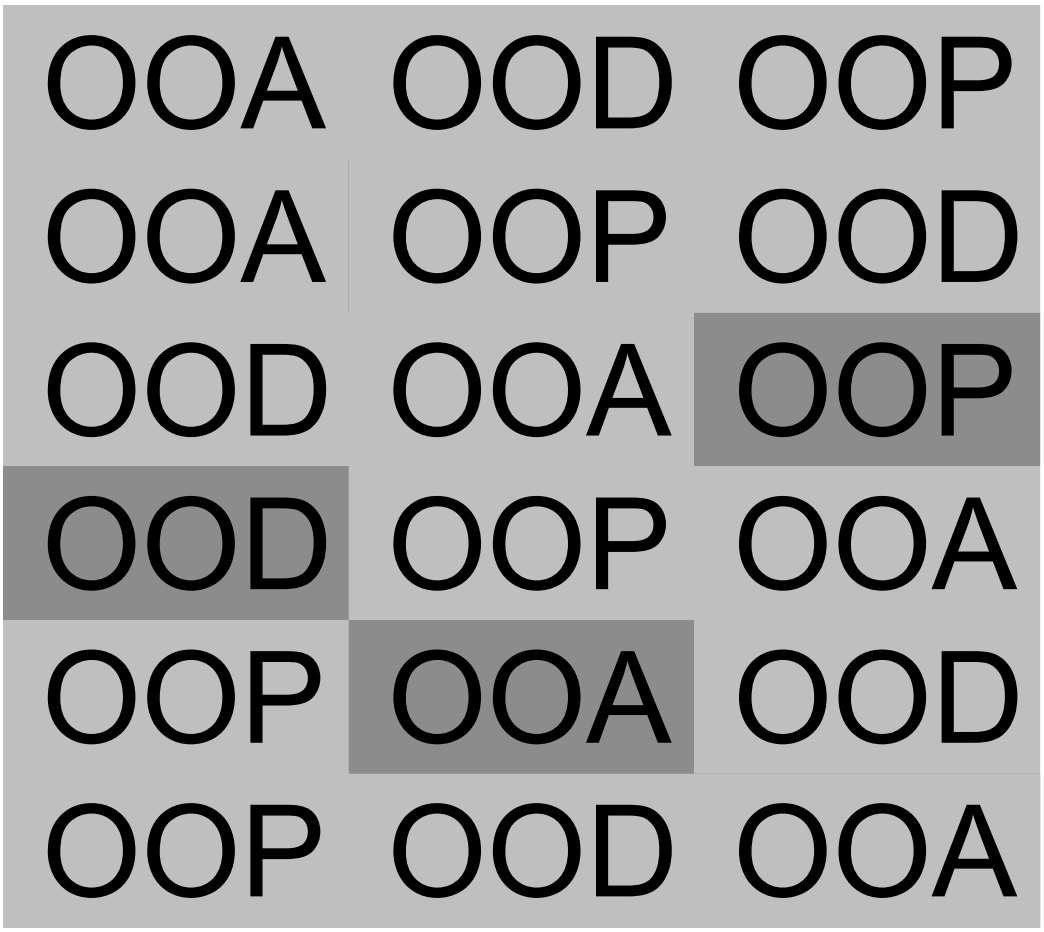
Java 8

Java goes functional



Johannes Nowak

Johannes Nowak

e-mail: johannes.nowak@t-online.de

Juni 2014
Januar / März 2015

Inhalt

1 Einleitung

Die folgende Einleitung umfasst drei Abschnitte:

Im ersten Abschnitt werden die wesentlichen Neuerungen von Java 8 in Form eines Überblicks vorgestellt.

Der zweite Abschnitt beschreibt die verwendeten Tools und die Struktur des Eclipse-Workspaces.

Im dritten Abschnitt schließlich werden einige Helper-Klassen vorgestellt, die in (fast) allen Beispiel-Projekten genutzt werden.

1.1 Was gibt's Neues?

Hier eine Liste der wesentlichen Neuerungen, die mit Java 8 einführt wurden.

Lambda-Ausdrücke

In Java 8 wurde endlich ein neues Sprachkonstrukt eingeführt, das viele Entwickler bislang immer schmerzlich vermißt haben: Lambda-Ausdrücke (auch als "Closures" bezeichnet).

Ein Lambda-Ausdruck repräsentiert namenlose Funktionalität – anders (aber auch ungenau!) gesagt: ein Lambda-Ausdruck ist eine anonyme Methode. Eine Methode also, welche nur Parameter und Code definiert, aber nicht unter einem eigenen Namen ansprechbar ist. Eine solche Methode kann an Variablen gebunden werden, als Parameter an andere Methoden übergeben oder von diesen als Return-Wert zurückgeliefert werden.

Wie fügt sich ein solches Konzept in das bisherige Typsystem von Java ein? Repräsentiert ein Lambda-Ausdruck (resp. die Referenz auf einen Ausdruck) einen komplett neuen Typ? Ist also das Typ-System von Java erweitert worden (etwa um so etwas wie die "delegates" von C#)?

Glücklicherweise fügen sich die Lambda-Ausdrücke recht nahtlos in das bisherige Typsystem ein: eine Lambda-Ausdruck kann verglichen werden mit einer Instanz einer anonymen Klasse, welche nur eine einzige Methode besitzt. Ebenso wie eine anonyme Klasse entweder i.d.R. ein Interface implementiert, implementiert auch ein Lambda-Ausdruck ein Interface – ein Interface allerdings, welches eben nur eine einzige Methode spezifiziert. Ein solches Interface wird als "funktionales Interface" bezeichnet. Vergleicht man nun eine anonyme Klasse, die ein solches Interface implementiert, mit einem Lambda-Ausdruck, so fällt auf, dass die Lambda-Notation wesentlich "schlanker" ist als die Notation in Form einer anonymen Klasse – und (nach einiger Übung!) auch verständlicher.

Ein kleines Beispiel zur Einstimmung: bei einem `Button` soll ein `ActionListener` registriert werden. Das Interface `ActionListener` spezifiziert nur eine einzige Methode: `actionPerformed(ActionEvent e)` – es handelt sich somit um ein funktionales Interface.

Die "alte" Notation:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Hello World");
```

```
    }
}
```

Derselbe Listener kann in der Lambda-Notation wesentlich kürzer formuliert werden:

```
button.addActionListener((ActionEvent e) -> {
    System.out.println("Hello World");
}
```

Es geht auch noch knapper:

```
button.addActionListener(e -> System.out.println("Hello World"));
```

Im folgenden wird genau herausgearbeitet werden müssen, worin die Gemeinsamkeiten von Lambda-Ausdrücken und anonyme Klassen bestehen und worin sich diese Konstrukte voneinander unterscheiden.

U.a. mit der Einführung von Lambdas geht Java den ersten Schritt in Richtung "funktionaler Programmierung".

Interfaces

Interfaces dienten bislang in erster Linie dazu, Funktionalität abstrakt zu spezifizieren. Mit einer Ausnahme: man konnte in einem Interface (und kann natürlich immer noch) auch statische Konstanten definieren.

In Java 8 wird der Begriff Interfaces erweitert. So können in Interfaces nun auch statische Methoden implementiert werden. Und auch nicht-statische Methoden können bereits implementiert werden – in Form sog. `default`-Methoden.

Damit kann in einem Interface bereits fast all das definiert werden, was auch in einer gewöhnlichen Klasse definiert werden kann – mit einer einzigen (aber entscheidenden) Ausnahme: Instanzvariablen können weiterhin auch nur in Klassen definiert werden, nicht aber in Interfaces.

Ein Beispiel aus dem Paket `java.util`:

```
public interface Iterator<E> {
    public abstract boolean hasNext();
    public abstract E next();
    public default void remove() {
        throw new UnsupportedOperationException("remove");
    }
    default void forEachRemaining(Consumer<? super E> action) {
        ...
```

```
    }
}
```

Eine Implementierung dieses Interfaces kann sich nun auf die Bereitstellung der `hasNext`- und `next`-Methoden beschränken – `remove` muss (im Gegensatz zum "alten" Interface) nicht mehr implementiert werden (kann aber überschrieben werden). Und es existiert eine neue Methode, die in dem alten Interface noch nicht enthalten war: `forEachRemaining`.

Da eine Klasse viele Interfaces implementieren kann, könnte der Eindruck entstehen, Java unterstütze nun "Mehrfachvererbung". Da aber Interfaces weiterhin keine nicht statischen Attribute (keine Instanzvariablen) definieren können, handelt es sich hierbei nicht um Mehrfachvererbung im strengen Sinne – und das ist auch gut so. Eine Sprache, die Mehrfachvererbung vollständig unterstützt, handelt sich nämlich eine Menge von Problemen ein. Und Java bleibt hoffentlich weiterhin eine (relativ) einfache Sprache...

Die Erweiterung des Interface-Begriffs hat offensichtliche Vorteile – aber auch Nachteile. Sowohl die Vorteile als auch die Nachteile werden ausführlich dargestellt werden.

Funktionale Interfaces der Standardbibliothek

Da Lambdas eine zentrale Bedeutung gewinnen, benötigt man ein Standard-Set an entsprechenden funktionalen Interfaces. Java 8 führt eine Vielzahl solcher neuer Interfaces ein: `Supplier`, `Consumer`, `Function`, `Predicate` etc. Diese Interfaces – und vor allem ihrer Verwendungsmöglichkeiten – werden ausführlich dargestellt.

Standardbibliothek

Neben den neu eingeführten funktionalen Interfaces sind einige zentrale "alte" Interfaces der Standardbibliothek erweitert worden. Insbesondere sind solche Interfaces ergänzt worden durch statische Methoden und durch `default`-Methoden. Dies gilt insbesondere auch für die Interfaces des Collection-Frameworks.

Darüber hinaus sind einige weitere wichtige Konzepte aufgenommen worden: so z.B. das `Optional`-Konzept und das `Spliterator`-Konzept. Das `Optional`-Konzept etwa erlaubt einen etwas bewußteren Umgang mit `null`-Werten und kann dazu verhelfen, `NullPointerException`s weitgehend zu vermeiden.

Und auch Reflection ist erweitert worden: endlich können nicht nur die Typen der Methodenparameter ermittelt werden, sondern auch deren Namen (und weitere Eigenschaften – etwa die Annotations, mit denen ein Parameter ausgezeichnet ist).

Streams

Bei der Verarbeitung von Listen (oder Sets oder Maps) geht es um immer wiederkehrende Aufgaben: eine Liste muss gefiltert werden (bestimmte Elemente müssen zur Weiterverarbeitung ausgewählt werden); die Elemente der Liste müssen irgendwie aggregiert werden (z.B. summiert werden); alle Elemente müssen auf jeweils andere Elemente abgebildet werden (aus einer Liste von Zahlen muss eine Liste von Strings erzeugt werden); auf alle Elemente einer Liste muss eine Ausgabe-Operation aufgerufen werden; etc.

Häufig müssen diese Operationen auch miteinander kombiniert werden: Filtern, Mappen, Ausgabe. Um solche Aufgaben elegant lösen zu können, führt Java 8 die sog. Streams ein. Ein Stream kann als eine Pipeline betrachtet werden. Die Daten einer Liste durchlaufen die Pipeline und werden an den verschiedenen Stationen dieser Pipeline jeweils unterschiedlich bearbeitet.

Ein Beispiel:

```
final List<Integer> list = new ArrayList<>();
// hier wird die Liste gefuellt....
Stream<Integer> stream = list.stream();
stream
    .map(x -> x * 3)
    .filter(x -> x % 2 == 0)
    .forEach(x -> System.out.print(x + " "));
```

Alle Elemente der Liste werden durch eine Pipe geschickt, innerhalb derer jedes Element zunächst mit 3 multipliziert wird. Bei der nächsten Station wird jedes Element daraufhin geprüft, ob es sich um eine gerade Zahl handelt – nur gerade Zahlen werden durchgelassen. Die letzte Station der Pipe gibt dann alle Zahlen, die zu ihr hingelangt sind, auf der Standardausgabe aus.

Das Stream-APIs wird ausführlich dargestellt werden – insbesondere auch die Konsequenzen dieses APIs in Bezug auf die parallele Verarbeitung.

Date / Time

Die meisten Methoden der `java.util.Date`-Klasse sind `deprecated`; und auch der `java.util.Calendar` hat's in sich:

```
Calendar c = GregorianCalendar.getInstance();
c.set(2015, 1, 25);
```

```
    out.println(DateFormat.getDateInstance(DateFormat.LONG)
        .format(c.getTime()));
```

Die Ausgabe:

```
25. Februar 2015
```

Sollte das `Calendar`-Objekt nicht den 25. Januar repräsentieren? (Klar: Monate beginnen bei 0, Tage aber bei 1!)

Grund genug also, ein neues Date/Time-API einzuführen. Dort gibt's u.a. die Klassen `Month`, `LocalDate`, `DateTimeFormatter` und `FormatStyle`:

```
    LocalDate d = LocalDate.of(2015, Month.JANUARY, 25);

out.println(DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG)
       .format(d));
```

Die Ausgaben:

```
25. Januar 2015
```

So ist es schön...

Multithreading

Java 8 erweitert das in Java 5 eingeführte und bereits in Java 7 erweiterte Paket `java.util.concurrent` um einige Klassen – z.B. um die Klassen `CompletableFuture` und `StampedLock`. Diese beiden Klassen werden genauer analysiert werden.

JavaScript-Engine

Java 8 stellt für JavaScript-Anwendungen eine neue JavaScript-Engine (einen JavaScript-Interpreter namens "Nashorn") zur Verfügung. Der alte Interpreter ("Rhino") wird abgelöst.

Die Liste der Features ist natürlich nicht vollständig. Aber alles geht nicht...

1.2 Verwendete Tools und Aufbau des Workspace

Der Workspace basiert auf folgendem JDK:

jdk1.8.0_31 (32-Bit Maschine)

Folgende Eclipse-Version wurde verwendet:

4.4.1 (Luna)

Der Workspace enthält eine Vielzahl von Projekten, deren Namen wie folgt aufgebaut sind:

`x<kkaa>-<Kapitel><Abschnitt>`

z.B.:

`x0301-Lambdas-ActionListener`

`03` ist die Nummer des "Kapitels", `01` die Nummer des "Abschnitts". `Lambdas` ist der Name des Kapitels, `ActionListener` ist der Name des Abschnitts.

Die Kapitel-Abschnitts-Nummerierung der Projekte entspricht exakt der Struktur der vorliegenden Skripts. Somit kann leicht zwischen dem Quellcode der Beispielprojekte und diesem Skript hin- und her geswitcht werden.

Die mit "u" präfixierten Projekte sind die Projekte, die für Übungen vorgesehen sind.

Zusätzlich zu den u- und x-Projekten existieren die Projekte `shared` und `db-util`. Das `shared`-Projekt enthält einige Klassen, die in vielen der x-Projekte verwendet werden. `db-util` kann benutzt werden, um auf einfache Weise eine Datenbank aufzubauen (`db-util` ist für Übungs-Projekte vorgesehen, die mit einer Datenbank arbeiten).

Das `dependencies`-Projekt enthält einige Tools (die Datenbank und ASM).

Ein letzter Hinweis: Der Quellcode der Beispielprojekte enthält keinerlei Kommentare (um Platz zu sparen...). Die Kommentare zu den Projekten befinden sich stattdessen in diesem Skript. (Das Skript besteht so gesehen aus den ausgelagerten Kommentaren.)

1.3 Klassen des shared-Projekts

In den Demonstrations-Beispielen werden bestimmte Utility-Klassen immer wieder verwendet. Dies sind deshalb in einem Projekt namens `shared` angesiedelt (im Package `util`). Dieses Projekt ist daher im `classpath` fast aller Beispielprojekte enthalten.

Hier seien bereits einige dieser Klassen kurz vorgestellt.

Die Klasse Util

Die Klasse `Util` enthält u.a. folgende statische Methoden:

```
package util;
// ..l
public class Util {

    public static void mlog() {
        final StackTraceElement[] elements =
            Thread.currentThread().getStackTrace();
        hlog(elements[2].getMethodName());
    }

    public static void hlog(String text) {
        final String LINE =
            "+--------------------------------------------";
        System.out.println(LINE);
        System.out.println("| " + text);
        System.out.println(LINE);
    }

    public static void tlog(String text, Object... args) {
        synchronized (System.out) {
            System.out.printf("[ %2d ] ",
Thread.currentThread().getId());
            System.out.printf(text, args);
            System.out.println();
        }
    }
}
```

Die Methode `mlog` kann benutzt werden, um am Anfang einer Methode den Namen der Methode auszugeben (dieser Name wird nicht übergeben, sondern dynamisch ermittelt). Ein Beispiel:

```
import static util.Util.mlog;

public class C {
    public static void main(String[] args) {
        mlog();
        foo();
    }
    static void foo() {
        mlog();
        // ...
        bar();
        // ...
    }
    static void bar() {
        mlog();
        // ...
    }
}
```

Die Ausgaben (hier verkürzt dargestellt):

```
main
foo
bar
```

Die Methode `hlog` kann verwendet werden, um eine "Überschrift" auszugeben (sie wird von `mlog` aufgerufen).

Mittels der Methode `tlog` kann die Id des aktuellen Threads auszugeben werden (mitsamt eines Info-Textes).

Die Klasse Features

Um der technischen Implementierung bestimmter Spracheigenschaften auf die Spur zu kommen, bietet es sich an, eine allgemein verwendbare Methode zur Reflection-basierten Untersuchung von Klassen zu benutzen. Zu diesem Zweck existiert im `shared`-Paket eine Klasse `util.Features`. Die `print`-Method dieser Klasse gibt alle in der an diese Methode übergebenen Klasse implementierten Attribute, Konstruktoren, Methoden und innere Klassen aus. Die `printInheritance`-Methode gibt die Ableitungs-Hierarchie aus.

```
package util;

import java.lang.reflect.Constructor;
```

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class Features {

    public static void print(Class<?> cls) {
        out.println(cls.getName()
            + " (" + cls.getDeclaringClass() + ")");
        final Constructor<?>[] constructors =
cls.getDeclaredConstructors();
        if (constructors.length > 0) {
            out.println("\tConstructors");
            for (Constructor<?> c : constructors)
                out.println("\t\t" + c);
        }
        final Field[] fields = cls.getDeclaredFields();
        if (fields.length > 0) {
            out.println("\tFields");
            for (Field f : fields)
                out.println("\t\t" + f);
        }
        final Method[] methods = cls.getDeclaredMethods();
        if (methods.length > 0) {
            out.println("\tMethods");
            for (Method m : methods)
                out.println("\t\t" + m);
        }
        final Class<?>[] classes = cls.getDeclaredClasses();
        if (classes.length > 0) {
            out.println("\tClasses");
            for (Class<?> c : classes)
                out.println("\t\t" + c);
        }
    }

    public static void printInheritance(Object obj) {
        String s = "";
        for(Class<?> cls = obj.getClass();
                cls != Object.class; cls = cls.getSuperclass()) {
            System.out.println(s + cls.getName());
            s += "\t";
        }
    }
}
```

Die Klasse SerializeUtil

In einigen der Beispielprojekte geht's u.a. um das Thema Serialisierung. In diesen Projekten wird folgende Klasse genutzt:

```
package util;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializeUtil {

    @SuppressWarnings("unchecked")
    public static <T> T serializeDeserialize(T obj) {
        return (T) deserialize(serialize(obj));
    }

    public static ByteArrayOutputStream serialize(Object obj) {
        final ByteArrayOutputStream out = new
ByteArrayOutputStream();
        try (final ObjectOutputStream oos = new
ObjectOutputStream(out)) {
            oos.writeObject(obj);
        }
        catch(Exception e) {
            throw new RuntimeException(e);
        }
        return out;
    }

    public static Object deserialize(ByteArrayOutputStream out) {
        final ByteArrayInputStream in =
            new ByteArrayInputStream(out.toByteArray());
        try (final ObjectInputStream ois = new
ObjectInputStream(in)) {
            return ois.readObject();
        }
        catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Damit die Festplatte nicht unnötig verschmutzt wird, wird für die Serialisierung einfach ein `ByteArrayOutputStream` verwendet. Bei der Deserialisierung wird ein `ByteArrayInputStream` als Quelle verwendet, welcher seine Daten von einem zuvor gefüllten `ByteArrayOutputStream` bezieht.

Die Klasse PerformanceRunner

Die Klasse kann zu Performance-Tests verwendet werden:

```
package util;
// ...
public class PerformanceRunner {

    public void run(String msg, int times, Runnable runnable) {
        final long start = System.nanoTime();
        try {
            for (int i = times; i > 0; --i) {
                runnable.run();
            }
        }
        catch (Throwable t) {
            System.out.println(t);
        }
        final long end = System.nanoTime();
        System.out.printf("%-30s : %5d\n", msg, (end - start) /
1_000_000);
    }

    public void run(String msg, int times,
                Runnable initRunnable, Runnable runnable) {
        long duration = 0;
        try {
            for (int i = times; i > 0; --i) {
                initRunnable.run();
                final long start = System.nanoTime();
                runnable.run();
                final long end = System.nanoTime();
                duration += (end - start);
            }
        }
        catch (Throwable t) {
            System.out.println(t);
        }
        System.out.printf("%-30s : %5d\n", msg, duration /
1_000_000);
```

