walker.forEach(f -> System.out.println(f));
});
}
```

Die Ausgaben (man beachte u.a. die zweite Zeile von oben):

```
jj.appl.Application.lambda$demoOptions$10(Application.java:100)
jj.appl.Application$$Lambda$16/1644443712.run(Unknown Source)
jj.appl.Gamma.gamma(Application.java:30)
jj.appl.Beta.beta(Application.java:24)
jj.appl.Alpha.alpha(Application.java:18)
jj.appl.Application.demoOptions(Application.java:95)
jj.appl.Application.main(Application.java:41)
```

Wir möchten nur die jeweils beiden obersten Elemente des Stacks anzeigen - und nur für diesen beiden oberen Elemente werden im Folgenden auch `StackFrame`-Objekte erzeugt werden!. Wir übergeben an `walk` eine Function, die den ihr übergebenen Stream nutzt, um via `limit` einen neuen Stream zu erzeugen, der nur zwei Elemente liefert – und um diese Elemente dann in einer Liste zu sammeln:

```
static void demoLimit() {
    Alpha.alpha(() -> {
        StackWalker walker = StackWalker.getInstance();
        List<StackFrame> stack = walker.walk(
            s -> s.limit(2).collect(Collectors.toList()));
        for (StackFrame f : stack) {
            System.out.println(f);
        }
    });
}
```

Die Ausgaben:

```
jj.appl.Application.lambda$demoLimit$12(Application.java:110)
jj.appl.Gamma.gamma(Application.java:30)
```

Wir wollen nun die beiden obersten Elemente gerade nicht(!) sehen – aber alle unteren. Statt den neuen Stream mittels `limit` zu erzeugen, erzeugen wir ihn mittels `skip`:

```
static void demoSkip() {
    Alpha.alpha(() -> {
        StackWalker walker = StackWalker.getInstance();
        List<StackFrame> stack = walker.walk(
            s -> s.skip(2).collect(Collectors.toList()));
        for (StackFrame f : stack) {
```

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

Die Ausgaben:

```
jj.appl.Beta.beta(Application.java:24)
jj.appl.Alpha.alpha(Application.java:18)
jj.appl.Application.demoSkip(Application.java:119)
jj.appl.Application.main(Application.java:43)
```

Wir wollen nur die Aufrufe derjenigen Methoden sehen, die zur Klasse `Alpha` gehören. Wir benutzen `filter` (man beachte die an `getInstance` übergebene `Option`!):

```
static void demoFilterClasses() {
    Alpha.alpha(() -> {
        StackWalker walker = StackWalker.getInstance(
            StackWalker.Option.RETAIN_CLASS_REFERENCE);
        List<StackFrame> stack = walker.walk(
            s -> s
                .filter(f -> f.getDeclaringClass() ==
Alpha.class)
                .collect(Collectors.toList()));
        for (StackFrame f : stack) {
            System.out.println(f);
        }
    });
}
```

Die Ausgaben:

```
jj.appl.Alpha.alpha(Application.java:18)
```

Mittels des Aufrufs der Methode `StackWalker.getCallerClass()` kann die Klasse derjenigen Methode ermittelt werden, in welcher der `StackWalker` ausgeführt wird:

```
static void demoCallerClass() {
    Alpha.alpha(() -> {
        StackWalker walker = StackWalker.getInstance(
            StackWalker.Option.RETAIN_CLASS_REFERENCE);
        Class<?> cls = walker.getCallerClass();
        System.out.println(cls);
    });
}
```

## Die Ausgabe:

```
class jj.appl.Gamma
```

Die folgende Methode gibt die Eigenschaften eines `StackFrame`s aus – sie benutzt dazu eine entsprechende `print`-Methode:

```
static void demoStackFrame() {  
    Alpha.alpha() -> {  
        StackWalker walker = StackWalker.getInstance(  
            StackWalker.Option.RETAIN_CLASS_REFERENCE);  
        walker.forEach(f -> print(f));  
    };  
}
```

Die `print`-Methode gibt u.a. den Namen der aktuellen Methode, den Namen der Klasse dieser Methode, den Dateinamen und die Zeilennummer aus. Um die `declaringClass` auszugeben, muss die `Option.RETAIN_CLASS_REFERENCE` gesetzt sein:

```
static void print(StackFrame f) {  
    System.out.println(f);  
    System.out.println("\tClassName          = " +  
f.getClassName());  
    System.out.println("\tDeclaringClass = " +  
f.getDeclaringClass());  
    System.out.println("\tMethodName          = " +  
f.getMethodName());  
    System.out.println("\tFileName           = " +  
f.getFileName());  
    System.out.println("\tLineNumber         = " +  
f.getLineNumber());  
}
```

## Die Ausgaben (ein kleiner Auszug!):

```
jj.appl.Application.lambda$demoStackFrame$20(Application.java:153)  
  ClassName          = jj.appl.Application  
  DeclaringClass     = class jj.appl.Application  
  MethodName         = lambda$demoStackFrame$20  
  FileName           = Application.java  
  LineNumber         = 153  
jj.appl.Gamma.gamma(Application.java:30)  
...
```

Ein Performance-Test zeigt übrigens, dass die `StackWalker`-Klasse performanter ist als die "alte" `Thread.getStackTrace()`-Variante – zumal dann, wenn wir mit `walk` in Kombination mit `limit` arbeiten (s. hierzu den Performance-Test im Workspace).

Abschließend sei ein kleiner `Tracer` vorgestellt.

Angenommen, die Methode `demoTracer` ruft eine `foo`-Methode auf, die ihrerseits eine `bar`-Methode aufruft. Die Ein- und Ausstiege aus den Methoden sollen protokolliert werden:

```
package jj.appl;

public class Application {
    ...

    static void demoTracer() {
        int result = foo(42, "Hello");
        System.out.println("result = " + result);
    }

    static int foo(int x, String s) {
        try (Tracer tracer = new Tracer(x, s)) {
            tracer.trace("foo starts work...");
            bar(null);
            tracer.trace("foo terminates work...");
            return tracer.value(2 * x);
        }
    }

    static void bar(Object obj) {
        try (Tracer tracer = new Tracer(obj)) {
            tracer.trace("bar working...");
        }
    }
}
```

Die Ausgaben:

```
>> jj.appl.Application.foo(42, Hello)
    foo starts work...
    >> jj.appl.Application.bar(null)
        bar working...
    << jj.appl.Application.bar
        foo terminates work...
<< jj.appl.Application.foo -> 84
result = 84
```

Das Protokoll verdeutlicht u.a. auch die Aufrufhierarchie.

Hier die Klasse `Tracer`:

```
package util;

import java.util.Arrays;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Tracer implements AutoCloseable {

    private static final StackWalker walker =
        StackWalker.getInstance();
    private static int indent;

    private final String name;
    private Object value;

    public Tracer(Object... args) {
        StackWalker.StackFrame frame = walker.walk(s ->
            s.limit(2).collect(Collectors.toList())).get(1);
        this.name = frame.getClassName() + "." +
frame.getMethodName();
        Stream<String> stream = Arrays.stream(args)
            .map(arg -> String.valueOf(arg));
        String argString = String.join(", ",
            stream.collect(Collectors.toList()));
        this.trace(">> " + name + "(" + argString + ")");
        indent++;
    }

    public void trace(String msg) {
        for(int i = 0; i < indent; i++)
            System.out.print("\t");
        System.out.println(msg);
    }

    public <T> T value(T value) {
        this.value = value;
        return value;
    }

    @Override
    public void close() {
        indent--;
    }
}
```



```
        String s = this.value == null ? "" : " -> " +  
            String.valueOf(this.value);  
        this.trace("<< " + name + s);  
    }  
}
```

## 6.6 Vereinheitlichtes Logging

Java 9 stellt ein Logger-API bereit, welches auch die VM zu Logging-Zwecken nutzt. Das API kann mit beliebigen Implementierungen arbeiten.

`Logger` ist das neue Interface, welches als ein inneres Interface der Klasse `System` definiert ist. Dieses Interface definiert seinerseits einen inneren `enum`-Typ namens `Level`:

```
import java.lang.System.Logger;  
import java.lang.System.Logger.Level;
```

Das `Logger`-Interface bietet u.a. folgende überladene `log`-Methoden:

```
void log(Level level, Object obj)  
void log(Level level, String str)  
void log(Level level, Supplier<String> supplier)  
void log(Level level, String format, Object... params)  
void log(Level level, String str, Throwable t)  
void log(Level level, Supplier<String> supplier, Throwable t)  
void log(Level level, ResourceBundle bundle,  
         String format, Object... params)  
void log(Level level, ResourceBundle bundle, String str, Throwable t)
```

Beim der "alten" `java.util.logging`-Lösung existierten folgende Log-Levels:

ALL, FINER, FINE, INFO, WARNING, SEVERE, OFF

Der neue `Level`-enum definiert folgende Levels:

ALL, TRACE, DEBUG, INFO, WARNING, ERROR, OFF

Das folgende Beispiel zeigt, wie die Standard-Implementierung des `Loggers` ermittelt und verwendet werden kann – es demonstriert insbesondere die Verwendung verschiedener `log`-Methoden:

```
System.Logger logger = System.getLogger("");  
  
System.out.println(logger.isLoggable(System.Logger.Level.ERROR));  
  
logger.log(Level.ERROR, "Water in drive A:");  
  
logger.log(Level.INFO, new Point(42, 77));  
  
logger.log(Level.INFO, "i = {0} s = {1}", 42, "Hello");
```

```
        logger.log(Level.INFO, "exception", new
IllegalStateException());

        logger.log(Level.INFO, () -> "Hello " + 42 + " World " +
3.14);
```

### Die Ausgaben:

```
true
Okt. 29, 2017 6:48:39 VORM. jj.appl.Application main
SCHWERWIEGEND: Water in drive A:
Okt. 29, 2017 6:48:39 VORM. jj.appl.Application main
INFORMATION: java.awt.Point[x=42,y=77]
Okt. 29, 2017 6:48:39 VORM. jj.appl.Application main
INFORMATION: i = 42 s = Hello
Okt. 29, 2017 6:48:39 VORM. jj.appl.Application main
INFORMATION: exception
java.lang.IllegalStateException
    at jj.appl.Application.main(Application.java:22)
Okt. 29, 2017 6:48:39 VORM. jj.appl.Application main
INFORMATION: Hello 42 World 3.14
```

Wie können eine eigene `Logger`-Klasse schreiben – `SimpleLogger`. Wir müssen vier Methoden implementieren: `getName`, `isLoggable` und zwei überladene `log`-Methoden:

```
package jj.mod;

import java.lang.System.Logger;
import java.util.ResourceBundle;
import static java.text.MessageFormat.format;

public class SimpleLogger implements Logger {

    @Override
    public String getName() {
        return "SimpleLogger";
    }

    @Override
    public boolean isLoggable(Level level) {
        switch (level) {
            case OFF:
            case TRACE:
            case DEBUG:
            case INFO:
            case WARNING:
```

```
        case ERROR:
        case ALL:
        default:
            return true;
    }
}

@Override
public void log(Level level, ResourceBundle bundle,
                String msg, Throwable thrown) {
    System.out.printf("%s: %s - %s%n", level, msg, thrown);
}

@Override
public void log(Level level, ResourceBundle bundle,
                String format, Object... params) {
    System.out.printf("%s: %s%n", level, format(format,
params));
}
}
```

Wir schreiben eine zweite `Logger`-Klasse, die etwas geschwätziger ist als `SimpleLogger` – die Klasse `VerboseLogger`:

```
package jj.mod;
// ...
public class VerboseLogger implements Logger {

    @Override
    public String getName() {
        return "VerboseLogger";
    }

    @Override
    public boolean isLoggable(Level level) {
        return true;
    }

    @Override
    public void log(Level level, ResourceBundle bundle,
                    String msg, Throwable thrown) {
        System.out.printf("VerboseLogger [%s]: %s - %s%n",
            level, msg, thrown);
    }

    @Override
```

```
        public void log(Level level, ResourceBundle bundle,
                        String format, Object... params) {
            System.out.printf("VerboseLogger [%s]: %s%n",
                             level, format(format, params));
        }
    }
}
```

Wollen wir nun eine dieser beiden Implementierungen verwenden, so müssen wir eine von `LoggerFinder` abgeleitete Klasse implementieren. Dabei überschreiben wir die `getLogger`-Methode, der u.a. eine Name verwendet wird. Diesen Namen nutzen wir, um entweder einen `SimpleLogger` oder einen `VerboseLogger` zu erzeugen und zurückzuliefern:

```
package jj.mod;

import java.lang.System.Logger;
import java.lang.System.LoggerFinder;

public class MyLoggerFinder extends LoggerFinder {

    @Override
    public Logger getLogger(String name, Module module) {
        System.out.println(this.getClass().getSimpleName() +
                           ".getLogger(" + name + ", " + module + ")");
        if (name.equals("VerboseLogger"))
            return new VerboseLogger();
        if (name.equals("SimpleLogger"))
            return new SimpleLogger();
        return null;
    }
}
```

Wir bauen ein Modul `jj.mod`, welches für den abstrakten Typ `LoggerFinder` unsere `MyLoggerFinder`-Implementierung bereitstellt:

```
module jj.mod {
    provides java.lang.System.LoggerFinder
        with jj.mod.MyLoggerFinder;
}
```

Wir bauen ein `jj.appl`-Modul, welches von `jj.mod` abhängig ist:

```
module jj.appl {
    requires jj.mod;
}
```

In diesem Modul definieren wir die `demo`-Methoden. Die erste nutzt den `SimpleLogger`, die zweite den `VerboseLogger`:

```
static void demoSimpleLogger() {  
    Logger logger = System.getLogger("SimpleLogger");  
    System.out.println(logger.getName());  
    logger.log(Level.ERROR, "Water in drive A:");  
    logger.log(Level.INFO, "Nice day");  
}
```

Die Ausgaben:

```
MyLoggerFinder.getLogger(SimpleLogger, module jj.appl)  
SimpleLogger  
ERROR: Water in drive A:  
INFO: Nice day
```

```
static void demoVerboseLogger() {  
    Logger logger = System.getLogger("VerboseLogger");  
    System.out.println(logger.getName());  
    logger.log(Level.ERROR, "Water in drive A:");  
    logger.log(Level.INFO, "Nice day");  
}
```

Die Ausgaben:

```
MyLoggerFinder.getLogger(VerboseLogger, module jj.appl)  
VerboseLogger  
VerboseLogger [ERROR]: Water in drive A:  
VerboseLogger [INFO]: Nice day
```

Wir können den `LoggerFinder` auch explizit über den `ServiceLoader` ermitteln. Dazu muss die `module-info` um folgenden `uses`-Eintrag erweitert werden:

```
uses java.lang.System.LoggerFinder;
```

Dann kann der `ServiceLoader` wie folgt genutzt werden:

```
final ServiceLoader<LoggerFinder> loader =  
    ServiceLoader.load(LoggerFinder.class);  
for (LoggerFinder finder : loader) {  
    System.out.println(finder);  
}
```

Die Ausgabe:

```
==> jj.mod.MyLoggerFinder@724af044
```



## 6.7 CompletableFuture

Die in Java-8 eingeführte `CompletableFuture`-Klasse ist in einige Methoden erweitert worden – insbesondere um solche, die mit Timeouts und Failures zu tun haben.

Hier zunächst eine Übersicht:

```
public class CompletableFuture<T> ... {  
  
    // ...  
  
    public CompletableFuture<T> completeAsync(  
        Supplier<? extends T> supplier)  
    public CompletableFuture<T> completeOnTimeout(  
        T value, long timeout, TimeUnit unit)  
    public CompletableFuture<T> orTimeout(long timeout, TimeUnit  
unit)  
  
    public static <U> CompletableFuture<U> completedFuture(U  
value)  
    public static <U> CompletableFuture<U> failedFuture(Throwable  
ex)  
    public static <U> CompletionStage<U> completedStage(U value)  
    public static <U> CompletionStage<U> failedStage(Throwable  
ex)  
  
    public Executor defaultExecutor()  
    public static Executor delayedExecutor(long delay, TimeUnit  
unit)  
}
```

Wir benötigen für unsere Demo-Anwendung folgende Importe:

```
import java.util.concurrent.CompletableFuture;  
import java.util.concurrent.ExecutionException;  
import java.util.concurrent.Executor;  
import java.util.concurrent.TimeUnit;  
import java.util.function.Function;
```

Wir werden folgende `sleep`-Methode nutzen:

```
static void sleep(int millis) {  
    try {  
        Thread.sleep(millis);  
    }  
}
```



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

Zunächst sei an einem hoffentlich einleuchtenden Beispiel vorgestellt, was es überhaupt mit `CompletableFuture` auf sich hat.

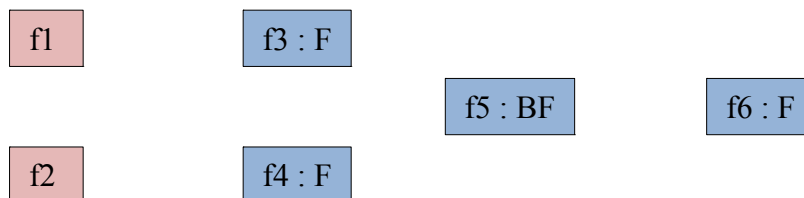
Pythagoras hat bekanntlich erkannt, dass bei einem rechtwinkligen Dreieck die Summe der Kathetenquadrate gleich dem Quadrat der Hypothense ist. Angenommen also, die Kathetenlängen sind gegeben ( $a$ ,  $b$ ). Dann kann die Hypothense wie folgt berechnet werden: `c = Math.sqrt(a * a + b * b)`.

Angenommen, die beiden Eingabewerte ( $a$  und  $b$ ) werden durch `CompletableFuture` repräsentiert (diese Objekte repräsentieren Werte, die erst später zur Verfügung stehen).

Wenn dann einer der Werte irgendwann zur Verfügung steht, kann das Quadrat dieses Wertes berechnet werden – und parallel zu dieser Berechnung kann auch das Quadrat des zweiten Werts berechnet werden (sofern es zur Verfügung steht).

Wenn dann die Quadrate der Katheten parallel berechnet sind, kann die Summe dieser beiden Quadrate berechnet werden. Und schließlich dann die Wurzel aus dieser Summe.

Das folgende Schaubild möge diesen Ablauf verdeutlichen:



`f1` und `f2` sind `CompletableFuture`s, welche die beiden Eingangswerte bereitstellen werden.

`f3` und `f4` sind `CompletableFuture`s, die jeweils mit einer `Function` verbunden sind (`x -> x * x`).

`f5` beginnt zu arbeiten, wenn sowohl `f3` als auch `f4` ihre Aufgaben erledigt haben. `f5` ist mit einer `BiFunction` verbunden: (`x`, `y`) `-> x + y`.

Das `CompletableFuture` `f6` beginnt seine Arbeit dann, wenn `f5` die seine erledigt hat. `f6` ist mit einer `Function` verbunden, welche die Wurzel zieht: `x -> Math.sqrt(x)`.

f3 und f4 können ihre Arbeiten parallel ausführen.

Dieser Ablauf kann wie folgt implementiert werden:

```
static void demoPythagoras() {
    CompletableFuture<Double> f1 = new CompletableFuture<>();
    CompletableFuture<Double> f2 = new CompletableFuture<>();
    CompletableFuture<Double> f3 = f1.thenApplyAsync(x -> x *
x);
    CompletableFuture<Double> f4 = f2.thenApplyAsync(x -> x *
x);
    CompletableFuture<Double> f5 =
        f3.thenCombine(f4, (x, y) -> x + y);
    CompletableFuture<Double> f6 = f5.thenApply(x ->
Math.sqrt(x));
    f1.complete(3.0);
    f2.complete(4.0);
    try {
        double result = f6.get();
        System.out.println(result);
    }
    catch (InterruptedException | ExecutionException e) {
        System.out.println(e);
    }
}
```

Nachdem der `CompletableFuture`-Graph aufgebaut ist, werden die Eingabewerte bereitgestellt: `f1.complete(3.0)` und `f2.complete(4.0)`. Erst dann beginnt die Maschinerie zu arbeiten. Nach der Bereitstellung der Eingangswerte wartet der Hauptthread auf das Ergebnis von `f6` – mittels des Aufrufs von `f6.get()`. Diese Methode liefert dann irgendwann die 5.0 zurück.

Man beachte, wie die `CompletableFuture`-Objekte erzeugt werden: die ersten beiden sind das Resultat der direkten Instanziierung der Klasse; die weiteren werden durch Methoden erzeugt, die ihrerseits auf `CompletableFuture`-Objekte aufgerufen werden (`thenApplyAsync`, `thenCombine` und `thenApply`).

Der Workspace enthält eine weitere Version der oben beschriebenen `demo`-Methode, welche auch die an den Aktionen beteiligten Threads ausgibt. Hier die Ausgabe der Methode (`demoPythagorasVerbose`):

```
[1] complete 3.0
[14] 3.0 * 3.0 => 9.0
[1] complete 4.0
[13] 4.0 * 4.0 => 16.0
```

```
[13] 9.0 + 16.0 => 25.0
[13] Math.sqrt(25.0) ==> 5.0
[1] 5.0
```

Die Berechnung des Quadrats von 3.0 besorgt der Thread 14, die Berechnung des Quadrats von 4.0 wird im Thread 13 ausgeführt. Letzterer führt dann auch die Addition und die Wurzelberechnung aus. Der Thread 1 wartet via `get` auf das Endresultat der Berechnung.

Auf die oben beschriebene beispielhafte Art und Weise können nun natürlich beliebig komplex Grafen erzeugt werden und damit zur Laufzeit beliebig komplexe Berechnungen teilweise parallel ausgeführt werden.

Wir begnügen uns im Folgenden aber mit etwas einfacheren Beispielen.

Hier ein einfaches Beispiel, in welchem nur zwei `CompletableFutures` verwendet werden:

```
static void demoSimple()
    throws InterruptedException, ExecutionException {
    CompletableFuture<Integer> f1 = new
CompletableFuture<>();
    CompletableFuture<Integer> f2 = f1.thenApplyAsync(x -> x
* x);
    f1.complete(3);
    int result = f2.get();
    System.out.println(result);
}
```

Die Ausgabe ist erwartungsgemäß 9.

Die folgende `demo`-Methode benutzt die Java-9-Methode `completeAsync`. An diese Methode wird ein `Supplier` übergeben, dessen Resultat dann als Eingabewert für `f1` verwendet wird:

```
static void demoCompleteAsync()
    throws InterruptedException, ExecutionException {
    CompletableFuture<Integer> f1 = new
CompletableFuture<>();
    CompletableFuture<Integer> f2 = f1.thenApplyAsync(x -> x
* x);
    f1.completeAsync(() -> 3);
    int result = f2.get();
    System.out.println(result);
}
```

Wie erwartet, wird auch hier der Wert 9 ausgegeben.

Die folgende `demo`-Methode verwendet die Java-9-Methode `completeOnTimeout`. Diese Methode liefert zwar eine `CompletableFuture`-Referenz zurück – aber diese Referenz zeigt auf dasselbe Objekt, auf das sie aufgerufen wurde. Das aktuelle `CompletableFuture` liefert den an die Methode `completeOnTimeout` übergebenen "Spezialwert" zurück, wenn der eingestellte Timeout überschritten wurde:

```
static void demoCompleteOnTimeout()
    throws InterruptedException, ExecutionException {
    CompletableFuture<Integer> f1 = new
CompletableFuture<>();
    CompletableFuture<Integer> f2 = f1.thenApplyAsync(x -> {
        sleep(2000);
        return x * x;
    });
    CompletableFuture<Integer> f3 =
        f2.completeOnTimeout(-1, 1000,
TimeUnit.MILLISECONDS);
    System.out.println(f3 == f2);

    f1.complete(3);
    int result = f3.get();
    System.out.println(result);
}
```

Die Ausgaben:

```
true
-1
```

Der eingestellte Timeout wird hier überschritten – weil die an `f2` übergebene Funktion sich sehr viel Zeit lässt ...

Hätte sich `f2` etwas beeilt, wäre natürlich der Wert 9 ausgegeben worden.

Mittels der Methode `orTimeout` kann veranlasst werden, dass eine Methode, die auf ein `CompletableFuture` wartet, entweder normal zurückkehrt oder aber – im Falle, dass der Timeout überschritten wurde – eine `ExecutionException` geworfen wird:

```
static void demoOrTimeout() {
    CompletableFuture<Integer> f1 = new
CompletableFuture<>();
    CompletableFuture<Integer> f2 = f1.thenApplyAsync(x -> {
        sleep(2000);
        return x * x;
    });
}
```

```
    });  
    CompletableFuture<Integer> f3 =  
        f2.orTimeout(1000, TimeUnit.MILLISECONDS);  
    System.out.println(f3 == f2);  
    f1.complete(3);  
    try {  
        int result = f3.get();  
        System.out.println(result);  
    }  
    catch (ExecutionException | InterruptedException e) {  
        System.out.println(e);  
    }  
}
```

Die Ausgaben (auch hier wurde der Timeout überschritten):

```
true  
java.util.concurrent.ExecutionException:  
    java.util.concurrent.TimeoutException
```

Ein `CompletableFuture`, welches mittels der Factory-Methode `completedFuture` erzeugt wurde, ist bereits "fertig" – und liefert also unmittelbar das Resultat:

```
static void demoCompletedFuture()  
    throws InterruptedException, ExecutionException {  
    CompletableFuture<Integer> f1 =  
        CompletableFuture.completedFuture(3); // java 8  
    CompletableFuture<Integer> f2 = f1.thenApplyAsync(x -> x  
* x);  
    int result = f2.get();  
    System.out.println(result);  
}
```

Die Berechnung liefert das erwartete Ergebnis: 9.

Ein `CompletableFuture` kann auch mittels `failedFuture` erzeugt werden. Es wirft dann genau diejenige Exception, welche an `failedFuture` übergeben wurde:

```
static void demoFailedFuture() {  
    CompletableFuture<Integer> f1 =  
    CompletableFuture.failedFuture(  
        new RuntimeException("Water in drive a:");  
    );  
    CompletableFuture<Integer> f2 = f1.thenApplyAsync(x -> x  
* x);  
    try {  
        int result = f2.get();  
    }  
}
```

```
        System.out.println(result);
    }
    catch (InterruptedException | ExecutionException e) {
        System.out.println(e);
    }
}
```

Die Ausgaben:

```
java.util.concurrent.ExecutionException:
  java.lang.RuntimeException: Water in drive a:
```

Mittels der statischen Methode `completedStage` kann ein `CompletedStage`-Objekt erzeugt werden (das genau dasjenige Resultat liefert, welches der Factory-Methode übergeben wird). Den Bearbeiter des Resultats kann dann in Form eines `BiConsumer`s an `whenComplete` übergeben werden:

```
static void demoCompletedStage()
    throws InterruptedException, ExecutionException {
    CompletionStage<Integer> f1 =
        CompletableFuture.completedStage(3);
    CompletionStage<Integer> f2 = f1.thenApplyAsync(x -> x *
x);
    f2.whenComplete((Integer v, Throwable t) ->
        System.out.println(v + " " + t));
}
```

Die Ausgaben:

```
9 null
```

By the way. `CompletedStage` ist die Basisklasse von `CompletedFuture`. Letztere implementiert zusätzlich das `Future`-Interface (also insbesondere die `get`-Methode).

Die Methode `failedState` erzeugt ein `CompletionStage`-Objekt, welches bei seiner Berechnung eine Exception wirft:

```
static void demoFailedStage()
    throws InterruptedException, ExecutionException {
    CompletionStage<Integer> f1 =
        CompletableFuture.failedStage(new
RuntimeException("ex"));
    CompletionStage<Integer> f2 = f1.thenApplyAsync(x -> x *
x);
    f2.whenComplete((Integer v, Throwable t) ->
        System.out.println(v + " " + t));
}
```

```
}
```

Die Ausgabe:

```
null java.util.concurrent.CompletionException:
    java.lang.RuntimeException: ex
```

Für die parallele Ausführung der `CompletedFuture`-Objekte wird ein `Executor` verwendet (also ein Thread-Pool). Mittels der Method `defaultExecutor` kann die tatsächliche Implementierung ermittelt werden (`Executor` ist nur ein Interface):

```
static void demoDefaultExecutor() {
    Log.logMethodCall();
    CompletableFuture<Integer> f = new CompletableFuture<>();
    Executor executor = f.defaultExecutor();
    System.out.println(executor);
}
```

Die Ausgabe zeigt, dass ein `ForkJoinPool` verwendet wird:

```
java.util.concurrent.ForkJoinPool@32e6e9c3[
    Running,
    parallelism = 3,
    size = 3,
    active = 1,
    running = 0,
    steals = 6,
    tasks = 0,
    submissions = 0
]
```

An `completeAsync` kann nun ein `Executor` übergeben werden, der die Aktion mit einer Verzögerung ausführt:

```
static void demoDelayedExecutor()
    throws InterruptedException, ExecutionException {
    Log.logMethodCall();
    CompletableFuture<Integer> f1 = new
CompletableFuture<>();
    CompletableFuture<Integer> f2 = f1.thenApplyAsync(x -> x
* x);
    f1.completeAsync(() -> 3,
        CompletableFuture.delayedExecutor(1,
TimeUnit.SECONDS));
    int result = f2.get();
    System.out.println(result);
}
```

Die Ausgabe ist auch hier erwartungsgemäß die 9 – aber die Berechnung läßt sich nun ein wenig Zeit...

Schließlich können wir von `CompletableFuture` eine eigene Klasse ableiten, die mit einem eigenen `Executor` operiert:

```
static class MyFuture<T> extends CompletableFuture<T> {

    static class MyExecutor implements Executor {
        @Override
        public void execute(Runnable command) {
            System.out.println(
                "MyExecutor.execute(" + command + ")");
            new Thread(command).start();
        }
    };

    @Override
    public CompletableFuture<T> newIncompleteFuture() {
        System.out.println("MyFuture.newIncompleteFuture()");
        return new MyFuture<T>();
    }

    @SuppressWarnings("unchecked")
    @Override
    public <R> MyFuture<R> thenApplyAsync(
        Function<? super T, ? extends R> function) {
        sleep(10);
        return (MyFuture<R>)
            super.thenApplyAsync(function, new MyExecutor());
    }
}
```

Der hier benutzte `Executor` ist trivial: für jede Execution wird ein neuer Thread benutzt (was natürlich nicht gerade performant ist...)

Die `MyFuture`-Klasse überschreibt die Methode `newCompletableFuture`. Diese Methode wird stets dann aufgerufen, wenn ein neues `CompletableFuture` mittels einer Factory-Methode erzeugt wird. Hier wird also festgelegt, dass alle Factory-Methoden ein `MyFuture`-Objekt erzeugen.

Die `thenApplyAsync`-Methode ist derart überschrieben, dass ein wenig Zeit verstreicht, bevor die eigentliche `thenApplyAsync`-Methode der Basisklasse aufgerufen wird (eine solche Konstruktion kann z.B. dann verwendet werden, wenn wir die Abläufe tracen oder visualisieren wollen).



Hier eine Anwendung dieser Klasse:

```
static void demoNewIncompleteFuture()
    throws InterruptedException, ExecutionException {
    CompletableFuture<Integer> f1 = new MyFuture<>();
    System.out.println(f1.getClass().getName());
    CompletableFuture<Integer> f2 = f1.thenApplyAsync(x -> x
*   x);
    System.out.println(f2.getClass().getName());
    f1.complete(3);
    int result = f2.get();
    System.out.println(result);
}
```

Die Ausgaben:

```
jj.appl.Application$MyFuture
MyFuture.newIncompleteFuture()
jj.appl.Application$MyFuture
MyExecutor.execute(
    java.util.concurrent.CompletableFuture$UniApply@402f32ff)
9
```

## 6.8 Kompakte Strings

Ein `String`-Objekt benutzte bislang einen Arrays von `chars`, um die Zeichen des Strings zu speichern. Für jedes Zeichen wurden zur Speicherung also zwei Byte benutzt. Werden hauptsächlich Strings verwendet, die nur Latin1-Zeichen enthalten, wurde somit immer ein Byte des `chars` "verschenkt". Für Zeichenfolgen, die UTF16-Zeichen enthalten, war diese `char`-Array-Speicherung natürlich angemessen.

Bei großen Mengen von zu verwaltenden Strings konnte diese Implementierung natürlich Problem bereiten (`OutOfMemoryError`).

Ab Java 9 werden Zeichenketten, die nur Latin1-Zeichen enthalten, in einem `byte`-Array gespeichert – jedes Zeichen belegt dann nur genau ein einzige Byte. Bei UTF16-Zeichenketten wird ebenfalls ein `byte`-Array verwendet – wobei dann aber ein Zeichen jeweils zwei Byte benutzt. Die Chance, dass wir uns `OutOfMemoryErrors` einfangen, wird somit also reduziert.

Wir zeigen den neuen Aufbau von Strings, indem wir ein wenig Reflection betreiben. Der folgenden `inspect`-Methode wird ein `char`-Array übergeben, aufgrund dessen ein `String` erzeugt wird (mittels `String.valueOf`). Dann werden die `String`-eigenen Felder `value` und `coder` inspiziert – der Wert dieser Felder wird ermittelt und ausgegeben. Das `value`-Feld enthält den `String`-eigenen Zeiger auf den `byte`-Array, `coder` enthält die Art der Codierung (ein oder zwei Byte pro `char`):

```
private static void inspect(char[] content) throws Exception
{
    Class<?> cls = String.class;
    Field fieldValue = cls.getDeclaredField("value");
    Field fieldCoder = cls.getDeclaredField("coder");
    fieldValue.setAccessible(true);
    fieldCoder.setAccessible(true);
    System.out.println(fieldValue);
    System.out.println(fieldCoder);
    String s = String.valueOf(content);
    byte[] value = (byte[]) fieldValue.get(s);
    System.out.println(value.length);
    for(byte v : value)
        System.out.print((char)v);
    System.out.println();
    byte coder = (byte) fieldCoder.get(s);
    System.out.println(coder);
}
```

Wir rufen `inspect` mit einem `char`-Array auf, der nur Latin1-Zeichen enthält:

```
inspect(new char[] { 'a', 'b', 'c' });
```

Die Ausgabe zeigt, dass der allokierte `byte`-Array die Größe 3 hat – und das der `0-coder` verwendet wird:

```
private final byte[] java.lang.String.value
private final byte java.lang.String.coder
3
abc
0
```

Wird nun aber an `inspect` ein Array mit UTF16 Zeichen übergeben:

```
inspect(new char[] { 500, 501, 502 });
```

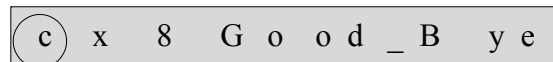
so wird ein `byte`-Array der Größe 6 alloliert – und `coder` enthält den Wert 1:

```
private final byte[] java.lang.String.value
private final byte java.lang.String.coder
6
???
1
```

Nach außen hin macht es natürlich keinerlei Unterschied, welche interne Form der Speicherung benutzt wird - Strings können also weiterhin genauso genutzt werden wie bislang.

Auf den ersten Blick mag es so ausschauen, als würde bei der Speicherung von Latin1-Strings im Vergleich zu den "alten" Strings 50% Speicher eingespart. Schauen wir näher hin, ist die Speicherersparnis aber deutlich geringer.

Im "alten" Java sah der Latin1-String "Good\_Bye" wir folgt aus (jedes Kästchen belegt 4 Byte):

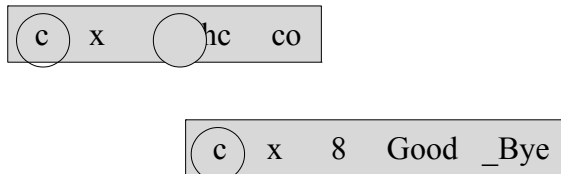


Wie jedes Objekt haben auch String- und Array-Objekte zunächst einmal einen Zeiger auf das `Class`-Objekt, welches die String- resp. die entsprechende Array-Klasse beschreibt (`c`). Weitere 4 Byte sind reserviert für den Garbage-Kollektor und Multithreading-Zwecke (`x`). Ein String hat einen Zeiger auf einen `char`-Array – und ein

Feld für den `hashCode` (`hc`). Dieser hat zusätzlich zu `x` und `c` einen Bereich, in dem die Länge des Arrays gespeichert ist (hier: 8). Dann kommen vier weitere 4-Byte-große Blöcke, in denen 8 `char`-Elemente Platz finden.

Insgesamt ergibt sich somit ein Speicherbedarf von  $11 * 4 = 44$  Bytes.

Der neue kompakte String sieht wie folgt aus:



Das `String`-Objekt enthält nun zusätzlich ein Feld für den `coder` (`co`). Die Anzahl der 4-Byte-Blöcke, in denen die eigentlichen Zeichen gespeichert werden, reduziert sich nun auf 2.

Insgesamt ergibt sich hier ein Speicherverbrauch von  $10 * 4 = 40$  Bytes.

Die Speicherersparnis beträgt also etwa 10%. Bei kleineren Strings ist die Ersparnis natürlich noch geringer – je größer allerdings die Strings sind, umso größer wird die Speicherersparnis.

Ein weiterer Hinweis: `Latin1`-Strings sind nicht nur (etwas) kompakter, sondern auch in ihrer Verarbeitung deutlich performanter als `UTF16`-Strings. Hier eine kleine `demo`-Methode:

```
static void demoPerformance() throws Exception {
    final int N = 10000;
    final int M = 5000;
    final int SIZE = 256;
    int n1 = 0;
    int n2 = 0;
    int duration1 = 0;
    int duration2 = 0;
    // Latin1-chars
    char[] array1 = new char[SIZE];
    for(int i = 0; i < SIZE; i++)
        array1[i] = (char)i;
    // UTF16-chars
    char[] array2 = new char[SIZE];
    for(int i = 0; i < SIZE; i++)
        array2[i] = (char)(i + SIZE);
    final String string1A = new String(array1);
    final String string1B = new String(array1);
```

```
final String string2A = new String(array2);
final String string2B = new String(array2);
for(int i = 0; i < N; i++) {
    {
        long start = System.nanoTime();
        for(int j = 0; j < M; j++)
            n1 += string1A.compareTo(string1B);
        duration1 += (System.nanoTime() - start);
    }
    {
        long start = System.nanoTime();
        for(int j = 0; j < M; j++)
            n2 += string2A.compareTo(string2B);
        duration2 += (System.nanoTime() - start);
    }
}
System.out.println(n1);
System.out.println(n2);
System.out.println("Latin1 : " + duration1);
System.out.println("UTF16  : " + duration2);
}
```

Die Ausgaben:

```
0
0
Latin1 : 641091205
UTF16  : 1396682141
```

Die Verarbeitung (hier: das Vergleichen) von Latin1-Strings ist also etwa doppelt so schnell wie die von UTF16-Strings...

Und ein letzter Hinweis: Zur effizienten Speicherung von Strings sollte bei Bedarf vielleicht auch die `String`-Methode `intern` herangezogen werden (eine Methode, die einen internen, globalen Cache nutzt)...

## 6.9 Arrays

Die Klasse `java.util.Arrays` ist um einige weitere statische Methoden erweitert worden.

Die `Arrays.equals`-Methode war bereits vor Java 9 mehrfach überladen – es gab `equals`-Methoden für Arrays mit primitiven Komponententyp und eine `equals`-Methode für Object-Arrays:

```
public static boolean equals(int[] a, int[] b)
// ...
public static boolean equals(Object[] a, Object[] b)
```

Die letzte `Object[]`-Variante vergleicht die Elemente mittels ihrer `equals`-Methoden.

Für jede dieser Methoden gibt's nun zusätzlich eine Überladung, die es erlaubt, Teilbereiche von Arrays auf Gleichheit zu testen (ein "Range-Equals"). Dabei werden für jeden der beiden am Vergleich beteiligten Arrays zwei Indizes übergeben: der Start-Index und der Ende-Index. Wobei dann jeweils die Elemente vom Start-Index (einschließlich) bis zum Ende-Index (ausschließlich) für den Vergleich herangezogen werden:

```
public static boolean equals(
    int[] a, int aFromIndex, int aToIndex,
    int[] b, int bFromIndex, int bToIndex)
// ...
public static boolean equals(
    Object[] a, int aFromIndex, int aToIndex,
    Object[] b, int bFromIndex, int bToIndex)
```

Ist `aToIndex-aFromIndex` ungleich `bToIndex-bFromIndex`, liefern die Methoden jeweils `false` zurück (eine Exception wird nur dann geworfen, wenn einer oder mehrere der Indizes außerhalb des gültigen Bereichs liegen).

Ein Beispiel:

```
static void demoEqualsRange() throws Exception {
    int[] ints1 = new int[] { 10, 11, 12, 13, 14, 15 };
    int[] ints2 = new int[] { 11, 12, 13, 14, 15, 16 };
    System.out.println(Arrays.equals(ints1, 1, 5, ints2, 0,
4));
}
```

Der `equals`-Vergleich liefert hier `true` (verglichen werden hier jeweils die `int`-Werte von 11 bis 14 (einschließlich)).

Für den Vergleich von `Object`-Arrays wurden zwei neue generische `equals`-Methoden eingeführt, welchen neben den beiden zu vergleichenden Arrays (und evtl. Indizes) ein `Comparator` übergeben wird:

```
public static <T> boolean equals(  
    T[] a,  
    T[] b,  
    Comparator<? super T> cmp)
```

```
public static <T> boolean equals(  
    T[] a, int aFromIndex, int aToIndex,  
    T[] b, int bFromIndex, int bToIndex,  
    Comparator<? super T> cmp)
```

Wir demonstrieren im Folgenden Beispiel nur die erste dieser beiden Methoden:

```
static void demoEquals() throws Exception {  
  
    class Foo {  
        public final int x;  
        public Foo(int x) {  
            this.x = x;  
        }  
    }  
  
    Foo[] foos1 = new Foo[] { new Foo(10), new Foo(11) };  
    Foo[] foos2 = new Foo[] { new Foo(10), new Foo(11) };  
  
    System.out.println(Arrays.equals(foos1, foos2));  
  
    Comparator<Foo> comparator =  
        Comparator.comparing(foo -> foo.x);  
    System.out.println(Arrays.equals(foos1, foos2,  
comparator));  
}
```

Da die Klasse `Foo` die `equals`-Methode nicht überschreibt, wird beim ersten Aufruf von `Arrays.equals` die `Object`-eigene `equals`-Methode zum Vergleich jeweils zweier Elemente herangezogen – es werden also Referenzen verglichen. Und schon der erste Vergleich (`new Foo(10).equals(new Foo(10))`) liefert `false`. – weshalb auch `Arrays.equals` als Resultat `false` liefert.

Dem zweiten Aufruf von `Arrays.equals` wird zusätzlich ein `Comparator` übergeben, der `Foo`-Elemente daraufhin miteinander vergleicht, ob sie denselben `x`-Wert besitzen. Dieser `Comparator` wird von `Arrays.equals` für alle zu vergleichenden Elemente aufgerufen – und liefert also im obigen Beispiel stets `0` zurück. Als Resultat liefert `Arrays.equals` dann `true` zurück.

Java 9 erweitert die `Arrays`-Klasse um eine Vielzahl überladener `compare`-Methoden.

Für jeden primitiven Element-Typ gibt's zwei `compare`-Methoden – hier am Beispiel des Komponenten-Typs `int` dargestellt:

```
public static int compare(  
    int[] a,  
    int[] b)  
  
public static int compare(  
    int[] a, int aFromIndex, int aToIndex,  
    int[] b, int aFromIndex, int aToIndex)  
  
// dito für alle weiteren primitiven Typen
```

Die erste Methode vergleicht alle Elemente beider Arrays; die zweite Methode ist der "Range-Compare".

`compare` liefert einen positiven Wert, wenn der erste Array lexikographisch größer ist als der zweite; im umgekehrten Falle wird ein negativer Wert geliefert. Ansonsten (wenn beide Arrays lexikographisch gleich sind) wird `0` geliefert

Eine Demo-Anwendung:

```
static void demoComparePrimitive() throws Exception {  
  
    int[] ints1 = new int[] { 11, 12, 13, 15 };  
    int[] ints2 = new int[] { 11, 12, 13, 15 };  
    System.out.println(Arrays.compare(ints1, ints2));  
  
    int[] ints3 = new int[] { 11, 12, 13, 15 };  
    int[] ints4 = new int[] { 11, 12, 13, 155 };  
    System.out.println(Arrays.compare(ints3, ints4));  
  
    int[] ints5 = new int[] { 11, 12, 13, 155 };  
    int[] ints6 = new int[] { 11, 12, 13, 15 };  
    System.out.println(Arrays.compare(ints5, ints6));  
}
```



Die Ausgaben: 0, -1, 1.

Zusätzlich existieren vier weitere generische `compare`-Methoden:

```
public static <T extends Comparable<? super T>> int compare(
    T[] a,
    T[] b)

public static <T extends Comparable<? super T>> int compare(
    T[] a, int aFromIndex, int aToIndex,
    T[] b, int aFromIndex, int aToIndex)

public static <T> int compare(
    T[] a,
    T[] b,
    Comparator<? super T> cmp)

public static <T> int compare(
    T[] a, int aFromIndex, int aToIndex,
    T[] b, int aFromIndex, int aToIndex,
    Comparator<? super T> cmp)
```

Die ersten beiden Methoden setzen voraus, dass der Element-Typ `Comparable` ist. Den letzten beiden Methoden ist jeder Element-Typ recht: sie vergleichen die Elemente mittels eines expliziten `Comparators`.

Hier eine Anwendung, welche die `Comparable`-basierte Variante zeigt:

```
static void demoCompareComparable() {

    class Foo implements Comparable<Foo> {
        public final int x;

        public Foo(int x) {
            this.x = x;
        }

        @Override
        public int compareTo(Foo other) {
            return Integer.compare(this.x, other.x);
        }
    }

    Foo[] foos1 = new Foo[] { new Foo(10), new Foo(11) };
    Foo[] foos2 = new Foo[] { new Foo(10), new Foo(11) };
    System.out.println(Arrays.compare(foos1, foos2));
}
```

```
Foo[] foos3 = new Foo[] { new Foo(10), new Foo(11) };
Foo[] foos4 = new Foo[] { new Foo(10), new Foo(111) };
System.out.println(Arrays.compare(foos3, foos4));

Foo[] foos5 = new Foo[] { new Foo(10), new Foo(111) };
Foo[] foos6 = new Foo[] { new Foo(10), new Foo(11) };
System.out.println(Arrays.compare(foos5, foos6));
}
```

Die Ausgaben: 0, -1, 1.

Und hier eine Anwendung, welche die `Comparator`-basierte Variante demonstriert (wobei der Abwechslung halber unterschiedliche `Comparator`-Implementierungen genutzt werden):

```
static void demoCompareComparator() {

    class Foo {
        public final int x;

        public Foo(int x) {
            this.x = x;
        }
    }

    Foo[] foos1 = new Foo[] { new Foo(10), new Foo(11) };
    Foo[] foos2 = new Foo[] { new Foo(10), new Foo(11) };
    System.out.println(Arrays.compare(foos1, foos2,
        new Comparator<Foo>() {
            @Override
            public int compare(Foo foo1, Foo foo2) {
                return Integer.compare(foo1.x, foo2.x);
            }
        }));

    Foo[] foos3 = new Foo[] { new Foo(10), new Foo(11) };
    Foo[] foos4 = new Foo[] { new Foo(10), new Foo(111) };
    System.out.println(Arrays.compare(foos3, foos4,
        (foo1, foo2) -> Integer.compare(foo1.x, foo2.x)));