```
    }}
```

Der ersten `run`-Methode wird neben einem Info-Test ein `Runnable` und `times`-Parameter übergeben. Die `run`-Methode dieses `Runnables` wird `times`-mal aufgerufen. Nach Beendigung der Schleife wird die gemessene Zeitdauer in Millisekunden ausgegeben.

Der zweiten `run`-Methode wird ein weiteres `Runnable` mitgegeben, welches jeweils vor dem "eigentlichen" `Runnable` ausgeführt – die Zeit, die dies Ausführung kostet, wird aber nicht mitgerechnet.

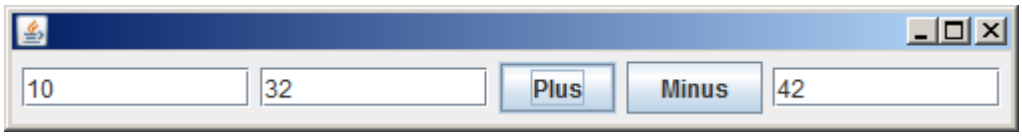
2 Typen von Klassen

Um das neu eingeführte Lambda-Konzept zu verstehen, ist es sinnvoll, zunächst noch einmal das "alte" Klassenkonzept näher zu beleuchten – insbesondere das Konzept der anonymen Klassen.

Wir untersuchen in diesem Kapitel folgende Klassen-Typen:

- Globale Klassen
- Statische Klassen
- Member-Klassen
- Lokale Klassen
- Anonyme Klassen

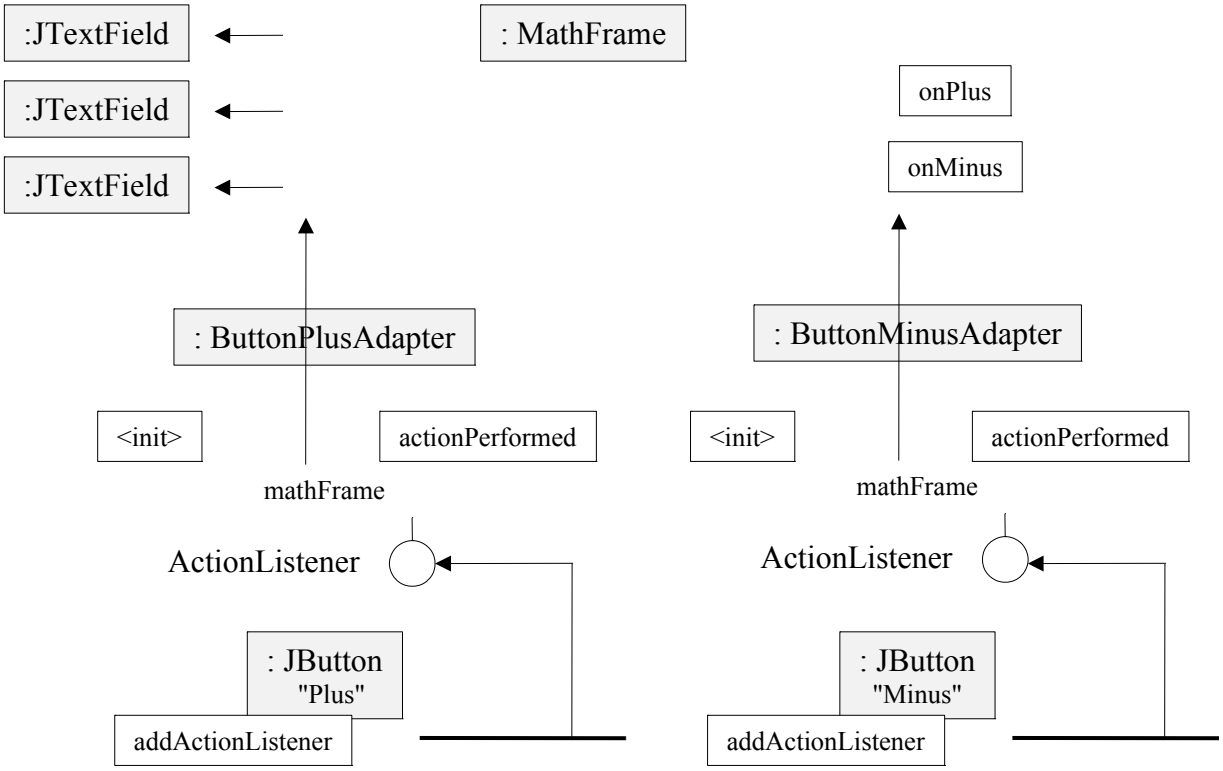
Als Beispiel verwenden wird eine einfache Swing-Anwendung – einen Kalkulator:



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

2.1 Global Classes

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



Das Diagramm beansprucht nicht, UML-konform zu sein...

Eine kurze Erläuterung der Anwendung:

Ein `MathFrame` (die Klasse ist abgeleitet von `JFrame`) besitzt Referenzen auf drei `JTextField`- und auf zwei `JButton`-Objekte.

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

Die folgende Anwendung definiert zwei globale Klassen, welche das `ActionListener`-Interface implementieren: `ButtonPlusAdapter` und `ButtonMinusAdapter`. Jede dieser

beiden Klassen wird genau einmal instantiiert – und die so erzeugten Adapter-Objekte bei den beiden `JButtons` registiert. Bei der Erzeugung der Adapter wird der Konstruktor der entsprechenden Klasse aufgerufen (im Bild als `<init>` bezeichnet). Diesem wird die Referenz auf den `MathFrame` als Parameter übergeben – welche dann in der Instanzvariablen `mathFrame` gespeichert wird.

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

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

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

Hier der komplette Quellcode:

```
public class Application {
    public static void main(String[] args) {
        new MathFrame();
    }
}
```

```
// ...
public class MathFrame extends JFrame {

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

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

```
        this.add(this.textFieldResult);
        this.registerListeners();
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.pack();
        this.setVisible(true);
    }

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

    public void onPlus() {
        try {
            int x = Integer.parseInt(this.textFieldX.getText());
            int y = Integer.parseInt(this.textFieldY.getText());
            int result = x + y;
            this.textFieldResult.setText(String.valueOf(result));
        }
        catch(NumberFormatException e) {
            this.textFieldResult.setText("Illegal input");
        }
    }

    public void onMinus() {
        try {
            int x = Integer.parseInt(this.textFieldX.getText());
            int y = Integer.parseInt(this.textFieldY.getText());
            int result = x - y;
            this.textFieldResult.setText(String.valueOf(result));
        }
        catch(NumberFormatException e) {
            this.textFieldResult.setText("Illegal input");
        }
    }
}
```

```
// ...
public class ButtonPlusAdapter implements ActionListener {
    final MathFrame mathFrame;
    public ButtonPlusAdapter(MathFrame mathFrame) {
        this.mathFrame = mathFrame;
    }
    public void actionPerformed(ActionEvent e) {
```

```
        this.mathFrame.onPlus();
    }
}
```

```
// ...
public class ButtonMinusAdapter implements ActionListener {
    final MathFrame mathFrame;
    public ButtonMinusAdapter(MathFrame mathFrame) {
        this.mathFrame = mathFrame;
    }
    public void actionPerformed(ActionEvent e) {
        this.mathFrame.onMinus();
    }
}
```

Die Anwendung hat zwei offensichtliche Schwächen: Erstens enthalten die `onPlus`- und `onMinus`-Methoden von `MathFrame` fast denselben Code (aus `onPlus` ist per Copy&Paste `onMinus` gemacht worden...). Zweitens unterscheiden sich die beiden Adapter-Klassen eigentlich nur durch die Implementierung der jeweiligen `actionPerformed`-Methode.

Das erste Problem werden wir später angehen. Hier geht's zunächst einmal um das zweite. Um gerade einmal die Clicks zweier Buttons an eine entsprechende Verarbeitungsmethode weiterzuleiten, werden zwei öffentliche Adapter-Klassen definiert - mit jeweils etwa 10 Zeilen (die sich jeweils nur in einer einzigen Zeile unterscheiden).

Trotzdem ist diese Lösung, was die Verwendung von Adaptern angeht, state of the art.

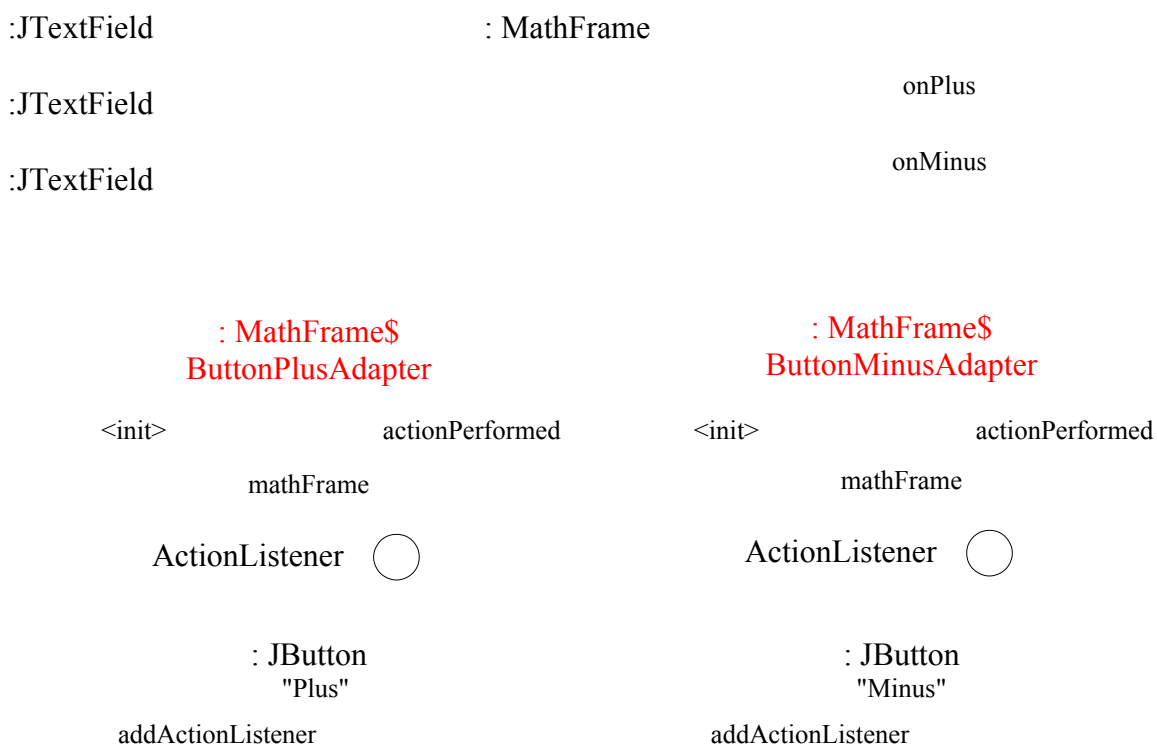
In den folgenden Abschnitten werden die inneren Klassen vorgestellt. Die Verwendung der inneren Klassen wird dazu führen, dass der Schreibaufwand radikal reduziert werden kann. Wobei aber die Technik – die Weiterleitung eines Events via Adapter – unverändert bleibt. Das Objektdiagramm bleibt also im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

2.2 Static Classes

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

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

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



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

Der Quellcode:

```
// ...
public class MathFrame extends JFrame {
```

```
    static class ButtonPlusAdapter implements ActionListener {
        final MathFrame mathFrame;
        public ButtonPlusAdapter(MathFrame mathFrame) {
            this.mathFrame = mathFrame;
        }
        public void actionPerformed(ActionEvent e) {
            this.mathFrame.onPlus();
        }
    }

    static class ButtonMinusAdapter implements ActionListener {
        final MathFrame mathFrame;
        public ButtonMinusAdapter(MathFrame mathFrame) {
            this.mathFrame = mathFrame;
        }
        public void actionPerformed(ActionEvent e) {
            this.mathFrame.onMinus();
        }
    }

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

    public MathFrame() {
        // wie gehabt ...
    }
    private void registerListeners() {
        this.buttonPlus.addActionListener(new
ButtonPlusAdapter(this));
        this.buttonMinus.addActionListener(new
ButtonMinusAdapter(this));
    }
    private void onPlus() {
        // wie gehabt ...
    }
    private void onMinus() {
        // wie gehabt ...
    }
}
```

Die statischen Adapter-Klassen hätten wir auch als `private` definieren können – außerhalb der umsschließenden Klassen ist ihre Verwendung wenig sinnvoll... Die

`actionPerformed`-Methode der Adapter-Klassen rufen `onPlus` und `onMinus` auf. Diese Methoden können nun `private` sein (anders als bei der Verwendung globaler Adapter-Klassen).

Der Compiler hat folgende `class`-Dateien erzeugt:

```
MathFrame$ButtonPlusAdapter
MathFrame$ButtonDiffAdapter
```

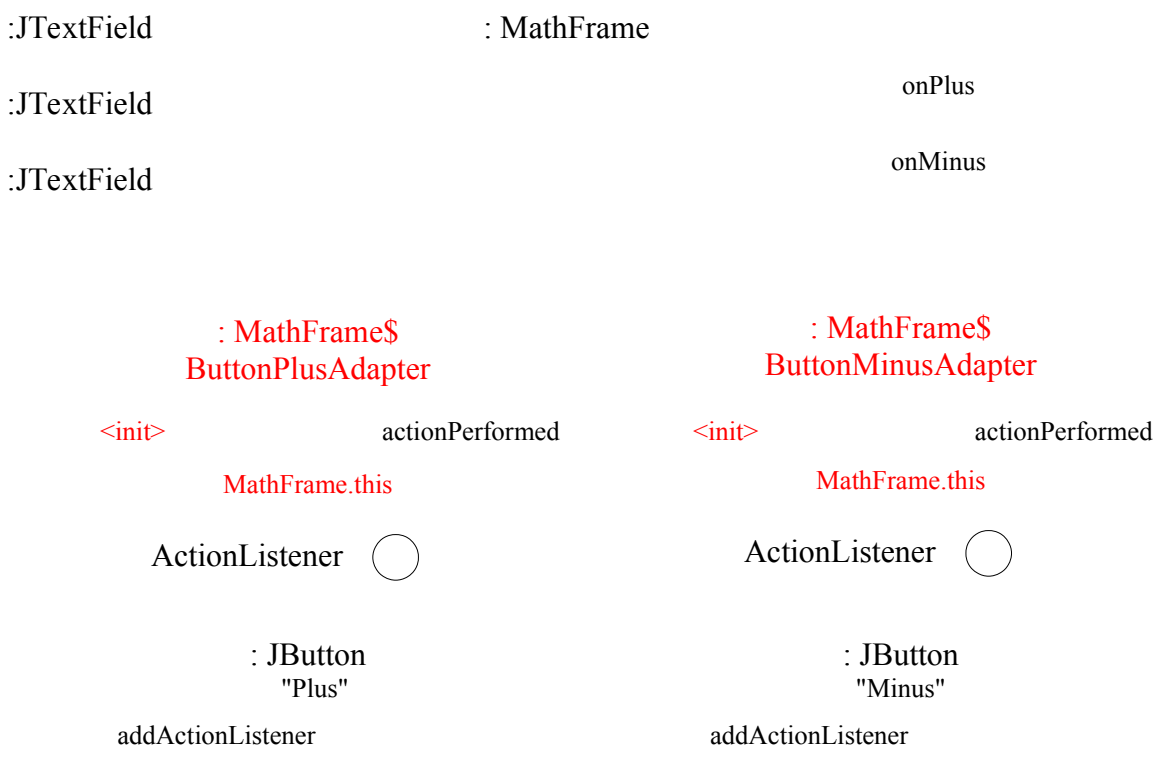
2.3 Member Classes

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

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

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

Das Objektdiagramm hat sich kaum verändert:



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

Hier der Quellcode (dessen Umfang deutlich geschrumpft ist):

```
// ...
public class MathFrame extends JFrame {

    private class ButtonPlusAdapter implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            MathFrame.this.onPlus();
        }
    }

    private class ButtonMinusAdapter implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            MathFrame.this.onMinus();
        }
    }

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

    public MathFrame() {
        // wie gehabt ...
    }
    private void registerListeners() {
        this.buttonPlus.addActionListener(new
ButtonPlusAdapter());
        this.buttonMinus.addActionListener(new
ButtonMinusAdapter());
    }
    private void onPlus() {
        // wie gehabt ...
    }
    private void onMinus() {
        // wie gehabt ...
    }
}
```

Die oben definierten Adapter-Klassen werden als Member-Klassen bezeichnet – weil sie auf derselben Ebene definiert sind wie die Member (Felder und Methoden) der umschließenden Klasse.

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

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

Der Compiler generiert aus den obigen Memberklassen die folgenden `class`-Dateien:

```
MathFrame$ButtonPlusAdapter
MathFrame$ButtonMinusAdapter
```

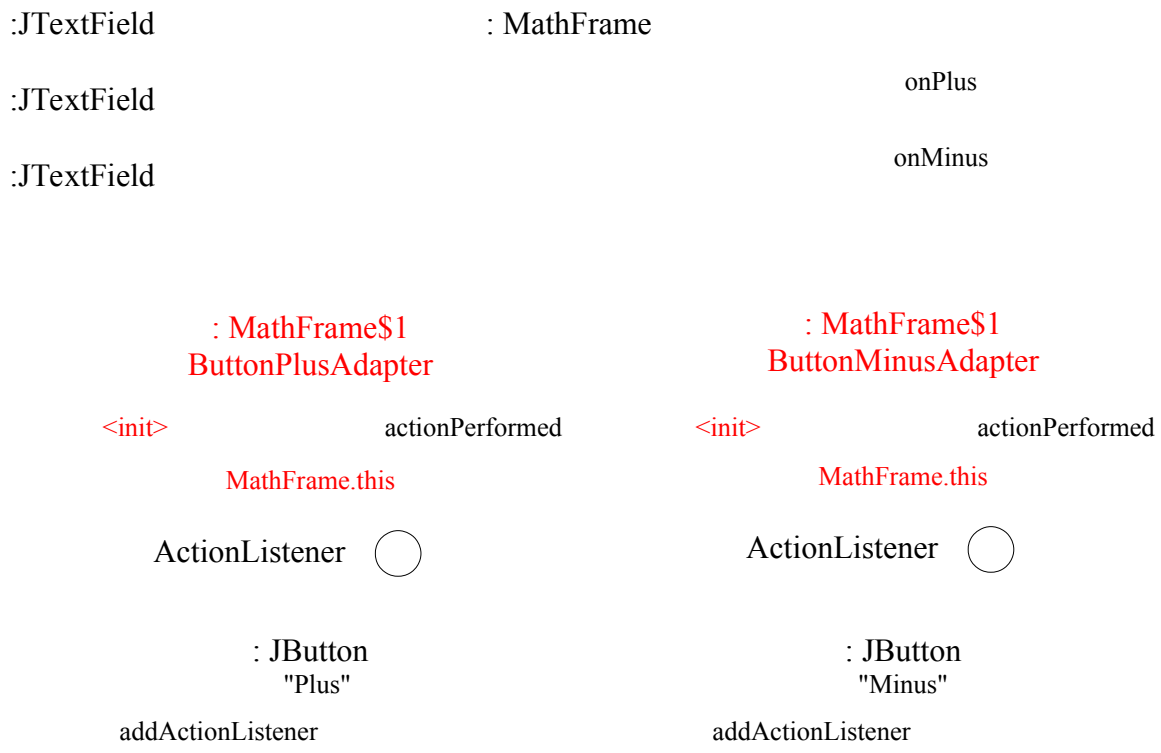
Wir könnten diese beiden Klassen per Reflection analysieren – und würden genau diejenigen Features wiederfinden, die wir auch bei den globalen Adapter-Klassen finden würden (technisch ist also alles beim alten geblieben).

2.4 Local Classes

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

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

Das Objektdiagramm ändert sich nur unwesentlich:



Der Compiler generiert nun folgende Klassen:

```
MathFrame$1ButtonPlusAdapter
MathFrame$1ButtonMinusAdapter
```

Hier der Quellcode:

```
// ...
public class MathFrame extends JFrame {

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

    public MathFrame() {
        // wie gehabt ...
    }

    private void registerListeners() {
        class ButtonPlusAdapter implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                MathFrame.this.onPlus();
            }
        }
        this.buttonPlus.addActionListener(new
ButtonPlusAdapter());

        class ButtonMinusAdapter implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                MathFrame.this.onMinus();
            }
        }
        this.buttonMinus.addActionListener(new
ButtonMinusAdapter());
    }
    private void onPlus() {
        // wie gehabt ...
    }
    private void onMinus() {
        // wie gehabt ...
    }
}
```

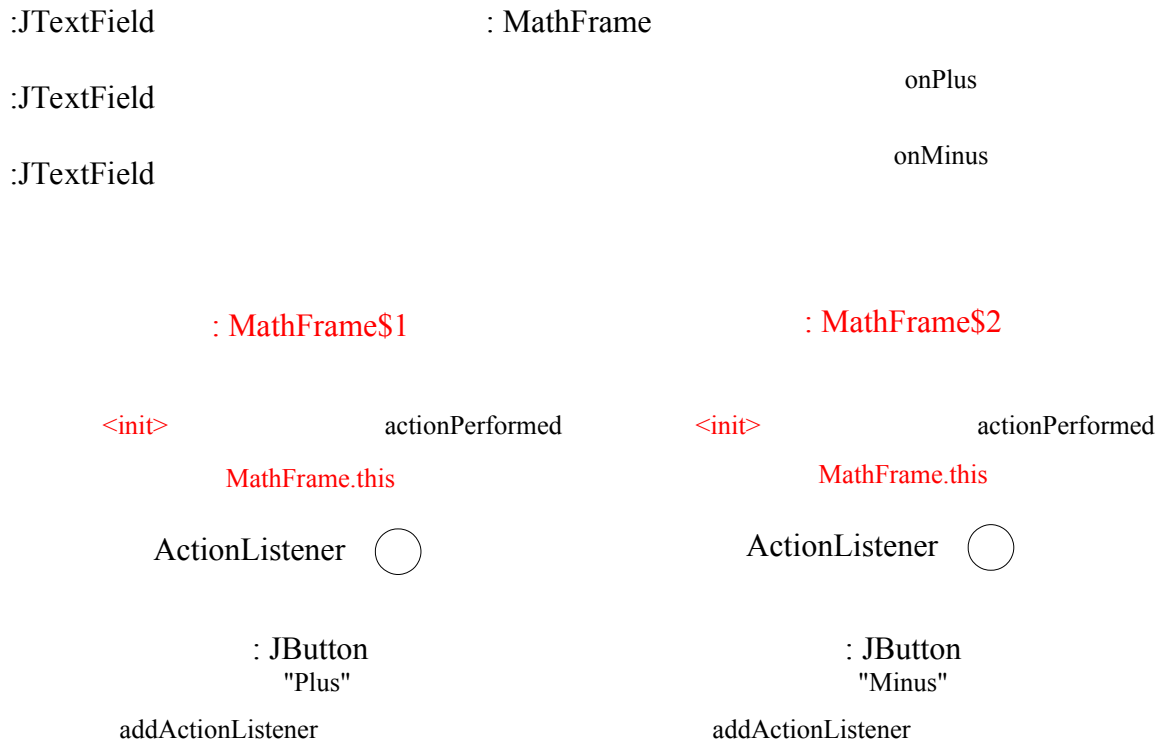
Natürlich kann eine lokale Klasse keinen Sichtbarkeits-Modifizierer besitzen – sie ist einfach nur lokal.

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

2.5 Anonymous Classes

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

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



Der Quellcode:

```
// ...
public class MathFrame extends JFrame {

    // wie gehabt ...
```

```
    private void registerListeners() {
        this.buttonPlus.addActionListener(new ActionListener {
            public void actionPerformed(ActionEvent e) {
                MathFrame.this.onPlus();
            }
        });
        this.buttonMinus.addActionListener(new ActionListener {
            public void actionPerformed(ActionEvent e) {
                MathFrame.this.onMinus();
            }
        });
    }

    // wie gehabt ...
}
```

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

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

2.6 Aufgaben

Typen von Klassen - 1

Eine Applikation besteht aus zwei Klassen:

```
package ex1;

public class CountingRunnable implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                Thread.sleep(1000);
                System.out.print(i + " ");
            }
            catch (InterruptedException e) {
            }
        }
    }
}
```

```
package ex1;

public class Application {

    public static void main(String[] args) {
        Runnable r = new CountingRunnable();
        Thread t = new Thread(r);
        t.start();
        try {
            t.join();
        }
        catch (InterruptedException e) {
        }
    }
}
```

Vereinfachen Sie die Anwendung, indem Sie die `CountingRunnable`-Klasse als anonyme Klasse implementieren.

Können Sie in der anonymen Klasse das "äußere" Objekt ansprechen? Wenn nicht: warum nicht?

Typen von Klassen - 2

Bauen Sie die Klasse derart um, dass Sie in der anonymen Klasse ein äußeres Objekt ansprechen können.

Typen von Klassen - 3

Benutzen Sie statt eines `Runnable`s eine Ableitung von `Thread`. Implementieren Sie diese Ableitung wieder als anonyme Klasse.

Typen von Klassen - 4

In lokalen und anonymen Klassen können Sie auch lokale Variablen resp. Parameter derjenigen Methode ansprechen, in welcher die innere Klasse jeweils definiert ist. Zeigen Sie dies, indem Sie die Anzahl der Schleifendurchläufe aus einer Variablen der umschließenden Methode auslesen.

3 Lambdas

Das Einstiegs-Kapitel zum Thema "Typen von Klassen" endete mit anonymen Klassen. Java 8 kennt nun zusätzlich auch anonyme Funktionen: "Lambdas". Anonyme Klassen sind Klassen, die namenlos sind (sie besitzen nur eine Implementierung). Anonyme Funktionen sind namenlose Funktionen (Funktionen, die ebenfalls nur eine Implementierung besitzen).

Die Oracle-Dokumentation:

Lambda Expressions, a new language feature, has been introduced in this release. They enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.

Im Folgenden wird die Verwendung solcher Lambdas vorgestellt - anhand einiger hoffentlich plausibler und einschlägiger Beispiele.

Das Kapitel verfolgt zunächst das im letzten Kapitel dargestellte "Taschenrechner"-Beispiel noch ein wenig weiter. Dann wechselt aber das Thema: wir stellen Lambdas im Zusammenhang mit Multithreading und im Zusammenhang mit der Verarbeitung von zeichenbasierten Eingaben und einigen weiteren Beispielen vor.

Abschließend wird der Einsatz von Lambdas im Kontext von Dynamic-Proxies gezeigt.

3.1 ActionListener

Als Einstieg sei hier noch einmal die im letzten Kapitel vorgestellt Taschenrechner-Variante mit anonymen Klassen vorgestellt:

```
public class MathFrame extends JFrame {
    // ...
    private void registerListeners() {
        this.buttonPlus.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                MathFrame.this.onPlus();
            }
        });
        this.buttonMinus.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                MathFrame.this.onMinus();
            }
        });
    }
    private void onPlus() {
        // wie gehabt ...
    }
    private void onMinus() {
        // wie gehabt ...
    }
}
```

Das einzige, worin sich die beiden anonymen Klassen unterscheiden, ist deren Implementierung: in der `actionPerformed`-Methode der ersten Klasse wird `onPlus` aufgerufen, in der zweiten Implementierung `onMinus`. Das gesamte "Drumherum" ist aber in beiden Fällen exakt dasselbe:

```
        ... .addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                ...
            }
        });
```

Dann wäre es schön, auf dieses "Drumherum" auch komplett verzichten zu können. Und genau dazu sind Lambdas gemacht. Hier das erste Lambda-Beispiel:

```
    private void registerListeners() {
        this.buttonPlus.addActionListener(
            (ActionEvent e) -> { this.onPlus(); }
        );
        this.buttonMinus.addActionListener(
```

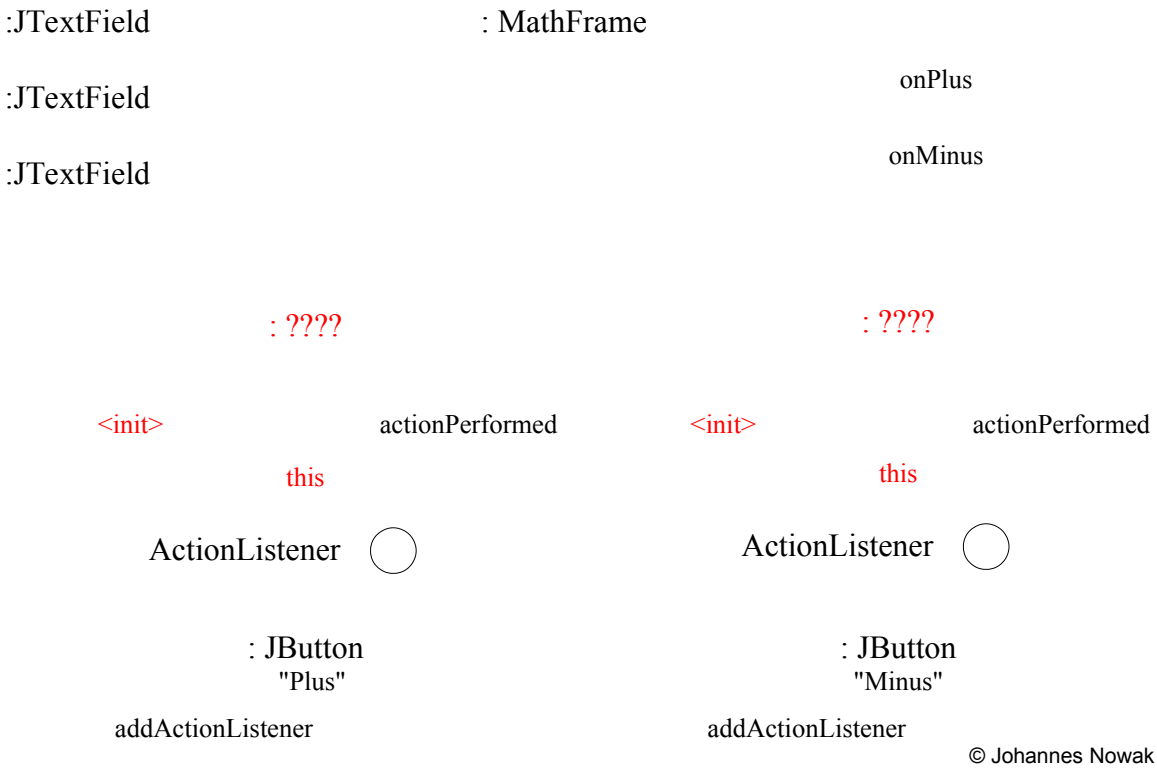
```
            (ActionEvent e) -> { this.onMinus(); }
        );
    }
```

Die Methode `addActionListener` verlangt natürlich weiterhin einen `ActionListener`. Ein `ActionListener` ist ein Objekt, auf welches exakt eine einzige Methode aufgerufen werden kann: `actionPerformed`. Diese ist `void` und hat ein Argument des Typs `ActionEvent`. Ein Interface, welches nur eine einzige anonyme Methode spezifiziert, wird als "funktionales Interfaces" bezeichnet (oder als "SAM-Interface": Single Abstract Method).

Statt nun aber an `addActionListener` eine Instanz einer anonymen, das Interface implementierenden Klasse zu übergeben, wird ihr nur eine "Funktion" übergeben: eine Funktion, deren Schnittstelle identisch ist mit derjenigen von `actionPerformed`. Diese Funktion hat keinen Namen; wir spezifizieren nur die Parameterliste der Funktion: `(ActionEvent e)`. Dieser Liste folgt (getrennt über den neuen `->` Operator) die Implementierung der Funktion. Die Implementierung ist im obigen Beispiel ein Block (also eine Folge von Anweisungen, welche mittels `{ }` zusammengefasst sind).

Tatsächlich wird eine "Funktion" jeweils als Methode in einem Objekt verpackt, dessen Referenz dann jeweils übergeben wird (dies wird im nächsten Kapitel genauer untersucht werden...).

Ein Objektdiagramm:



Die "Funktion" wird hier als gestricheltes Objekt dargestellt. Man beachte, dass die Referenz auf das äußere Objekt nun als `this` dargestellt wird. Tatsächlich kann das "Funktions-Objekt" als solches in der Methode dieses Objekts nicht angesprochen werden; `this` verweist immer auf das äußere Objekt (auch dies wird später noch sehr genau analysiert werden). Dies ist ein erster wichtiger Unterschied zu anonymen Klassen.

Allgemein gilt:

Sei `I` ein funktionales Interface, welches eine `f`-Methode spezifiziert:

```
public interface I {
    public abstract R f(P0 p1, P1 p2, ... PN pn);
}
```

Dann kann überall dort, wo ein Objekt einer Klasse verlangt wird, welche das Interface `I` implementiert, auch eine namenlose "Funktion" angegeben werden – sofern diese `R` zurückliefert und eine Parameterliste der Form `(P0 p0, P1 p1, ... PN pn)` besitzt.

Eine "Funktion", welche ein Interface `I` implementiert, wird im folgenden der Einfachheit halber als `I`-Funktion bezeichnet. Die beiden Funktionen, die im obigen Beispiel an `addActionListener` übergeben werden, bezeichnen wir also als `ActionListener`-Funktionen.

Der Typ eines Lambda-Ausdrucks wird vom Compiler anhand desjenigen Typs ermittelt, an welchen der Ausdruck zugewiesen wird ("target typing"). Hier es der Typ des von `addActionListener` verlangten Parameters – also: `ActionListner`.

Was das obige Beispiel angeht, können wir uns noch etwas knapper ausdrücken:

Da die beiden an `addActionListener` übergebenen Funktionen jeweils nur eine einzige Anweisung besitzen (den Aufruf von `onPlus` resp. `onMinus`), braucht diese Anweisung nicht einmal per `{ }` geklammert zu werden. Dann muss allerdings auch das die Anweisung terminierende Semikolon entfallen. Hier die kürzere Variante:

```
    private void registerListeners() {
        this.buttonPlus.addActionListener((ActionEvent e) ->
this.onPlus());
        this.buttonMinus.addActionListener((ActionEvent e) ->
this.onMinus());
    }
```

Und es geht noch kürzer:

Die Parameterlisten der anonymen Funktionen haben jeweils die Form `(ActionEvent e)`. Der Compiler weiß nun aber doch, dass an `addActionListener` ein Objekt einer Klasse übergeben werden muss, deren `actionPerformed`-Methode einen `ActionEvent`-Parameter besitzt. Wenn nun statt einer Instanz einer solchen Klasse einfach eine Funktion übergeben werden darf, muss der Typ des Parameters auch nicht mehr explizit spezifiziert werden – der Compiler kann diesen Typ automatisch berechnen (Typ-Inferenz). Daher ist auch die folgende noch kürzere Variante erlaubt:

```
    private void registerListeners() {
        this.buttonPlus.addActionListener((e) -> this.onPlus());
        this.buttonMinus.addActionListener((e) ->
this.onMinus());
    }
```

Und hier schließlich die allerkürzeste Variante: Da die Parameterliste nur ein einziges Element hat, kann sogar noch die Klammerung per `()` weggelassen werden:

```
    private void registerListeners() {
        this.buttonPlus.addActionListener(e -> this.onPlus());
        this.buttonMinus.addActionListener(e -> this.onMinus());
    }
```

Natürlich hätte man den `e`-Parameter auch an `onPlus` resp `onMinus` weiterreichen können (vorausgesetzt, diese beiden Methoden würden einen solchen `ActionEvent`-Parameter verlangen):

```
    private void registerListeners() {
        this.buttonPlus.addActionListener(e -> this.onPlus(e));
        this.buttonMinus.addActionListener(e -> this.onMinus(e));
    }
```

Das Hinzufügen eines expliziten `ActionListeners` verlangte bei der Lösung mit anonymen Klassen fünf Zeilen – bei der Benutzung von Lambdas (also bei der Übergabe einer `ActionListeners`-Funktion) reicht jeweils eine einzige Zeile. Der Code wird wesentlich knapper – ohne dabei aber an Verständlichkeit einzubüßen (im Gegenteil).

Wenn eine anonyme Funktion als Parameter eines Methodenaufrufs übergeben werden kann, muss sie natürlich auch einer Variablen zugewiesen werden können:

```
        ActionListener l = (e -> out.println("Hello"));
        l.actionPerformed(new ActionEvent(this, 0, ""));
```

Der Aufruf von `actionPerformed` führt dazu, dass `"Hello"` ausgegeben wird.

3.2 Operators

Die `onPlus`- und die `onMinus`-Methode unseres Taschenrechners sehen annähernd gleich aus:

```
    private void onPlus() {
        try {
            int x = Integer.parseInt(this.textFieldX.getText());
            int y = Integer.parseInt(this.textFieldY.getText());
            int result = x + y;
            this.textFieldResult.setText(String.valueOf(result));
        }
        catch(NumberFormatException e) {
            this.textFieldResult.setText("Illegal input");
        }
    }

    private void onMinus() {
        try {
            int x = Integer.parseInt(this.textFieldX.getText());
            int y = Integer.parseInt(this.textFieldY.getText());
            int result = x - y;
            this.textFieldResult.setText(String.valueOf(result));
        }
        catch(NumberFormatException e) {
            this.textFieldResult.setText("Illegal input");
        }
    }
```

Wie kann die Redundanz, die hier offensichtlich vorliegt, vermieden werden?

Wir definieren ein Interface:

```
public interface BinaryOperator {
    public abstract int apply(int x, int y);
}
```

Ein binärer Operator kann auf zwei `int`-Werte angewendet werden – die Anwendung des Operators (`apply`) resultiert in einem neuen `int`-Wert.

`onPlus` und `onMinus` können dann durch eine einzige Methode ersetzt werden – einer Methode, welcher ein `BinaryOperator` übergeben wird:

```
    private void onCalc(BinaryOperator op) {
        try {
```

```
            int x = Integer.parseInt(this.textFieldX.getText());
            int y = Integer.parseInt(this.textFieldY.getText());
            int result = op.apply(x, y);
            this.textFieldResult.setText(String.valueOf(result));
        }
        catch(NumberFormatException e) {
            this.textFieldResult.setText("Illegal input");
        }
    }
```

Das Resultat wird hier mittels der Anwendung (`apply`) des der Methode übergebenen `BinaryOperators` berechnet.

Die `registerListeners`-Methode registriert bei den beiden Buttons dann jeweils eine `ActionListener`-Funktion, innerhalb derer die `onCalc`-Methode mit jeweils einem Objekt einer von `BinaryOperator` abgeleiteten anonymen Klasse aufgerufen wird:

```
    private void registerListeners() {
        this.buttonPlus.addActionListener(e -> this.onCalc(
            new BinaryOperator() {
                public int apply(int x, int y) {
                    return x + y;
                }
            }
        ));
        this.buttonPlus.addActionListener(e -> this.onCalc(
            new BinaryOperator() {
                public int apply(int x, int y) {
                    return x - y;
                }
            }
        ));
    }
```

Da `BinaryOperator` ein funktionales Interface ist, können diese Objekte nun wiederum ersetzt werden durch anonyme Funktionen:

```
    private void registerListeners() {
        this.buttonPlus.addActionListener(e -> this.onCalc((x, y)
-> x + y));
        this.buttonMinus.addActionListener(e -> this.onCalc((x,
y) -> x - y));
    }
```

An `addActionListener` wird nun also jeweis eine `ActionListener`-Funktion übergeben, deren Implementierung `onCalc` aufruft – und dieser wird wiederum eine

weitere anonyme Funktion übegeben – eine `BinaryOperator`-Funktion. Die ganze Konstruktion bleibt trotzdem verständlich und lesbar...

3.3 Operators-Map

Eine `Map` kann Schlüssel auf Objekte abbilden – und natürlich auch auf Objekte einer von einem Interface abgeleiteten anonymen Klasse. Dann muss eine `Map` auch Schlüssel auf anonyme Funktionen abbilden können:

Angenommen, wir definieren folgende `Map`:

```
    private final Map<String, BinaryOperator> operators
        = new LinkedHashMap<>();
```

Dann können wird in diese `Map` vier anonyme `BinaryOperator`-Funktionen eintragen:

```
    {
        operators.put("Plus",  (x, y) -> x + y);
        operators.put("Minus", (x, y) -> x - y);
        operators.put("Times", (x, y) -> x * y);
        operators.put("Div",   (x, y) -> x / y);
    }
```

Der Taschenrechner muss nunmehr die Textfelder als Instanzvariablen definieren. Die Buttons werden "dynamisch" erzeugt – aufgrund der in der `Map` enthaltenen Einträge:

```
    private final JTextField textFieldX = new JTextField(10);
    private final JTextField textFieldY = new JTextField(10);
    private final JTextField textFieldResult = new
JTextField(10);

    public MathFrame() {
        this.add(this.textFieldX);
        this.add(this.textFieldY);
        this.addButtons();
        this.add(this.textFieldResult);
        // ...
    }
    private void addButtons() {
        for (Map.Entry<String, BinaryOperator> entry :
operators.entrySet()) {
            JButton button = new JButton(entry.getKey());
            button.addActionListener(e ->
onCalc(entry.getValue()));
            this.add(button);
        }
    }
    private void onCalc(BinaryOperator op) {
```

```
        // wie gehabt ...
    }
}
```

3.4 Das Standard-Interface BinaryOperator

In den letzten beiden Abschnitten wurde das funktionale Interface `BinaryOperator` benutzt:

```
public interface BinaryOperator {
    public abstract int apply(int x, int y);
}
```

Java 8 enthält ein Paket `java.util.function`, in welchem ein ganz ähnliches Interface bereits definiert ist – allerdings eines, welches einem Typ parametrisiert ist:

```
public interface BinaryOpreator<T> {
    T apply(T t0, T t1);
}
```

Statt unseres eigenen Interfaces können wird dann natürlich auch dieses Standard-Interface nutzten – indem als Typ-Parameter `Integer` ersetzt werden:

```
import java.util.function.BinaryOperator;

public class MathFrame extends JFrame {
    private final Map<String, BinaryOperator<Integer >>
        operators = new LinkedHashMap<>();
    {
        operators.put("Plus",  (x, y) -> x + y);
        operators.put("Minus", (x, y) -> x - y);
        operators.put("Times", (x, y) -> x * y);
        operators.put("Div",   (x, y) -> x / y);
    }
    // ...
    private void addButtons() {
        for (Map.Entry<String, BinaryOperator<Integer>> entry :
                operators.entrySet()) {
            JButton button = new JButton(entry.getKey());
            button.addActionListener(e ->
onCalc(entry.getValue()));
            this.add(button);
        }
    }
    private void onCalc(BinaryOperator<Integer> op) {
        // ...
            int result = op.apply(x, y);
        // ...
    }
```

```
}
```

3.5 Enums

Man könnte auch eine `enum`-Klasse mit vier Operatoren definieren:

```
public enum BinaryOperators implements BinaryOperator<Integer > {

    PLUS("Plus",   (v1, v2) -> v1 + v2),
    MINUS("Minus", (v1, v2) -> v1 - v2),
    TIMES("Times", (v1, v2) -> v1 * v2),
    DIV("Div",     (v1, v2) -> v1 / v2)
    ;

    private final String displayName;
    private final BinaryOperator<Integer > op;

    private BinaryOperators(
            String displayName, BinaryOperator<Integer > op) {
        this.displayName = displayName;
        this.op = op;
    }
    public Integer apply(Integer v1, Integer v2) {
        return this.op.apply(v1, v2);
    }
    public String displayName() {
        return this.displayName;
    }
}
```

In der `MathFrame`-Klasse könnte diese `enum`-Klasse wie folgt genutzt werden:

```
    private void addButtons() {
        for (BinaryOperators op : BinaryOperators.values()) {
            JButton button = new JButton(op.displayName());
            button.addActionListener(e -> onCalc(op));
            this.add(button);
        }
    }
```

3.6 Multithreading

Wir vergessen den Taschenrechner und betreiben ein wenig Multithreading. Wir wollen einen Thread starten, dessen Arbeit darin besteht, eine Sekunde lang zu schlafen. Der Hauptthread soll auf die Terminierung dieses abgespaltenen Threads starten:

```
        Thread t = new Thread(new Runnable() {
            public void run() {
                out.println("Thread starts");
                try {
                    Thread.sleep(1000);
                }
                catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                out.println("Thread terminates");
            }
        });
        t.start();
        try {
            t.join();
        }
        catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        out.println("Thread terminated");
```

Die Lösung ist korrekt. Aber auch einigermaßen "komplex" – und zwar deshalb, weil sie zwei `try-catch`-Blöcke enthält. Die `sleep`- und die `join`-Methoden können jeweils `InterruptedExceptions` werfen – und diese müssen nun einmal abgefangen werden (zumindest in der `run`-Methode können sie nicht per `throws` zum Weiterwerfen deklariert werden).

Man müsste diese checked-Exceptions zu `RuntimeExceptions` "umbiegen" können.

Sei folgendes Interface gegeben:

```
public interface XRunnable {
    public abstract void run() throws Throwable;
    static void xrun(XRunnable runnable) {
        try {
            runnable.run();
        }
        catch(Throwable e) {
            throw new RuntimeException(e);
```

```
        }
    }
}
```

Das Interface ist ein funktionales Interface. Es definiert nur eine einzige abstrakte Methode: `run`. Diese Methode ist parameterlos und `void`.

Zusätzlich definiert es eine weitere statische Methode (in Java 8 kann ein Interface nicht nur statische Attribute, sondern auch statische Methoden implementieren – hierzu später noch mehr).