    Foo[] foos5 = new Foo[] { new Foo(10), new Foo(111) };
    Foo[] foos6 = new Foo[] { new Foo(10), new Foo(11) };
    System.out.println(Arrays.compare(foos5, foos6,
        Comparator.comparing(foo -> foo.x)));
}
```

Auch hier wieder dieselben Ausgaben: 0, -1, 1.

Schließlich führt Java 9 eine Reihe von überladenen `mismatch`-Methoden, welche die Position der ersten ungleichen Elemente zweier Arrays liefert (oder `-1`, wenn beide Arrays gleich sind). Zur Demonstration eine einfache `int[]`-basierte Variante:

```
static void demoMismatch() throws Exception {  
  
    int[] ints1 = new int[] { 11, 12, 13, 14 };  
    int[] ints2 = new int[] { 11, 12, 13, 14 };  
    System.out.println(Arrays.mismatch(ints1, ints2));  
  
    int[] ints3 = new int[] { 11, 12, 13, 14 };  
    int[] ints4 = new int[] { 11, 12, 133, 14 };  
    System.out.println(Arrays.mismatch(ints3, ints4));  
}
```

Die Ausgaben: `-1` und `2`.

## 6.10 Objects

Zum Testen von Preconditions stellt die Klasse `Objects` zusätzlich zu der bereits seit Java 7 bekannten `requireNonNull`-Methode einige weitere Methoden zur Verfügung:

```
public static <T> T requireNonNullElse(T obj, T defaultObj)

public static <T> T requireNonNullElseGet(
    T obj, Supplier<? extends T> supplier)

public static int checkIndex(int index, int length)

public static int checkFromToIndex(int fromIndex, int toIndex,
int length)
```

Die Verwendung der ersten beiden Methoden demonstriert die folgende `Foo`-Klasse:

```
import java.util.Objects;

public class Foo {

    static void alpha(String s) {
        Objects.requireNonNull(s); // since 1.7
        System.out.println(s.toUpperCase());
    }

    static void beta(String s) {
        s = Objects.requireNonNullElse(s, "hello");
        System.out.println(s.toUpperCase());
    }

    static void gamma(String s) {
        s = Objects.requireNonNullElseGet(s, () -> "h" + "ello");
        System.out.println(s.toUpperCase());
    }
}
```

Hier die `demo`-Methode:

```
static void demoRequireNonNull() throws Exception {
    try {
        Foo.alpha(null);
    }
    catch (NullPointerException e) {
        System.out.println("Expected" + e);
    }
}
```

```
    }  
    Foo.beta(null);  
    Foo.gamma(null);  
}
```

### Die Ausgaben:

```
Expected java.lang.NullPointerException  
HELLO  
HELLO
```

Die Verwendung der letzten beiden Methoden demonstriert die folgende `Bar`-Klasse:

```
import java.util.Objects;  
  
class Bar {  
  
    static void alpha(int[] array, int index) {  
        System.out.println(array[index]);  
    }  
  
    static void beta(int[] array, int index) {  
        Objects.checkIndex(index, array.length);  
        System.out.println(array[index]);  
    }  
  
    static void gamma(int[] array, int fromIndex, int toIndex) {  
        Objects.checkFromToIndex(fromIndex, toIndex,  
array.length);  
        for (int i = fromIndex; i < toIndex; i++)  
            System.out.println(array[i]);  
    }  
}
```

### Die demo-Methode:

```
static void demoCheckIndex() throws Exception {  
    int[] array = new int[] { 10, 11, 12, 13, 14 };  
    try {  
        Bar.alpha(array, 5);  
    }  
    catch (Exception e) {  
        System.out.println("Expected: " + e);  
    }  
    try {  
        Bar.beta(array, 5);  
    }
```

```
    }  
    catch (Exception e) {  
        System.out.println("Expected: " + e);  
    }  
    Bar.gamma(array, 0, 5);  
    try {  
        Bar.gamma(array, -1, 6);  
    }  
    catch (Exception e) {  
        System.out.println("Expected: " + e);  
    }  
}
```

Und die Ausgaben:

```
Expected: java.lang.ArrayIndexOutOfBoundsException: 5  
Expected: java.lang.IndexOutOfBoundsException:  
    Index 5 out-of-bounds for length 5  
10  
11  
12  
13  
14  
Expected: java.lang.IndexOutOfBoundsException:  
    Range [-1, 6) out-of-bounds for length 5
```

Natürlich könnten wir einfach darauf vertrauen, dass ein Array-Zugriff mit einem ungültigen Index ohnehin eine `ArrayIndexOutOfBoundsException` wirft – aber die Fehlertexte sind informativer, wenn eine der Precondition-Testmethode verwendet wird.

## 6.11 Deprecated

Die `@Deprecated`-Annotation ist um zwei Attribute erweitert worden: `since` und `forRemoval` (deren Bedeutung nicht weiter erklärt werden muss):

```
package java.lang;
// ...
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={
    CONSTRUCTOR, FIELD, LOCAL_VARIABLE,
    METHOD, PACKAGE, MODULE, PARAMETER, TYPE
})
public @interface Deprecated {
    String since() default "";
    boolean forRemoval() default false;
}
```

Die Annotation kann nun z.B. wie folgt verwendet werden:

```
@Deprecated(forRemoval = true, since = "9")
class Nonsense {
}
```

(Die unterstrichenen Texte sollen die Warnungen repräsentieren, die der Compiler generiert.)

Die direkte Instanziierung (via `new`) der Wrapper-Klassen für die primitiven Datentypen ist nun deprecated:

```
static void demoWrapper() throws Exception {

    Integer i1 = new Integer(32);
    System.out.println(i1);
    Integer i2 = new Integer("32");
    System.out.println(i2);
    Integer i3 = Integer.valueOf(32);
    System.out.println(i3);

    Double d1 = new Double(3.14);
    System.out.println(d1);
    Double d2 = new Double("3.14");
    System.out.println(d2);
    Double d3 = Double.valueOf(3.14);
    System.out.println(d3);
}
```

```
}
```

Statt `new Integer(n)` sollte nun also die Factory-Methode `Integer.valueOf(n)` verwendet werden. Diese Methode nutzt den `Integer`-eigenen Cache (für alle `int`-Werte im Bereich von -128 bis 127 existieren vorgefertigte `Integer`-Objekte).

Die Klasse `Observable` und das Interface `Observer` sind ebenfalls deprecated (aus Design-Sicht sind diese Typen äußerst fragwürdig):

```
static void demoObserverObservable() {  
  
    Observer observer = new Observer() {  
        @Override  
        public void update(Observable source, Object arg) {  
            // ...  
        }  
    };  
    Observable observable = new Observable();  
}
```

Und schließlich sollte auch `Object.finalize` nicht mehr überschrieben werden (mit dieser Methode sind eine Reihe technischer Probleme verbunden – siehe die JavaDoc):

```
static void demoFinalize() {  
  
    class Foo {  
        @Override  
        public void finalize() throws Throwable {  
            super.finalize();  
            System.out.println("finalize");  
        }  
    }  
    new Foo();  
    System.gc();  
}
```

Statt `finalize` kann die neue `Cleaner`-Klasse verwendet werden, die im nächsten Abschnitt vorgestellt wird.

Java 9 bietet zudem ein neues Tool namens `jdeprscan`, welches es ermöglicht, deprecated Elemente aufzufinden.



## 6.12 Cleaner

Statt `Object.finalize`-zu verwenden, sollte der in Java 9 eingeführte `Cleaner`-Mechanismus verwendet werden.

Der technische Hintergrund dieses neuen Konzepts (die `java.lang.ref`-Klassen `Reference`, `WeakReference`, `SoftReference` und `PhantomReference`) können hier nicht weiter erläutert werden – erleichtern aber das Verständnis dieses Konzepts.

Wie benötigen zunächst eine Instanz der Java 9-Klasse `java.lang.ref.Cleaner`. Zu diesem Zweck definieren wir eine Singleton-Klasse namens `DefaultCleaner`:

```
package jj.util;

import java.lang.ref.Cleaner;

public class DefaultCleaner {
    public static final Cleaner instance = Cleaner.create();
    private DefaultCleaner() {
    }
}
```

Bei einem `Cleaner`-Objekt können beliebig viele Objekte registriert werden. Jedes Objekt wird zusammen mit einem `Runnable` (oder mehreren(!) `Runnables`) registriert. Immer dann, wenn eines dieser registrierten Objekte nicht mehr referenziert wird (mit Ausnahme natürlich von der `Cleaner`-Registrierung selbst – diese benutzt aber ein `Reference`-Objekt – s.o.) und also für die GC bereitstellt, wird das mit diesem Objekt assoziierte `Runnable` aufgerufen. Dieses `Runnable` darf jedoch das Objekt, dessen GC ansteht, nicht referenzieren...

Zur Demonstration definieren wir folgende `Resource`-Klasse:

```
package jj.appl;

public class Resource {
    public final String name;
    public Resource(String name) {
        this.name = name;
    }
    public void cleanup() {
        System.out.println(this + " : cleanup()");
    }
    @Override
    public String toString() {
```

```
        return "Resource [" + name + "]\n";\n    }\n}
```

Eine `Resource` hat einen Namen und eine `cleanup`-Methode, die dann aufgerufen werden soll, wenn das Objekt, das diese `Resource` nutzt, für die GC bereitsteht. Man beachte genau die "wenn"-Bedingung...

Angenommen, wir definieren nun eine `Foo`-Klasse, die zwei solcher `Resource`-Objekte besitzt. Wenn das `Foo`-Objekt für die GC bereitsteht, die `cleanup`-Methode auf jedes dieser beiden `Resource`-Objekte aufgerufen werden:

```
package jj.appl;\n\nimport jj.util.DefaultCleaner;\n\npublic class Foo {\n\n    final Resource resource0 = new Resource("Hello");\n    final Resource resource1 = new Resource("World");\n\n    public Foo() {\n        DefaultCleaner.instance.register(this,\n            () -> this.resource0.cleanup());\n        DefaultCleaner.instance.register(this,\n            () -> this.resource1.cleanup());\n    }\n\n    public void doSomething() {\n        System.out.println("doSomething with " +\n            this.resource0 + " and " + this.resource1);\n    }\n}
```

Für `this` werden zwei `Runnable`s beim `DefaultCleaner` registriert, deren `run`-Methoden jeweils die `cleanup`-Methode einer der `Resource`-Objekte aufruft.

Eine Anwendung:

```
static void demoFoo() throws Exception {\n    Foo foo = new Foo();\n    foo.doSomething();\n    foo = null;\n    System.gc();\n    Thread.sleep(1000);\n}
```

Die Ausgaben zeigen, dass der beabsichtigte Effekt leider ausbleibt:

```
doSomething with Resource [Hello] and Resource [World]
```

Was läuft falsch? In den `run`-Methoden der beiden `Runnable`s wird `this` referenziert – also genau dasjenige `Foo`-Objekt, das für den GC "eigentlich" bereitsteht. Da es nun also wieder via `this` referenziert, kann es nicht freigegeben werden...

Wir müssen die Referenzen auf die beiden `Resource`-Objekte, deren `cleanup`-Methode aufgerufen werden soll, in einem statischen Kontext kopieren (in einen Kontext, der keinen Bezug zur `Foo.this`-Referenz hat) – und sie innerhalb des jeweiligen `Runnable`s über diesen statischen Kontext referenzieren.

Bei der folgenden `Bar`-Klasse funktioniert die Freigabe der `Resource`-Objekte:

```
package jj.appl;

import jj.util.DefaultCleaner;

public class Bar {

    final Resource resource0 = new Resource("Hello");
    final Resource resource1 = new Resource("World");

    static class ResourceHolder implements Runnable {
        final Resource r;
        ResourceHolder(Resource r) {
            this.r = r;
        }
        public void run() {
            this.r.cleanup();
        }
    }

    final ResourceHolder resourceHolder0 = new
ResourceHolder(resource0);
    final ResourceHolder resourceHolder1 = new
ResourceHolder(resource1);

    public Bar() {
        DefaultCleaner.instance.register(this, resourceHolder0);
        DefaultCleaner.instance.register(this, resourceHolder1);
    }

    public void doSomething() {
```

```
        System.out.println("doSomething with " +  
            this.resource0 + " and " + this.resource1);  
    }  
}
```

Ebenso wie ein `Foo`-Objekt referenziert auch ein `Bar`-Objekt zwei `Resource`-Objekte.

Zusätzlich werden aber zwei Instanzen der statischen inneren Klasse `ResourceHolder` erzeugt – wobei jeder `ResourceHolder` genau eine `Resource` hält. Die `ResourceHolder` sind `Runnable`. Sie können somit beim `Cleaner` registriert werden. Und die `run`-Methode dieser `ResourceHolder` kann dann die `cleanup`-Methode der `Resource`-Objekte aufrufen – ohne dass hierzu das zur GC anstehende `Bar`-Objekt angesprochen werden muss.

Man beachte, dass die `ResourceHolder` statisch sein muss – wäre sie nicht `static`, würde ein `ResourceHolder` implizit das Objekt der umschließenden Klassen (also das `Bar`-Objekt) referenzen. Und der ganze Aufwand wäre vergebens.

Hier die `demo`-Anwendung:

```
static void demoBar() throws Exception {  
    Bar bar = new Bar();  
    bar.doSomething();  
    bar = null;  
    System.gc();  
    Thread.sleep(1000);  
}
```

Die Ausgabe zeigt, dass bei der Freigabe eines `Bar`-Objekts nun die `cleanup`-Methode seiner beiden `Resource`-Objekte tatsächlich aufgerufen wird:

```
doSomething with Resource [Hello] and Resource [World]  
Resource [World] : cleanup()  
Resource [Hello] : cleanup()
```

## 7 Flow

Zunächst einige Zitate aus einem Oracle-Dokument:

### ***What is Reactive Programming ?***

*Reactive programming is about processing an asynchronous stream of data items, where applications react to the data items as they occur. A stream of data is essentially a sequence of data items occurring over time. This model is more memory efficient because the data is processed as streams, as compared to iterating over the in-memory data.*

*In the Reactive Programming model, there is a Publisher and a Subscriber. The Publisher publishes a stream of data, to which the Subscriber is asynchronously subscribed.*

*The model also provides a mechanism to introduce higher order functions to operate on the stream by means of Processors. Processors transform the data stream without the need for changing the Publisher or the Subscriber. The Processor (or a chain of Processors) sit between the Publisher and the Subscriber to transform one stream of data to another. The Publisher and the Subscriber are independent of the transformation that happen to the stream of data.*

### ***Why Reactive Programming ?***

- *Simpler code, making it more readable.*
- *Abstracts away from boiler plate code to focus on business logic.*
- *Abstracts away from low-level threading, synchronization, and concurrency issues.*
- *Stream processing implies memory efficient*
- *The model can be applied almost everywhere to solve almost any kind of problem.*

### ***JDK 9 Flow API***

*The Flow APIs in JDK 9 correspond to the Reactive Streams Specification, which is a defacto standard. The Reactive Streams Specification is one of the initiatives to standardize Reactive Programming. Several implementations already support the Reactive Streams Specification.*

*The Flow API (and the Reactive Streams API), in some ways, is a combination of ideas from Iterator and Observer patterns. The Iterator is a pull model, where the application pulls items from the source. The Observer is a push model, where the items from the*

*source are pushed to the application. Using the Flow API, the application initially requests for N items, and then the publisher pushes at most N items to the Subscriber. So its a mix of Pull and Push programming models.*

(<https://community.oracle.com/docs/DOC-1006738>)

## Die Flow-Interfaces

Aus der Java-Doc:

*Interrelated interfaces and static methods for establishing flow-controlled components in which Publishers produce items consumed by one or more Subscribers, each managed by a Subscription.*

*These interfaces correspond to the reactive-streams specification. They apply in both concurrent and distributed asynchronous settings: All (seven) methods are defined in void "one-way" message style. Communication relies on a simple form of flow control (method `Flow.Subscription.request(long)`) that can be used to avoid resource management problems that may otherwise occur in "push" based systems.*

(<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Flow.html>)

```
@FunctionalInterface
public static interface Flow.Publisher<T> {
    public abstract void subscribe(
        Flow.Subscriber<? super T> subscriber);
}
```

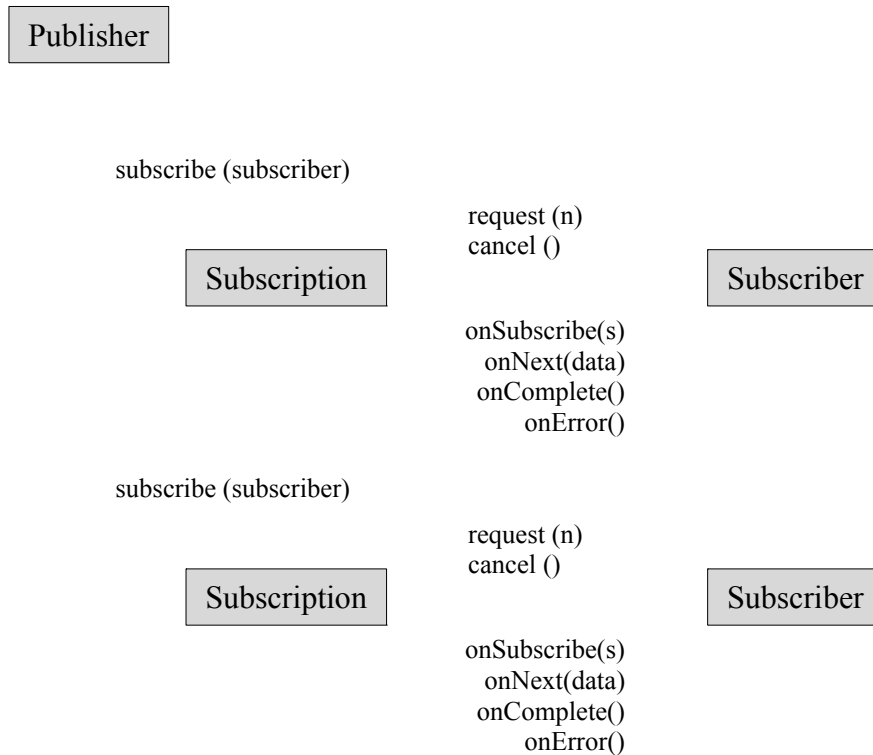
```
public static interface Flow.Subscriber<T> {
    public abstract void onSubscribe(
        Flow.Subscription subscription);
    public abstract void onNext(T item) ;
    public abstract void onError(Throwable throwable) ;
    public abstract void onComplete() ;
}
```

```
public static interface Flow.Subscription {
    public abstract void request(long n);
    public abstract void cancel() ;
}
```

```
public static interface Flow.Processor<T,R> extends
    Flow.Subscriber<T>, Flow.Publisher<R> {
}
```

Zusätzlich zum Interface `Publisher` existiert in Java 9 eine Klasse namens `SubscriptionPublisher`.

Eine Übersicht über den Zusammenhang zwischen `Publisher`, `Subscription` und `Subscriber`:



In den folgenden Beispielen benutzen wir zwei Utility-Klassen: `Log` und `XRunnable`.

Die Methoden der `Log`-Klasse geben zusätzlich zu den eigentlichen Meldungen die ID des aktuellen Threads aus. Diese Klasse ist im `shared`-Projekt enthalten. Sie enthält folgende Methoden:

```

public class Log {
    public static void tlogEnter(Object... params)
    public static void tlog(Object msg)
    public static void tlogExit()
}
  
```

`tlogEnter` und `tlogExit` geben zusätzlich den Namen der Methode aus, in welcher sie aufgerufen werden (und `tlogEnter` beginnt die Ausgabe mit `>>`, `tlogExit` stellt der

Ausgabe << voran). Mittels `tlogEnter` wird der Einstieg in eine Methode protokolliert, mittels `tlogExit` der Austieg aus einer Methode.

Die Ausgaben sind thread-spezifisch eingerückt: alle Ausgaben, die in einem bestimmten Thread stattfinden, beginnen an ein- und derselben Spaltenposition.

Wir benutzen weiterhin eine Klasse `XRunnable`, die ebenfalls im `shared`-Projekt definiert ist:

```
package jj.util.base;

import java.util.function.Function;

@FunctionalInterface
public interface XRunnable {

    public static void xrun(XRunnable runnable) {
        xrun(e -> new RuntimeException(e), runnable);
    }

    public static void xrun(
        Function<Throwable, ? extends RuntimeException>
wrapper,
        XRunnable runnable) {
        try {
            runnable.run();
        }
        catch (Throwable e) {
            throw wrapper.apply(e);
        }
    }

    public abstract void run() throws Throwable;
}
```

Mittels der statischen `xrun`-Methoden können auf einfache Weise Lambdas ausgeführt werden, die eine `Exception` werfen – ohne diese `Exception` selbst abfangen zu müssen (sie wird in eine `RuntimeException` eingewickelt).

Eine beispielhafte Anwendung:

```
XRunnable.xrun(() -> Thread.sleep(1000));
```

Die Items, die im Folgenden von den `Pushlischern` veröffentlicht und von den `Subscribern` verarbeitet werden, sind vom Typ `Message` (eine solche `Message` enthält der Einfachheit halber nur eine einfache `sequenceNumber`):



```
package jj.flow;  
  
public class Message {  
  
    public final int sequenceNumber;  
  
    public Message(int sequenceNumber) {  
        this.sequenceNumber = sequenceNumber;  
    }  
  
    @Override  
    public String toString() {  
        return this.getClass().getSimpleName() +  
            " [sequenceNumber=" + sequenceNumber + "];"  
    }  
  
    @Override  
    public int hashCode() { ... }  
  
    @Override  
    public boolean equals(Object other) { ... }  
}
```

Die Klasse ist im Projekt `flow-commons` definiert.

## 7.1 Ein einfacher Subscriber

Hier ein einfacher `SimpleSubscriber`, der das Interface `Subscriber` implementiert:

```
package jj.app1;

import java.util.concurrent.Flow.Subscriber;
import java.util.concurrent.Flow.Subscription;

import jj.util.base.XRunnable;

public class SimpleSubscriber<T> implements Subscriber<T> {

    private Subscription subscription;

    @Override
    public void onSubscribe(Subscription subscription) {
        Log.tlog("\tonSubscribe(" +
            subscription.getClass().getSimpleName() + ")");
        this.subscription = subscription;
        subscription.request(1);
    }

    @Override
    public void onNext(T item) {
        Log.tlog("\tonNext(" + item + ")");
        XRunnable.xrun(() -> Thread.sleep(100));
        this.subscription.request(1);
    }

    @Override
    public void onError(Throwable t) {
        Log.tlog("\tonError(" + t + ")");
    }

    @Override
    public void onComplete() {
        Log.tlog("\tonComplete()");
    }
}
```

Die obige Klasse muss alle vier Methoden des Interfaces implementieren. Alle `Subscriber`-Methoden sind vom Typ `void` – also "one way methods". Keine der Methoden darf eine checked `Exception` werfen:

- `onSubscribe` wird vom `Publisher` aufgerufen werden, wenn der `SimpleSubscriber` bei ihm via `subscribe` registriert wird. Der Methode wird ein `Subscription`-Objekt übergeben, welches einer Instanzvariablen zugewiesen wird (damit beim Aufruf von `onNext` eine neue Nachricht (resp. neue Nachrichten) angefordert werden kann (resp. können)). `onSubscribe` erklärt den `Subscriber` dann via `request(1)` bereit, genau eine folgende Nachricht verarbeiten zu können (der `Subscriber` ist somit auf den ersten Aufruf von `onNext` gefasst).
- Über den Aufruf von `onNext` empfängt der `Subscriber` eine neue Nachricht. Diese kann dann in `onNext` verarbeitet werden (z.B.: `sleep(100)`). Anschließend muss der `Subscriber` wiederum via `request` sich bereit erklären, eine neue Nachricht (resp. neue Nachrichten) zu empfangen. Der `SimpleSubscriber` ist bereit, wieder genau eine einzige Nachricht zu empfangen (`request(1)`).
- `onError` wird aufgerufen, wenn der `Publisher` einen Fehler erkannt hat. Nachdem einmal `onError` aufgerufen wurde, wird anschließend keine weitere Methode auf den `Subscriber` aufgerufen werden.
- `onComplete` wird aufgerufen, wenn der `Publisher` via `close()` geschlossen wird. Auch hier wird anschließend keine weitere `Subscriber`-Methode mehr aufgerufen werden.

Hinweis: Wird die `request`-Methode mit dem Wert `Long.MAX_VALUE` aufgerufen, erklärt sich der `Subscriber` bereit, beliebig viele Items zu empfangen).

Um Items zu veröffentlichen, wird die folgende Klasse `SubmissionPublisher` genutzt (die Klasse, die das Interface `Publisher` implementiert):