Der statischen `xrun`-Methode wird ein `XRunnable`-Objekt übergeben (oder: eine `XRunnable`-Funktion – also eine Funktion, welche parameterlos und `void` ist). Innerhalb eines `try`-Blocks wird dann die `run`-Methode auf das übergebene `XRunnable`-Objekt bzw. auf die übergebene `XRunnable`-Funktion aufgerufen. Eine hier möglicherweise geworfene `Exception` wird im `catch`-Block aufgefanen und dort in einer `RuntimeException` eingewickelt, welche dann ihrerseits geworfen wird.

Dann kann die obige Anwendung wesentlich knapper formuliert werden:

```
        Thread t = new Thread(() -> {
            out.println("Thread starts");
            XRunnable.xrun(() -> Thread.sleep(1000));
            out.println("Thread terminates");
        });
        t.start();
        XRunnable.xrun(() -> t.join());
        out.println("Thread terminated");
```

Man beachte den zweifachen Aufruf von `XRunnable.xrun`.

Beim zweiten Aufruf:

```
        XRunnable.xrun(() -> t.join());
```

wird innerhalb der an `xrun` übergebenen `XRunnable`-Funktion auf die Variable `t` zugegriffen. `t` ist eine lokale Variable der "Umgebung". Hätte man statt der `XRunnable`-Funktion eine Objekt einer anonymen Klasse übergeben, hätte `t` als `final` deklariert sein müssen. Diese Einschränkung ist nun bei Funktionen aufgehoben worden – die Variable muss nurmehr "effektiv final" sein – ihr darf nur ein einziges Mal ein Wert zugewiesen worden sein.

Das hier verwendete `XRunnable`-Interfaces ist im `shared`-Projekt definiert (als statisches innere Interface der Klasse `util.Util`). Sofern folgender statischer Import verwendet wird:

```
import static util.Util.XRunnable.xrun;
```

kann die Methode natürlich auch einfach als `xrun` angesprochen werden.

3.7 CharacterProcessor

Als weiteres Beispiel bauen wir einen `CharacterProcessor`.

Ein `CharacterProcessor` bekommt eine Folge von Zeichen, welche er zeichenweise lesen wird. Die Verarbeitung jedes Zeichens soll an einen `Handler` delegiert werden.

Das `Handler`-Interface sei wie folgt definiert:

```
public interface Handler<T> {
    public abstract void handle(T value);
}
```

`Handler` ist ein funktionales Interface.

`CharacterProcessor` enthält eine statische `process`-Methode, welcher ein `Reader` und ein `Handler<Character>` übergeben wird:

```
class CharacterProcessor {
    public static void process(Reader reader, Handler<Character>
handler) {
        try(Reader r = reader) {
            int ch = r.read();
            while (ch != -1) {
                handler.handle((char) ch);
                ch = r.read();
            }
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Als `Reader` könnte der Methode z.B. ein `StringReader` oder ein `InputStreamReader` übergeben werden (beide Klasse sind von `Reader` abgeleitet).

Die `process`-Methode liest alle Zeichen der Eingabe. Jedes Zeichen wird der `handle`-Methode des `Handler`s zur Bearbeitung übergeben. An Ende wird (per Auto-Close) der `Reader` geschlossen. Eine mögliche `Exception`, welche die auf den `Reader` aufgerufene `read`-Methode werfen kann, wird als `RuntimeException` weitergeworfen.

Hier eine mögliche Anwendung:

```
        Reader reader = new StringReader("Hello");
        CharacterProcessor.process(reader, new
Handler<Character>() {
            public void handle(Character ch) {
                out.println(ch);
            }
        });
```

Und hier deren Ausgaben:

```
H
e
l
l
o
```

In der obigen Anwendung wurde an `process` ein `Handler`-Objekt übergeben (ein Objekt einer von `Handler` abgeleiteten anonymen Klasse). Statt eines solchen Objekts können wir natürlich auch eine anonyme Funktion übergeben – eine Funktion, welche mit `char` parametrisiert ist und `void` liefert:

```
        Reader reader = new StringReader("Hello");
        CharacterProcessor.process(reader, ch ->
out.println(ch) );
```

Ein andere Anwendung könnte die an das `Handler`–Objekt resp. an die `Handler`-Funktion übergeben Zeichen in einer Liste speichern – in einer Liste, welche dann nach Beendigung von `process` ausgegeben wird:

```
        Reader reader = new StringReader("Hello ");
        List<Character> chars = new ArrayList<>();
        CharacterProcessor.process(reader, ch -> chars.add(ch));
        for (Character ch : chars)
            out.println(ch);
```

Wäre an `process` ein `Handler`-Objekt einer anonymen Klasse übergeben worden, hätte die `chars`-Variable `final` sein müssen. Wird eine anonyme `Handler`-Funktion übergeben, ist das Schlüsselwort `final` nicht erforderlich. Die Variable muss nur "effektiv final" sein.

Wir möchten nun die Anzahl der in der Eingabe enthaltenen Zeichen zählen. Folgende Lösung wird vom Compiler als illegal verworfen:

```
        Reader reader = new StringReader("Hello");
        int n = 0;
```

```
        CharacterProcessor.process(reader, ch -> n++);  //
illegal!
        out.println(n);
```

Auf `n` (also auf eine Variable der umschließenden Methode) darf nur lesend zugegriffen werden (hier verhält es sich also bei anonymen Funktionen genauso wie bei Objekten anonymer Klassen).

Wir definieren eine Klasse `Box`:

```
package appl;

public class Box<T> {
    public T value;
    public Box(T start) {
        this.value = start;
    }
    public String toString() {
        return this.value.toString();
    }
}
```

Und ein `Box`-Objekt, welches effektiv final ist:

```
        Reader reader = new StringReader("Hello");
        Box<Integer> n = new Box<>(0);
        CharacterProcessor.process(reader, ch -> n.value++);
        out.println(n);
```

Dann darf die anonyme Funktion selbstverständlich den Inhalt der `Box` inkrementieren.

Wir möchten die Anzahl der in der Eingabe enthaltenen "schwarzen Zeichen" ermitteln:

```
        Reader reader = new StringReader("Hello");
        Box<Integer> n = new Box<>(0);
        CharacterProcessor.process(reader, ch -> {
            if (! Character.isWhitespace(ch)) n.value++;
        });
        out.println(n);
```

Die an `process` übergebene Funktion ist nun relativ "komplex": sie enthält eine `if`-Anweisung. Wie kann diese Komplexität reduziert werden?

Wir definieren ein funktionales Interface `Tester`:

```
public interface Tester<T> {
```

```
    public abstract boolean test(T value);
}
```

Dann kann die Klasse `CharacterProcessor` wie folgt refaktoriert werden:

```
public class CharacterProcessor {
    public static void process(Reader reader,
            Tester<Character> tester, Handler<Character> handler)
{
        try(Reader r = reader) {
            int ch = r.read();
            while (ch != -1) {
                if (tester.test((char)ch))
                    handler.handle((char) ch);
                ch = r.read();
            }
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    public static void process(Reader reader, Handler<Character>
handler) {
        process(reader, (ch) -> true, handler);
    }
}
```

Der ersten der beiden `process`-Methoden wird neben einem `Handler` nun ein weiteres Objekt (oder eine weitere Funktion) übergeben: ein `Tester`. Und nur dann, wenn das jeweilige Zeichen den Test (was das auch immer sei) bestanden hat, wird es dem `Handler` übergeben.

Die zweite `process`-Methode wird auf die erste zurückgeführt: indem ein `Tester` übergeben wird, der mit jedem Zeichen einverstanden ist: `(ch) -> true`.

Hier ein Aufruf der ersten dieses beiden `process`-Methoden:

```
        Reader reader = new StringReader("Hello ");
        Box<Integer> n = new Box<>(0);
        CharacterProcessor.process(reader,
            ch -> ! Character.isWhitespace(ch),
            ch -> n.value++);
        out.println(n);
```

Die `if`-"Komplexität" ist also von der Anwendung in die `CharacterProcessor`-Klasse verschoben worden.

Das von uns definierte Interface existiert natürlich genau wie `BinaryOperator` bereits wieder in der Standardbibliothek – allerdings unter einem anderen Namen:

```
public interface Predicate<T> {
    public abstract boolean test(T value);
}
```

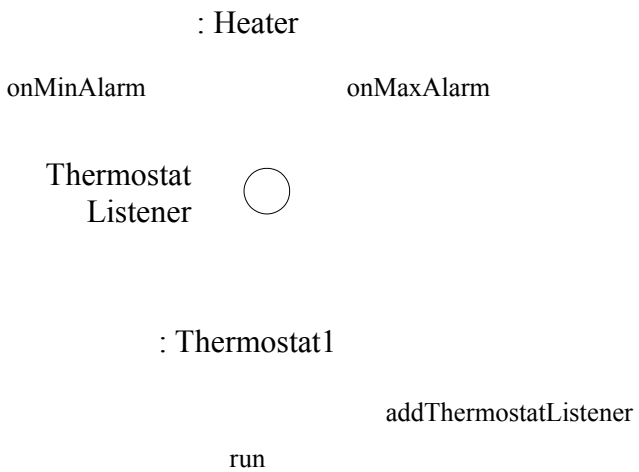
3.8 Thermostat / Heater

Das folgende Beispiel demonstriert, wie aus einem nicht-funktionalen Interface funktionale Interfaces gebaut werden können.

Eine Heizung (oder ein Kühlschrank...) soll mittels eines Thermostats gesteuert werden können.

Im folgenden werden zwei Lösungen präsentiert – die erste benutzt ein nicht-funktionales Interface, die zweite ein funktionales Interface (und ist damit für die Verwendung von Lambdas prädestiniert):

Zunächst zur ersten Lösung. Ein kleines Objektdiagramm:



Das Interface `ThermostatListener` definiert die beiden Methoden `onMinAlarm` und `onMaxAlarm`:

```
public interface ThermostatListener {
    public abstract void onMinAlarm();
    public abstract void onMaxAlarm();
}
```

Die Klasse `Thermostat1`:

```
public class Thermostat1 {

    private final List<ThermostatListener> listeners = new
ArrayList<>();
```

```
    public void addThermostatListener(ThermostatListener
listener) {
        this.listeners.add(listener);
    }

    public void run() {
        // too cold...
        for (ThermostatListener listener : this.listeners)
            listener.onMinAlarm();
        // too hot...
        for (ThermostatListener listener : this.listeners)
            listener.onMaxAlarm();
    }
}
```

Dann könnte eine Heizung wie folgt definiert werden:

```
public class Heater1 implements ThermostatListener {
    public void onMinAlarm() {
        out.println("Heater on");
    }
    public void onMaxAlarm() {
        out.println("Heater off");
    }
}
```

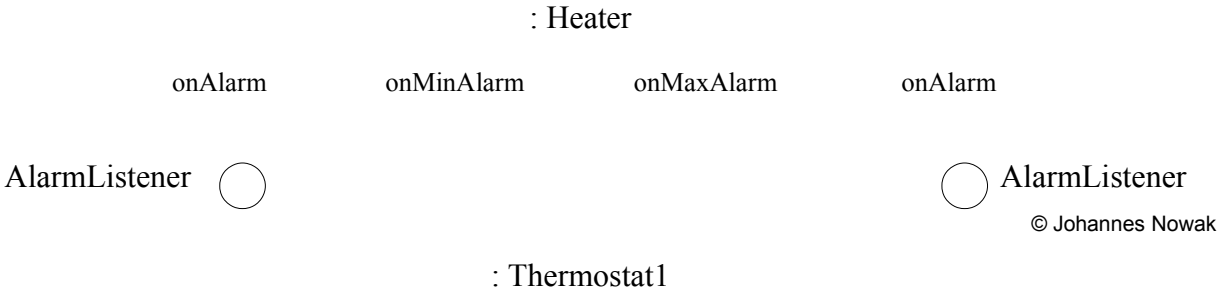
`Heater1` implementiert `ThermostatListener`; daher kann ein `Heater1` an die `Thermostat1`-Methode `addThermostatListener` übergeben werden:

```
        Thermostat1 thermostat = new Thermostat1();
        Heater1 heater = new Heater1();
        thermostat.addThermostatListener(heater);
        thermostat.run();
```

Die Ausgaben:

```
Heater on
Heater off
```

Nun zur zweiten Lösung.



Statt des nicht-funktionalen Interfaces `ThermostatListener` könnte man nun ein funktionales Interface definieren:

```
public interface AlarmListener {
    public abstract void onAlarm();
}
```

Die Thermostat-Klasse (`Thermostat2`) hat dann zwei Listener-Listen:

```
public class Thermostat2 {

    private final List<AlarmListener> maxAlarmListeners = new
ArrayList<>();
    private final List<AlarmListener> minAlarmListeners = new
ArrayList<>();

    public void addMaxAlarmListener(AlarmListener listener) {
        this.maxAlarmListeners.add(listener);
    }
    public void addMinAlarmListener(AlarmListener listener) {
        this.minAlarmListeners.add(listener);
    }

    public void run() {
        // too cold...
        for (AlarmListener listener : this.minAlarmListeners)
            listener.onAlarm();
        // too hot...
        for (AlarmListener listener : this.maxAlarmListeners)
            listener.onAlarm();
    }
}
```

Die Heizung (`Heater2`) muss dann keinerlei Interface mehr implementieren (und damit sind auch die Namen der Methoden Schall und Rauch):

```
public class Heater2 {
    public void onMinAlarm() {
```

```
        out.println("Heater on");
    }
    public void onMaxAlarm() {
        out.println("Heater off");
    }
}
```

Das Thermostat und die Heizung können nun wie folgt zusammengeschraubt werden:

```
        Thermostat2 thermostat = new Thermostat2();
        Heater2 heater = new Heater2();

        thermostat.addMaxAlarmListener(() ->
heater.onMaxAlarm());
        thermostat.addMinAlarmListener(() ->
heater.onMinAlarm());

        thermostat.run();
```

Oder, noch einfacher (dazu später mehr):

```
        Thermostat2 thermostat = new Thermostat2();
        Heater2 heater = new Heater2();

        thermostat.addMaxAlarmListener(heater::onMaxAlarm);
        thermostat.addMinAlarmListener(heater::onMinAlarm);

        thermostat.run();
```

Welche der beiden oben vorgestellten Lösungen ist semantisch ausdrucksstärker? Welche der beiden Lösungen ist "lockerer"?

3.9 Comparator

Ein letztes Beispiel – zu einem bekannten funktionalen Interface der Standard-Bibliothek: `Comparator`.

Gegeben sei die Klasse `Book` (und eine Liste von `Books`):

```
public class Book {
    public String isbn;
    public String title;
    public String author;
    public double price;

    public Book(String isbn, String title, String author, double
price) {
        // ...
    }

    @Override
    public String toString() { ... }

    public static final List<Book> list = new ArrayList<>();
    static {
        list.add(new Book("1111", "Pascal", "Wirth", 44.44));
        list.add(new Book("2222", "Modula", "Wirth", 33.33));
        list.add(new Book("3333", "Oberon", "Wirth", 22.22));
        list.add(new Book("4444", "Eiffel", "Meyer", 11.11));
    }
}
```

Die Liste soll nach den Namen der Autoren sortiert werden.

Hier die "traditionelle" Lösung (es wird aufsteigend sortiert):

```
    Collections.sort(Book.list, new Comparator<Book>() {
        public int compare(Book b1, Book b2) {
            return b1.title.compareTo(b2.title);
        }
    });
```

Die Ausgaben:

```
Book [4444, Eiffel, Meyer, 11.11]
Book [2222, Modula, Wirth, 33.33]
Book [3333, Oberon, Wirth, 22.22]
```

```
Book [1111, Pascal, Wirth, 44.44]
```

Und hier die wesentlich knappere Lösung mit einer anonymen Funktion (hier wird absteigend sortiert):

```
    Collections.sort(Book.list, (b1, b2) -> -
b1.title.compareTo(b2.title));
```

Die Ausgaben:

```
Book [1111, Pascal, Wirth, 44.44]
Book [3333, Oberon, Wirth, 22.22]
Book [2222, Modula, Wirth, 33.33]
Book [4444, Eiffel, Meyer, 11.11]
```

Welche weitere bekannten funktionalen Interfaces enthält die Standardbibliothek?

3.10 Dynamic Proxy

Mittels eines Dynamic-Proxies sollen Aufrufe von Service-Methoden geloggt werden.

Sei z.B. folgendes Service-Interface gegeben:

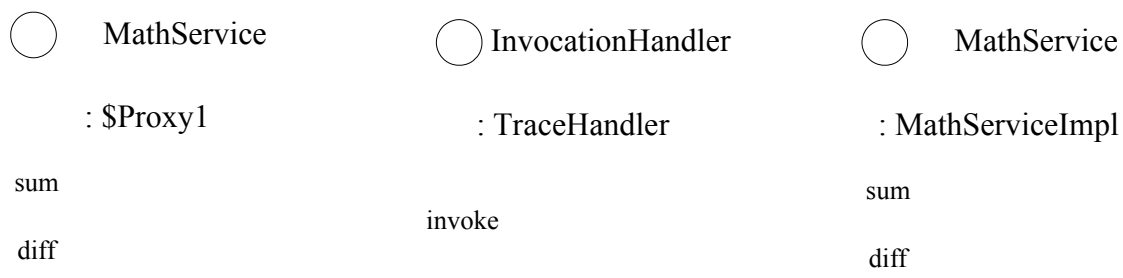
```
package appl;

public interface MathService {
    public abstract int sum(int x, int y);
    public abstract int diff(int x, int y);
}
```

Und folgende Implementierung:

```
package appl;

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;
    }
}
```

Ein Dynamic-Proxy ist eine Klasse, die ein bestimmtes Interface implementiert (z.B. `MathService`). Eine solche Proxy-Klasse wird zur Laufzeit automatisch generiert. Sie delegiert an einen `InvocationHandler` (im Paket `java.lang.reflect` definiert). `InvocationHandler` spezifiziert eine `invoke`-Methode, welcher ein `Method`-Objekt und eine Liste von Parametern übergeben wird. Eine Implementierung dieses Interfaces kann z.B. die Aufrufe tracen – und dann an das eigentliche Zielobjekt (z.B. ein Objekt vom Typ `MathServiceImpl`) weiterleiten. `InvocationHandler` könnte z.B. in von einer Klasse `TraceHandler` implementiert sein.

Ein kleines Schaubild:



Hier die Implementierung der `TraceHandler`-Klasse:

```
package util;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Arrays;

public class TraceHandler implements InvocationHandler {

    private final Object target;

    public TraceHandler(Object target) {
        this.target = target;
    }

    public Object invoke(Object proxy, Method method, Object[]
args)
                                                    throws
Throwable {
        System.out.println(">> " + method.getName() + " " +
            Arrays.toString(args));
        Object result;
        try {
            if (this.target instanceof InvocationHandler)
                result = ((InvocationHandler)
this.target).invoke(
                        proxy, method, args);
            else
                result = method.invoke(this.target, args);
            System.out.println("<< " + method.getName() + " " +
                Arrays.toString(args) + " -> " + result);
            return result;
        }
        catch(InvocationTargetException e) {
            System.out.println("<< Exception " + method.getName()
+ " " +
                Arrays.toString(args) + " -> " + e.getCause());
            throw e.getCause();
        }
        catch(Throwable e) {
```

```
            System.out.println("<< Exception " + method.getName()
+ " " +
                Arrays.toString(args) + " -> " + e);
            throw e;
        }
    }
}
```

Ohne hier auf nähere Einzelheiten einzugehen: Man muss bei der Implementierung eines `InvocationHandlers` offensichtlich auf eine Vielzahl von Einzelheiten achten.

Hier eine Anwendung:

```
    static void demoOld() {
        final MathServiceImpl mathServiceImpl = new
MathServiceImpl();
        InvocationHandler traceHandler = new
TraceHandler(mathServiceImpl);
        final MathService mathService = (MathService)
Proxy.newProxyInstance(
                ClassLoader.getSystemClassLoader(),
                new Class[] { MathService.class },
                traceHandler);

        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(40, 2));
    }
```

Und ihre Ausgaben:

```
>> sum [40, 2]
<< sum [40, 2] -> 42
42
>> diff [40, 2]
<< diff [40, 2] -> 38
38
```

Es wäre schön, wir könnten eine einzige Handler-Klasse schreiben – eine Klasse, welche die jeweils spezifischen Aktionen (hier: die Trace-Aktionen) an Objekte delegiert, deren Klassen jeweils ein bestimmtes funktionales Interface implementieren.

Hier die funktionalen Interfaces:

```
package util;
// ...
public interface Before {
    abstract public void before(Method m, Object[] args);
```

```
}
```

```
package util;
// ...
public interface AfterReturning {
    abstract public void after(Method m, Object[] args, Object
result);
}
```

```
package util;
// ...
public interface AfterThrowing {
    abstract public void after(Method m, Object[] args, Throwable
t);
}
```

Und hier die allgemeine Handler-Klasse (implementiert als anonyme Klasse der `create`-Methode der `XProxy`-Klasse):

```
package util;
// ...
public class XProxy {

    public static <T> T create(
                final Class<T> iface,
                final T target,
                Before before,
                AfterReturning after,
                AfterThrowing afterException) {
        InvocationHandler handler = new InvocationHandler() {
            public Object invoke(Object proxy, Method method,
                                 Object[] args) throws Throwable {
                if (before != null)
                    before.before(method, args);
                try {
                    Object result = method.invoke(target, args);
                    if (after != null)
                        after.after(method, args, result);
                    return result;
                }
                catch (InvocationTargetException e) {
                    if (afterException != null)
                        afterException.after(method, args,
e.getCause());
                    throw e.getCause();
                }
```

```
                catch (Throwable e) {
                    if (afterException != null)
                        afterException.after(method, args, e);
                    throw e;
                }
            }
        };

        return (T) Proxy.newProxyInstance(
            ClassLoader.getSystemClassLoader(),
            new Class<?>[] { iface },
            handler);
    }
}
```

Und hier eine Anwendung (welche dieselben Trace-Ausgaben erzeugt, die auch unter Verwendung des `TraceHandler`s erzeugt wurden):

```
    static void demoNew() {

        final MathService mathService = XProxy.create(
                MathService.class,
                new MathServiceImpl(),
                (m, a) -> System.out.println(">> " +
                    m.getName() + " "    + Arrays.toString(a)),
                (m, a, r) -> System.out.println("<< " +
                    m.getName() + " " + Arrays.toString(a) + " ->
" + r),
                null);