```
import java.util.concurrent.SubmissionPublisher;
```

In der `demoSubmit`-Methode werden 10 Messages via `submit` gesendet:

```
static void demoSubmit() {
    SubmissionPublisher<Message> publisher =
        new
SubmissionPublisher<>(ForkJoinPool.commonPool(), 1);

    System.out.println("MaxBufferCapacity : " +
        publisher.getMaxBufferCapacity());

    SimpleSubscriber<Message> subscriber = new
SimpleSubscriber<>();
    publisher.subscribe(subscriber);
}
```

```
        for (int i = 1000; i < 1010; i++) {
            Message message = new Message(i);
            Log.tlog("submit(" + message + ")");
            publisher.submit(new Message(i));
        }

        publisher.close();
        XRunnable.xrun(() -> Thread.sleep(2000));
    }
```

Bei der Erzeugung des `SubmissionPublishers` wird ein Konstruktor benutzt, dem zwei Argumente übergeben werden: ein `Executor` und eine Puffer-Größe. Der `Executor` wird vom `SubmissionPublisher` als Thread-Pool genutzt. Die `Subscription`, die von `subscribe` erzeugt wird, wird einen Puffer der Größe 1 besitzen.

(Wir hätten auch den parameterlosen Konstruktor von `SubmissionPublisher` benutzen können – die Puffergröße wäre dann 256 gewesen. Zu groß, um das Verhalten des `SubmissionPublishers` näher untersuchen zu können...)

Der erzeugte `SimpleSubscriber` wird beim `Publisher` via `subscribe` registriert. Dieser Aufruf wird dazu führen, dass ein neues `Subscription`-Objekt erzeugt wird und die `Subscriber`-Methode `onSubscribe` aufgerufen wird (wobei die `Subscription` übergeben wird). Der Aufruf von `onSubscribe` erfolgt dabei aber in einem anderen Thread als der Aufruf von `subscribe`! (`subscribe` wartet also nicht darauf, dass `onSubscribe` aufgerufen wird und zurückkehrt. Dasselbe gilt für das Verhältnis von `submit` und `onNext`.)

Da die `onSubscribe`-Methode auf die übergebene `Subscription` die Methode `request(1)` aufruft, wird beim ersten Aufruf der `Publisher.submit`-Methode diese die `Subscription`-Methode `onNext` aufgerufen – wobei die an `submit` übergebene `Message` an `onNext` weitergereicht wird. `onNext` verarbeitet die `Message` und fordert dann über die `Subscription` ein weiteres Item an (via `request(1)`).

Wir können auf den `Publisher` also beliebig häufig die `submit`-Methode aufrufen – stets wird dann die `onNext`-Methodes des `Subscribers` aufgerufen. Wenn der `Publisher` via `close()` geschossen wird, wird dies dem `Subscriber` via `onComplete()` mitgeteilt.

Man beachte, dass die auf den `Publisher` aufgerufene `submit`-Methode blockiert – sie blockiert solange, bis der `Subscriber` mittels des `request`-Aufrufs sich für den Empfang einer weiteren Nachricht bereit erklärt hat.

Die Geschwindigkeit des `Publishers` hängt also von der Geschwindigkeit des `Subscribers` ab (diese Kopplung wäre auch bei einem größeren Puffer gegeben – sie

würde nur anfangs nicht sofort in Erscheinung treten...). Der (die) `Subscriber` bremsen also den `Publisher` aus.

Hier die Ausgaben des Aufrufs der obigen `demoSubmit`-Methode (sie zeigen, dass alle an `submit` übergebenen Nachrichten früher oder später den `Subscriber` zugestellt werden):

```
[ 1] -- MaxBufferCapacity : 1
[12] -- onSubscribe(BufferedSubscription)
[ 1] -- submit(Message [sequenceNumber=1000])
[ 1] -- submit(Message [sequenceNumber=1001])
[ 1] -- submit(Message [sequenceNumber=1002])
[ 1] -- submit(Message [sequenceNumber=1003])
[12] -- onNext(Message [sequenceNumber=1000])
[12] -- onNext(Message [sequenceNumber=1001])
[ 1] -- submit(Message [sequenceNumber=1004])
[12] -- onNext(Message [sequenceNumber=1002])
[ 1] -- submit(Message [sequenceNumber=1005])
[ 1] -- submit(Message [sequenceNumber=1006])
[12] -- onNext(Message [sequenceNumber=1003])
[12] -- onNext(Message [sequenceNumber=1004])
[ 1] -- submit(Message [sequenceNumber=1007])
[12] -- onNext(Message [sequenceNumber=1005])
[ 1] -- submit(Message [sequenceNumber=1008])
[ 1] -- submit(Message [sequenceNumber=1009])
[12] -- onNext(Message [sequenceNumber=1006])
[12] -- onNext(Message [sequenceNumber=1007])
[12] -- onNext(Message [sequenceNumber=1008])
[12] -- onNext(Message [sequenceNumber=1009])
[12] -- onComplete()
```

In der Methode `demoOffer` wird nun statt der `submit`-Methode die `offer`-Methode des `Publishers` aufgerufen – eine Methode, die sich komplett anders verhält als `submit`.

```
static void demoOffer() {
    SubmissionPublisher<Message> publisher = new
SubmissionPublisher<>(
        ForkJoinPool.commonPool(), 1);

    System.out.println("MaxBufferCapacity : " +
        publisher.getMaxBufferCapacity());

    SimpleSubscriber<Message> subscriber = new
SimpleSubscriber<>();
    publisher.subscribe(subscriber);

    for (int i = 1000; i < 1020; i++) {
        XRunnable.xrun(() -> Thread.sleep(50));
    }
}
```

```
        Message message = new Message(i);
        System.out.println("publish(" + message + ")");
        publisher.offer(message, null);
    }

    publisher.close();
    XRunnable.xrun(() -> Thread.sleep(2000));
}
```

`demoOffer` bietet dem Publisher 20 Messages an. Zwischen der Produktion jeder Message vergehen 50 ms (man erinnere sich daran, dass der `SimpleSubscriber` 100 ms für die Verarbeitung einer Nachricht benötigt).

Hier die Ausgaben:

```
[ 1] -- MaxBufferCapacity : 1
[12] -- onSubscribe(BufferedSubscription)
[ 1] -- publish(Message [sequenceNumber=1000])
[12] -- onNext(Message [sequenceNumber=1000])
[ 1] -- publish(Message [sequenceNumber=1001])
[12] -- onNext(Message [sequenceNumber=1001])
[ 1] -- publish(Message [sequenceNumber=1002])
[ 1] -- publish(Message [sequenceNumber=1003])
[12] -- onNext(Message [sequenceNumber=1002])
[ 1] -- publish(Message [sequenceNumber=1004])
[ 1] -- publish(Message [sequenceNumber=1005])
[12] -- onNext(Message [sequenceNumber=1003])
[ 1] -- publish(Message [sequenceNumber=1006])
[ 1] -- publish(Message [sequenceNumber=1007])
[12] -- onNext(Message [sequenceNumber=1004])
[ 1] -- publish(Message [sequenceNumber=1008])
[ 1] -- publish(Message [sequenceNumber=1009])
[12] -- onNext(Message [sequenceNumber=1006])
[ 1] -- publish(Message [sequenceNumber=1010])
[ 1] -- publish(Message [sequenceNumber=1011])
[12] -- onNext(Message [sequenceNumber=1008])
[ 1] -- publish(Message [sequenceNumber=1012])
[ 1] -- publish(Message [sequenceNumber=1013])
[12] -- onNext(Message [sequenceNumber=1010])
[ 1] -- publish(Message [sequenceNumber=1014])
[ 1] -- publish(Message [sequenceNumber=1015])
[12] -- onNext(Message [sequenceNumber=1012])
[ 1] -- publish(Message [sequenceNumber=1016])
[ 1] -- publish(Message [sequenceNumber=1017])
[12] -- onNext(Message [sequenceNumber=1014])
[ 1] -- publish(Message [sequenceNumber=1018])
[ 1] -- publish(Message [sequenceNumber=1019])
[12] -- onNext(Message [sequenceNumber=1016])
```

```
[12] -- onNext(Message [sequenceNumber=1018])  
[12] -- onComplete()
```

Wie man erkennt, kehrt `offer` sofort zurück (und wartet also nicht darauf, dass die angebotene Nachricht tatsächlich versendet wurde). Der `Publisher` wird hier also im Gegensatz zur Verwendung von `submit` nicht(!) ausgebremst. D.h. natürlich auch, dass nicht alle Nachrichten den `Subscriber` erreichen – im Schnitt wird jede zweite Nachricht "verschluckt".

Die Geschwindigkeit des `Publishers` ist somit völlig unabhängig von der des (der) `Subscriber` – allerdings eben um den Preis, dass nicht mehr alle an `offer` übergebenen Nachrichten auch den (die) `Subscriber` erreichen.

Neben der oben benutzten `offer`-Methoden existieren zwei überladene `offer`-Methoden, die wie folgt aufgerufen werden können:

```
publisher.offer(message,  
    (x, y) -> true);
```

```
publisher.offer(message,  
    0, TimeUnit.MILLISECONDS, (x, y) -> true);
```

Die Bedeutung dieser Methoden entnehme man der JavaDoc...

## 7.2 Ein StdSubscriber für's Testen

Für die Test-Methoden, die wir im Folgenden entwickeln, werden wir einen Standard-Subscriber nutzen. Dieser ist im Projekt `flow-common` implementiert (im package `jj.flow`). Der `StdSubscriber` ist seinerseits abgeleitet von einem `DelegatingHandler`, welcher die Aufrufe der `on...-Methoden` an `Consumer` resp. `BiConsumer` delegiert.

Wir definieren zunächst die Klasse `DelegatingSubscriber`, welche alle `on`-Aufrufe (`onSubscribe`, `onNext`, `onError`, `onComplete`) delegiert an `Consumer`- resp. `BiConsumer`-Objekte – welche aber zusätzlich Logging-Funktionalität und zwei `await`-Methoden implementiert:

```
package jj.flow;

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.Flow.Subscriber;
import java.util.concurrent.Flow.Subscription;
import java.util.function.BiConsumer;
import java.util.function.Consumer;

import static jj.util.base.XRunnable.xrun;
import static jj.util.log.Log.tlogEnter;
import static jj.util.log.Log.tlog;
import static jj.util.log.Log.tlogExit;

public class DelegatingSubscriber<T> implements Subscriber<T> {

    private final CountDownLatch done = new CountDownLatch(1);

    private Subscription subscription;
    private List<T> receivedItems = new ArrayList<>();

    private Consumer<Subscription> onSubscribeHandler = s -> { };
    private BiConsumer<Subscription, T> onNextHandler = (s, i) ->
{ };
    private BiConsumer<Subscription, Throwable> onErrorHandler =
(s, t) -> { };
    private Consumer<Subscription> onCompleteHandler = s -> { };

    public void onSubscribeHandler(
```



```
        Consumer<Subscription> onSubscribeHandler) {
            this.onSubscribeHandler =
Objects.requireNonNull(onSubscribeHandler);
        }
        public void onNextHandler(
            BiConsumer<Subscription, T> onNextHandler) {
            this.onNextHandler =
Objects.requireNonNull(onNextHandler);
        }
        public void onErrorHandler(
            BiConsumer<Subscription, Throwable> onErrorHandler) {
            this.onErrorHandler =
Objects.requireNonNull(onErrorHandler);
        }
        public void onCompleteHandler(
            Consumer<Subscription> onCompleteHandler) {
            this.onCompleteHandler =
Objects.requireNonNull(onCompleteHandler);
        }

        @Override
        public void onSubscribe(Subscription subscription) {
            tlogEnter(subscription.getClass().getSimpleName());
            this.subscription = subscription;
            this.onSubscribeHandler.accept(this.subscription);
            tlogExit();
        }

        @Override
        public void onNext(T item) {
            tlogEnter(item);
            this.receivedItems.add(item);
            this.onNextHandler.accept(this.subscription, item);
            tlogExit();
        }

        @Override
        public void onError(Throwable t) {
            tlogEnter(t);
            this.onErrorHandler.accept(this.subscription, t);
            done.countDown();
            tlogExit();
        }

        @Override
        public void onComplete() {
```

```
        tlogEnter();
        this.onCompleteHandler.accept(this.subscription);
        done.countDown();
        tlogExit();
    }

    public List<T> await() {
        xrun(() -> done.await());
        return this.receivedItems;
    }

    public void await(List<T> expectedItems) {
        xrun(() -> done.await());
        if (! this.receivedItems.equals(expectedItems))
            tlog("ERROR: expected: " + expectedItems +
                " but received: " + this.receivedItems);
        else
            tlog("SUCCESS");
    }
}
```

`DelegatingSubscriber<T>` implementiert das Interface `Flow.Subscriber<T>` - also die Methoden `onSubscribe`, `onNext`, `onError` und `onComplete`.

Die Klasse definiert vier `on...Handler`-Attribute, die über gleichnamige Methoden initialisiert werden: `onSubscribeHandler`, `onNextHandler`, `onErrorHandler` und `onCompleteHandler`. Mittels dieser Methoden können `Consumer` resp. `BiConsumer` registriert werden, an welche die Aufrufe von `onSubscribe`, `onNext` etc. delegiert werden.

Jede `on...-Methode` (`onSubscribe` etc.) diagnostiziert zunächst ihren Ein- und Ausstieg – mit der Methode `tlogEnter` resp. `tlogExit`. Alle Methoden delegieren jeweils an den passenden `on...Handler`.

Die an `onSubscribe` übergebene `Flow.Subscription` wird in einer Instanzvariablen gespeichert. Diese Instanzvariable wird in `onNext` an die `accept`-Methode des `onNextHandlers` übergeben.

Das jeweils an `onNext` übergebene Item wird zu einer Liste der empfangenen Items hinzugefügt (`receivedItems`).

Schließlich benutzt die Klasse einen `CountDownLatch`, der mit 1 initialisiert wird. Beim Aufruf von `onError` resp. `onComplete` (also bei der letzten Interaction des Publishers mit einem `DelegatingSubscriber`) wird der `CountDownLatch` um 1 dekrementiert. Eine Anwendung kann somit mittels des Aufrufs von `await` darauf warten, dass der

`DelegatingSubscriber` terminiert wurde. `await` liefert schließlich die `receivedItems`-Liste zurück.

`await` ist überladen. Der zweiten `await`-Methode wird die Liste der erwarteten Items übergeben. Die Methode vergleicht die Liste der tatsächlich empfangenen Items mit der Liste der erwarteten – und gibt im Falle einer Nicht-Übereinstimmung beide Listen in Form einer Fehlermeldung aus.

`DelegatingSubscriber` implementiert also eine technische Infrastruktur, welche insb. für Test- resp. Demonstrationszwecke genutzt werden kann.

Hier nun die Klasse `StdSubscriber`, die für fast alle folgenden Projekte genutzt wird:

```
package jj.flow;

import static jj.util.base.XRunnable.xrun;
import static jj.util.log.Log.tlog;

public class StdSubscriber<T> extends DelegatingSubscriber<T> {

    public StdSubscriber(int sleepTime) {
        this.onSubscribeHandler(s -> {
            int n = 1;
            tlog("request(" + n + ")");
            s.request(n);
        });
        this.onNextHandler((s, item) -> {
            int n = 1;
            tlog("sleep(" + sleepTime + ")");
            xrun(() -> Thread.sleep(sleepTime));
            tlog("request(" + n + ")");
            s.request(n);
        });
    }
}
```

Die Klasse ist abgeleitet von `DelegationSubscriber`.

Dem Konstruktor von `StdSubscriber` wird eine `sleepTime` mitgegeben, die zur Simulation "harter Arbeit" genutzt wird. Der Konstruktor ruft den Konstruktor der Basisklasse mit einem `onSubscribeConsumer` und einem `onNextConsumer` auf.

In der `accept`-Methode des `onSubscribeConsumers` wird `request` bereits das erste Mal aufgerufen, um das erste Item anzufordern (um die Bereitschaft zu signalisieren, dass der Subscriber ein weiteres Item verarbeiten kann).

In der `accept`-Methode des `onNextConsumers` wird harte Arbeit simuliert. Anschließend wird wieder via `request` die Bereitschaft bekundet, ein weiteres Item zu verarbeiten.

Hier eine Anwendung dieser `StdSubscriber`-Klasse:

```
static void demoSubscriber1() {  
    SubmissionPublisher<Message> publisher =  
        new SubmissionPublisher<>(ForkJoinPool.commonPool(),  
1);  
  
    StdSubscriber<Message> subscriber = new  
StdSubscriber<>(0);  
  
    Log.tlog("subscribe");  
    publisher.subscribe(subscriber);  
  
    Log.tlog("submit 1000");  
    publisher.submit(new Message(1000));  
  
    Log.tlog("close");  
    publisher.close();  
    subscriber.await(List.of(new Message(1000)));  
}
```

Die letzte Zeile wartet darauf, dass die `Subscriber`-Methode `onComplete` aufgerufen wird und sichert zu, dass die Liste der vom `Subscriber` empfangenen Items genau ein einziges Element enthält: `Message(1000)`.

Hier die Ausgaben der obigen `demo`-Methode:

```
[ 1] -- subscribe  
[ 1] -- submit 1000  
[12] >> DelegatingSubscriber.onSubscribe(BufferedSubscription)  
[ 1] -- close  
[12] -- request(1)  
[12] << DelegatingSubscriber.onSubscribe()  
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1000])  
[12] -- sleep(0)  
[12] -- request(1)  
[12] << DelegatingSubscriber.onNext()  
[12] >> DelegatingSubscriber.onComplete()  
[12] << DelegatingSubscriber.onComplete()  
[ 1] -- SUCCESS
```

Die Ausgabe zeigt, dass alle Interaktionen zwischen der Subscription und dem Subscriber im Thread mit der ID 12 ablaufen. Der Hauptthread (ID = 1) stößt also nur die eigentliche Verarbeitung an, die dann asynchron verläuft.

Wir könnten eine Variante des `StdSubscribers` nutzen, welche für die Reaktion auf `onSubscribe` und für die eigentliche Verarbeitung der empfangenen Items jeweils einen neuen Thread startet – und in diesem dann auch die Bereitschaft zur Verarbeitung weiterer Items bekundet (via `Subscription.request()`):

```
package jj.appl;

import static jj.util.log.Log.tlog;
import static jj.util.base.XRunnable.xrun;

import jj.flow.DelegatingSubscriber;

public class AnotherSubscriber<T> extends DelegatingSubscriber<T>
{
    public AnotherSubscriber(int sleepTime) {
        this.onSubscribeHandler(s -> {
            new Thread(() -> {
                int n = 1;
                tlog("request(" + n + ")");
                s.request(n);
            }).start();
        });
        this.onNextHandler((s, item) -> {
            new Thread(() -> {
                int n = 1;
                xrun(() -> Thread.sleep(sleepTime));
                tlog("request(" + n + ")");
                s.request(n);
            }).start();
        });
    }
}
```

Die demo-Methode:

```
static void demoSubscriber2() {
    SubmissionPublisher<Message> publisher =
        new SubmissionPublisher<>(
            ForkJoinPool.commonPool(), 1);
}
```

```
AnotherSubscriber<Message> subscriber =
    new AnotherSubscriber<>(100)

Log.tlog("subscribe");
publisher.subscribe(subscriber);

Log.tlog("submit 1000");
publisher.submit(new Message(1000));

Log.tlog("close");
publisher.close();

subscriber.await(List.of(new Message(1000)));
XRunnable.xrun(() -> Thread.sleep(1000));
}
```

(Man beachte die letzte Zeile...)

Hier die Ausgaben dieser Methode:

```
[ 1] -- subscribe
[ 1] -- submit 1000
[ 1] -- close
[12] >> DelegatingSubscriber.onSubscribe(BufferedSubscription)
[12] << DelegatingSubscriber.onSubscribe()
[13] -- request(1)
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1000])
[12] << DelegatingSubscriber.onNext()
[12] >> DelegatingSubscriber.onComplete()
[12] << DelegatingSubscriber.onComplete()
[ 1] -- SUCCESS
[14] -- request(1)
```

Die `request`-Methode wird nun in den Threads mit den IDs 13 und 14 aufgerufen. Zusätzlich zu den asynchronen Aufrufen des Subscribers kann also auch dieser die eigentliche Verarbeitung wiederum asynchron betreiben...

## 7.3 Requests

Im Folgenden wird ein `Subscriber` benutzt, der jeweils via `request(5)` fünf weitere Items anfordert. Natürlich darf dann der `request(5)`-Aufruf nur bei jedem fünften `onNext`-Aufruf stattfinden.

Die Items, die an `onNext` übergeben werden, werden in einem Array gesammelt. Die Elemente des Arrays werden immer dann verarbeitet, bevor in `onNext` die `request`-Methode aufgerufen wird.

```
package jj.app1;

import static jj.util.base.XRunnable.xrun;
import static jj.util.log.Log.tlog;

import jj.flow.DelegatingSubscriber;

public class MySubscriber<T> extends DelegatingSubscriber<T> {

    private int counter = 0;
    private final int N = 5;
    private final T[] array = (T[]) new Object[N-1];

    public MySubscriber(int sleepTime) {
        this.onSubscribeHandler(s -> {
            this.counter = 0;
            tlog("request(" + this.N + ")");
            s.request(this.N);
        });
        this.onNextHandler((s, item) -> {
            if (counter < N - 1) {
                this.array[this.counter] = item;
                this.counter++;
                return;
            }
            tlog("==> working...");
            for(T elem : this.array) {
                tlog("    " + elem);
            }
            xrun(() -> Thread.sleep(sleepTime));
            this.counter = 0;
            tlog("request(" + this.N + ")");
            s.request(this.N);
        });
    }
}
```

```
}  
}
```

```
static void demo() {  
  
    SubmissionPublisher<Message> publisher = new  
SubmissionPublisher<>();  
  
    MySubscriber<Message> subscriber = new  
MySubscriber<>(100);  
  
    publisher.subscribe(subscriber);  
  
    List<Message> expected = new ArrayList<>();  
    for (int i = 1000; i < 1015; i++) {  
        Message message = new Message(i);  
        expected.add(message);  
        tlog("submit " + i);  
        xrun(() -> Thread.sleep(100));  
        publisher.submit(message);  
    }  
  
    tlog("close");  
    publisher.close();  
    subscriber.await(expected);  
}
```

### Die Ausgaben:

```
[12] >> DelegatingSubscriber.onSubscribe(BufferedSubscription)  
[ 1] -- submit 1000  
[12] -- request(5)  
[12] << DelegatingSubscriber.onSubscribe()  
[ 1] -- submit 1001  
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1000])  
[12] << DelegatingSubscriber.onNext()  
[ 1] -- submit 1002  
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1001])  
[12] << DelegatingSubscriber.onNext()  
[ 1] -- submit 1003  
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1002])  
[12] << DelegatingSubscriber.onNext()  
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1003])  
[12] << DelegatingSubscriber.onNext()  
[ 1] -- submit 1004  
[ 1] -- submit 1005  
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1004])
```



```
[12] -- ==> working...
[12] --      Message [sequenceNumber=1000]
[12] --      Message [sequenceNumber=1001]
[12] --      Message [sequenceNumber=1002]
[12] --      Message [sequenceNumber=1003]
[ 1] -- submit 1006
[12] -- request(5)
[12] << DelegatingSubscriber.onNext()
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1005])
[12] << DelegatingSubscriber.onNext()
[ 1] -- submit 1007
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1006])
[12] << DelegatingSubscriber.onNext()
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1007])
[12] << DelegatingSubscriber.onNext()
[ 1] -- submit 1008
[ 1] -- submit 1009
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1008])
[12] << DelegatingSubscriber.onNext()
[ 1] -- submit 1010
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1009])
[12] -- ==> working...
[12] --      Message [sequenceNumber=1005]
[12] --      Message [sequenceNumber=1006]
[12] --      Message [sequenceNumber=1007]
[12] --      Message [sequenceNumber=1008]
[ 1] -- submit 1011
[12] -- request(5)
[12] << DelegatingSubscriber.onNext()
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1010])
[12] << DelegatingSubscriber.onNext()
[ 1] -- submit 1012
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1011])
[12] << DelegatingSubscriber.onNext()
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1012])
[12] << DelegatingSubscriber.onNext()
[ 1] -- submit 1013
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1013])
[12] << DelegatingSubscriber.onNext()
[ 1] -- submit 1014
[ 1] -- close
[12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=1014])
[12] -- ==> working...
[12] --      Message [sequenceNumber=1010]
[12] --      Message [sequenceNumber=1011]
[12] --      Message [sequenceNumber=1012]
[12] --      Message [sequenceNumber=1013]
[12] -- request(5)
[12] << DelegatingSubscriber.onNext()
[12] >> DelegatingSubscriber.onComplete()
[12] << DelegatingSubscriber.onComplete()
```

```
[ 1] -- SUCCESS
```

## 7.4 SubmissionPublisher - Details

Auf einen `SubmissionPublisher` kann die Methode `consume` aufgerufen werden. Diese verlangt als Parameter einen `Consumer` und liefert ein `CompletableFuture` zurück. Dieses `CompletableFuture`-Objekt kann dann via `get()` ausgeführt werden:

```
static void demoConsume() {  
  
    SubmissionPublisher<Message> publisher =  
        new SubmissionPublisher<>(ForkJoinPool.commonPool(),  
1);  
  
    CompletableFuture<Void> future =  
publisher.consume((Message m) -> {  
        xrun(() -> Thread.sleep(10));  
        tlog("accept " + m);  
    });  
  
    for (int i = 0; i < 10; i++) { // 300 !!  
        tlog("submit " + i);  
        publisher.submit(new Message(i));  
    }  
  
    tlog("close");  
    publisher.close();  
  
    xrun(() -> future.get());  
}
```

Die Ausgaben:

```
[ 1] -- submit 0  
[ 1] -- submit 1  
[ 1] -- submit 2  
[ 1] -- submit 3  
    [12] -- accept Message [sequenceNumber=0]  
[ 1] -- submit 4  
    [12] -- accept Message [sequenceNumber=1]  
[ 1] -- submit 5  
    [12] -- accept Message [sequenceNumber=2]  
[ 1] -- submit 6  
    [12] -- accept Message [sequenceNumber=3]  
[ 1] -- submit 7  
    [12] -- accept Message [sequenceNumber=4]  
[ 1] -- submit 8  
    [12] -- accept Message [sequenceNumber=5]
```

```
[ 1] -- submit 9
    [12] -- accept Message [sequenceNumber=6]
[ 1] -- close
    [12] -- accept Message [sequenceNumber=7]
    [12] -- accept Message [sequenceNumber=8]
    [12] -- accept Message [sequenceNumber=9]
```

Die folgende `demo`-Methode zeigt die Benutzung einer Reihe weiterer Methoden der Klasse `SubmissionPublisher`:

```
static void demoInspect() {

    SubmissionPublisher<Message> publisher = new
SubmissionPublisher<>();

    StdSubscriber<Message> subscriber1 = new
StdSubscriber<>(10);
    StdSubscriber<Message> subscriber2 = new
StdSubscriber<>(10);

    publisher.subscribe(subscriber1);
    publisher.subscribe(subscriber2);

    tlog(publisher.getExecutor());
    tlog(publisher.getMaxBufferCapacity());
    tlog(publisher.getNumberOfSubscribers());
    tlog(publisher.hasSubscribers());
    tlog(publisher.getSubscribers());
    tlog(publisher.isSubscribed(subscriber1));

    publisher.close();
    subscriber1.await();
    subscriber2.await();
}
```

Die Ausgaben:

```
[ 1] -- java.util.concurrent.ForkJoinPool@7ca48474[...]
[ 1] -- 256
[ 1] -- 2
[ 1] -- true
    [13] >> DelegatingSubscriber.onSubscribe(...)
[ 1] -- [jj.flow.StdSubscriber@337d0578,
        jj.flow.StdSubscriber@59e84876]
    [12] >> DelegatingSubscriber.onSubscribe(...)
    [12] -- request(1)
    [12] << DelegatingSubscriber.onSubscribe()
```

```
[ 1] -- true
[13] -- request(1)
[13] << DelegatingSubscriber.onSubscribe()
[13] >> DelegatingSubscriber.onComplete()
[13] << DelegatingSubscriber.onComplete()
[12] >> DelegatingSubscriber.onComplete()
[12] << DelegatingSubscriber.onComplete()
```

Die folgende `demo`-Methode schließlich zeigt die Benutzung der `estimate...-Methoden` der Klasse `SubmissionPublisher`:

```
static void demoEstimate() {

    SubmissionPublisher<Message> publisher = new
SubmissionPublisher<>();

    StdSubscriber<Message> subscriber = new
StdSubscriber<>(100);

    publisher.subscribe(subscriber);

    tlog(publisher.estimateMinimumDemand());
    tlog(publisher.estimateMaximumLag());

    for (int i = 0; i < 5; i++) {
        tlog("submit " + new Message(i));
        publisher.submit(i);
    }

    tlog(publisher.estimateMinimumDemand());
    tlog(publisher.estimateMaximumLag());

    tlog("close");
    publisher.close();
    subscriber.await();
}
```

Die Ausgaben:

```
[ 1] -- 0
[ 1] -- 0
[ 1] -- submit 0
[ 1] -- submit 1
[ 1] -- submit 2
[ 1] -- submit 3
[ 1] -- submit 4
[ 1] -- -5
[ 1] -- 5
```

```
[ 1] -- close
    [12] >> DelegatingSubscriber.onSubscribe(BufferedSubscription)
    [12] -- request(1)
    [12] << DelegatingSubscriber.onSubscribe()
    [12] >> DelegatingSubscriber.onNext(Message [sequenceNumber=0)
    [12] -- sleep(100)
    [12] -- request(1)
...

```

## 7.5 Ein PeriodicPublisher

Wir können `SubmissionPublisher` als Basisklasse für spezielle `Publisher` verwenden. Im Folgenden wird ein `Publisher` vorgestellt, der periodisch Meldungen verschickt:

```
package jj.appl;

import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
// ...

class PeriodicPublisher<T> extends SubmissionPublisher<T> {

    private final ScheduledFuture<?> periodicTask;
    private final ScheduledExecutorService scheduler;

    public PeriodicPublisher(int millis, Supplier<? extends T>
supplier) {
        this.scheduler = new ScheduledThreadPoolExecutor(1);
        this.periodicTask = scheduler.scheduleAtFixedRate(
            () -> {
                T element = supplier.get();
                tlog("submit " + element);
                this.offer(element, null);
            },
            0, millis, TimeUnit.MILLISECONDS);
    }

    public void close() {
        this.periodicTask.cancel(false);
        this.scheduler.shutdown();
        super.close();
    }
}
```

```
static void demo() {

    AtomicInteger number = new AtomicInteger(42);

    tlog("creating Publisher");
    PeriodicPublisher<Message> publisher =
        new PeriodicPublisher<>(1000,
```

```
        () -> new Message(number.getAndIncrement())));

        StdSubscriber<Message> subscriber = new
StdSubscriber<>(1);
        xrun(() -> Thread.sleep(2000));

        tlog("subscribe");
        publisher.subscribe(subscriber);

        xrun(() -> Thread.sleep(5000));

        tlog("close");
        publisher.close();
        subscriber.await();
    }
}
```

### Die Ausgaben:

```
[ 1] -- creating Publisher
[12] -- submit 42
[12] -- submit 43
[12] -- submit 44
[ 1] -- subscribe
[13] >> DelegatingSubscriber.onSubscribe(BufferedSubscription)
[13] -- request(1)
[13] << DelegatingSubscriber.onSubscribe()
[12] -- submit 45
[13] >> DelegatingSubscriber.onNext(
        Message [sequenceNumber=45])
[13] -- sleep(1)
[13] -- request(1)
[13] << DelegatingSubscriber.onNext()
[12] -- submit 46
[13] >> DelegatingSubscriber.onNext(
        Message [sequenceNumber=46])
[13] -- sleep(1)
[13] -- request(1)
[13] << DelegatingSubscriber.onNext()
[12] -- submit 47
[13] >> DelegatingSubscriber.onNext(47)
[13] -- sleep(1)
[13] -- request(1)
[13] << DelegatingSubscriber.onNext()
[12] -- submit 48
[13] >> DelegatingSubscriber.onNext(
        Message [sequenceNumber=48])
[13] -- sleep(1)
[13] -- request(1)
[13] << DelegatingSubscriber.onNext()
```



```
[12] -- submit 49
      [13] >> DelegatingSubscriber.onNext(
            Message [sequenceNumber=49])
      [13] -- sleep(1)
      [13] -- request(1)
      [13] << DelegatingSubscriber.onNext()
[ 1] -- close
      [13] >> DelegatingSubscriber.onComplete()
      [13] << DelegatingSubscriber.onComplete()
```

## 7.6 Processors

Das `Flow.Processor`-Interface ist sowohl von `Flow.Subscriber` als auch von `Flow.Publisher` abgeleitet. Ein `Processor` kann also sowohl als `Subscriber` bei einem `Publisher` registriert werden als auch selbst wiederum als `Publisher` fungieren.

Ein Item, welches ein `Processor` von einem `Publisher` via `onNext` zugestellt wird, kann z.B. in ein anderes Item transformiert werden, welches dann an den bei einem `Processor` registrierten `Subscribern` gesendet wird. Oder ein `Processor` kann als Filter fungieren etc.

Da `Processor` nur ein Interface ist, benötigt eine `Processor`-Implementierung natürlich einen "konkreten" `Publisher`. Wir leiten also im Folgenden die `Processor`-Klassen einfach von `SubmissionPublisher` ab (und implementieren somit das `Publisher`-Interface).

Dem folgenden `MapProcessor` wird eine `Function` übergeben, welche den Input des `Processors` auf den Output mappt:

```
package jj.utils;

import java.util.concurrent.Flow.Processor;
import java.util.concurrent.Flow.Subscription;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.SubmissionPublisher;
import java.util.function.Function;

import jj.flow.DelegatingSubscriber;

public class MapProcessor<T, R>
    extends SubmissionPublisher<R>
    implements Processor<T, R> {

    private DelegatingSubscriber<T> subscriber =
        new DelegatingSubscriber<>();

    public MapProcessor(Function<? super T, ? extends R>
function) {
        super(ForkJoinPool.commonPool(), 1);
        this.subscriber.onSubscribeHandler(s -> s.request(1));
        this.subscriber.onNextHandler((s, item) -> {
            this.submit((R) function.apply(item));
            s.request(1);
        });
    }
}
```

```
        this.subscriber.onCompleteHandler(s -> this.close());
    }

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscriber.onSubscribe(subscription);
    }

    @Override
    public void onNext(T item) {
        this.subscriber.onNext(item);
    }

    @Override
    public void onError(Throwable t) {
        this.subscriber.onError(t);
    }

    @Override
    public void onComplete() {
        this.subscriber.onComplete();
    }
}
```

Der folgenden `FilterProcessor`-Klasse wird ein `Predicate` übergeben. Nur solche Input-Items, mit denen das `Predicate` zufrieden ist, werden weiter verschickt:

```
package jj.utils;

import java.util.concurrent.Flow.Processor;
import java.util.concurrent.Flow.Subscription;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.SubmissionPublisher;
import java.util.function.Predicate;

import jj.flow.DelegatingSubscriber;

public class FilterProcessor<T>
    extends SubmissionPublisher<T>
    implements Processor<T, T> {

    private final DelegatingSubscriber<T> subscriber =
        new DelegatingSubscriber<>();

    public FilterProcessor(Predicate<? super T> predicate) {
        super(ForkJoinPool.commonPool(), 1);
        this.subscriber.onSubscribeHandler(s -> s.request(1));
        this.subscriber.onNextHandler((s, item) -> {
            if (predicate.test(item))
```

```

        this.submit(item);
        s.request(1);
    });
    this.subscriber.onCompleteHandler(s -> this.close());
}

@Override
public void onSubscribe(Subscription subscription) {
    this.subscriber.onSubscribe(subscription);
}
@Override
public void onNext(T item) {
    this.subscriber.onNext(item);
}
@Override
public void onError(Throwable t) {
    this.subscriber.onError(t);
}
@Override
public void onComplete() {
    this.subscriber.onComplete();
}
}

```

Die folgenden `demo`-Methoden benutzen allesamt eine kleine Hilfsmethode, die mit Hilfe eines an ihr übergebenen `SubmissionPublishers` die Strings "red", "green" und "blue" verschickt:

```

static private void submitSomeItemsAndClose(
    SubmissionPublisher<String> publisher) {
    String[] colors = new String[] { "red", "green",
"blue" };
    for (String color : colors) {
        tlog("submit " + color);
        publisher.submit(color);
    }
    tlog("close");
    publisher.close();
}

```

Hier eine Anwendung der oben vorgestellten `MapProcessor`-Klasse – die Klasse mappt Strings auf deren Längen:

```

static void demoMapProcessor() {

    SubmissionPublisher<String> publisher =

```

```
        new SubmissionPublisher<>(ForkJoinPool.commonPool(),
1);

        MapProcessor<String, Integer> processor =
            new MapProcessor<>(s -> s.length());
        tlog("subscribe processor");
        publisher.subscribe(processor);

        StdSubscriber<Integer> subscriber = new
StdSubscriber<>(100);
        tlog("subscribe subscriber");
        processor.subscribe(subscriber);

        submitSomeItemsAndClose(publisher);
        subscriber.await(List.of(3, 5, 4));
    }
```

Die Ausgaben sollen hier nicht weiter analysiert werden...

Hier eine Anwendung der `FilterProcessor`-Klasse (es werden aus dem Input-Strom alle Strings herausgefiltert, deren Länge kleiner als 4 sind – resp.: es werden nur solche Strings weitergereicht, die 4 oder mehr Zeichen beinhalten):

```
static void demoFilterProcessor() {

    SubmissionPublisher<String> publisher =
        new SubmissionPublisher<>(ForkJoinPool.commonPool(),
1);

    FilterProcessor<String> processor =
        new FilterProcessor<>(s -> s.length() >= 4);
    tlog("subscribe processor");
    publisher.subscribe(processor);

    StdSubscriber<String> subscriber = new
StdSubscriber<>(100);
    tlog("subscribe subscriber");
    processor.subscribe(subscriber);

    submitSomeItemsAndClose(publisher);
    subscriber.await(List.of("green", "blue"));
}
```

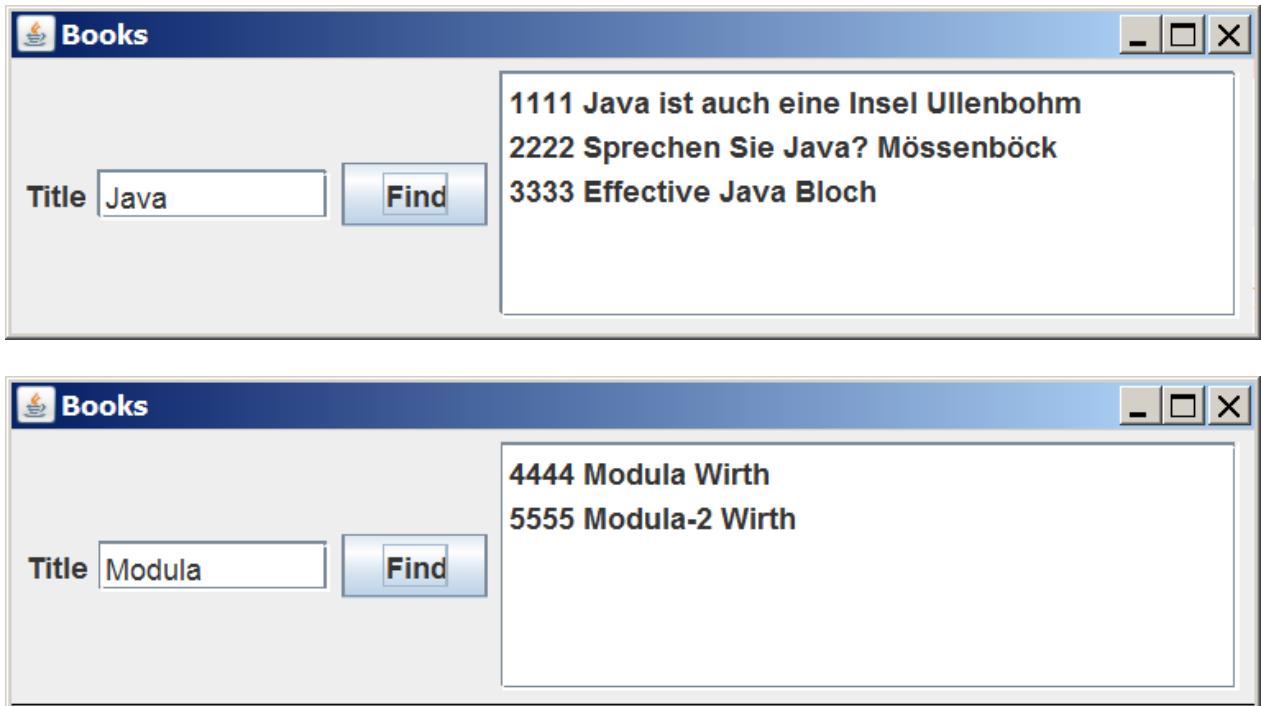
Auch hier sollen die Ausgaben nicht weiter analysiert werden...

Und hier schließlich eine Anwendung, die einen `MapProcessor` und einen `FilterProcessor` "in Reihe" schaltet:

```
static void demoMapFilterProcessor() {  
  
    SubmissionPublisher<String> publisher =  
        new SubmissionPublisher<>(ForkJoinPool.commonPool(),  
1);  
  
    MapProcessor<String, Integer> processor1 =  
        new MapProcessor<>(s -> s.length());  
    tlog("subscribe processor 1");  
    publisher.subscribe(processor1);  
  
    FilterProcessor<Integer> processor2 =  
        new FilterProcessor<>(i -> i >= 4);  
    tlog("subscribe processor 2");  
    processor1.subscribe(processor2);  
  
    StdSubscriber<Integer> subscriber = new  
StdSubscriber<>(100);  
    tlog("subscribe subscriber");  
    processor2.subscribe(subscriber);  
  
    submitSomeItemsAndClose(publisher);  
    subscriber.await(List.of(5, 4));  
}
```

## 7.7 Beispiel: Datenbank

Die folgende GUI-Anwendung benutzt eine Datenbank (`BookDatabase`):



Mittels der Betätigung des "Find"-Buttons wird eine Anfrage an die Datenbank geschickt, welche diese in einem neuen Thread bearbeitet. In diesem Thread wird ein `Publisher` erzeugt, welcher die der Anfrage entsprechenden `Book`-Objekte ermittelt und jedes dieser Objekte via `submit` verschickt.

Der Anfrage wird neben dem Buchtitel ein `Subscriber<Book>` übergeben. Für jedes von der Datenbank ermittelte und via `submit` verschickte `Book`-Objekt wird die `onNext`-Methode dieses `Subscribers` aufgerufen werden. Diese fügt dann das jeweilige `Book` zu der listenförmige Anzeige hinzu.

Der `Publisher` wird bewußt via `Thread.sleep` verlangsamt. Die `Book`-Objekte werden an `submit` im Abstand von jeweils einer Sekunde übergeben. Entsprechend langsam wird die Anzeige erfolgen. Die Oberfläche ist aber unmittelbar nach Betätigung des "Find"-Buttons wieder aktiv – denn über diesen Button wird ja nur ein neuer Thread gestartet.

Die GUI-Komponente (`BookFrame`) kennt die `BookDatabase` (sie ruft die `find`-Methode auf) – die `BookDatabase` allerdings weiß nichts von der Oberfläche. Die `BookDatabase` ist also an den `BookFrame` nur über das `Subscriber`-Interface gekoppelt.

## jj.domain.Book

```
package jj.domain;

public class Book {

    public final String isbn;
    public final String title;
    public final String author;

    public Book(String isbn, String title, String author) { ... }

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

## jj.database.BookDatabase

```
package jj.database;
// ...
public class BookDatabase {

    private final List<Book> books = new ArrayList<>();
    {
        books.add(new Book("1111", "Java ist auch eine Insel",
"Ulllenbohm"));
        books.add(new Book("2222", "Sprechen Sie Java?",
"Mössenböck"));
        books.add(new Book("3333", "Effective Java", "Bloch"));
        books.add(new Book("4444", "Modula", "Wirth"));
        books.add(new Book("5555", "Modula-2", "Wirth"));
    }

    public void find(Subscriber<Book> subscriber, String title) {
        new Thread(() -> this.doFind(subscriber, title)).start();
    }

    private void doFind(Subscriber<Book> subscriber, String
title) {
        try (final SubmissionPublisher<Book> publisher =
            new SubmissionPublisher<>()) {
            publisher.subscribe(subscriber);
            this.books.forEach(book -> {
                if (book.title.contains(title)) {
                    XRunnable.xrun(() -> Thread.sleep(1000));
                    publisher.submit(book);
                }
            });
        }
    }
}
```



```

        }
    });
}
}
}

```

Die `find`-Methode, die mit einer Titel und einem `Subscriber` aufgerufen wird, startet einen neuen Thread, in welchem ein `Publisher` erzeugt wird, bei dem der `Subscriber` registriert wird. Dieser `Publisher` ermittelt alle zur Anfrage passenden `Book`-Objekte und verschickt jedes dieser Objekte via `submit`. Dabei wird "harte" Arbeit simuliert. Die `find`-Methode kehrt sofort zum Aufrufer zurück.

## jj.gui.BookSubscriber

```

package jj.gui;
// ...
import javax.swing.DefaultListModel;
import javax.swing.JList;
import javax.swing.SwingUtilities;

public class BookSubscriber implements Flow.Subscriber<Book> {

    private final DefaultListModel<Book> listModel;
    private Subscription subscription;

    public BookSubscriber(DefaultListModel<Book> listModel) {
        this.listModel = listModel;
    }
    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        this.subscription.request(1);
    }

    @Override
    public void onNext(Book book) {
        System.out.println("onNext(" + book + ")");
        SwingUtilities.invokeLater(() ->
this.listModel.addElement(book));
        this.subscription.request(1);
    }

    @Override
    public void onError(Throwable throwable) { ... }

    @Override

```

```
    public void onComplete() { ... }  
}
```

Die Klasse `BookSubscriber` implementiert `Flow.Subscriber<Book>`. Beim Erzeugen eines `BookSubscriber`s wird ein `DefaultListModel` übergeben, über welches die Anzeige aktualisiert werden wird. `onSubscribe` wird das erste `Book` angefordert. Beim Aufruf von `onNext` wird das übergebene `Book` in das `DefaultListModel` eingetragen und jeweils ein weiteres `Book` angefordert.