        System.out.println(mathService.sum(40, 2));
        System.out.println(mathService.diff(40, 2));
    }
```

3.11 Aufgaben

Lambdas – 1

Das Thermostat-Heater-Beispiel soll erweitert werden.

Die `onAlarm`-Methode des Interfaces `AlarmListener` soll einen `AlarmEvent` als Parameter übergeben bekommen:

```
public interface AlarmListener {
    public abstract void onAlarm(AlarmEvent e);
}
```

`AlarmEvent` sei wie folgt definiert:

```
public class AlarmEvent<S> {
    public final S source;
    public final String message;
    public AlarmEvent(S source, String message) { ... }
}
```

Dann muss natürlich auch die Anwendung angepasst werden...

Lambdas – 2

Gegeben sei folgende Klasse:

```
package ex2;

import java.util.Arrays;

public class Array<T> {
    @SuppressWarnings("unchecked")
    private T[] elements = (T[]) new Object[2];
    private int size;

    public void add(T element) {
        this.ensureCapcity();
        this.elements[this.size] = element;
        this.size++;
    }
    public int size() {
        return this.size;
```

```
    }

    public T get(int index) {
        if (index < 0 || index >= this.size)
            throw new IndexOutOfBoundsException();
        return this.elements[index];
    }
    private void ensureCapcity() {
        if (this.elements.length == size) {
            this.elements = Arrays.copyOf(elements, this.size *
2);
        }
    }
}
```

Die Klasse implementiert eine Reallokations-Strategie: immer dann, wenn der interne Array voll ist, wird ein neuer erzeugt, der doppelt so groß ist wie der alte. Diese Strategie ist in der Klasse "fest verdrahtet". Sie soll ersetzt werden durch eine vom Benutzer definierbare Strategie.

Hier ein Interface:

```
package ex2;

@FunctionalInterface
public interface Reallocator {
    public int newSize(int oldSize);
}
```

Ändern Sie die `Array`-Klasse derart, dass dem Konstruktor die gewünschte Strategie übergeben werden kann. Falls der Benutzer keine Strategie übergibt, soll die "Verdopplungs"-Strategie verwendet werden.

Weitere Aufgabe: Ersetzten Sie `Reallocator` durch ein anderes Interface: durch das Standard-Interface `IntFunction`.

Lambdas – 3

Die meisten Listener-Interfaces des AWT sind keine funktionalen Interfaces. `WindowListener` z.B. hat acht Methoden. Es wäre schön, wenn auch zum Zwecke des Event-Handlings Lambdas genutzt werden könnten.

Gegeben ist folgende `MyFrame`-Klasse:

```
package ex3;
```

```
// ...
public class MyFrame extends Frame {
    private final TextField textFieldFoo = new TextField("Foo",
10);
    private final TextField textFieldBar = new TextField("World",
10);

    public MyFrame() {
        this.setLayout(new FlowLayout());
        this.add(this.textFieldFoo);
        this.add(this.textFieldBar);
        this.pack();

        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                MyFrame.this.dispose();
            }
        });

        this.textFieldFoo.addFocusListener(new FocusListener() {
            public void focusGained(FocusEvent e) {

MyFrame.this.textFieldFoo.setBackground(Color.yellow);
            }
            public void focusLost(FocusEvent e) {

MyFrame.this.textFieldFoo.setBackground(Color.white);
            }
        });

        this.setVisible(true);
    }
}
```

Studieren Sie nun folgende Klasse:

```
package ex3;
// ...
import java.util.function.Consumer;

public class FocusHandler implements FocusListener {

    private Consumer<FocusEvent> gained;
    private Consumer<FocusEvent> lost;

    public static FocusHandler focusListener() {
```

```
        return new FocusHandler();
    }

    public void focusGained(FocusEvent e) {
        if (this.gained != null)
            this.gained.accept(e);
    }
    public void focusLost(FocusEvent e) {
        if (this.lost != null)
            this.lost.accept(e);
    }

    public FocusHandler gained(Consumer<FocusEvent> consumer) {
        this.gained = consumer;
        return this;
    }
    public FocusHandler lost(Consumer<FocusEvent> consumer) {
        this.lost = consumer;
        return this;
    }
}
```

Es existiert eine weitere äquivalente Klasse `WindowHandler`.
Ändern Sie die `MyFrame`-Klasse derart, dass sie die `FocusHandler`- und `WindowHandler`-Klasse nutzt. Verwenden Sie in der `MyFrame`-Klasse nur noch Lambdas.

Lambdas – 4

Gegeben sei folgende Klasse:

```
package ex4;
// ...
public class MyFrame extends JFrame {
    private final TextField textField = new TextField(10);

    private int counter;

    public MyFrame() {
        this.setLayout(new FlowLayout());
        this.add(this.textField);
        this.setBounds(100, 100, 200, 100);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Thread progressThread = new Thread() {
```

```
            @Override
            public void run() {
                while(true) {
                    try {
                        Thread.sleep(1000);
                    }
                    catch(Exception e) {
                        throw new RuntimeException(e);
                    }
                    SwingUtilities.invokeLater(new Runnable() {
                        public void run() {
                            MyFrame.this.textField.setText(
                                String.valueOf(+
+MyFrame.this.counter));
                        }
                    });
                }
            }
        };
        progressThread.setDaemon(true);
        progressThread.start();
        this.setVisible(true);
    }
}
```

Studieren Sie diese Klasse. Ersetzen Sie dann die in der Klasse verwendeten anonyme Klassen durch Lambdas.

4 Details zu Lambdas

Syntax

Hier zunächst einmal die formale Definition eines Lambda-Ausdrucks:

```
lambda = ArgList "->" Body

ArgList = Identifier
        | "(" Identifier [ "," Identifier ]* ")"
        | "(" Type Identifier [ "," Type Identifier ]* ")"

Body = Expression
        | "{" [ Statement ";" ]+ "}
```

Anonyme Klassen vs. Lambdas

Worin liegen die Gemeinsamkeiten von anonymen Klassen und Lambdas? Worin liegen die Unterschiede?

Die Gemeinsamkeiten

- Sowohl anonyme Klassen als auch Lambdas sind namenlos.
- Zur Laufzeit existieren sowohl für anonyme Klassen als auch für Lambdas Objekte.
- Sofern anonyme Klassen resp. Lambdas in einem nicht-statischen Kontext definiert sind, haben die Objekte, die zur Laufzeit erzeugt werden, jeweils eine Referenz auf das "äußere", sie erzeugende Objekt (auf ihren Erzeuger).
- Sowohl in anonymen Klassen als auch in Lambdas können finale resp. effektiv finale Elemente der umschließenden Methode angesprochen werden.

Die Unterschiede:

- Der Zeiger auf das "äußere" Objekt wird in anonymen Klassen über den Namen `Outer.this` angesprochen (wobei `Outer` der Name der umschließenden Klasse ist). In Lambdas wird dieser Zeiger als `this` angesprochen.
- In einer anonymen Klasse können weitere Methoden und Attribute definiert werden. In einem Lambda-Ausdruck ist so etwas natürlich nicht möglich.

- Lambdas haben von sich aus keinen eigenen Typ (siehe den folgenden Abschnitt zum "Target-Typing"); anonyme Klassen haben einen Typ.

- Anonyme Klassen können nicht nur von Interfaces, sondern auch von Klassen abgeleitet sein. Lambdas können nur (funktionale) Interfaces implementieren.

Unterschiede, welche die technische Implementierung betreffen, werden später in diesem Kapitel noch ausführlich dargestellt werden.

Übersicht

In den folgenden Abschnitten werden Lambdas nun etwas genauer untersucht.

Zunächst geht's um das Thema "Target-Typing" – oder: welchen Typ hat ein Lambda?

Dann werden die sog. Methoden-Referenzen vorgestellt. Methoden-Referenzen können als "verkürzte" Lambdas angesehen werden.

Dann wird das Performance-Verhalten von Lambda-Funktionen analysiert – und mit der Performance von Methoden anonymer Klassen verglichen.

Die folgenden Abschnitte analysieren dann die Implementierung von anonymen Klassen und Lambdas.

Diese letzten Abschnitte bieten Hintergrundwissen – und ein solches Wissen ist sicherlich immer nützlich. Trotzdem müssen diese Abschnitte nicht unbedingt im Detail studiert werden (sofern die Zeit nicht reicht, können diese Abschnitte also übersprungen werden).

Im allerletzten Abschnitt stellt der Autor dieses Skripts ein kleines Experiment vor – ein fluent and typesafe select-from-where...

4.1 Target-Typing

Ein Lambda hat von sich aus keinen Typ. Wie kann dann trotzdem der Typ eines Lambdas bestimmt werden (er muss bestimmt werden!)? Es gibt vier Möglichkeiten.

Gegeben seien die folgenden beiden funktionalen Interfaces:

```
public interface Bar {
    public abstract int f(int x, int y);
}
```

```
public interface Bar {
    public abstract int b(int x, int y);
}
```

Man beachte, dass die beiden Interfaces "strukturell" äquivalent sind – ihre Methoden haben dieselbe Signatur und denselben Return-Typ.

Angenommen, wir schreiben nun folgende Zeile:

```
        Object obj = (x, y)-> x * y;   // illegal!
```

Der Compiler wird diese Zeile zurückweisen. Wenn irgendeine Zuweisung stattfindet, muss natürlich der Typ des zugewiesenen Ausdrucks ermittelt werden können. Genau dies ist hier nicht möglich.

Folgende Zuweisung dagegen ist korrekt:

```
        Foo foo = (x, y) -> x * y;
```

Der Compiler kennt den Typ von `foo` (nämlich `Foo`) – und weiß, dass es sich um ein funktionales Interface handelt. Dann kann geprüft werden, ob der Lambda-Ausdruck zur Signatur und zum Return-Typ der in `Foo` definierten Methode passt – er passt. (Hierbei kann dann gleichzeitig auch der Typ von `x` und `y` (nämlich `int`) deduziert werden.

Über `foo` ist dann natürlich `f` aufrufbar.

Und `foo` kann dann problemlos einer `Object`-Referenz zugewiesen werden:

```
        Object obj = foo;
```

Natürlich könnte exakt derselbe Lambda-Ausdruck auch einer `Bar`-Referenz zugewiesen werden:

```
        Bar bar = (x, y) -> x * y;
```

Der Typ eines Lambdas kann also aufgrund des Typs derjenigen Variablen berechnet werden, welcher er zugewiesen wird (oder auch nicht – sieht das `Object`-Beispiel).

Die zweite Variante der Typ-Bestimmung:

Der Typ eines Lambdas kann durch einen expliziten Cast bestimmt werden:

```
        Object obj1 = (Foo) (x, y) -> x * y;
        Object obj2 = (Bar) (x, y) -> x * y;
```

Der erste Lambda-Ausdruck ist nun vom Typ `Foo`, der zweite vom Typ `Bar`.

Die dritte Variante:

Sei eine Methode gegeben, welche ein `Foo` als Parameter verlangt:

```
    static void hello(Foo foo) {
        System.out.println(foo.f(20, 42));
    }
```

Dann kann diese Methode wie folgt aufgerufen werden:

```
        hello((x, y) -> x * y);
```

Der Compiler kann den Typ des Lambdas als `Foo` bestimmen – denn die `hello`-Methode verlangt ein `Foo`.

Das funktioniert allerdings nicht immer. Angenommen, es existiert noch eine weitere `hello`-Methode:

```
    static void hello(Bar bar) {
        System.out.println(bar.b(20, 42));
    }
```

Wie sollte der Compiler entscheiden, welche der beiden `hello`-Methoden aufgerufen werden soll? Also kann er den Typ des Lambdas nicht berechnen...

Und die vierte Variante:

Sei eine `world`-Methode gegeben, die ein `Foo` liefert:

```
    static Foo world() {
```

```
        return (x, y) -> x * y;
    }
```

Der Compiler kann natürlich auch hier den Typ des Lambdas berechnen: der Lambda-Ausdruck ist vom Typ `Foo`.

Der Compiler kann den Typ eines Lambdas also in folgenden Kontexten ermitteln (oder auch nicht!):

- Zuweisung
- Cast
- Parameter
- Return-Wert

Erst aufgrund dessen, was das "Ziel" ist, kann der Typ berechnet werden. Daher der Ausdruck "Target-Typing".

4.2 Methoden-Referenzen

In Fällen, wo eine Lambda keinen Block benötigt, kann ein solcher Ausdruck häufig auch durch eine sog. Methoden-Referenz ersetzt werden.

Sei z.B. folgende Klasse gegeben:

```
public class Foo {

    public static void f() {
        out.println("f()");
    }
    public static int g(int v) {
        return v * 2;
    }

    public void r() {
        out.println("r()");
    }
    public int s(int v) {
        return v * 2;
    }
}
```

Die Klasse enthält zwei statische und zwei nicht-statische Methoden. Jeweils eine der Methoden ist parameterlos, und die jeweils andere ist mit einem `int` parametrisiert.

Seien weiterhin folgende funktionale Interfaces gegeben:

```
@FunctionalInterface
public interface Mapper {
    public int map(int x);
}
```

```
@FunctionalInterface
public interface Action<T> {
    public void execute(T arg);
}
```

`map` liefert `int` zurück, `execute` liefert `void`. Das `Action`-Interface ist im Gegensatz zu `Mapper` ein generisches Interface.

Im folgenden wird via Lambda-Ausdruck ein `Runnable` erzeugt, dessen `run`-Methode die statische, parameterlose `f`-Methode der Klasse `Foo` aufruft:

```
Runnable r = () -> Foo.f();
r.run();
```

Dasselbe funktioniert auch mit einer "Methoden-Referenz":

```
Runnable r = Foo::f;
r.run();
```

Die Methoden-Referenz besteht hier aus dem Namen der Klasse, in welcher die aufzurufende Methode vereinbart ist, und dem Namen der Methode selbst. Klassen- und Methodenname sind durch `::` getrennt.

Nun wird ein `Foo`-Objekt erzeugt und in der `run`-Methode des mittels eines Lambda-Ausdrucks definierten `Runnable`s die nicht-statische `r`-Methode auf dieses `Foo`-Objekt aufgerufen:

```
Foo foo = new Foo();
Runnable r = () -> foo.r();
r.run();
```

Auch das funktioniert mit einer Methoden-Referenz:

```
Foo foo = new Foo();
Runnable r = foo::r;
r.run();
```

Statt des Namens der Klasse wird hier dem `::` der Name einer Referenz vorangestellt – einer Referenz, die auf ein `Foo`-Objekt zeigt.

In der `map`-Methode des im folgenden aufgrund des Lambda-Ausdruck erzeugten `Mappers` wir die statische, parametrisierte `g`-Methode von `Foo` aufgerufen:

```
Mapper m = v -> Foo.g(v);
int result = m.map(21);
out.println(result);
```

Die Ausgabe ist natürlich `42`.

Auch dies funktioniert mit einer Methoden-Referenz:

```
Mapper m = Foo::g;
int result = m.map(21);
out.println(result);
```

Der Compiler muss hier natürlich sicherstellen, dass der Parameter der `g`-Methode von `Foo` zu dem von der `map`-Methode verlangten Parameter passt (`int` passt zu `int`)...

Im folgenden wird mittels eines Lambdas zur Laufzeit ein Mapper erzeugt, dessen `map`-Methode die nicht-statische `s`-Methode auf das zuvor erzeugte `Foo`-Objekt aufruft:

```
            Foo foo = new Foo();
            Mapper m = v -> foo.s(v);
            int result = m.map(21);
            out.println(result);
```

Auch das funktioniert mittels einer Methoden-Referenz:

```
            Foo foo = new Foo();
            Mapper m = foo::s;
            int result = m.map(21);
            out.println(result);
```

Die Mechanismus funktionieren natürlich auch bei generischen Interfaces.

Hier die Lambda-Variante:

```
            Action<String> c = v -> out.println(v);
            c.execute("Hello");
```

Und hier schließlich die Variante mit einer Methoden-Referenz:

```
            Action<String> c = out::println;
            c.execute("Hello");
```

Insbesondere `out::println` wird in Zukunft wahrscheinlich häufig zu sehen sein...

Ein letztes Beispiel. Sei eine Klasse `Bar` gegeben:

```
public class Bar {
   public Bar(int value) {
      System.out.println("Bar(" + value + ")");
   }
}
```

Dann könnte man folgende `Action` erzeugen:

```
        Action<Integer> a = v -> new Bar(v);
        a.execute(42);
```

Auch hier kann eine Methoden-Referenz verwendet werden – eine Konstruktor-Referenz:

```
        Action<Integer> a = Bar::new;
        a.execute(42);
```

Nach Einschätzung des Autors dieses Skripts erfordern Methoden-Referenzen zwar weniger Schreibaufwand als die Definition expliziter Lambdas, sind aber i.d.R. schwerer zu verstehen. Daher wird in den folgenden Beispielprojekte auch weitgehend auf sie verzichtet – zumeist werden "richtige" Lambdas verwendet werden.

4.3 Performance

Unterscheidet sich die Performance von Methoden anonymer Klassen und Lambda-Funktionen?

Wir benutzen den `PerformanceRunner` aus den `shared`-Projekt.

Hier die Anwendung:

```
    private static final int TIMES = 1_000_000_000;
    private static final int LOOPS = 5;

    private static int n;
```

```
    static void demo() {
        PerformanceRunner runner = new PerformanceRunner();
        for (int i = 0; i < LOOPS; i++) {
            {
                Runnable r = new Runnable() {
                    public void run() {
                        n++;
                    }
                };
                runner.run("anonymous class", TIMES, r);
            }
            {
                Runnable r = () -> n++;
                runner.run("lambda", TIMES, r);
            }
            out.println(n);
            n = 0;
        }
```

```
    }
```

Das Protokoll zeigt, dass der Unterschied minimal ist:

```
Performace-Test: this will take some time ...
anonymous class                :  4637
lambda                         :  5668
2000000000
anonymous class                :  3560
lambda                         :  4863
2000000000
anonymous class                :  5025
lambda                         :  5042
2000000000
anonymous class                :  4913
lambda                         :  4859
2000000000
anonymous class                :  4885
lambda                         :  4851
2000000000
```

Schalten man die VM in den Server-Modus (mit dem VM-Argument `-server`), werden folgende Ausgaben erzeugt:

```
Performace-Test: this will take some time ...
anonymous class                :    88
lambda                         :    84
2000000000
anonymous class                :    65
lambda                         :    67
2000000000
anonymous class                :    65
lambda                         :    69
2000000000
anonymous class                :    69
lambda                         :    67
2000000000
anonymous class                :    68
lambda                         :    75
2000000000
```

Auch hier sind keine signifikanten Unterschiede erkennbar.

4.4 Anonyme Klassen und Lambdas

In den folgenden Abschnitten werden die Gemeinsamkeiten von anonymen Klassen und deren Unterschiede genauer beleuchtet.

(Die Entdeckungsreise wird etwas länger dauern – sie bietet aber hoffentlich genug Überraschungen...)

Die Analyse benutzt natürlich Reflection. Dabei wird insbesonder die im `shared`-Projekt definierte Klasse `Features` eingesetzt. Deren `print`-Methode kann benutzt werden, um alle Konstruktoren, Attribute und Methoden einer Klasse auflisten zu lassen.

Dabei muss unterschieden werden, ob die anonymen Klassen resp. Lambdas in einem statischen oder nicht-statischen Kontext definiert sind.

Es existieren vier startbare `Application`-Klassen:

```
as.Application
ls.Application
ai.Application
as.Application
```

Hier die Bedeutung der Paket-Namen:

- `as`: anonyme Klassen in einem statischen Kontext
- `ls`: Lambdas in einem statischen Kontext
- `ai`: anonyme Klassen in einem Instanz-Kontext
- `li`: Lambdas in einem Instanz-Kontext

Alle Methoden, die im folgenden vorgestellt werden, haben eine `throws`-Klausel – damit kein umständliches `Exception`-Handling stattfinden muss. Auch die `main`-Methoden haben jeweils diese Klausel...

Anonyme Klassen im statischen Kontext

Beginnen wir mit einer anonymen Klasse, die in einem statischen Kontext definiert ist. Zu Beginn von `main` wird die `Application`-Klasse analysiert; dann wird eine statische `demo`-Methode aufgerufen. Dort wird eine anonyme Klasse definiert, die das Interface `Runnable` implementiert. Schließlich wird dann diese anonyme Klasse analysiert.