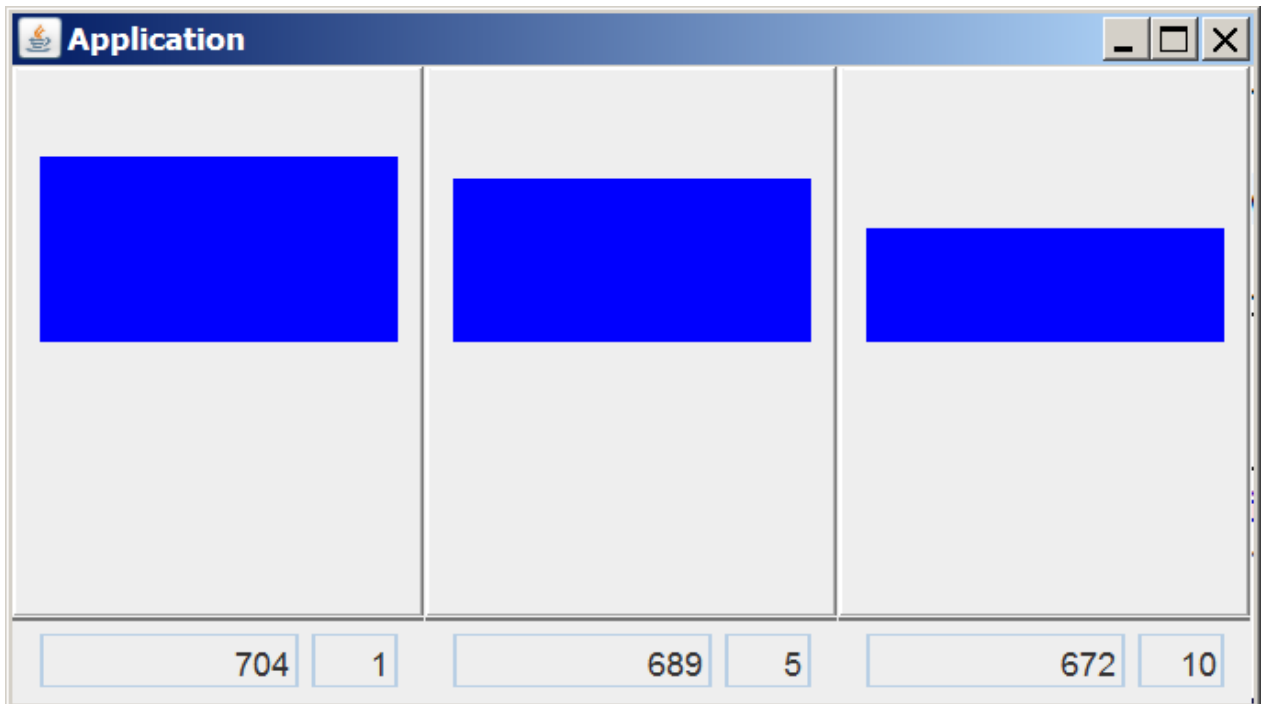
## jj.gui.BookFrame

```
package jj.gui;  
// ...  
import javax.swing.DefaultListModel;  
  
import jj.database.BookDatabase;  
import jj.domain.Book;  
  
public class BookFrame extends JFrame {  
  
    private final BookDatabase bookDatabase;  
  
    private final DefaultListModel<Book> bookListModel =  
        new DefaultListModel<>();  
  
    private final JLabel labelTitle = new JLabel("Title");  
    private final JTextField textFieldTitle = new JTextField(10);  
    private final JButton buttonFind = new JButton("Find");  
    private final JList<Book> listBook = new  
JList<>(this.bookListModel);  
  
    public BookFrame(BookDatabase bookDatabase) {  
        // Aufbau der GUI...  
        this.buttonFind.addActionListener(e -> onFind());  
    }  
  
    private void onFind() {  
        this.bookListModel.clear();  
        final BookSubscriber subscriber =  
            new BookSubscriber(this.bookListModel);  
        final String title = this.textFieldTitle.getText();  
        this.bookDatabase.find(subscriber, title);  
    }  
}
```

In `onFind` wird ein `BookSubscriber` erzeugt, dem das `DefaultListModel` übergeben wird. Auf die `BookDatabase` wird dann deren `find`-Methode aufgerufen, welcher der `Subscriber` übergeben wird. Diese `find`-Methode kehrt sofort zurück.

Es handelt sich hier um eine sehr einfache Anwendung (die auch mit "traditionellen" Mitteln hätte implementiert werden können) – eine Anwendung, die aber immerhin zeigt, dass die Datenquelle mit der Datensenke sehr lose gekoppelt werden kann – nur über das Interface `Flow.Subscriber`.

## 7.8 Beispiel: Swing-Diagramme



Der Hauptthread der Anwendung hat eine "Endlosschleife", in der fortwährend Sinus-Werte für aufeinanderfolgende x-Werte (Abstand: 0.01) berechnet werden (die berechneten Sinus-Werte sollen irgendwelche Meßdaten repräsentieren.). Jeder dieser Werte wird jeweils in einer `Message` verpackt, die zusätzlich eine `sequenceNumber` enthält, die fortwährend hochgezählt wird. Diese `Messages` werden von einem `Publisher` versandt.

Diese Werte werden in drei `DiagramPanel`s angezeigt. Die jeweilige Anzeige ist mit einer Verarbeitung des vom Hauptthread gelieferten Wertes verknüpft. Die Verarbeitung beim linken Panel ist sehr flott, diejenige beim mittleren Panel mittel, und diejenige beim rechten Panel sehr langsam. Dabei soll der (sehr schnelle) `Publisher` aber nicht ausgebremst werden.

Ein "schneller" `DiagramPanel` wird die Werte, die der `Publisher` liefert, mehr oder weniger lückenlos anzeigen; ein "langsamer" `DiagramPanel` wird dagegen nicht alle Werte anzeigen sollen (denn sonst würde der `Publisher` ausgebremst werden). Die Anzeige in einem solchen langsamen `DiagramPanel` wird also eher "ruckeln" (und der Anzeige eines schnellen `DiagramPanel`s etwas hinterher hinken).

Hier zunächst die kleine Klasse `Message`, in deren Objekten die "Meßdaten" verpackt werden:

```
package jj.core;

public class Message {

    public final long sequenceNumber;
    public final double value;

    public Message(long sequenceNumber, double value) {
        this.sequenceNumber = sequenceNumber;
        this.value = value;
    }

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

Die Klasse `DialogFrame` muss hier nicht im einzelnen vorgestellt werden. Die Köpfe der öffentlichen Methoden dieser Klasse sollen reichen:

```
package jj.gui;
// ...
public class DiagramFrame extends JFrame {

    public DiagramFrame(String title, int x, int y, Color color,
        int panelCount) { ... }

    public boolean isClosed() { ... }

    public void processMessage(int index, Message message)
    { ... }
}
```

Die Klasse `Application` erzeugt den `Publisher` und die Oberfläche (einen `DiagramFrame`) und implementiert die Endlosschleife (in der Methode `generateMessages`), in der die "Meßdaten" von dem `Publisher` versandt werden:

```
package jj.appl;
// ...
import java.awt.Color;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.SubmissionPublisher;
import jj.core.Message;
import jj.flow.DelegatingSubscriber;
import jj.gui.DiagramFrame;

public class Application {
```

```
public static void main(String[] args) {
    Log.enabled = false;
    new Application().run();
}

private final SubmissionPublisher<Message> publisher =
    new SubmissionPublisher<>(ForkJoinPool.commonPool(), 1);

private int[] delays = new int[] { 10, 50, 100 };

private final DiagramFrame frame = new DiagramFrame(
    this.getClass().getSimpleName(), 100, 100, Color.blue,
    delays.length);

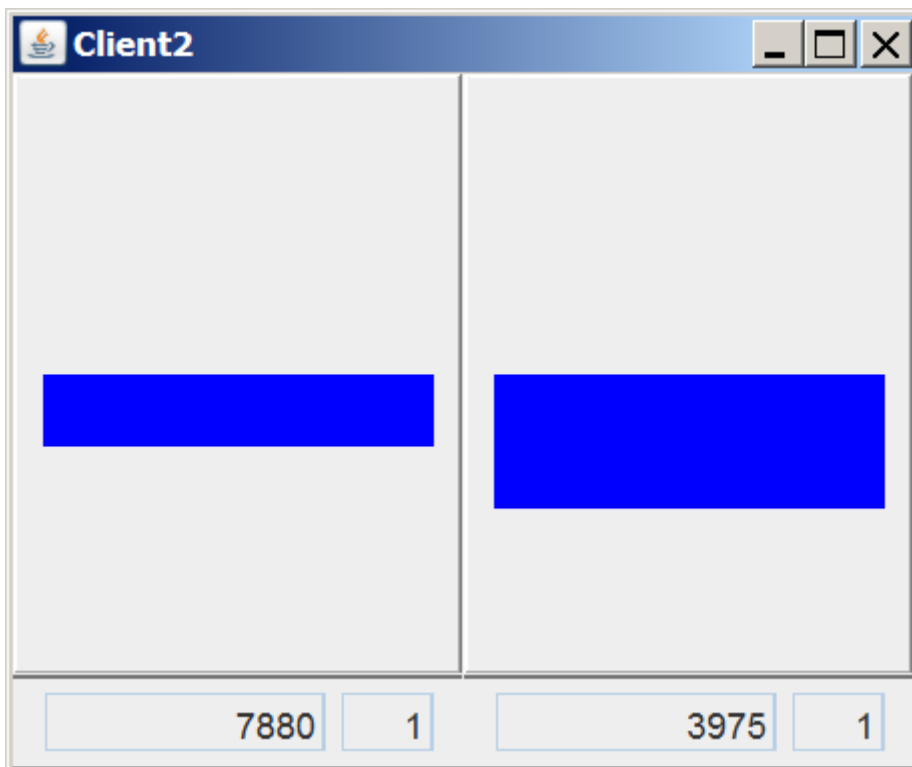
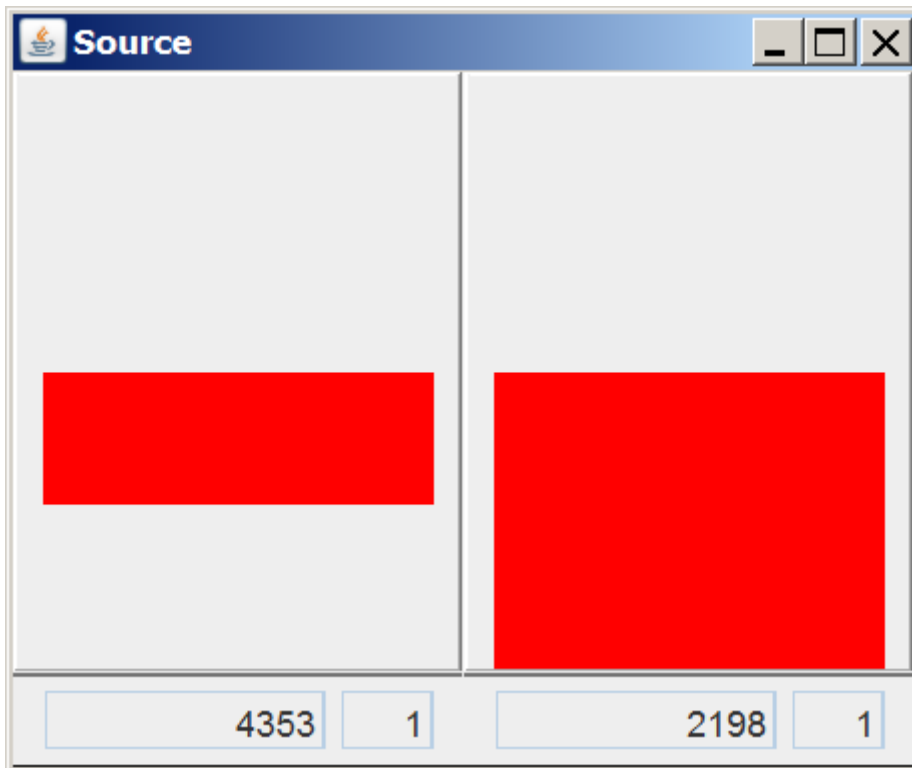
private void run() {
    for (int i = 0; i < this.delays.length; i++)
        this.subscribe(i, this.delays[i]);
    new Thread(() -> generateMessages(10)).start();
}

private void subscribe(int index, int delay) {
    DelegatingSubscriber<Message> subscriber =
        new DelegatingSubscriber<>();
    subscriber.onSubscribeHandler(s -> s.request(1));
    subscriber.onNextHandler((s, msg) -> {
        this.frame.processMessage(index, msg);
        xrun(() -> Thread.sleep(delay));
        s.request(1);
    });
    publisher.subscribe(subscriber);
}

private void generateMessages(int delay) {
    long sequenceNumber = 0;
    for (double x = 0; !this.frame.isClosed(); x += 0.01) {
        xrun(() -> Thread.sleep(delay));
        sequenceNumber++;
        final Message message = new Message(sequenceNumber,
Math.sin(x));
        this.publisher.offer(message, null);
    }
    this.publisher.close();
}
}
```

Würde in der `generateMessage`-Methode der `offer`-Aufruf durch den Aufruf von `submit` ersetzt, würde der `Publisher` stark ausgebremst. Die langsamste Anzeige würde allen anderen Komponenten (insb. dem `Publischer`) ihre eigene Geschwindigkeit aufzwingen.

## 7.9 Beispiel: Swing-Diagramme Client-Server





Die Anwendung erweitert das letzte Beispiel zu eine Client-Server-Anwendung.

Die Server-Anwendung wird über `jj.source.Source` gestartet; mittels `jj.appl.Client1` und `jj.appl.Client2` können zwei Clients gestartet werden.

Der Server misst permanent irgendwelche Daten (Temperatur und Luftfeuchtigkeit). Er enthält eine GUI, welche die jeweils aktuellen Daten anzeigt (eine "schnelle" GUI – sie ist genauso schnell wie die Datenquelle).

Mit diesem Server können sich nun Clients verbinden (via Sockets), die ihrerseits ebenfalls eine GUI zur Anzeige der Temperatur und der Luftfeuchtigkeit enthalten. Diese GUIs sind allerdings "billige" und als "langsame" GUIs.

Die Wetterstation (also der Server) soll nun natürlich nicht über die langsamen externen Clients (Nebenstellen der Wetterstation) ausgebremst werden. Also wir der im Server laufende `SubmissionPublisher` die Daten jeweils nur anbieten (`offer`), aber nicht auf deren Verarbeitung warten (`submit`).

Das heißt natürlich auch, dass die entfernten Clients längst nicht alle Daten bekommen – sondern etwa vielleicht nur jedes fünfte Daten (die anderen Daten werden "verschluckt"). Das reicht aber für eine "langsame" Anzeige auch völlig aus.

Das Programmsystem ist relativ komplex. Es soll hier auch nicht tiefergehend vorgestellt werden. Der interessierte Leser / die interessierte Leserin möge einfach den Quellcode studieren...

## 8 Tools

Im Folgenden werden einige Werkzeuge vorgestellt, die in Java 9 hinzugekommen sind.

- `jlink` ist ein Werkzeug, mittels dessen kompakte Laufzeit-Images erzeugt werden können (ein solches Image enthält nur die absolut notwendigen jar-Dateien).
- `jdeps` ermöglicht die Ermittlung der Abhängigkeiten zwischen mehreren Modulen.
- Mittels der `jshell` können Java-Ausdrücke und –Anweisungen interaktiv ausgeführt werden.

## 8.1 jlink

### tools-jlink-mod

```
build.xml
```

```
mod.jar  
appl.jar  
image
```

### tools-jlink-mod

```
module jj.mod {  
    //exports jj.mod.pri;  
    exports jj.mod.pub;  
}  
jj.mod.pri  
    Bar  
jj.mod.pub  
    Foo
```

### tools-jlink-appl

```
module jj.appl {  
    requires jj.mod;  
}  
jj.appl  
    Application (main)
```

Die build.xml:

```
<property name="appl" value="${basedir}-appl" />  
<property name="mod" value="${basedir}-mod" />  
<property name="image" value="${basedir}/image"/>
```

```
<target name="build-mod">  
    <echo message="building ${mod}" />  
    <exec executable="${javac}">  
        <arg value="-d" />  
        <arg value="${mod}/build" />  
        <arg value="${mod}/src/module-info.java" />  
        <arg value="${mod}/src/jj/mod/pri/*.java" />  
        <arg value="${mod}/src/jj/mod/pub/*.java" />  
    </exec>  
    <exec executable="${jar}">
```

```
        <arg value="--create" />
        <arg value="--file=${basedir}/mod.jar" />
        <arg value="--module-version=1.0" />
        <arg value="-C" />
        <arg value="${mod}/build" />
        <arg value="." />
    </exec>
</target>
```

```
<target name="build-appl">
    <echo message="building ${appl}" />
    <exec executable="${javac}">
        <arg value="--module-path" />
        <arg value="${mod}/build" />
        <arg value="-d" />
        <arg value="${appl}/build" />
        <arg value="${appl}/src/module-info.java" />
        <arg value="${appl}/src/jj/appl/*.java" />
    </exec>
    <exec executable="${jar}">
        <arg value="--create" />
        <arg value="--file=${basedir}/appl.jar" />
        <arg value="--module-version=1.0" />
        <arg value="-C" />
        <arg value="${appl}/build" />
        <arg value="." />
    </exec>
</target>
```

```
<target name="jlink">
    <echo message="jlink ${image}" />
    <exec executable="${jlink}">
        <arg value="--output" />
        <arg value="${image}" />
        <arg value="--module-path" />
        <arg value="${jdk}/jmods;${basedir}" />
        <arg value="--add-modules" />
        <arg value="java.base,jj.appl" />
        <arg value="--compress=2"/>
        <arg value="--launcher"/>
        <arg value="jj=jj.appl/jj.appl.Application"/>
    </exec>
</target>
```

```
<target name="listModules">
    <exec executable="${image}/bin/java">
        <arg value="--list-modules" />
        <arg value="--module-path" />
        <arg value="${image}" />
    </exec>
</target>
```

```
listModules:
    java.base@9
    jj.appl@1.0
    jj.mod@1.0
```

```
<target name="run"
    depends="clean, build-mod, build-appl, jlink,
listModules">
    <exec executable="${image}/bin/java">
        <arg value="-m" />
        <arg value="jj.appl/jj.appl.Application" />
    </exec>
</target>
```

## 8.2 jdeps

### tools-jlink-mod

```
build.xml
```

### tools-jdeps-mod

```
module jj.mod {  
    //exports jj.mod.pri;  
    exports jj.mod.pub;  
}  
jj.mod.pri  
    Bar  
jj.mod.pub  
    Foo
```

### tools-jdeps-appl

```
module jj.appl {  
    requires jj.mod;  
}  
jj.appl  
    Application (main)
```

Die build.xml:

```
// ...  
  
<target name="jdeps">  
    <echo message="jdeps ${appl}" />  
    <echo message="===== mod =====" />  
    <exec executable="${jdeps}">  
        <arg value="${mod}/build" />  
    </exec>  
    <echo message="===== appl =====" />  
    <exec executable="${jdeps}">  
        <!--  
        <arg value="-R" />  
        -->  
        <arg value="--module-path" />  
        <arg value="${mod}/build" />  
        <arg value="${appl}/build" />  
    </exec>
```

```
</target>
```

```
===== mod =====
```

```
jj.mod
```

```
[file:///...x0802-tools-jdeps-mod/build/]
```

```
requires mandated java.base (@9)
```

```
jj.mod -> java.base
```

```
jj.mod.pri          -> java.io          java.base
```

```
jj.mod.pri          -> java.lang        java.base
```

```
jj.mod.pub          -> java.io          java.base
```

```
jj.mod.pub          -> java.lang        java.base
```

```
jj.mod.pub          -> jj.mod.pri       jj.mod
```

```
===== appl =====
```

```
jj.appl
```

```
[file:///...x0802-tools-jdeps-appl/build/]
```

```
requires mandated java.base (@9)
```

```
requires jj.mod
```

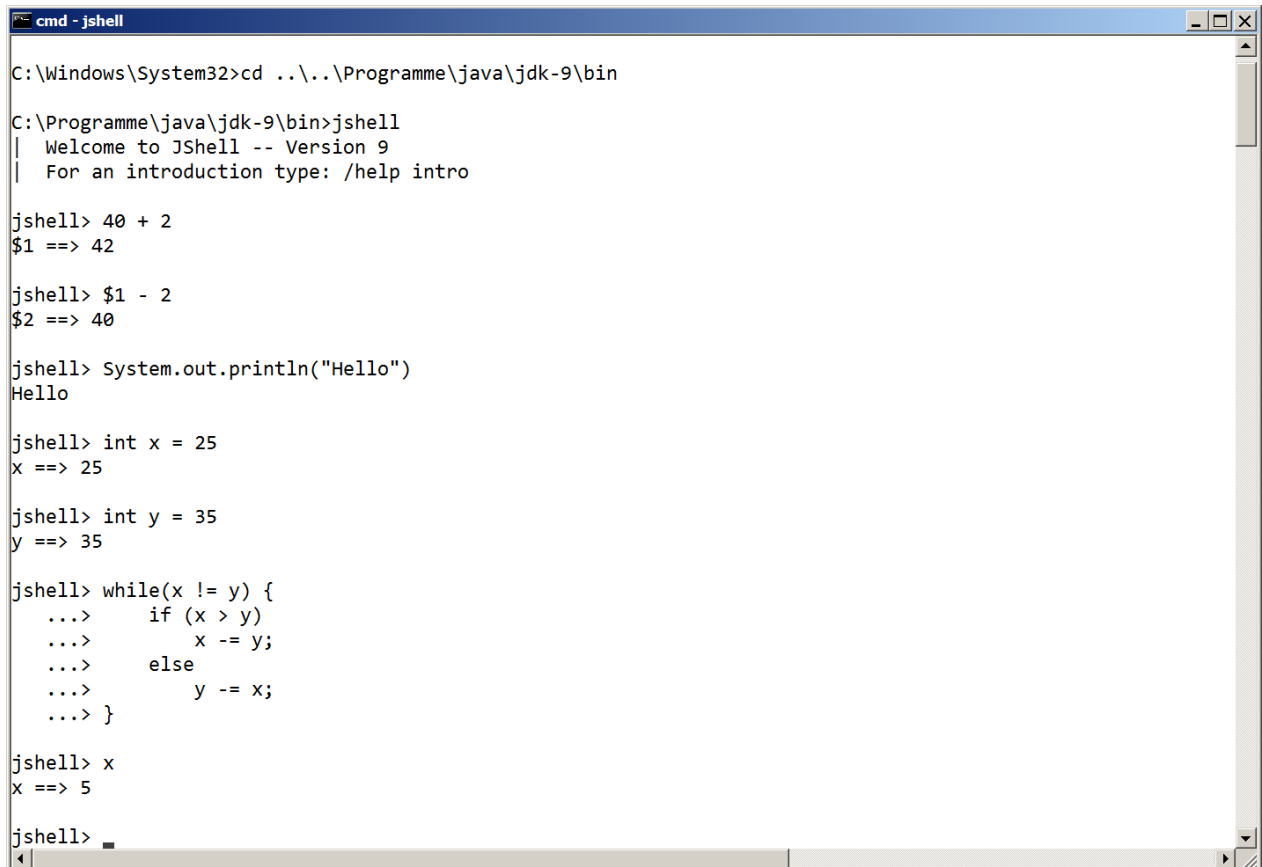
```
jj.appl -> java.base
```