```
package as;
// ...
public class Application {
```

```
    public static void main(String[] args) throws Exception {
        Features.print(Application.class);
        demo();
    }

    static void demo() throws Exception {
        final Runnable r = new Runnable() {
            public void run() {
                out.println("Hello");
            }
        };
        Features.print(r.getClass());
    }
}
```

Die Ausgaben (etwas verkürzt und umformatiert dargestellt):

```
as.Application (null)
   Constructors
      public as.Application()
   Methods
      static void as.Application.demo()
      public static void as.Application.main(java.lang.String[])

as.Application$1 (null)
   Constructors
      as.Application$1()
   Methods
      public void as.Application$1.run()
```

Wie aus dem Listing hervorgeht, hat die Klasse `as.Application` einen parameterlosen Konstruktor und zwei statische Methoden: `main` und `demo`. Aufgrund der Definition der anonymen Klasse hat der Compiler eine Klasse `as.Application$1` generiert. Diese hat ebenfalls einen parameterlosen Konstruktor (mit `package`-Sichtbarkeit) und eine nicht-statische `run`-Methode.

Der Bytecode von `as.Application` ist in der `class`-Datei `as/Application.class` abgelegt; der von der anonymen Klasse in `as/Application$1.class`.

Könnte man – just for fun – ein weiteres Objekt der anonymen Klasse erzeugen? Man kann – aber nur via Reflection (denn die Klasse ist ja namenlos). `demo` könnte wie folgt erweitert werden:

```
    static void demo() throws Exception {
        final Runnable r = new Runnable() {
            public void run() {
```

```
                out.println("Hello");
            }
        };
        // ...
        final Class<? extends Runnable> cls = r.getClass();
        final Runnable rr= cls.newInstance();
        rr.run();
    }
```

Die Ausgabe:

```
Hello
```

Wir können also zur Erzeugung eines Objekts einer anonymen Klasse die `Class`-Methode `newInstance` nutzen.

Lambdas im statischen Kontext

Nun zur Lambda-basierten Lösung:

```
package ls;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        Features.print(Application.class);
        demo();
    }

    static void demo() throws Exception {
        final Runnable r = () -> out.println("Hello");
        Features.print(r.getClass());
    }
}
```

Die Ausgaben:

```
ls.Application (null)
    Constructors
        public ls.Application()
    Methods
        static void ls.Application.demo()
        private static void ls.Application.lambda$0()
        public static void ls.Application.main(java.lang.String[])

ls.Application$$Lambda$1/12251916 (null)
```

```
    Constructors
        private ls.Application$$Lambda$1/12251916()
    Methods
        public void ls.Application$$Lambda$1/12251916.run()
```

Die Klasse `ls.Application` hat nun zusätzlich eine statische, private Methode namens `lambda$0` (hierzu gleich mehr).

Aufgrund des definierten Lambdas hat der Compiler eine Klasse mit vielen `$$` und einer Nummer erzeugt. Wie erwartet, hat auch diese Klasse einen parameterlosen Konstruktor und eine `run`-Methode. Der Konstruktor ist hier allerdings `private`.

Anders als bei anonymen Klassen wird der Bytecode dieser Klasse aber in keiner eigenen `class`-Datei abgelegt – sondern zusammen mit dem `Application`-Code in der Datei `ls/Application.class`. Es existiert hier also nur diese eine `class`-Datei.

Was hat es nun mit der `Application`-eigenen `lambda$0`-Methode auf sich? Hier ein Test (am Ende der `demo`-Methode):

```
        Method m =
Application.class.getDeclaredMethod("lambda$0");
        m.setAccessible(true);
        m.invoke(null);
```

Und hier das Resultat:

```
hello
```

`lambda$0` ist offenbar genau diejenige Methode, die den Bytecode für den Lambda-Ausdruck enthält. Hier der Bytecode dieser Methode (etwas "bereinigt"):

```
private static synthetic void lambda$0();
    0  getstatic System.out : PrintStream [61]
    3  ldc <String "Hello"> [67]
    5  invokevirtual PrintStream.println(String) : void [69]
    8  return
```

Da der Konstruktor der für das Lambda generierten Klasse `private` ist, kann nun nicht mehr via `Class.newInstance` ein neues Objekt dieser Klasse erzeugt werden. Man muss sich daher etwas mehr Mühe geben (über den "Umweg" eines `Constructor`-Objekts gehen):

```
    static void demo() throws Exception {
        final Runnable r = () -> out.println("Hello");
        // ....
         final Class<? extends Runnable> cls = r.getClass();
```

```
        final Constructor<? extends Runnable> ctor =
        cls.getDeclaredConstructor();
        ctor.setAccessible(true);
        final Runnable rr = ctor.newInstance();
        rr.run();
    }
```

Dieser Umweg ist nur deshalb erforderlich, weil nur über ein `Constructor`-Objekt, dessen `accessible`-Property auf `true` gesetzt wird, ein Objekt einer Klasse erzeugt werden, dessen Konstruktor `private` ist.

Anonyme Klassen im nicht-statischen Kontext

Gehen wir vom statischen Kontext zum nicht-statischen Kontext. Hier zunächst eine anonyme Klasse:

```
package ai;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        Features.print(Application.class);
        final Application appl = new Application();
        appl.demo();
    }

    void demo() throws Exception {
        final Runnable r = new Runnable() {
            public void run() {
                out.println("Hello");
            }
        };
        Features.print(r.getClass());
    }
}
```

In `main` wird ein `Application`-Objekt erzeugt und auf diese dann die nicht-statische `demo`-Methode aufgerufen.

Die Ausgaben:

```
ai.Application (null)
    Constructors
        public ai.Application()
    Methods
```

```
        void ai.Application.demo()
        public static void ai.Application.main(java.lang.String[])

ai.Application$1 (null)
    Constructors
        ai.Application$1(ai.Application)
    Fields
        final ai.Application ai.Application$1.this$0
    Methods
        public void ai.Application$1.run()
```

Wie man sieht, ist die `demo`-Methode von `ai.Application` nun nicht-`static`.

Die vom Compiler für die anonyme Klasse generierte Klasse `ai.Application$1` hat nun einen parametrisierten Konstruktor – einen Konstruktor, der ein Argument des Typs der umschließenden Klasse (`ai.Application`) verlangt. Dieser Konstruktor-Parameter wird dann offenbar dem Attribut `this$0` zugewiesen (dieses Attribut war in der statischen Variante nicht vorhanden). Es ist dies genau dasjenige Feld, welches im Java-Quellcode der anonymen Klasse als `Application.this` angesprochen wird.

Ein kurzer Blick in den Bytecode von `ai.Application$0`:

```
Application$1(ai.Application this$0);
     0  aload_0 [this]
     1  aload_1 [this$0]
     2  putfield ai.Application$1.this$0 : ai.Application [12]
     ...
```

Man erkennt, dass die dem Konstruktor übergebene `ai.Application`-Referenz im Feld `this$0` eingetragen wird.

Natürlich erzeugt der Compiler auch hier wieder (wie auch bei der statischen Varianten) zwei `class`-Dateien: `ai/Application.class` und `ai/Application$1.class`.

Wie kann in einem nicht-statischen Kontext via Reflection ein Objekt der anonymen Klasse erzeugt werden? Per `Class.forInstance` wird's nicht funktionieren. Man benötigt auch hier ein `Constructor`-Objekt, welches den parametrisierten Konstruktor repräsentiert. Da dieser aber nicht `private` ist, muss hier allerdings `accessible` nicht auf `true` gesetzt werden:

```
    void demo() throws Exception {
        final Runnable r = new Runnable() {
            public void run() {
                out.println("Hello");
            }
        };
```

```
        // ...
        final Class<? extends Runnable> cls = r.getClass();
        final Constructor<? extends Runnable> ctor =
                cls.getDeclaredConstructor(Application.class);
        final Runnable rr = ctor.newInstance(this);
        rr.run();
    }
```

Man beachte, dass beim Aufruf `getDeclaredConstructor` als Parameter die umschließende Klasse (`Application.class`) übergeben wird. Beim Aufruf von `newInstance` wird das `Application`-Objekt (`this`) übergeben.

Lambdas im nicht-statischen Kontext

Nun schließlich zu Lambdas in nicht-statischen Kontext:

```
package li;
// ...
public class Application {

    public static void main(String[] args) throws Exception {
        Features.print(Application.class);
        final Application appl = new Application();
        appl.demo();
    }

    void demo() throws Exception {
        final Runnable r = () -> out.println("Hello");
        Features.print(r.getClass());
    }
}
```

Die Ausgaben:

```
li.Application (null)
    Constructors
        public li.Application()
    Methods
        void li.Application.demo()
        private static void li.Application.lambda$0()
        public static void li.Application.main(java.lang.String[])

li.Application$$Lambda$1/12251916 (null)
    Constructors
        private li.Application$$Lambda$1/12251916()
    Methods
```

```
public void li.Application$$Lambda$1/12251916.run()
```

`li.Application` hat wieder die bereits von `ls.Application` bekannte `lambda$0`-Methode.

Die Klasse, die für den Lambda-Ausdruck generiert wird, hat im Unterschied zur Lösung mit anonymen Klassen nur einen einfachen, nicht-parametrisierten Konstruktor. Und demensprechend existiert auch kein Feld, in welchem die Adressen eine `Application`-Objekts gespeichert wird.

Die Lambda-Variante ist hier offensichtlich etwas "schlanker" an die Variante mit der anonymen Klasse. Die für den Lambda-Ausdruck generierte Klasse hat aber nur deshalb kein `Application`-Attribut, weil in dem Ausdruck kein expliziter Bezug (`this`) auf das "äußere" Objekt genommen wird!

Wie kann per Reflection ein Objekt der Lambda-Klasse erzeugt werden? Man benötigt ein `Constructor`-Objekt, dessen `accessible`-Eigenschaft auf `true` gesetzt werden muss – ein `Constructor`-Objekt, welches den parameterlosen Konstruktor repräsentiert:

```
void demo() throws Exception {
    final Runnable r = () -> out.println("Hello");
    // ...
    final Class<? extends Runnable> cls = r.getClass();
    final Constructor<? extends Runnable> ctor =
            cls.getDeclaredConstructor();
    ctor.setAccessible(true);
    final Runnable rr = ctor.newInstance();
    rr.run();
}
```

Resultate

Für jede anonyme Klassen erzeugt der Compiler eine eigene `class`-Datei. Für Lambdas werden keine `class`-Dateien erzeugt – der Bytecode, der für Lambdas generiert wird, wird in der `class`-Datei der die Lambdas umschließenden Klassen hinterlegt (in Methoden namens `lambda$...`).

Die Konstruktoren von anonymen Klasse haben die `package`-Sichtbarkeit; die Konstruktoren von Lambdas sind `private`.

In statischen Kontexten sind die Konstruktoren sowohl von anonymen Klassen als auch von Lambdas parameterlos.

Werden anonyme Klassen im nicht-statischen Kontext definiert, wird ein Konstruktor generiert, dem die Referenz auf eine Instanz der äußeren Klasse übergeben werden muss. Diese Referenz wird in einem eigenen Attribut gespeichert.

Für Lambdas wird auch dann ein parameterloser Konstruktor generiert, wenn diese in einem nicht-statischen Kontext definiert sind (aber nur dann, wenn im Lambda-Ausdruck kein ausdrücklicher Bezug auf das äußere Objekt genommen wird).

4.5 Bezug auf Elemente der äußeren Klasse

Sowohl in anonymen Klassen als auch in Lambda-Klassen können Elemente der umschließenden Klasse angesprochen werden. Statische Elemente der äußeren Klasse können immer angesprochen werden – gleichgültig, ob die anonyme- resp. die Lambda-Kasse in einen statischen oder nicht-statischem Kontext definiert sind. Nicht-statische Elemente der äußeren Klasse (also Instanz-Elemente) können nur von solchen anonymen Klassen resp. Lambdas angesprochen werden, die in einem nicht-statischen Kontext definiert sind.

Anonyme Klassen im statischen Kontext

`Application` definiert eine statische Variable `staticVar`. In der statischen `demo`-Methode gibt's eine anonyme Klasse, innerhalb derer diese Variable angesprochen wird:

```
package as;
// ...
public class Application {

    private static int staticVar = 42;

    public static void main(String[] args) throws Exception {
        Features.print(Application.class);
        demo();
    }

    static void demo() throws Exception {
        final Runnable r = new Runnable() {
            public void run() {
                out.println("Hello " + Application.staticVar);
            }
        };
        Features.print(r.getClass());
    }
}
```

(Natürlich hätte man innerhalb der `run`-Methode auf die Qualifizierung von `staticVar` auch verzichten können – statt `Application.staticVar` hätte man einfach `staticVar` schreiben können.)

Die Ausgaben:

```
as.Application (null)
   Constructors
      public as.Application()
   Fields
      private static int as.Application.staticVar
   Methods
      static void as.Application.demo()
      static int as.Application.access$0()
      public static void as.Application.main(java.lang.String[])

as.Application$1 (null)
   Constructors
      as.Application$1()
   Methods
      public void as.Application$1.run()
```

Für die `Application`-Klasse generiert der Compiler eine Methode `access$0` – diese wird in der `run`-Methode der anonymen Klasse aufgerufen, um auf `staticVar` zugreifen zu können (`staticVar` ist als `private` definiert). Diese `access$0`-Methode würde nicht generiert werden, wenn `staticVar` nicht `private` wäre – dann könnte `run` direkt auf die Variable zugreifen.

Für die anonyme Klasse wird ein parameterloser Konstruktor definiert (was nicht weiter überraschend ist).

Lambdas im statischen Kontext

```
package ls;
// ...
public class Application {

    private static int staticVar = 42;

    public static void main(String[] args) throws Exception {
        Features.print(Application.class);
        demo();
    }

    static void demo() throws Exception {
        final Runnable r = () -> out.println("Hello " +
staticVar);
        Features.print(r.getClass());
    }
}
```

```
ls.Application (null)
```

```
   Constructors
      public ls.Application()
   Fields
      private static int ls.Application.staticVar
   Methods
      static void ls.Application.demo()
      private static void ls.Application.lambda$0()
      public static void ls.Application.main(java.lang.String[])

ls.Application$$Lambda$1/12251916 (null)
   Constructors
      private ls.Application$$Lambda$1/12251916()
   Methods
      public void ls.Application$$Lambda$1/12251916.run()
```

Da in `run` ohnehin bereits die für die `Application`-Klasse generierte `lamdda$0`-Methode aufgerufen wird, bedarf es hier keiner weiteren `access$0`-Methode. Und die `lambda$0`-Methode wird natürlich unabhängig von der Sichtbarkeit von `staticVar` generiert.

Anonyme Klassen im nicht-statischen Kontext

Im folgenden wird in `Appliation` eine Instanzvariable `instanceVar` definiert. In `main` wird ein `Application`-Objekt erzeugt, auf welches `demo` aufgerufen wird. Die in `demo` implementierte anonyme Klasse greift via `Application.this.instanceVar` auf die Instanzvariable des äußeren Objekts zu:

```
package ai;
// ...
public class Application {

    private int instanceVar = 42;

    public static void main(String[] args) throws Exception {
        Features.print(Application.class);
        Application appl = new Application();
        appl.demo();
    }

    void demo() throws Exception {
        final Runnable r = new Runnable() {
            public void run() {
                out.println("Hello " +
Application.this.instanceVar);
            }
        };
```

```
        Features.print(r.getClass());
    }
}
```

```
ai.Application (null)
   Constructors
      public ai.Application()
   Fields
      private int ai.Application.instanceVar
   Methods
      void ai.Application.demo()
      static int ai.Application.access$0(ai.Application)
      public static void ai.Application.main(java.lang.String[])

ai.Application$1 (null)
   Constructors
      ai.Application$1(ai.Application)
   Fields
      final ai.Application ai.Application$1.this$0
   Methods
      public void ai.Application$1.run()
```

Die `access$0`-Methode von `Application` hat nun einen `Application`-Parameter – weil sie `static` ist, benötigt sie diesen Parameter, um auf `instanceVar` zugreifen zu können.

Die anonyme Klasse hat einen Konstruktor, dem das äußere Objekt übergeben wird – und ein Attribut `this$0`, in dem die Referenz auf dieses äußere Objekt gehalten wird. Wird `run` aufgerufen, so wird `run` die `access$0`–Methode aufrufen – und dabei `this$0` als Parameter übergeben.

Wäre `instanceVar` nicht `private`, gäb's keine `access$0`-Methode. `run` könnte dann direkt via `this$0` auf `instanceVar` zugreifen.

Lambdas im nicht-statischen Kontext

Sofern in einem Lambda-Ausdruck keine Instanz-Elemente der äußeren Klasse angesprochen werden, wird für die Lambda-Klasse ein parameterloser Konstruktor erzeugt (s. letzten Abschnitt). Sofern aber solche Elemente angesprochen werden, benötigt natürlich auch eine Lambda-Kasse eine parametrisierten Konstruktor – und eine von diesem Konstruktor initialisierte Referenzvariable, die auf das äußere Objekt zeigt.

Im folgenden nicht-statischen Lambda wird via `this.instanceVar` auf das `instanceVar`-Attribut des äußeren Objekts zugegriffen (statt `this.instanceVar` hätte man natürlich auch einfach `instanceVar` schreiben können):

```
package li;
// ...
public class Application {

    private int instanceVar = 42;

    public static void main(String[] args) throws Exception {
        Features.print(Application.class);
        Application appl = new Application();
        appl.demo();
    }

    void demo() throws Exception {
        final Runnable r =
           () -> out.println("Hello " + this.instanceVar);
        Features.print(r.getClass());
    }
}
```

```
li.Application (null)
   Constructors
      public li.Application()
   Fields
      private int li.Application.instanceVar
   Methods
      void li.Application.demo()
      private void li.Application.lambda$0()
      public static void li.Application.main(java.lang.String[])

li.Application$$Lambda$1/8460669 (null)
   Constructors
      private li.Application$$Lambda$1/8460669(li.Application)
   Fields
      private final li.Application
                    li.Application$$Lambda$1/8460669.arg$1
   Methods
      public void li.Application$$Lambda$1/8460669.run()
      private static java.lang.Runnable
          li.Application$$Lambda$1/8460669.get$Lambda(li.Application)
```

`lambda$0` von `Application` ist nun eine Instanzmethode (nicht `static`).

Genau wie anonyme Klassen hat auch eine Lambda-Klasse, die in einem nicht-statischen Kontext definiert wird, eine Referenz auf das äußere Objekt, die über den Konstruktor parametrisiert wird – hier hat diese Referenz allerdings den Namen `arg$1`.

Zusätzlich gibt's eine private, statische Methode `get$Lambda`, die mit einer `Application`-Referenz aufgerufen werden muss. Die Methode erzeugt ein neues Objekt der Lambda-Klasse und liefert dieses zurück. Um dies zu zeigen, erweitern wir die `demo`-Methode:

```
    void demo() throws Exception {
        final Runnable r =
           () -> out.println("Hello " + this.instanceVar);
        // ...
        Method m = r.getClass().getDeclaredMethod(
            "get$Lambda", Application.class);
        m.setAccessible(true);
        Runnable rr = (Runnable) m.invoke(r, this);
        rr.run();
        out.println(r == rr);
        out.println(r.getClass() == rr.getClass());
    }
```

Die Ausgaben:

```
Hello 42
false
true
```

Resultate

Der Zugriff auf statische Elemente der umschließenden Klasse ist stets unproblematisch – gleichgültig, ob die anonyme resp. die Lambda-Klasse in einem statischen oder nicht-statischen Kontext definiert sind.

Sofern eine anonyme Methode in einem nicht-statischen Kontext definiert ist, besitzt sie immer eine Referenz auf eine Instanz der äußeren Klasse – auf diejenige Instanz, innerhalb derer das Objekt der anonymen Klasse erzeugt wurde. Nur über diese Referenz kann das Objekt der anonymen Klasse auf die Instanz-Elemente des äußeren Objekts zugreifen.

Bei Lambdas sieht die Sache ähnlich aus – mit dem Unterschied allerdings, dass die Referenz auf das äußere Objekt nur dann generiert wird, wenn sie tatsächlich benötigt wird (wenn die anonyme Klasse also via `this` auf Instanz-Elemente des äußeren Objekts zugreift).

Eine Lambda-Klasse, die in einem nicht-statischen Kontext definiert ist und eine Referenz auf das äußere Objekt hat, besitzt zusätzlich eine `get$Lambda`-Methode, mittels derer ein neues Objekt der Lambda-Klass erzeugt werden kann.

Was hier am Beispiel von Attributen demonstriert wurde (`staticVar`, `instanceVar`), gilt natürlich auch für Methoden (statische-, nicht-statische Methoden).

4.6 Bezug auf Elemente der umschließenden Methode

Anonyme- und Lambda-Klassen können auf lokale Elemente der umschließenden Methode zugreifen, sofern diese entweder explizit oder effektiv final sind – also auf lokale Variablen und auf Parameter der umschließenden Methode.

Der Kürze halber beschränkt sich die folgende Darstellung auf anonyme- resp. Lambda-Klassen, die in einem nicht-statischen Kontext definiert sind. Auch die Analyse wird sich auf anonyme- resp. Lambda-Klasse beschränken (die Klasse `Application` wird also nicht weiter betrachtet – hier würde man auch nichts Neues entdecken).

Anonyme Klassen

In der `demo`-Methode, in welcher die anonyme Klasse instantiiert wird, werden zwei lokale Variablen definiert: eine Variable `foo`, die effektiv `final` ist, und eine Variable `bar`, die explizit als `final` deklariert ist. Zusätzlich gibt's in `Application` eine Instanzvariable `hello`:

```
package ai;
// ...
public class Application {

    String hello = "Hello";

    public static void main(String[] args) throws Exception {
        Application appl = new Application();
        appl.demo();
    }

    void demo() throws Exception {
        int foo = 42;
        final int bar = 77;
        final Runnable r = new Runnable() {
            public void run() {
                out.println(Application.this.hello + " " + foo +
" " + bar);
            }
        };
        Features.print(r.getClass());
    }
}
```

Die Ausgaben:

```
ai.Application$1 (null)
   Constructors
      ai.Application$1(ai.Application,int)
   Fields
      final ai.Application ai.Application$1.this$0
      private final int ai.Application$1.val$foo
   Methods
      public void ai.Application$1.run()
```

Das Feld `this$0` ist bereits bekannt. Zusätzlich existiert ein weiters als `final` definierte Feld: `val$foo` vom Typ `int`. Beide Felder werden vom Konstruktor initialisiert (der dementsprechend zwei Parameter besitzt).

Beim Erzeugen einer Instanz der anonymen Klasse wird also erstens die Referenz auf das äußere `Application`-Objekt übergeben und zweitens offensichtlich der Wert der `foo`-Variablen. Der Wert der `foo`-Variablen wird also in das erzeugte Objekt der anonymen Klasse hineinkopiert – und kann dann natürlich in `run` angesprochen werden.

Der Wert von `bar` wird aber ebenfalls in `run` angesprochen – anders als der `foo`-Wert wird der Wert von `bar` aber nicht kopiert. Der Compiler unterscheidet also: ist ein lokales Element nur effektiv, aber nicht explizit `final`, so wird kopiert; ist ein lokales Element explizit `final`, wird nicht kopiert. Stattdessen wird in der `run`-Method der Wert eines solchen Elements einfach in Literalform abgelegt. Das Literal `77` existiert im Bytecode also zweimal: einmal in der `demo`-Methode und ein zweites Mal in der `run`-Methode. (Das Literal `42` existiert im Bytecode dagegen nur einmal: in der `demo`-Methode.)

Wir erweitern `demo`, um die Werte von `this$0` und `val$foo` auszulesen:

```
        // ...
        Class<? extends Runnable> cls = r.getClass();

        Field field0 = cls.getDeclaredField("this$0");
        Object obj0 = field0.get(r);
        out.println(obj0 == this);

        Field field1 = cls.getDeclaredField("val$foo");
        field1.setAccessible(true);
        Object obj1 = field1.get(r);
        out.println(obj1);
```

Die Ausgaben: `true` und `42`.

Lambdas

```
package li;
// ...
public class Application {

    String hello = "Hello";

    public static void main(String[] args) throws Exception {
        Application appl = new Application();
        appl.demo();
    }

    void demo() throws Exception {
        int foo = 42;
        final int bar = 77;
        final Runnable r =
           () -> out.println(this.hello + " " + foo + " " + bar);
        Features.print(r.getClass());
    }
}
```