```
jj.appl -> jj.mod
```

```
jj.appl          -> java.lang        java.base
```

```
jj.appl          -> jj.mod.pub       jj.mod
```

## 8.3 jshell



```
cmd - jshell

C:\Windows\System32>cd ../../Programme/java/jdk-9\bin

C:\Programme\java\jdk-9\bin>jshell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro

jshell> 40 + 2
$1 ==> 42

jshell> $1 - 2
$2 ==> 40

jshell> System.out.println("Hello")
Hello

jshell> int x = 25
x ==> 25

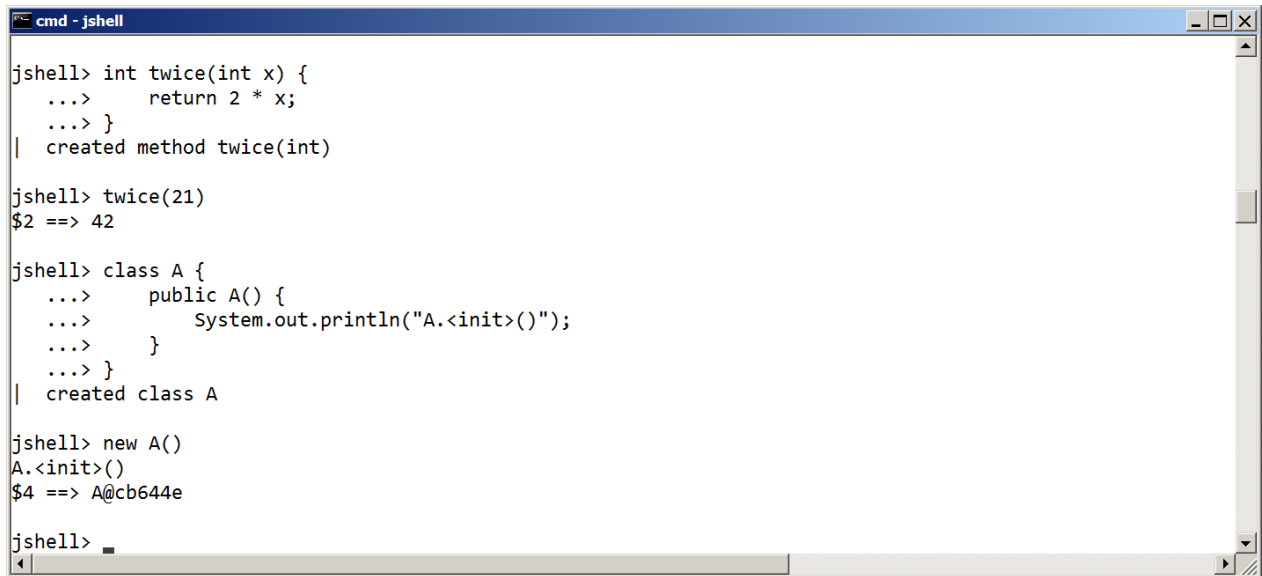
jshell> int y = 35
y ==> 35

jshell> while(x != y) {
...>     if (x > y)
...>         x -= y;
...>     else
...>         y -= x;
...> }

jshell> x
x ==> 5

jshell> _
```





```
cmd - jshell

jshell> int twice(int x) {
...>     return 2 * x;
...> }
| created method twice(int)

jshell> twice(21)
$2 ==> 42

jshell> class A {
...>     public A() {
...>         System.out.println("A.<init>()");
...>     }
...> }
| created class A

jshell> new A()
A.<init>()
$4 ==> A@cb644e

jshell> 
```

## 8.4 Weitere Werkzeuge

Unified Logging: `xlog`-Option für `java`

Erstellen von `jmod`-Dateien: `jmod`

Verbinden mit Prozess, Post Mortem Debugger: `jhsdb`

## 9 Java 10

Dieses Kapitel stellt wesentliche Erweiterungen von Java 10 vor.

- Im ersten Abschnitt wird das neue Feature "Local Variable Type Inference" vorgestellt. (Vor Java 10 war Java die einzige Main-Stream-Sprache, der dieses neue Feature fehlte...)
- Der zweite Abschnitt stellt Neuerungen in den `Collection`- und den `Collectors`-Klassen vor
- Im dritten Abschnitt geht's um eine kleine Erweiterung der `Optional`-Klasse.
- Im letzten Abschnitt werden die Erweiterungen von `Runtime.Version` vorgestellt.

Java 10 bietet darüber hinaus insbesondere Erweiterungen des Garbage-Collectors und Erweiterungen im Bereich Security. Insbesondere die letzten Erweiterungen sind sehr speziell – und werden eben deshalb auch nicht weiter vorgestellt.

## 9.1 Local Variable 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.

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 "Typ-lose" 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".

Folgende Definition ist also unzulässig:

```
//      illegal:  
class var  
}
```

Wir können mittels `var` auch finale Variablen definieren:

```
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.)

Wir definieren und initialisieren eine `List`-Variable:

```
var list = List.of(10, 20, 30);
```

Und zeigen, dass diese Variable vom Typ `List<Integer>` ist:

```
List<Integer> l = list;
```

Folgende Zuweisung ist illegal:

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

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

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

Die Variable `i` ist implizit vom Typ `int`.

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

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:

```
final int foo;
if ("1".equals("1"))
    foo = 42;
else
    foo = 77;
```

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

```
// illegal
final var bar;
if ("1".equals("1"))
    bar = 42;
else
    bar = 77;
```

Angenommen, wir initialisieren eine Variablen mit einem Lambda-Ausdruck:

```
Function<String, Integer> foo = s -> s.length();
var result = foo.apply("Hello");
System.out.println(result);
```

Da der Typ eines Lambda-Ausdruck aus diesem selbst nicht hervorgeht, sondern nur aus dem Ziel einer Zuweisung, ist folgende Zeile illegal:

```
var bar = s -> s.length(); // illegal (-> Target Typing)
```

Der Compiler kann den Typ des Lambda-Ausdrucks und damit auch den Typ von `bar` nicht ermitteln.

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:

```
var function = new Function<String, Integer>() {
    @Override
    public Integer apply(String s) {
        return s.length();
    }
};
```

`listener` hat nun exakt den Typ der anonymen Klasse (etwa: `...$0`).

An dieselbe Variable kann also kein(!) zweiter `ActionListener` zugewiesen werden:

```
//      illegal:
      function = new Function<String, Integer>() {
          @Override
          public Integer apply(String s) {
              return s.length();
          }
      };
```

Denn der Ausdruck auf der rechten Seite hat nun einen anderen(!) Typ (etwa ...\$1) – und ist daher nicht kompatibel zum Typ von `listener`.

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:

```
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:

```
class Foo {
    // var i = 42; // illegal
}
```

Und ebenso wenig können natürlich auch Methoden-Parameter oder Return-Typen mit `var` definiert werden...

## 9.2 Collections und Collectors

### List.copyOf

Die Interfaces `List`, `Set` und `Map` enthalten nun jeweils eine statische `copyOf`-Methode. Diese Methoden liefern aufgrund einer ihnen jeweils übergebenen `Collection` eine immutable Kopie dieser `Collection` zurück.

Wir erzeugen eine `ArrayList`:

```
List<String> list = new ArrayList<>();  
list.add("red");  
list.add("green");  
list.add("blue");
```

Und erzeugen dann mittels `List.copyOf` eine unveränderliche Kopie dieser Liste:

```
List<String> copy = List.copyOf(list);  
System.out.println(copy.size());  
for (String s : copy)  
    System.out.println(s);  
try {  
    copy.add("yellow");  
}  
catch (UnsupportedOperationException e) {  
    System.out.println("expected: " + e);  
}
```

Auf die neue Liste können alle "Lese"-Operationen aufgerufen werden. Und über die neue Liste kann natürlich iteriert werden.

Der Aufruf aber einer Methode, welche den Zustand der Liste ändern würde, wird zur Laufzeit mit einer `UnsupportedOperationException` quittiert.

Auch die Interfaces `Set` und `Map` enthalten nun eine entsprechende `copyOf`-Methode.

### Collectors.toUnmodifiableList

Die `Collectors`-Klasse ist um folgende Methoden erweitert worden:

```
Collectors.toUnmodifiableList  
Collectors.toUnmodifiableSet  
Collectors.toUnmodifiableMap
```



`Collectors.toConcurrentMap`

Die `toList`-Methode der `Collectors`-Klasse liefert eine modifiable List zurück:

```
List<String> list = Stream.of("red", "green", "blue")
    .collect(Collectors.toList());
list.add("yellow");
list.forEach(System.out::println);
```

Die Ausgaben:

```
red
green
blue
yellow
```

Die neue `toUnmodifiableList` liefert eine unmodifiable List zurück:

```
List<String> list = Stream.of("red", "green", "blue")
    .collect(Collectors.toUnmodifiableList());
try {
    list.add("yellow");
}
catch (UnsupportedOperationException e) {
    System.out.println("expected: " + e);
}
list.forEach(System.out::println);
```

Die Ausgaben:

```
expected: java.lang.UnsupportedOperationException
red
green
blue
```

## **`Collectors.toUnmodifiableMap`**

Hier eine Anwendung der alten `Collectors.toMap`-Methode:

```
Map<String, Integer> map = Stream.of("red", "green",
"blue")
    .collect(Collectors.toMap(s -> s, s -> s.length()));
map.put("yellow", 5);
System.out.println(map);
```

Die Ausgaben:

```
{red=3, green=5, blue=4, yellow=5}
```

Und hier eine kleine Anwendung der neuen `toUnmodifiableMap`:

```
Map<String, Integer> map = Stream.of("red", "green",  
"blue")  
    .collect(Collectors.toUnmodifiableMap(s -> s, s ->  
s.length()));  
try {  
    map.put("yellow", 5);  
} catch (UnsupportedOperationException e) {  
    System.out.println("expected: " + e);  
}  
System.out.println(map);
```

Die Ausgaben:

```
expected: java.lang.UnsupportedOperationException  
{blue=4, green=5, red=3}
```

## 9.3 Optional

Eine Anwendung der ("alten") `Optional.get`-Methode:

```
try {
    Optional<String> string1 = Optional.of("Hello");
    if (string1.isPresent()) {
        String s = string1.get();
        System.out.println(s);
    }
    Optional<String> string2 = Optional.empty();
    String s = string2.get(); // throws an exception
    System.out.println(s);
}
catch (Exception e) {
    System.out.println(e);
}
```

Die Ausgaben:

```
Hello
java.util.NoSuchElementException: No value present
```

Da Entwickler nicht unbedingt erwarten, dass eine Methode namens `get` eine Exception liefern kann, gibt's nun zusätzlich die äquivalente Methode `orElseThrow`:

```
try {
    Optional<String> string1 = Optional.of("Hello");
    if (string1.isPresent()) {
        String s = string1.orElseThrow();
        System.out.println(s);
    }
    Optional<String> string2 = Optional.empty();
    String s = string2.orElseThrow(); // throws an
exception
    System.out.println(s);
}
catch (Exception e) {
    System.out.println(e);
}
```

Die Ausgaben sind dieselben wie im ersten Beispiel.

Die neue `orElseThrow`-Methode passt dann auch zu derjenigen `orElseThrow`-Methode, die bereits in Java 9 existierte:

```
try {  
    Optional<String> string = Optional.empty();  
    String s = string.orElseThrow(  
        () -> new RuntimeException("zzz"));  
    System.out.println(s);  
}  
catch (Exception e) {  
    System.out.println(e);  
}
```

## 9.4 Runtime.Version

Die Klasse `Runtime.Version` war bereits in Java 9 enthalten. Sie wurde in Java 10 erweitert. Hier eine Anwendung:

```
Runtime.Version v = Runtime.version();
System.out.println(v);
List<Integer> vlist = v.version();
System.out.println(vlist);
System.out.println("feature = " + v.feature());
System.out.println("interim = " + v.interim());
System.out.println("update = " + v.update());
System.out.println("patch = " + v.patch());
System.out.println(v.build());
```

Die Ausgaben:

```
10.0.1+10
[10, 0, 1]
feature = 10
interim = 0
update = 1
patch = 0
Optional[10]
```

Die Methoden `major()` und `minor()` sind nun deprecated.

## 10 Java 11

Dieses Kapitel stellt wesentliche Erweiterungen von Java 11 vor – beschränkt sich aber auf diejenigen Erweiterungen, die für den Entwickler unmittelbar interessant sind.

- Im ersten Abschnitt wird gezeigt, wie mittels des `java`-Kommandos nun auch einfache Quellcode-Dateien ausgeführt werden können.
- Der zweite Abschnitt zeigt, wie auch Lambda-Parameter nun mittels `var` definiert werden können.
- Im dritten Abschnitt geht's um Erweiterung der String-Klasse.
- Im vierten Abschnitt geht's um Erweiterungen der Files-Klasse – um Methoden, die Strings auf einfache Weise in eine Datei schreiben resp. solche Strings aus Dateien auslesen.
- Im fünften Abschnitt wird eine kleine Erweiterung der `Optional`-Klasse vorgestellt: `isEmpty`.
- Im sechsten Abschnitt wird die neue statische `not`-Methode des `Predicate`-Interfaces vorgestellt.
- Im siebten Abschnitt geht's um das `HttpClient`-API (welches bereits in Java 9 eingeführt wurde – bislang aber experimentellen Charakter hatte).
- Im achten Abschnitt schließlich zeigen wir, wie mittels des `HttpClient`-APIs `WebSockets` implementiert werden können.

## 10.1 Ausführen von Single-File Sourcecode

Mittel des `java`-Kommandos kann nun auch Sourcecode ausgeführt werden, der in einer einzigen Datei enthalten ist:

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

Angenommen, man befindet sich in einem Verzeichnis namens `src`, das ein Verzeichnis `appl` enthält, welches die Datei `Application.java` enthält. Dann kann der Quellcode dieser Datei wie folgt ausgeführt werden:

```
.../src > java appl/Application.java
```

## 10.2 Benutzung von var in Lambda-Parametern

Auch Lambda-Parameter können nun mittels var definiert werden:

```
package appl;

import java.util.function.BiConsumer;

public class Application {

    public static void main(String[] args) {

        BiConsumer<String, Integer> c1 =
            (@Nullable String s, Integer i) -> { };

        BiConsumer<String, Integer> c2 =
            (@Nullable var s, var i) -> { };

    }
}
```

Die `c1`-Variante funktionierte schon immer, die `c2`-Variante erst mit Java-11.

Die folgende Zeile aber ist illegal (Parameter-Annotationen können also nur dann verwendet werden, wenn der Typ des Parameters definiert ist – und sei es als `var`):

```
BiConsumer<String, Integer> c3 = (@Nullable s, i) -> { };
```



## 10.3 Erweiterungen der String-Klasse

Die Klasse `String` besitzt eine neue Instanz-Methode `repeat`, die einen `String` erzeugt, der `n`-mal denjenigen `String` enthält, auf den sie aufgerufen wird:

```
static void demoRepeat() {  
    String s1 = "Hello".repeat(5);  
    System.out.println(s1);  
    String s2 = "-".repeat(50);  
    System.out.println(s2);  
}
```

Die Ausgaben:

```
HelloHelloHelloHelloHello  
-----
```

Neben der alten `trim`-Methode gibt's nun die Methoden `strip`, `stripLeading` und `stripTrailing` (wobei die `strip`-Methode allgemeingültiger definiert ist als die alte `trim`-Methode):

```
static void demoStrip() {  
    System.out.println("'" + " Hello " + ".trim() + "'");  
    System.out.println("'" + " Hello " + ".strip() + "'");  
    System.out.println("'" + " Hello " + ".stripLeading() +  
    "'"");  
    System.out.println("'" + " Hello " + ".stripTrailing() +  
    "'"");  
}
```

Die Ausgaben:

```
'Hello'  
'Hello'  
'Hello '  
' Hello'
```

Die Instanzmethode `lines` liefert einen `Stream<String>` derjenigen Zeilen zurück, aus denen der `String` besteht, auf den sie aufgerufen wird:

```
static void demoLines() {  
    String s = "red\ngreen\nblue";  
    Stream<String> stream = s.lines();  
    stream.forEach(System.out::println);  
}
```

Die Ausgaben:

```
red
green
blue
```

`isBlank` liefert `true`, wenn der String, auf den sie aufgerufen wird, leer ist oder nur aus weißen Zeichen besteht:

```
static void demoIsBlank() {
    System.out.println("").isBlank());
    System.out.println(" ".isBlank());
    System.out.println("\n \t ".isBlank());
    System.out.println(" n t ".isBlank());
}
```

Die Ausgaben:

```
true
true
true
false
```

An `Character.toString` wird ein `int` übergeben. Sie liefert einen String zurück, der das Zeichen mit diesem Wert enthält:

```
static void demoCharacterToString() {
    String s = Character.toString(65);
    System.out.println(s);
}
```

Die Ausgabe liefert den String "A" zurück.

## 10.4 Erweiterungen der Files-Klasse

Die Klassen der Files-Klasse des Packages `java.nio.file` sind erweitert worden. Die folgenden Beispiele benutzen drei Klassen:

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

Der statischen Files-Methode `writeString` wird ein `Path` und ein `String` übergeben. Sie erzeugt im Folgenden stets eine neue Datei, die den jeweils an `writeString` übergebenen `String` enthält. Die `readString`-Methode liest die gesamte Datei aus und liefert den Inhalt in Form eines einzigen `String` zurück:

```
static void demo1() throws IOException {
    var path = Path.of("abc.txt");

    Files.writeString(path, "Hello\n");
    Files.writeString(path, "World\n");
    Files.writeString(path, "Good\nBye\n");

    final String line1 = Files.readString(path);
    final String line2 = Files.readString(path);
    final String line3 = Files.readString(path);

    System.out.println(line1);
    System.out.println(line2);
    System.out.println(line3);
}
```

Die Ausgaben:

Good  
Bye

Good  
Bye

Good  
Bye

(Die `readString`-Methode liest natürlich dreimal dasselbe: das, was beim letzten Aufruf von `writeString` geschrieben wurde...)

An `writeString` kann eine `StandardOpenOption` übergeben werden. Bei Angabe der `APPEND`-Option erzeugt `writeString` keine neue Datei, sondern hängt den an die Methode übergebenen String ans Ende einer bereits bestehenden Datei an:

```
,
    static void demo2() throws IOException {
        var path = Path.of("abc.txt");

        Files.writeString(path, "Hello\n");
        Files.writeString(path, "World\n",
StandardOpenOption.APPEND);

        String line = Files.readString(path);
        System.out.println(line);
    }
}
```

Die Ausgaben:

```
Hello
World
```

Natürlich kann das Resultat von `readString` auch als Stream weiterverarbeitet werden:

```
    static void demo3() throws IOException {
        var path = Path.of("abc.txt");

        Files.writeString(path, "Hello\n");
        Files.writeString(path, "World\n",
StandardOpenOption.APPEND);

Files.readString(path).lines().forEach(System.out::println);
    }
}
```

Die Ausgaben:

```
Hello
World
```

## 10.5 Erweiterungen der Optional-Klasse

Neben der `isPresent`-Methode gibt's nun auch die inverse Methode - `isEmpty`:

```
package appl;

import java.io.IOException;
import java.util.Optional;

public class Application {

    public static void main(String[] args) throws IOException {
        Optional<String> s = Optional.empty();
        if (! s.isPresent())
            System.out.println("empty");
        if (s.isEmpty())
            System.out.println("empty");
    }
}
```

Die Ausgaben:

```
empty
empty
```

## 10.6 Erweiterungen des Predicate-Interfaces

Angenommen, wir wollen ein `Predicate` erzeugen, welches ein gegebenes `Predicate` negiert. Bislang mussten wir die Instanz-Methode `negate()` verwenden (wozu i.d.R. das `Predicate`, auf das diese Methode aufgerufen wird, einer Variablen zugewiesen werden musste):

```
static void demoOld() {  
    final Predicate<String> isEmpty = s -> s.isEmpty();  
    final Predicate<String> isEmpty2 = isEmpty.negate();  
  
    System.out.println(isEmpty.test("Hello"));  
    System.out.println(isEmpty2.test("Hello"));  
    System.out.println(isEmpty.test(""));  
    System.out.println(isEmpty2.test(""));  
}
```

Die Ausgaben:

```
false  
true  
true  
false
```

Das `Predicate`-Interface enthält nun zusätzlich eine statische Methode `not` (die am besten mittels eines statischen Imports bekanntgemacht wird):

```
import static java.util.function.Predicate.not;
```

Hier einige Beispiele zur Verwendung dieser `not`-Methode (man beachte, dass an `not` wiederum einfach ein Lambda-Ausdruck übergeben werden kann – dieser muss nicht erst an eine Variablen gebunden werden):

```
static void demoNew() {  
    final Predicate<String> isEmpty = s -> s.isEmpty();  
    final Predicate<String> isEmpty2 = not(isEmpty);  
    final Predicate<String> isEmpty3 = not(s ->  
s.isEmpty());  
  
    System.out.println(isEmpty.test("Hello"));  
    System.out.println(isEmpty2.test("Hello"));  
    System.out.println(isEmpty3.test(""));  
    System.out.println(isEmpty2.test(""));  
}
```

Die Ausgaben sind dieselben wie im ersten Beispiel.

## 10.7 Die Klasse HttpClient

Statt HTTP-Clients mittels der alten Klasse `URLConnection` zu implementieren, sollte nun das HTTP-Client-API genutzt werden. Dieses wurde bereits als experimentelles API in Java 9 eingeführt – ist nun aber standardisiert worden. Dieses API unterstützt u.a. HTTP/2, WebSockets und HTTP/2 Server Push. Es ermöglicht synchrone und asynchrone Aufrufe.

Das neue API ist recht komplex – daher kann im Folgenden nur ein kleiner Einblick vermittelt werden. Es macht insbesondere rege Gebrauch von `CompletableFuture`.

Wir zeigen drei kleine Servlet-basierte Anwendungen. Die erste demonstriert den synchronen Aufruf einer `GET`-Methode, die zweite den asynchronen Aufruf einer solchen Methode, und die dritte den Aufruf einer `POST`-Methode.

Um diese Anwendungen auszuführen, muss Tomcat gestartet werden:

```
dependencies/apache-tomcat-9.0.17/bin/startup.bat
```

Das Projekt besteht aus zwei Teil-Projekten:

```
java-11-HttpClient  
java-11-HttpClient-Server
```

Das erste Projekt enthält den Client (eine einfache `main`-Anwendung). Das Server-Projekt enthält eine Servlet-Klasse. Und es enthält eine `build.xml`, mittels derer die erforderliche `war`-Datei erzeugt und in das `webapps`-Verzeichnis von Tomcat kopiert werden kann.

Hier zunächst die Servlet-Klasse. Sie implementiert sowohl `doGet` als auch `doPost`. Es werden jeweils zwei Parameter erwartet: `x` und `y`. Als Antwort wird die Summe dieser beiden Werte zurückgeschickt. Die `GET`-Anwendung erwartet die Parameter natürlich in Form eines Query-Strings, die `POST`-Anwendung erwartet die Parameter jeweils in einer eigenen Zeilen des Request-Bodies:

```
package servlets;  
// ...  
@WebServlet("/*")  
public class MathServlet extends HttpServlet {  
  
    @Override
```

```
public void doGet(
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    System.out.println("MathServlet.doGet()");
    try {
        int x = Integer.parseInt(request.getParameter("x"));
        int y = Integer.parseInt(request.getParameter("y"));
        int sum = x + y;
        response.getWriter().write(String.valueOf(sum));
        Thread.sleep(2000);
    }
    catch (Exception e) {
        throw new ServletException(e);
    }
}

@Override
public void doPost(
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    System.out.println("MathServlet.doPost()");
    try {
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(request.getInputStream()));
        int x = Integer.parseInt(reader.readLine());
        int y = Integer.parseInt(reader.readLine());
        int sum = x + y;
        response.getWriter().write(String.valueOf(sum));
    }
    catch (Exception e) {
        throw new ServletException(e);
    }
}
}
```

Hier der erste Client, der einen synchronen Aufruf der `GET`-Methode implementiert:

```
package client;

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpClient.Version;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
```



```
import java.net.http.HttpResponse.BodyHandlers;

public class Client1 {

    public static void main(String[] args) throws Exception {

        final HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create("http://localhost:8080/math?
x=40&y=2"))
            .GET()
            .build();

        final HttpClient client = HttpClient.newBuilder()
            .version(Version.HTTP_2)
            .build();

        final HttpResponse<String> response =
            client.send(request, BodyHandlers.ofString());

        System.out.println("status = " + response.statusCode());
        System.out.println("body    = " + response.body());
    }
}
```

Ein `HttpRequest` repräsentiert den Request. Er wird mittels des Builder-Patterns erzeugt (und ist immutable). An `uri` wird die URL übergeben, die u.a. den Query-String enthält (mit den Parametern `x=40` und `y=2`).

Ein `HttpClient` wird ebenfalls mittels eines Builder erzeugt (und ist daher ebenfalls immutable); er wird benutzt, um beliebig häufig Requests zu versenden.

Der `send`-Methode des `HttpClient`s wird der Request und ein `BodyHandler` übergeben. Der `BodyHandler` dekodiert den im Response enthaltene Antwort. Die `send`-Methode bockiert – und liefert nach Eintreffen der Antwort einen `HttpResponse<String>` (weil `BodyHandlers.toString()` einen `BodyHandler<String>` erzeugt). Dieser enthält neben dem HTTP-Status-Code den Body des Requests (der im obigen Fall "42" enthält).

Der folgende Client demonstriert den asynchronen Aufruf der `GET`-Methode:

```
package client;

// ...
import java.util.concurrent.CompletableFuture;

public class Client2 {
```

```
public static void main(String[] args) throws Exception {  
    final HttpRequest request = HttpRequest.newBuilder()  
        .uri(URI.create("http://localhost:8080/math?  
x=40&y=2"))  
        .GET()  
        .build();  
  
    final HttpClient client = HttpClient.newBuilder()  
        .version(Version.HTTP_2)  
        .build();  
  
    final CompletableFuture<HttpResponse<String>> future =  
        client.sendAsync(request, BodyHandlers.ofString());  
  
    future.thenAccept(response -> {  
        System.out.println("status = " +  
response.statusCode());  
        System.out.println("body    = " + response.body());  
    }).join();  
}
```

Statt `send` wird nun `sendAsync` verwendet. Auch an `sendAsync` wird ein `HttpRequest` und ein `BodyHandler` übergeben. Im Unterschied zu `send` kehrt `sendAsync` nun aber unmittelbar zurück und liefert dabei ein `CompletableFuture` (welches dann nach Eintreffen der Antwort den `HttpResponse` enthält).

Der `thenAccept`-Methode des `CompletableFuture`s wird ein `Consumer` übergeben, dem der `HttpRequest` übergeben werden wird (sobald die Antwort denn eintrifft). Sie liefert ein `CompletableFuture<Void>`, dessen `join`-Methode benutzt wird, um auf das Eintreffen des Requests zu warten (`join` blockiert).

Der letzte hier vorgestellte Client nutzt die `POST`-Methode:

```
package client;  
  
// ...  
import java.net.http.HttpRequest.BodyPublisher;  
import java.net.http.HttpRequest.BodyPublishers;  
  
public class Client3 {  
    public static void main(String[] args) throws Exception {
```

```
        final BodyPublisher publisher =
            BodyPublishers.ofString("40\n2\n");

        final HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create("http://localhost:8080/math"))
            .POST(publisher)
            .build();

        final HttpClient client = HttpClient.newBuilder()
            .version(Version.HTTP_2)
            .build();

        final HttpResponse<String> response =
            client.send(request, BodyHandlers.ofString());

        System.out.println("status = " + response.statusCode());
        System.out.println("body    = " + response.body());
    }
}
```

Wir benötigen nun einen `BodyPublisher` (der hier einen zweizeiligen String mit den vom Server zu verarbeitenden Werte liefert). Der `HttpRequest` wird nun nicht mit `GET()`, sondern mit `POST(publisher)` erzeugt. Mittels `send` wird wie auch beim ersten der hier vorgestellten Clients ein synchroner Request ausgeführt.

## 10.8 WebSockets mit dem HttpClient

Auch WebSockets können nun mittels eines `HttpClient`s erzeugt werden. Als Beispiel verwenden wir auch hier wieder einen Server, der die Summe zweier Zahlen berechnet.

Auch dieses Projekt besteht aus zwei Teilprojekten:

```
java-11-HttpClient-WebSocket
java-11-HttpClient-WebSocket-Server
```

Der Server kann wieder mittels einer `build.xml` beim Tomcat deployt werden.

Hier der Server:

```
package websockets;

import javax.websocket.OnClose;
import javax.websocket.OnError;
```

```
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/math")
public class MathServer {

    public MathServer() {
        this.println("CTOR");
    }

    @OnOpen
    public void onOpen() {
        this.println("onOpen()");
    }

    @OnClose
    public void onClose() {
        this.println("onClose()");
    }

    @OnMessage
    public String onMessage(String message) throws Exception {
        this.println(">> onMessage(" + message + ")");
        String[] tokens = message.split(",");
        try {
            final int x = Integer.parseInt(tokens[0]);
            final int y = Integer.parseInt(tokens[1]);
            final int result = x + y;
            Thread.sleep(1000);
            this.println("<< onMessage(" + message + ")");
            return String.valueOf(result);
        }
        catch (Exception e) {
            return e.getMessage();
        }
    }

    @OnError
    public void onError(Throwable e) {
        this.println("onError(" + e + ")");
    }

    private void println(String message) {
        System.out.printf("[%3d] %s\n",
            Thread.currentThread().getId(), message);
    }
}
```

```
}  
}
```

Und hier der Client:

```
package client;  
  
import java.net.URI;  
import java.net.http.HttpClient;  
import java.net.http.HttpClient.Version;  
import java.net.http.WebSocket;  
import java.util.concurrent.CompletionStage;  
import java.util.concurrent.CountDownLatch;  
  
public class Client {  
  
    public static void main(String[] args) throws Exception {  
  
        final int N = 3;  
        final CountDownLatch done = new CountDownLatch(N);  
  
        WebSocket.Listener listener = new WebSocket.Listener() {  
            @Override  
            public void onOpen(WebSocket webSocket) {  
                println("onOpen");  
                WebSocket.Listener.super.onOpen(webSocket);  
            }  
  
            @Override  
            public CompletionStage<?> onText(WebSocket webSocket,  
                CharSequence data, boolean last) {  
                println("onText(" + data + ")");  
                return WebSocket.Listener.super.onText(  
                    webSocket, data, last);  
            }  
  
            @Override  
            public CompletionStage<?> onClose(WebSocket  
webSocket,  
                int statusCode, String reason) {  
                println("onClose(" + statusCode + ", " + reason +  
" )");  
                done.countDown();  
                return WebSocket.Listener.super.onClose(  
                    webSocket, statusCode, reason);  
            }  
        }  
    }  
}
```

```
};

final HttpClient client = HttpClient.newBuilder()
    .version(Version.HTTP_2)
    .build();

Watch w = new Watch("duration with " + N + "
WebSockets");
for (int i = 0; i < N; i++) {
    WebSocket webSocket = client.newWebSocketBuilder()
        .buildAsync(
URI.create("ws://localhost:8080/ws/math"),
        listener
    ).join();

    println("sendText(40,2)");
    webSocket.sendText("40,2", true);
    println("sendText(70,7)");
    webSocket.sendText("70,7", true);

    println("sendClose()");
    webSocket.sendClose(WebSocket.NORMAL_CLOSURE, "ok");
}
println("await()");
done.await();
println(w.toString());
}

private static void println(String message) {
    System.out.printf("[%2d] %s\n",
        Thread.currentThread().getId(), message);
}
}
```

Die `HttpClient`-Methode `newWebSocketBuilder` erzeugt einen Builder, mittels dessen ein `WebSocket` erzeugt wird. Ihm wird die URL und ein Objekt einer Klasse übergeben, die von `WebSocket.Listener` abgeleitet ist. Mittels `sendText` kann dann ein asynchroner Request abgesetzt werden (`sendText` kehrt also sofort zurück).

Sobald die Antwort auf einen Request eintrifft, wird sie an die `onText`-Methode des für den `WebSocket` erzeugten `WebSocket.Listener` zugestellt.

Man interpretiere nun die Ausgaben.

**Die Client-seitigen Ausgaben:**

```
[16] onOpen
[ 1] sendText(40,2)
[ 1] sendText(70,7)
[ 1] sendClose()
[16] onOpen
[ 1] sendText(40,2)
[ 1] sendText(70,7)
[ 1] sendClose()
[16] onOpen
[ 1] sendText(40,2)
[ 1] sendText(70,7)
[ 1] sendClose()
[ 1] await()
[12] onText(42)
[12] onText(42)
[12] onText(42)
[12] onText(77)
[12] onClose(1000, ok)
[12] onText(77)
[12] onClose(1000, ok)
[12] onText(77)
[12] onClose(1000, ok)
[ 1] duration with 3 WebSockets : 2227
```

**Die Server-seitigen Ausgaben:**

```
[228] CTOR
[228] onOpen()
[224] >> onMessage(40,2)
[225] CTOR
[225] onOpen()
[221] >> onMessage(40,2)
[231] CTOR
[231] onOpen()
[230] >> onMessage(40,2)
[224] << onMessage(40,2)
[224] >> onMessage(70,7)
[221] << onMessage(40,2)
[221] >> onMessage(70,7)
[230] << onMessage(40,2)
[230] >> onMessage(70,7)
[224] << onMessage(70,7)
[224] onClose()
[221] << onMessage(70,7)
[221] onClose()
[230] << onMessage(70,7)
[230] onClose()
```





## 11 Java 12

Dieses Kapitel stellt wesentliche Erweiterungen von Java 12 vor – beschränkt sich aber auf diejenigen Erweiterungen, die für den Entwickler unmittelbar interessant sind.

- In Java 12 wird das `switch`-Statement erweitert (allerdings handelt es sich hierbei um einen sog. "Preview" – die entgültige Version steht also noch nicht definitiv fest).
- Und die Klasse `String` ist um einige Methoden erweitert worden.

## 11.1 Switch

Da die Neuerungen im Preview-Status sind, sollten eine `@SuppressWarnings`-Annotation verwendet werden:

```
@SuppressWarnings("preview")
```

Im herkömmlichen `switch` benötigten wir mehrere `case`-Zweige, um für die Werte dieser Zweige ein gemeinsames Verhalten zu implementieren - wobei wir uns dabei den `fall throughs` zunutze machten:

```
static void demo1() {  
    int day = 2;  
    switch (day) {  
        case 1:  
        case 2:  
        case 3:  
        case 4:  
        case 5:  
            System.out.println("workday");  
            break;  
        case 6:  
            System.out.println("saturday");  
            break;  
        case 7:  
            System.out.println("sunday");  
            break;  
    }  
}
```

Bei dem erweiterten `switch` können mehrere Werte in derselben `case`-Marke definiert werden:

```
static void demo2() {  
    int day = 2;  
    switch (day) {  
        case 1, 2, 3, 4, 5:  
            System.out.println("workday");  
            break;  
        case 6:  
            System.out.println("saturday");  
            break;  
        case 7:  
            System.out.println("sunday");  
            break;  
    }  
}
```

```
    }  
}
```

Wir können den obigen `switch` noch knapper formulieren, indem wir den bereits aus Lambda-Ausdrücken bekannten `->`-Operator nutzen – dann benötigen wir keinen `break` mehr:

```
static void demo3() {  
    int day = 2;  
    switch (day) {  
        case 1, 2, 3, 4, 5 -> System.out.println("workday");  
        case 6 -> System.out.println("saturday");  
        case 7 -> System.out.println("sunday");  
    }  
}
```

Eine `switch`-Anweisung kann nun auch einen Wert besitzen – einen Wert, der z.B. als Resultat einer Methode zurückgeliefert werden kann:

```
static int calc1(char op, int x, int y) {  
    return switch (op) {  
        case '+': yield x + y;  
        case '-': yield x - y;  
        case '*': yield x * y;  
        case '/': yield x / y;  
        default: throw new RuntimeException();  
    };  
}
```

(In Java 13 wird `yield` verwendet – in Java 12 wurde stattdessen noch `break` verwendet!)

Ein möglicher Aufruf von `calc1`:

```
static void demo4() {  
    int result = calc1('+', 40, 2);  
    System.out.println(result);  
}
```

Statt wie in `calc1` den Doppelpunkt und `break` zu verwenden, können wir auch hier den Pfeil-Operator benutzen:

```
static int calc2(char op, int x, int y) {  
    return switch (op) {  
        case '+' -> x + y;  
        case '-' -> x - y;  
    }  
}
```

```
        case '*' -> x * y;
        case '/' -> x / y;
        default -> throw new RuntimeException();
    };
}
```

Die `switch`-Kontrollstruktur wird häufig bei `enums` verwendet. Sei z.B. folgende enum-Klasse gegeben:

```
static enum Operator { PLUS, MINUS, TIMES, DIV};
```

Dann können wir folgende `calc3`-Methode schreiben:

```
static int calc3(Operator op, int x, int y) {
    return switch (op) {
        case PLUS -> x + y;
        case MINUS -> x - y;
        case TIMES -> x * y;
        case DIV -> x / y;
    };
}
```

Hier ist auch kein `default` erforderlich – der Compiler verlangt, dass für alle `enum`-Werte ein entsprechender `case` existiert.

Die Typen der von den `case`-Zweigen gelieferten Werte können verschieden sein. Das Resultat der `switch`-Anweisung hat dann den "größten" gemeinsamen Basistyp dieser Typen:

```
static void demo5() {
    int x = 2;
    Number n = switch (x) {
        case 1 -> 42;
        case 2 -> 77L;
        case 3 -> 3.14;
        default -> 0;
    };
    System.out.println(n + " " + n.getClass().getName());
}
```

Die Ausgabe:

```
77 java.lang.Long
```

Wir hätten statt `Number` auch `double` verwenden könne. Dann würde folgende Zeile ausgegeben:

```
Double n = (double)switch (x) {
```

77.0 java.lang.Double

Was passiert, wenn wir den Compiler den Typ berechnen lassen – wenn wir die Zielvariable also mit `var` definieren?:

```
static void demo6() {  
    int x = 2;  
    var v = switch (x) {  
        case 1 -> 42;  
        case 2 -> 77L;  
        case 3 -> 3.14;  
        default -> 0;  
    };  
    Number n = v;  
    System.out.println(n + " " + n.getClass().getName());  
}
```

Die Ausgabe:

77.0 java.lang.Double

## 11.2 Strings

Die Klasse String ist um eine indent-Methode erweitert worden:

```
static void demo1() {  
    String s1 = "Hello";  
    String s2 = s1.indent(8);  
    System.out.println(s1);  
    System.out.println(s2);  
}
```

Die Ausgaben:

```
Hello  
      Hello
```

Und sie enthält nun eine transform-Methode, der eine Function übergeben wird:

```
static void demo2() {  
    String s = "3.14";  
    double d1 = s.transform(str -> Double.parseDouble(str));  
    System.out.println(d1);  
    double d2 = s.transform(Double::parseDouble);  
    System.out.println(d1);  
}
```

Die Ausgaben:

```
3.14  
3.14
```

## 12 Java 13

Dieses Kapitel stellt Erweiterungen von Java 13 vor.

- Im ersten Abschnitt werden die neuen Text-Blöcke vorgestellt.
- Im zweiten Abschnitt geht's um neue String-Methoden.

## 12.1 Text Blocks

Ein SQL-Select-String konnte bislang wie folgt aufgebaut werden:

```
static void demo1() {  
    String sql = "" +  
        "select\n" +  
        "    isbn, title, author\n" +  
        "from\n" +  
        "    book\n" +  
        "where\n" +  
        "    title = 'Pascal';  
    System.out.println(sql);  
}
```

Die Ausgaben:

```
select  
    isbn, title, author  
from  
    book  
where  
    title = 'Pascal'
```

Das ist natürlich alles andere als übersichtlich.

Java 13 bietet nun (in Form eines Previews) die Möglichkeit, solche String in der syntaktischen Form eines Textblocks zu definieren:

```
static void demo2() {  
    String sql = ""  
        select  
            isbn, title, author  
        from  
            book  
        where  
            title = 'Pascal'  
        "";  
    System.out.println(sql);  
}
```

Ein Textblock beginnt und endet mit drei Double-Quotes. Die Start-Quotes müssen am Ende einer Zeile stehen.

Die Ausgaben der obigen Methode sind dieselben wie diejenigen von `demo1`.



Soll ein gewöhnliches String-Literals Double-Quotes enthalten, so müssen diese maskiert werden:

```
static void demo3() {  
    String json = "" +  
        "{\n" +  
        "    isbn : \"1111\", \n" +  
        "    title : \"Pascal\", \n" +  
        "    author : \"Wirth\", \n" +  
        "    year : 1970\n" +  
        "}";  
    System.out.println(json);  
}
```

Die Ausgaben:

```
{  
  isbn : "1111",  
  title : "Pascal",  
  author : "Wirth",  
  year : 1970  
}
```

In Text-Blöcken werden Double-Quotes nicht maskiert:

```
static void demo4() {  
    String json = ""  
        {  
            isbn : "1111",  
            title : "Pascal",  
            author : "Wirth",  
            year : 1970  
        }  
        "";  
    System.out.println(json);  
}
```

Die Ausgaben sind dieselben wie die von `demo3`.

Durch die Position der abschließenden drei Double-Quotes kann die Einrückungs-Tiefe der Zeilen eines Text-Block-Literals beeinflusst werden:

```
static void demo5() {  
    String json = ""  
        {  
            isbn : "1111",
```

```
        title : "Pascal",
        author : "Wirth",
        year : 1970
    }
    """;
    System.out.println(json);
}
```

Jede Zeile der Ausgabe wird nun um vier Zeichen eingerückt:

```
{
    isbn : "1111",
    title : "Pascal",
    author : "Wirth",
    year : 1970
}
```

Und natürlich kann ein Text-Block auch Formatierungs-Elemente (%s, %d etc.) enthalten, die via `String.format` mit aktuellen Werten ersetzt werden können:

```
static void demo6() {
    String json = String.format("""
        {
            isbn : "%s",
            title : "%s",
            author : "%s",
            year : %d
        }
        """,
        "1111", "Pascal", "Wirth", 1970);
    System.out.println(json);
}
```

Die Ausgaben:

```
{
    isbn : "1111",
    title : "Pascal",
    author : "Wirth",
    year : 1970
}
```

## 12.2 Neue String-Methoden

Auch die folgenden neuen String-Methoden haben noch Preview-Status.

Die String-Klasse hat nun eine Instanzmethode `formatted`:

```
static void demo1() {  
    String s = "alpha %s beta %d";  
    String s1 = String.format(s, "Hello", 42);  
    System.out.println(s1);  
    String s2 = s.formatted("Hello", 42);  
    System.out.println(s2);  
}
```

```
alpha Hello beta 42  
alpha Hello beta 42
```

Mittels der Methode `stripIndent` kann ein mehrzeiliger String derart formatiert werden, dass die die Einrückungen der String-Zeilen entfernt werden:

```
static void demo2() {  
    String s = "  alpha      \n      beta";  
    System.out.println(s);  
    System.out.println(s.stripIndent());  
}
```

```
alpha  
  beta  
alpha  
  beta
```

Mittels `translateEscapes` kann makierte Escape-Sequenzen demaskiert werden:

```
static void demo3() {  
    String s = "alpha      \\n  \\t  beta";  
    System.out.println(s);  
    System.out.println(s.translateEscapes());  
}
```

```
alpha      \n  \t  beta  
alpha  
  beta
```

## 13Anhang: Build mit Maven

Im Folgenden wird anhand eines kleinen Beispiels gezeigt, wie Maven zum Bau modularer `jar`s genutzt werden kann.

Die Anwendung besteht aus zwei Eclipse-Projekten:

```
x9901-MathService  
x9901-MathServiceApp.
```

Dabei ist das zweite Projekt vom ersten abhängig.

Jedes Projekt enthält eine `pom.xml`, mittels derer jeweils eine `jar` gebaut wird:

```
MathService-0.0.1-SNAPSHOT.jar  
MathServiceApp-0.0.1-SNAPSHOT.jar
```

Zunächst seien hier die Sourcen der beiden Projekte vorgestellt:

### Das Projekt MathService

```
package jj.mathService;  
  
import jj.mathService.impl.MathServiceImpl;  
  
public interface MathService {  
  
    public abstract int sum(int x, int y);  
    public abstract int diff(int x, int y);  
  
    public static MathService create() {  
        return new MathServiceImpl();  
    }  
}
```

`jj.mathService.MathService` spezifiziert zwei Methoden und fungiert zugleich als Factory für Objekte, deren Klasse das `MathService`-Interface implementiert.

`jj.mathService.impl.MathServiceImpl` implementiert das obige Interface:

```
package jj.mathService.impl;  
  
import jj.mathService.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 Projekt enthält eine kleine Testanwendung:

```
package jj.mathService.test;  
  
import org.junit.jupiter.api.Assertions;  
import org.junit.jupiter.api.Test;  
  
import jj.mathService.MathService;  
  
public class MathServiceTest {  
    @Test  
    public void testSum() {  
        Assertions.assertEquals(42, MathService.create().sum(40,  
2));  
    }  
    @Test  
    public void testDiff() {  
        Assertions.assertEquals(77, MathService.create().diff(80,  
3));  
    }  
}
```

Nur das Package, welches das Interface enthält, wird exportiert (als Modul-Name wird der Package-Name verwendet):

```
module jj.mathService {  
    exports jj.mathService;  
}
```

## Das Projekt MathServiceAppl

Das `MathServiceAppl`-Projekt enthält eine kleine `main`-Klasse, welche den `MathService` nutzt:

```
package jj.mathServiceAppl;
```

```
import jj.mathService.MathService;

public class Application {
    public static void main(String[] args) {
        MathService mathService = MathService.create();
        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(80, 3));
    }
}
```

```
module jj.mathServiceAppl {
    requires jj.mathService;
}
```

Hier die pom.xml für das erste Projekt:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 ...
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>jj</groupId>
    <artifactId>MathService</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>MathService</name>
    <url>http://maven.apache.org</url>

    <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>5.1.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.6.0</version>
      <configuration>
        <source>9</source>
        <target>9</target>
        <showWarnings>true</showWarnings>
        <showDeprecation>true</showDeprecation>
      </configuration>
    </plugin>
  </plugins>
</build>

</project>
```

Mittels `mvn pom -> run` als -> Test kann der Test gestartet werden; mittels `mvn pom -> run as -> install` kann die `jar`-Datei (`MathService-0.0.1-SNAPSHOT.jar`) erzeugt und im Maven-Repository abgestellt werden.

Für das zweite Projekt sieht die `pom.xml` wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>

  <groupId>jj</groupId>
  <artifactId>MathServiceAppl</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>MathServiceAppl</name>
  <url>http://maven.apache.org</url>

  <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
```

```
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.6.0</version>
        <configuration>
            <source>9</source>
            <target>9</target>
            <showWarnings>true</showWarnings>
            <showDeprecation>true</showDeprecation>
        </configuration>
    </plugin>
</plugins>
</build>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>jj</groupId>
            <artifactId>MathService</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
</dependencyManagement>

</project>
```

Mittels `mvn pom -> run as -> install` kann die `jar`-Datei (`MathServiceAppl-0.0.1-SNAPSHOT.jar`) erzeugt und im Maven-Repository abgestellt werden.

Nach der Installation liegen beide `jars` im Repository. Zusätzlich existieren sie aber auch im `target`-Ordner des entsprechenden Projekts.

Um das zweite Projekt nun auch in Eclipse übersetzen zu können, können wir den `module-path` um die `MathService-0.0.1-SNAPSHOT.jar` erweitern.

Im zweiten Projekt befindet sich schließlich eine kleine `run.bat`, mittels derer die Anwendung gestartet werden kann:

```
java --module-path
    ..\x9901-MathService\target\MathService-0.0.1-SNAPSHOT.jar;
    target\MathServiceAppl-0.0.1-SNAPSHOT.jar
    --module jj.mathServiceAppl/jj.mathServiceAppl.Application
```



## **14Literatur**

Guido Oelmann: Modularisierung mit Java 9. DPunkt-Verlag

Michael Inden: Java 9 – Die Neuerungen. DPunkt-Verlag

Cay S. Horstmann: Core Java Volume 2 - Advanced Features. Pearson