```
li.Application$$Lambda$1/12251916 (null)
   Constructors
      private li.Application$$Lambda$1/12251916(li.Application,int)
   Fields
      private final li.Application
                    li.Application$$Lambda$1/12251916.arg$1
      private final int li.Application$$Lambda$1/12251916.arg$2
   Methods
      private static java.lang.Runnable
                     ....get$Lambda(li.Application,int)
      public void li.Application$$Lambda$1/12251916.run()
```

Auch ein Objekt der hier definierten Lambda-Klasse hat zwei Felder: eine Referenz auf das umschließende Objekt (`arg$1`) und ein `int`-Feld für den Wert von `foo` (`arg$2`). Wir lesen auch hier die Daten aus:

```
        // ...
        Class<? extends Runnable> cls = r.getClass();

        Field field0 = cls.getDeclaredField("arg$1");
        field0.setAccessible(true);
        Object obj0 = field0.get(r);
        out.println(obj0 == this);

        Field field1 = cls.getDeclaredField("arg$2");
        field1.setAccessible(true);
        Object obj1 = field1.get(r);
```

```
        out.println(obj1);
```

Und erhalten auch hier die Ausgaben `true` und `42`.

Um eine neue Instanz der Lambda-Klasse zu erzeugen, kann `get$Lambda` aufgerufen werden – mit zwei Parametern:

```
        Method m = cls.getDeclaredMethod(
            "get$Lambda", Application.class, int.class);
        m.setAccessible(true);
        Runnable rr = (Runnable)m.invoke(r, this, 43);
        rr.run();
```

Die Ausgabe:

```
Hello 43 77
```

Man beachte die Typen, die an `getDeclaredMethod` übergeben werden; und die Werte, die an `invoke` übergeben werden.

Resulate

Eine anonyme- resp. Lambda-Klasse kann nur dann lokale Elemente (Variablen oder Parameter) der umschließenden Methode referenzieren, wenn diese entweder effektiv oder explizit `final` sind.

Ist ein Element nicht explizit `final`, so wird bei der Instantiierung der anonymen- resp. der Lambda-Klasse der Wert des Elements in das erzeugte Objekt hineinkopiert. Methoden der anonymen resp. der Lambda-Klassen beziehen sich dann auf diese Kopie.

Ist ein Element explizit `final`, so findet keine Kopie statt. Stattdessen nutzen die Methoden der anonymen- resp. der Lambda-Klasse einfach das Literal.

4.7 Serialisierung

Können Objekte anonymer- resp. Lambda-Klassen problemlos serialisiert werden? Sieht Gibt's Unterschiede bezüglich der Serialisierung von Objekten anonymer Klassen und der Serialiseriung von Objekten von Lambda-Klassen?

Um die Serialisierung / Deserialisierung einfach handhaben zu können, wird im folgenden stets die Klasse `SerializeUtil` aus dem `shared`-Projekt verwendet.

Wie setzen die Kenntnis eines wenig bekannten Mechanismus voraus, der es erlaubt, bei der Serialisierung eines Objekts nicht dieses selbst, sondern etwas "ganz anderes" zu serialisieren:

Besitzt eine serialisierbare Klasse eine Methode `writeReplace`, dann wird diese im Kontext der Serialisierung eines Objekte dieser Klasse aufgerufen – und es wird dasjenige Objekt serialisiert, welches diese Methode zurückliefert. Besitzt eine Klasse eine Methode namens `readResolve`, so wird diese im Kontext der Deserialisierung aufgerufen. Und die Deserialisierung liefert als Resultat genau dasjenige Objekt zurück, welches `readResolve` liefert. `readResolve` ist also "invers" zu `writeReplace`.

Hier ein kleines Demo-Beispiel:

```
package demo;
// ...
public class Point implements Serializable {

    public static void main(String[] args) {
        Point p0 = new Point(1, 2);
        out.println(p0);
        Point p1 = SerializeUtil.serializeDeserialize(p0);
        out.println(p1);
        out.println(p0 == p1);
    }

    static class SerializedPoint implements Serializable {
        final private String s;
        SerializedPoint(Point point) {
            this.s = point.y + "#" + point.x;
        }
        private Object readResolve() {
            out.println("SerializedPoint.readResolve");
            out.println("\tthis = " + this);
            final String[] tokens = this.s.split("#");
            final Point p = new Point(
```

```
                Integer.parseInt(tokens[1]),
Integer.parseInt(tokens[0]));
            out.println("\treturning "+ p);
            return p;
        }
        @Override
        public String toString() {
            return this.getClass().getSimpleName() + " [" +
this.s + "]";
        }
    }

    public int x;
    public int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    private Object writeReplace() {
        out.println("Point.writeReplace");
        out.println("\tthis = " + this);
        final SerializedPoint sp = new SerializedPoint(this);
        out.println("\treturning " + sp);
        return sp;
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName()
            + " [" + this.x + ", " + this.y + "]";
    }
}
```

Anhand der Ausgaben wird deutlich, was hier passiert:

```
Point [1, 2]
Point.writeReplace
   this = Point [1, 2]
   returning SerializedPoint [2#1]
SerializedPoint.readResolve
   this = SerializedPoint [2#1]
   returning Point [1, 2]
Point [1, 2]
false
```

Ein `Point` wird nicht als solcher, sondern in der Form eines `SerializedPoint`-Objekts serialisiert.

Nun zum eigentlichen Thema.

Im Package `f` existiert das funktionales Interface `Foo` (welches das Interface `Serializable` erweitert):

```
package f;

import java.io.Serializable;

@FunctionalInterface
public interface Foo extends Serializable {
    public void run();
}
```

Wir beginnen mit Klassen, die in einem statischen Kontext definiert sind.

Serialisierung im statischen Kontext

Die Klasse `s.Application` (`s` steht für `static`) ruft zwei statische `demo`-Methoden auf. `Application` selbst ist nicht serialisierbar:

```
package s;
// ...
public class Application {

   public static void main(String[] args) throws Exception {
      demoAnonymous();
      demoLambda();
   }

   // s.u.
}
```

Die erste `demo`-Methode serialisiert ein Objekt einer anonymen Klasse

```
   static void demoAnonymous() {
      Foo f0 = new Foo() {
         public void run() {
            out.println("Hello");
         }
      };
      Features.print(f0.getClass());
      f0.run();
```

```
      Foo f1 = SerializeUtil.serializeDeserialize(f0);
      out.println(f1 == f0);
      f1.run();
   }
```

Die Serialisierung ist völlig unproblematisch. Hier die Ausgaben:

```
s.Application$1 (null)
   Constructors
      s.Application$1()
   Methods
      public void s.Application$1.run()
Hello
false
Hello
```

Dass die Sache unproblematisch ist, liegt natürlich am statischen Kontext – also daran, dass die anonyme Klasse keine Referenz auf ein äußeres Objekt hat.

Ist die Serialisierung von Objekten anonymer Klassen überhaupt sinnvoll? Aber sicher: sofern sie Daten enthalten. Und anonyme `Foo`-Klassen können natürlich Instanzvariablen besitzen (auf die z.B. die `run`-Methode zugreift):

```
      Foo f = new Foo() {
          private int x = 42;
         public void run() {
            out.println(x++);
         }
      };
```

Auch die Serialisierung von Objekten von Lambda-Klassen ist problemlos – sofern auch die Lambdas in einem statischen Kontext definiert sind:

```
   static void demoLambda() {
      Foo f0 = () -> out.println("Hello");
      Features.print(f0.getClass());
      f0.run();
      Foo f1 = SerializeUtil.serializeDeserialize(f0);
      out.println(f1 == f0);
      f1.run();
   }
}
```

Die Ausgaben:

```
s.Application$$Lambda$1/5358504 (null)
   Constructors
```

```
      private s.Application$$Lambda$1/5358504()
   Methods
      public void s.Application$$Lambda$1/5358504.run()
      private final java.lang.Object
          s.Application$$Lambda$1/5358504.writeReplace()
Hello
false
Hello
```

Was auffällt: Anders als bei anonymen Klassen gibt's bei Lambdas, die serialisierbar sind, die bereits oben erläuterte Methode `writeReplace`... (mit dieser Methode werden wir uns noch näher beschäftigen).

Eine Serialisierung solcher Objekte ist natürlich weniger sinnvoll – denn Objekte von Lambda-Klassen können keine Attribute besitzen (also keinen eigentlichen Zustand).

Serialisierung im nicht-statischen Kontext

Im Unterschied zu der letzten `Application`-Klasse ist die Klasse `i.Application` (`i` steht für "Instance") serialisierbar (sie muss serialisierbar sein). Sie ruft vier nicht-statische `demo`-Methoden auf eine `Application`-Instanz auf:

```
package i;
// ...
import java.lang.invoke.SerializedLambda;

public class Application implements Serializable {

    public static void main(String[] args) throws Exception {
        Application appl = new Application();

        appl.demoAnonymous();
        appl.demoLambda();

        appl.demoSerializedLambda();
        appl.demoDeserializeSerializedLambda();
    }

    // s.u.
}
```

Es wird sich zeigen, dass im Kontext der Serialisierung von Objekten anonymer resp. Lambda-Klassen auch das in `main` erzeugte `Appliation`-Objekt serialisert / deserialisiert werden wird. Um dies zu zeigen, definieren wird die `writeReplace`- und

die `readResolve`-Methode – um die Serialisierung / Deserialisierung beobachten zu können (wobei das Standardverhalten wird beibehalten wird):

```
    private Object writeReplace() {
        out.println(">> serializing " + this);
        return this;
    }
    private Object readResolve() {
        out.println("<< deserializing " + this);
        return this;
    }
```

Wir serialisieren zunächst eine Instanz einer anonymen Klasse:

```
    void demoAnonymous() {
        Foo f0 = new Foo() {
            public void run() {
                out.println("Hello " + Application.this);
            }
        };
        Features.print(f0.getClass());
        f0.run();
        Foo f1 = SerializeUtil.serializeDeserialize(f0);
        out.println(f1 == f0);
        f1.run();
    }
```

Die Ausgaben:

```
i.Application$1 (null)
   Constructors
      i.Application$1(i.Application)
   Fields
      final i.Application i.Application$1.this$0
   Methods
      public void i.Application$1.run()
Hello i.Application@52e922
>> serializing i.Application@52e922
<< deserializing i.Application@1c7c054
false
Hello i.Application@1c7c054
```

Die Ausgaben zeigen, dass nicht nur das `Foo`-Objekt, sondern auch das `Application`-Objekt serialisiert / deserialisiert wird – weil die Instanz der anonymen Klasse eine interne Referenz auf das `Application`-Objekt besitzt. Und diese Referenz ist nicht `transient`. Sie kann auch nicht einfach auf `null` gesetzt werden – denn sie ist `final` (was mit normalen Java-Mitteln unmöglich ist, ist allerdings per Reflection möglich...).

Die Ausgaben zeigen weiterhin, dass das via `f1` referenzierte `Foo`-Objekt nun eine Referenz auf ein neues(!) `Application`-Objekt besitzt. (Es stellt sich natürlich die Frage, ob so etwas sinnvollerweise gewollt sein kann...)

In der folgenden Methode wird ein Objekt einer Lambda-Klasse serialisiert:

```
    void demoLambda() {
        Foo f0 = () -> out.println("Hello " + this);
        Features.print(f0.getClass());
        f0.run();
        Foo f1 = SerializeUtil.serializeDeserialize(f0);
        out.println(f1 == f0);
        f1.run();
    }
```

Die Ausgaben:

```
i.Application$$Lambda$1/11043253 (null)
   Constructors
      private i.Application$$Lambda$1/11043253(i.Application)
   Fields
      private final i.Application i.Application$
$Lambda$1/11043253.arg$1
   Methods
      public void i.Application$$Lambda$1/11043253.run()
      private final java.lang.Object
                    i.Application$$Lambda$1/11043253.writeReplace()
      private static f.Foo i.Application$
$Lambda$1/11043253.get$Lambda(i.Application)
Hello i.Application@52e922
>> serializing i.Application@52e922
<< deserializing i.Application@1fc625e
false
```

Auch hier wird im Kontext der Deserialisierung nicht nur ein neues `Foo`-Objekt erstellt, sondern auch ein neuen `Application`-Objekt.

Wie wir gezeigt haben, besitzt eine Lambda-Klasse, welche ein Interface implementiert, welches seinerseits `Serializable` erweitert, eine `writeReplace`-Methode besitzt. Was hat es mit dieser Methode auf sich? Wir können sie per Reflection aufrufen:

```
    void demoSerializedLambda() throws Exception {
        Foo f0 = () -> out.println("Hello " + this);
        f0.run();
        Method writeReplaceMethod =
            f0.getClass().getDeclaredMethod("writeReplace");
```

```
        writeReplaceMethod.setAccessible(true);
        SerializedLambda lambda =
            (SerializedLambda)writeReplaceMethod.invoke(f0);
        out.println(lambda);
        Method readResolveMethod =
            lambda.getClass().getDeclaredMethod("readResolve");
        readResolveMethod.setAccessible(true);
        Foo f1 = (Foo)readResolveMethod.invoke(lambda);
        f1.run();
        out.println(f1 == f0);
    }
```

`writeReplace` liefert offenbar ein Objekt der Klasse `SerializedLambda` zurück. Ein solches Objekt ist vergleichbar mit der in der Einleitung vorgestellten Demo-Klasse `SerializedPoint`. Also enthält diese Klasse dann die zu `writeReplace` inverse Methode `readResolve`. Rufen wir auch diese per Reflection auf das `SerializedLambda` auf, so erhalten wir ein neues `Foo`. Da aber keine Serialisierung stattgefunden hat, zeigt auch das neue `Foo` auf die "alte" `Application`.

Hier die Ausgaben:

```
Hello i.Application@52e922
SerializedLambda[
    capturingClass=class i.Application,
    functionalInterfaceMethod=f/Foo.run:()V,
    implementation=invokeSpecial i/Application.lambda$1:()V,
    instantiatedMethodType=()V,
    numCaptured=1]
Hello i.Application@52e922
false
```

In der letzten Methode wird – just for fun – gezeigt, wie das `SerializedLamba` eines `Foo`-Objekts als Input für die Serialisierung verwendet werden kann. Das Resultat der Deserialisierung ist dann wiederum ein `Foo`-Objekt (das hört sich auf den ersten Blick alles recht merkwürdig an, erscheint aber nach näheren Hinsehen plausibel):

```
    void demoDeserializeSerializedLambda() throws Exception {
        Foo f0 = () -> out.println("Hello " + this);
        f0.run();
        Method writeReplaceMethod =
            f0.getClass().getDeclaredMethod("writeReplace");
        writeReplaceMethod.setAccessible(true);
        SerializedLambda lambda =
            (SerializedLambda)writeReplaceMethod.invoke(f0);
        Foo f1 =
```

```
(Foo)SerializeUtil.deserialize(SerializeUtil.serialize(lambda));
        f1.run();
        out.println(f1 == f0);
    }
```

Die Ausgaben:

```
Hello i.Application@52e922
>> serializing i.Application@52e922
<< deserializing i.Application@20c684
Hello i.Application@20c684
false
```

Hier wurde auch wieder ein neues `Application`-Objekt erzeugt.

Resultate

Sofern anonyme- resp. Lambda-Klassen in einem statischen Kontext definiert sind, ist die Serialisierung / Deserialisierung ihrer Objekte unproblematisch (Voraussetzung ist natürlich, dass das von den Klassen implementierte Interface seinerseits serialisierbar ist).

Sind die anonymen- resp. Lambda-Klassen in einen nicht-statischen Kontext definiert, wird bei der Serialisierung ihrer Objekte immer auch das "äußere Objekt" mit in die Serialisierung einbezogen. Und bei der Deserialisierung entsteht nicht nur ein neues Objekt der anonymen- resp. Lambda-Klasse, sondern auch immer ein neues "äußeres Objekt".

Objekte anonymer Klassen werden als solche serialisiert; Objekte von Lambda-Klassen werden in Form von `SerializedLambdas` serialisiert.

4.8 Generics

Im folgenden untersuchen wird den Zusammenhang zwischen anonymen- resp. Lambda-Klassen und Generics. (Und by the way: Hierbei werden uns erneut die `SerializedLambdas` begegnen.)

Die folgenden Demo-Methoden benutzen jeweils anonyme- resp. Lambda-Klassen, die in einem statischen Kontext definiert sind. Wir hätten auch einen nicht-statischen Kontext wählen können – für die hier darzustellenden Zusammenhänge ist die Wahl des Kontextes nicht weiter relevant.

Wir benutzen nun ein generisches Interface (welche allerdings wieder `Serializable` erweitert):

```
package appl;

import java.io.Serializable;

@FunctionalInterface
public interface Foo<T> extends Serializable {
    public void run(T value);
}
```

Es geht um folgende Frage: Gegeben sei eine Referenz vom Typ `Foo<?>`, die auf irgendein `Foo`-Objekt zeigt – z.B. auf ein `Foo<String>`, ein `Foo<Integer>` oder ein `Foo<Foo>`. Kann aufgrund einer solchen Referenz zur Laufzeit der aktuelle Typ des Typ-Parameters ermittelt werden (also `String`, `Integer` oder `Foo`)?

Eine etwa vorschnelle Antwort lautet: "nein" – aber diese Anwort ist eben etwas vorschnell. Wenn die Antwort aber "ja" lautet – wozu kann man ein solches Wissen dann nutzen? Wissen sollte nicht nutzlos sein...

Analyse einer anonymen generischen Klasse

In der ersten (statischen) Methode wird eine dieses Interface implementierende Klasse analysiert. Als aktueller Typ-Parameter wird `String` verwendet:

```
    static void analyseAnonymous() throws Exception {
        Foo<String> f = new Foo<String>() {
            public void run(String value) {
                out.println(value);
            }
        };
        f.run("Hello");
        Class<?> cls = f.getClass();
        Features.print(cls);

        out.println("getGenericInterfaces...");
        Type[] ifaces = cls.getGenericInterfaces();
        for(Type iface : ifaces)
            out.println("\t" + iface);
        Type iface = ifaces[0];
        System.out.println(iface);
        System.out.println(iface.getClass());
        ParameterizedType pt = (ParameterizedType)iface;
```

```
        out.println("getActualTypeArguments...");
        Type[] argTypes = pt.getActualTypeArguments();
        for (Type argType : argTypes)
            out.println("\t" + argType);
        Class<?> argClass = (Class<?>) argTypes[0];
        out.println(argClass);
    }
```

Hier die Ausgaben (an einer Stelle etwas verkürzt):

```
Hello

appl.Application$1 (null)
   Constructors
      appl.Application$1()
   Methods
      public void appl.Application$1.run(java.lang.String)
      public void appl.Application$1.run(java.lang.Object)

getGenericInterfaces...
   appl.Foo<java.lang.String>
appl.Foo<java.lang.String> class sun...ParameterizedTypeImpl
getActualTypeArguments...
   class java.lang.String
class java.lang.String
```

Die anonyme Klasse enthält zwei `run`-Methoden – die eine ist mit `Object`, die andere mit `String` parametrisiert. Eigentlich aber hätten wird doch nur die mit `Object` parametrisierte Methode erwartet – da wir doch wissen, dass im Zuge der Kompilation die ganzen generischen Informationen "verdampfen".

Also scheint der Umstand, dass bei der Instantiierung der Klasse als aktueller Typ-Parameter `String` verwendet wurde, doch irgendwo auch zur Laufzeit wieder "auffindbar" zu sein. Man könnte per Reflection diejenige `run`-Methode ermitteln, deren Parameter-Typ nicht `Object` ist – der Typ dieses Parameters ist dann der aktuelle, bei der Instantiierung der Klasse verwendete Typ-Parameter.

Es geht aber auch noch anders, einfacher:

Die `Class`-Methode `getGenericInterfaces` liefert einen Array von `Type`-Objekten zurück (`java.lang.reflect.Type`) – genauer: einen Array von Objekten, deren Klassen das Interface `Type` implementieren. In unserem Falle implementiert die anonyme Klasse nur ein einziges Interface: `Foo`. Also wird auch nur ein einziges `Type`-Objekt geliefert. Die `Type`-Referenz kann gecastet werden auf `ParameterizedType` (denn das Interface `Foo` ist ein parametrisierter Typ). Auf die `ParameterizedType`-Referenz kann die Methode `getActualTypeArguments` aufgerufen werden. Da `Foo` nur

ein einziges Typ-Argument hat, wird nur ein einziger `Type` zurückgeliefert – und dieser kann auf `Class` gecastet werden. Und das Resultat lautet: `String.class`!

Analyse einer generischen Lambda-Klasse

Wie sieht's aus, wenn `Foo<T>` nicht in Form einer anonymen, sondern in Form einer Lambda-Klasse implementiert ist?

```
    static void analyseLambda() throws Exception {
        Foo<String> f = value -> out.println(value);
        f.run("Hello");
        Class<?> cls = f.getClass();
        Features.print(cls);

        out.println("getGenericInterfaces...");
        Type[] ifaces = cls.getGenericInterfaces();
        for (Type iface : ifaces)
            out.println("\t" + iface);
        Type iface = ifaces[0];
        System.out.println(iface);
        // ParameterizedType pt = (ParameterizedType)iface; //
runtime error
    }
```

Dort gibt's offenbar keinen `ParameterizedType`. Hier die Ausgabe:

```
Hello
appl.Application$$Lambda$1/17699851 (null)
   Constructors
      private appl.Application$$Lambda$1/17699851()
   Methods
      public void appl.Application$
$Lambda$1/17699851.run(java.lang.Object)
      private final java.lang.Object appl.Application$
$Lambda$1/17699851.writeReplace()

getGenericInterfaces...
   interface appl.Foo
interface appl.Foo
```

Es gibt nur eine einzige `run`-Methode – und deren Parameter ist vom Typ `Object`.

Was also bei anonymen Klasse möglich ist, scheint bei Lambda-Klassen nicht zu funktionieren.

Es sei denn, man benutzt einen Trick:

Benutzung von SerializedLambda

```
    static void analyseSerializedLambda() throws Exception {
        Foo<String> f0 = value -> out.println(value);
        Class<? extends Foo> cls0 = f0.getClass();

        Method writeReplaceMethod =
cls0.getDeclaredMethod("writeReplace");
        writeReplaceMethod.setAccessible(true);
        SerializedLambda lambda =
            (SerializedLambda) writeReplaceMethod.invoke(f0);

        final String implClassName =
lambda.getImplClass().replace('/', '.');
        final Class<?> implClass = Class.forName(implClassName);
        Features.print(implClass);

        final String methodName = lambda.getImplMethodName();
        out.println(methodName);
        Method method = null;
        for (final Method m : implClass.getDeclaredMethods()) {
            if (m.getName().equals(methodName)) {
                method = m;
                break;
            }
        }
        Class<?> argType = method.getParameterTypes()[0];
        out.println(argType);
    }
```

Aufgrund eines Objekts einer Lambda-Klasse besorgen wird uns dessen serialisierte Form: wir besorgen uns ein `SerializedLambda`. Dieses Objekt enthält u.a. den Namen derjenigen Klasse, welche den Bytecode der Lambda-Klasse enthält (wir erinnern uns: der Bytecode ist in der `class`-Datei der umschließenden Klasse gespeichert – hier: der Klasse `Application`). Das `SerializedLambda` enthält zudem den Namen der `lambda$`...-Methode, die den Bytecode der Lambda-Methode enthält.

Aufgrund des Namens der äußeren Klasse können wird das `Class`-Objekt dieser Klasse ermitteln (hier: `Application.class`). Dann können wir in dieser Klasse nach einer Methode suchen, deren Namen dem im `SerializedLambda` enthaltenen Methodennamen gleicht. Wenn schließlich diese Methode gefunden haben, können wird den Typ des Parameters dieser Methode bestimmen. Und erhalten (in unserem Falle): `String.class`!

Hier die Ausgaben der obigen `demo`-Methode:

```
appl.Application (null)
   Constructors
      public appl.Application()
   Methods
      public static void appl.Application.main(java.lang.String[])
      // ...
      static void appl.Application.analyseSerializedLambda()
      static void appl.Application.analyseAnonymous()
      static void appl.Application.analyseLambda()
      static void appl.Application.demoLambdaUtil()
      private static void appl.Application.lambda$0(java.lang.String)
      private static void appl.Application.lambda$1(java.lang.String)
      private static void appl.Application.lambda$2(java.lang.String)

lambda$1
class java.lang.String
```

Die Utilitiy-Klasse LambdaUtil

Im `shared`-Projekt existiert eine Klasse `LambdaUtil`, welche auf Grundlage eines von einer generischen Lambda-Klasse stammenden Objekts die "Implementierung"-Methode liefert – und zwar derart, dass aus dem Parameter (resp. den Parametern) dieser Methode der aktuelle, bei der Instantiierung der Lambda-Klasse verwendete aktuelle Typ-Parameter (die aktuell verwendeten Typ-Parameter) ermittelt werden kann (können). Hier eine `demo`:

```
    static void demoLambdaUtil() throws Exception {
        Foo<String> f0 = value -> out.println(value);

        Method m = LambdaUtil.getMethod(f0);
        Class<?>[] argTypes = m.getParameterTypes();
        for (Class<?> argType : argTypes) {
            out.println("\t" + argType);
        }
        Class<?> argType = argTypes[0];
        out.println(argType);
    }
```

Die Ausgaben:

```
   class java.lang.String
class java.lang.String
```

Der Nutzen

Angenommen, wird definieren folgendes funktionale Interface (welches `Serializable` beerbt):

```
@FunctionalInterface
interface Consumer<T> extends Serializable {
    public abstract void consume(T value);
}
```

Ein `Consumer<X>` kann ein `X` konsumieren (und kann natürlich auch ein `Y` konsumieren – vorausgesetzt, `Y extends X`).

Angenommen, wir bauen einen Array, in welchen `String`-, `Integer`- und `Double`-Objekte herumliegen:

```
        final Object[] array = {
                3.14, 10, "Hello", 20, 2.71, "World" };
```

Angenommen weiterhin, wird bauen vier `Consumer` – der erste kann `Strings`, der zweite `Integers`, der dritte `Doubles` und der vierte `Numbers` jedweder Sorte konsumieren:

```
        final Consumer<String> stringConsumer = (v) ->
out.println("\t" + v);
        final Consumer<Integer> intConsumer = (v) ->
out.println("\t" + v);
        final Consumer<Double> doubleConsumer = (v) ->
out.println("\t" + v);
        final Consumer<Number> numberConsumer = (v) ->
out.println("\t" + v);
```

Dann wäre es schön, folgende Zeilen schreiben zu können:

```
        consume("all Strings", array, stringConsumer);
        consume("all Ints", array, intConsumer);
        consume("all Doubles", array, doubleConsumer);
        consume("all Numbers", array, numberConsumer);
```

Wir setzen also die Existenz einer `consume`-Methode voraus, welcher neben einer Überschrift ein `Object[]`-Array und ein beliebiger `Consumer` übergeben werden kann.

Wir verlangen von den vier `consume`-Aufrufen folgende Ausgabe:

```
all Strings
   Hello
```

```
   World
all Ints
   10
   20
all Doubles
   3.14
   2.71
all Numbers
   3.14
   10
   20
   2.71
```

Hier die `consume`-Methode (welche wieder die `LambdaUtil`-Klasse des `shared`-Projekts nutzt):

```
    public static void consume(String info,
            Object[] array, Consumer<?> consumer) {

        out.println(info);
        final Method m = LambdaUtil.getMethod(consumer);
        final Class<?> parameterType = m.getParameterTypes()[0];
        for (Object value : array) {
            if (parameterType.isAssignableFrom(value.getClass()))
{
                ((Consumer)consumer).consume(value);
            }
        }
    }
}
```

Resultate

Sei folgendes funktionales Interface gegeben:

```
@FunctionalInterface
public interface IFace<A,B> {
    public abstract void func(A a, B b);
}
```

Und sei folgende anonyme Klasse definiert und instantiiert:

```
    IFace<String,Integer> iface = new IFace<String,Integer>() {
        func(String s, Integer I) {
            ...
        }
```

```
    }
```

Dann können zur Laufzeit die aktuellen Typ-Parameter der anonymen Klasse ermittelt werden(also: `String` und `Integer`) – indem das Interface, welches dieser anonymen Klasse zugrunde liegt, als `ParameterizedType` betrachtet wird. Aus diesem können dann die `ActualTypeArgument`s ermittelt werden.

Bei einer Lambda-Implementierung ist dies nicht so leicht möglich. Wenn aber das Interface von `Serializable` erbt:

```
@FunctionalInterface
public interface IFace<A,B> implements Serializable {
    public abstract void func(A a, B b);
}
```

dann können auch hier die Typ-Parameter ermittelt werden.

Sei z.B. folgende Lambda-Klasse definiert und instantiiert:

```
    IFace<String,Integer> iface = (s, i) -> {
            ...
    };
```

Dann kann von dem "Lambda-Objekt" das `SerializedLamda`-Objekt ermittelt werden – und aufgrund dieses Objekts dann die Implementierungs-Methode. Diese Methode ist mit den tatsächlichen, aktuell übergebenen Typ-Parametern parametrisiert.

4.9 Fluent and typesafe Select-From-Where

Das folgende Beispiel ist ein erster Versuch des Autors dieses Skripts, so etwas wie LINQ (.NET) in Java zu implementieren – ein allererster Versuch. Im `util`-Package dieses Projekts sind eine Vielzahl von Klassen enthalten, die hier nicht weiter dargestellt werden können (sie arbeiten übrigens wieder mit den `SerializedLambda`s – ohne die geht's nicht; zudem wird der Bytecode mit ASM untersucht etc.). Wer's studieren will, soll's tun...

Wie gesagt: ein erster Versuch - das Resultat ist aber vielleicht bereits ganz schön.

Sei folgende `Book`-Klasse gegeben:

```
package appl;
// ...
public class Book {
    public String isbn;
    public String title;
    public int price;
}
```

Dann können wir folgende `Application` scheiben:

```
package appl;

import static util.Query.from;
import static util.Util.mlog;

public class Application0 {

    public static void main(String[] args) {
        demo1();
        demo2();
        demo3();
    }
    static void demo1() {
        from(Book.class)
                .select(b -> b.title)
                .where(b -> b.isbn == "2222")
                .log();
    }
    static void demo2() {
        from(Book.class)
                .select(b -> b.isbn, b -> b.price)
                .where(b -> b.isbn == "2222")
```

```
                .log();
    }
    static void demo3() {
        from(Book.class)
                .select(b -> b.isbn, b -> b.title, b -> b.price)
                .where(b -> b.price > 40)
                .log();
    }
}
```

Das Ganze ist typsicher – und liest sich natürlich sehr flüssig.

Die Ausgaben:

```
select title from Book where isbn = '2222'
select isbn, price from Book where isbn = '2222'
select isbn, title, price from Book where price >= 40
```

Wie man sieht, werden automatisch korrekte SQL-Strings produziert.

Man beachte aber, dass es sich um nichts anderes als um ein Experiment handelt!

4.10 Aufgaben

Studieren Sie die folgende Anwendung!

```
package ex1;
// ...
public interface Handler<T extends Component> extends
Serializable {
    public abstract void handle(T c);
}
```

```
package ex1;
// ...
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;

import util.LambdaUtil;

public class Traverser {
    @SuppressWarnings("unchecked")
    public static <T extends Component> void traverse(
            Component component, Handler<T> handler) {
        final Class<?> cls;
        if (handler.getClass().isSynthetic()) {
            final Method m = LambdaUtil.getMethod(handler);
            cls = m.getParameterTypes()[0];
        }
        else {
            final ParameterizedType pt =
                (ParameterizedType)handler.getClass()
                    .getGenericInterfaces()[0];
            cls = (Class<?>)pt.getActualTypeArguments()[0];
        }
        if (cls.isAssignableFrom(component.getClass()))
            handler.handle((T)component);
        if (component instanceof Container) {
            final Container container = (Container)component;
            for (int i = 0; i < container.getComponentCount(); i+
+) {
                traverse(container.getComponent(i), handler);
            }
        }
    }
}
```

```
package ex1;
// ...
import ex1.Handler;
import ex1.Traverser;

public class MyFrame extends Frame {
    private final Panel panelLeft = new Panel();
    private final Panel panelRight = new Panel();
    private final Button buttonHello = new Button("Hello");
    private final Button buttonWorld = new Button("World");
    private final TextField textFieldFoo = new TextField("Foo",
10);
    private final TextArea textAreaBar = new TextArea("World", 2,
10);

    public MyFrame() {
        this.setLayout(new FlowLayout());
        this.panelLeft.setLayout(new FlowLayout());
        this.panelRight.setLayout(new FlowLayout());
        this.add(this.panelLeft);
        this.add(this.panelRight);
        this.panelLeft.add(this.buttonHello);
        this.panelLeft.add(this.textFieldFoo);
        this.panelRight.add(this.buttonWorld);
        this.panelRight.add(this.textAreaBar);
        this.pack();
        this.setVisible(true);
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                MyFrame.this.dispose();
            }
        });

        this.buttonHello.addActionListener(
             e -> textComponentsToUpperCase());
        this.buttonWorld.addActionListener(
             e -> textComponentsToLowerCase());
    }

    private void textComponentsToUpperCase() {
        Traverser.traverse(this, new Handler<TextComponent>() {
            public void handle(TextComponent c) {
                c.setText(c.getText().toUpperCase());
            }
        });
    }
```

```
    private void textComponentsToLowerCase() {
        Traverser.traverse(this,
                (TextComponent tc) ->
tc.setText(tc.getText().toLowerCase()));
        //Traverser.<TextComponent> traverse(this,
        //        tc -> tc.setText(tc.getText().toLowerCase()));
    }
}
```

5 Interfaces

In Interfaces können mit Java 8 nicht nur abstrakte Methoden und statische Konstanten definiert werden, sondern zusätzlich auch statische Methoden und sog. `default`-Methoden. Java unterstützt damit das, was man als "mixin inheritance" bezeichnet.

Der Vorteil ist klar. Einem Interface eine weitere abstrakte Methode hinzuzufügen, würde bedeuten, dass alle dieses Interface nutzende Klienten ihrerseits erweitert werden müssten: sie müssten die neue abstrakte Methode implementieren. Ein Interface kann aber problemlos um bereits implementierte Funktionalität erweitert werden, ohne dass die bisherigen Klienten des Interfaces sich darum kümmern müssen. Und zukünftige Klienten können von diesen Erweiterungen profitieren.

Die Interfaces von APIs können also unter Beibehaltung der Rückwärtskompatibilität erweitert werden – ohne den Vertrag mit den diese Interfaces nutzenden Klienten zu brechen. Die in Interfaces implementierten Methoden können dabei natürlich die bereits vorhandenen abstrakten Methoden nutzten. Und das von `default`-Methoden beschriebene Standardverhalten schließlich kann von konkreten Klassen jederzeit überschrieben werden.

Der Nachteil ist aber ebenso klar: Bislang dienten Interfaces nur der Spezifikation (von der Möglichkeit der Definition statischer Konstanten einmal abgesehen). Dieser "saubere" Interface-Begriff wird nun verwässert. Die Schönheit von Interfaces, die auf ihrer vollständigen Abstraktheit beruhte, geht verloren.

Resultat: Man sollte die neuen Möglichkeiten nur mit Bedacht nutzen.

5.1 Start

Bislang konnten in einem Interface nur öffentliche abstrakte Methoden, öffentliche statische Konstanten und öffentliche statische innere Klassen definiert werden:

```
public interface Foo {

    int x = 42;
    final int y = 43;
    public static final int z = 44;

    void f();
    public void g();
    public abstract void h();

    class C { }
    public static class D { }
}
```

Die Variable `i` sieht zwar aus wie eine nicht öffentliche Instanzvariable, ist aber `static`, `public` und `final` (implizit). Dasselbe gilt auch für `y`. Die Definition von `z` ist die ausführlichste Definition – `x` und `y` sind implizit aber genauso definiert.

Auch bei der `f`-Definition fügt der Compiler die Modifizierer `public` und `abstract` automatisch hinzu; bei `g` wird `abstract` hinzugefügt. `f` und `g` haben also (implizit) exakt dieselben Modifizierer, die bei `h` explizit notiert sind.

Auch die hier definierte Klasse `C` ist `public` und `static` – genauso wie `D`.

Hier eine mögliche Implementierung des obigen Interfaces:

```
public class FooImpl implements Foo {
    public void f() {
        out.println("f()");
    }
    public void g() {
        out.println("g()");
    }
    public void h() {
        out.println("h()");
    }
}
```

Ein `FooImpl`-Objekt kann nun über eine `Foo`-Referenz genutzt werden:

```
    static void demo() {
        Foo foo = new FooImpl();
        out.println(Foo.x);
        out.println(Foo.y);
        out.println(Foo.z);
        foo.f();
        foo.g();
        foo.h();
    }
```

Soweit zum bisherigen Stand der Dinge.

5.2 Statische Methoden

In Java 8 kann ein Interface auch statische Methoden enthalten:

```
public interface Foo {
    static final int x = 42;
    static void printX() {
        out.println(x);
    }
    void f();
}
```

Statische Methoden eines Interfaces können natürlich auf statische Attribute dieses Interfaces zugreifen (oder eine andere statische Methode des Interfaces aufrufen).

Hier eine Implementierung des `Foo`-Interfaces:

```
public class FooImpl implements Foo {
    public void f() {
        out.println("f()");
    }
}
```

Um die `f`-Methode aufzurufen, benötigt man natürlich eine Instanz einer konkreten, das Interfaces implementierenden Klasse; `x` und `printX` können aber über den Namen des Interfaces angesprochen resp. aufgerufen werden:

```
    static void demo() {
        Foo foo = new FooImpl();
        out.println(Foo.x);
        Foo.printX();
        foo.f();
    }
```

5.3 Default-Methoden

Ein Interface kann in Java 8 auch Instanz-Methoden implementieren – vorausgesetzt, sie sind als `default` definiert und nicht `final` (`default` und `final` schließen sich natürlich aus gutem Grunde aus...):

```
public interface Foo {
    void f();
    default void g() {
        out.print("g");
        out.println("g()");
    }
    //  default final void h() {     // illegal
    //     out.println("h()");
    //  }
}
```

Semantisch gesehen sind `default`-Implementierungen ausdrücklich zum Überschreiben vorgesehen (eine konkrete Klasse könnte z.B. eine wesentliche performantere Implementierung anbieten - weil in ihr die konkrete Struktur der Daten bekannt ist, auf denen diese Methode operiert). Und dieser Semantik sollte man sich auch bewußt sein, wenn man eigene Interfaces mit solchen `default`-Methoden ausstattet.

Hier eine konkrete Implementierungsklasse:

```
public class FooImpl implements Foo {
    public void f() {
        out.println("f()");
    }
    @Override
    public void g() {
        out.println("gg()");
    }
}
```

Die `g`-Methode von `FooImpl` ist wahrscheinlich performanter als die `g`-Methode des Interfaces...

Eine Anwendung:

```
    static void demo() {
        Foo foo = new FooImpl();
        foo.f();
        foo.g();
```

```
    }
```

Hier wird natürlich die überschriebene `g`-Methode aufgerufen.

Ein weiteres, diesmal tatsächlich einigermaßen "sinnvolles" Interface – das Interface `YAC` ("Yet another Comparator"):

```
public interface YAC<T> {

    public abstract boolean eq(T v0, T v1);

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

    public default boolean ge(T v0, T v1) {
        return this.gt(v0, v1) || this.eq(v0, v1);
    }

    public default boolean lt(T v0, T v1) {
        return ! this.ge(v0, v1);
    }

    public default boolean le(T v0, T v1) {
        return this.eq(v0, v1) || this.lt(v0, v1);
    }
}
```

Eine von `YAC` abgeleitete instantiierbare Klasse muss nur zwei Methoden implementieren: `eq` und `gt`. Die drei weiteren Methoden des Interfaces werden alle auf diese beiden Methoden zurückgeführt.

Ein konkreter `YAC` zum Vergleich von `Integer`-Objekten:

```
        YAC<Integer> yac = new YAC<Integer>() {
            public boolean eq(Integer v0, Integer v1) {
                return v0.equals(v1);
            }
            public boolean gt(Integer v0, Integer v1) {
                return v0.compareTo(v1) > 0;
            }
        };
```

Alle folgenden Aufrufe der `yac`-Methoden liefern `true`:

```
    static void demoYAC() {
        out.println(yac.eq(1, 1));
```

```
        out.println(yac.gt(2, 1));
        out.println(yac.ge(1, 1));
        out.println(yac.ge(2, 1));
        out.println(yac.lt(1, 2));
        out.println(yac.le(1, 2));
        out.println(yac.le(1, 1));
    }
```

5.4 Konflikte

Was passiert, wenn zwei Interfaces Methode definieren, welche dieselbe Signatur haben – und eine Klasse dennoch beide Interfaces implementieren möchte? Solche Probleme gab's auch bereits im "alten" Java:

```
public interface Foo {
    public abstact void f();
}
```

```
public interface Bar {
    public abstact void f();
}
```

`Foo` und `Bar` spezifizieren beide eine parameterlose `f`-Methode vom Typ `void`. Eine Klasse, die beide Interfaces implementiert, kann natürlich nur eine einzige `f`-Methode enthalten:

```
public class FooBar implements Foo, Bar {
    public void f() { ... }
}
```

Die Lösung war nie so richtig zufriedenstellend (in C# z.B. kann es dagegen für jede der beiden `f`-Methoden eine eigene Implementierung geben). Dieses Manko des "alten" Java konnte natürlich nicht beseitigt werden. Man könnte das Problem mit dem Hinweis abtun, solche Schwierigkeiten seien rein akademischer Natur und würden in der Praxis nicht auftreten (wer definiert schon eine Methode namens `f`???)

Bei den neuen `default`-Methoden hat man nun aber solche möglichen Konflikte sauber gelöst:

Sei sowohl in `Foo` und `Bar` die `default`-Methode `f` definiert:

```
public interface Foo {
    public default void f() {
        out.println("Foo.f");
    }
}
```

```
public interface Bar {
    public default void f() {
        out.println("Bar.f");
    }
}
```

Man möchte nun eine Klasse `FooBar` bauen, die beide Interfaces implementiert.

Folgende "Lösung" weist der Compiler zurück:

```
public class FooBar implements Foo, Bar {
}
```

Man kann aber in der Klasse eine eigene `f`-Methode implementieren:

```
public class FooBar1 implements Foo, Bar {
    public void f() {
        out.println("FooBar1.f()");
    }
}
```

Egal, ob ein `FooBar1`-Objekt nun über eine `Foo`- oder über ein `Bar`-Referenz angesprochen wird – es wird immer die eine `f`-Methode von `FooBar1` aufgerufen werden. Die `default`-Methoden der Interfaces werden also überhaupt nicht aufgerufen.

In einer Methode der Implementierungsklasse können dann aber wieder die `default`-Implementierungen der Interfaces aufgerufen werden – über die Notation `<Interface>.super.<Methode>`:

```
public class FooBar2 implements Foo, Bar {
    public void f() {
        Foo.super.f();
        Bar.super.f();
    }
}
```

Eine Anwendung:

```
    static void demo() {
        Foo foo = new FooBar1();
        Bar bar = new FooBar2();
        foo.f();
        bar.f();
    }
```

Die Ausgaben:

```
FooBar1.f()
Foo.f
Bar.f
```

5.5 Fluent Programming

Im nächsten Kapitel werden die neuen funktionalen Interfaces der Standardbibliothek vorgestellt werden. Diese ermöglichen das, was man als "fluent programming" bezeichnet. Dabei wird eine Technik benutzt, die auf den ersten Blick recht unverständlich ist. Diese Technik wird im folgenden näher beleuchtet (sie benutzt funktionale Interfaces mit `default`-Methoden).

Die folgende Methode verwendet eine Klasse `WorkerC`:

```
    static void demoCSimple() {
        WorkerC w1 = v -> out.println("w1 : " + v);
        WorkerC w2 = v -> out.println("w2 : " + v);
        w1.andThen(w2).work(42);
    }
```

Die Ausgaben:

```
w1 : 42
w2 : 42
```

Man könnte die Methode erweitern:

```
    static void demoCSimple() {
        WorkerC w1 = v -> out.println("w1 : " + v);
        WorkerC w2 = v -> out.println("w2 : " + v);
        WorkerC w3 = v -> out.println("w3 : " + v);
        w1.andThen(w2).andThen(w3).work(42);
    }
```

Die Ausgaben:

```
w1 : 42
w2 : 42
w3 : 42
```

(Die letzte Zeile der beiden obigen Code-Blöcke kann als "flüssiger Text" angesehen werden – daher der Name "fluent programming".)

Die Lösung sieht wie folgt aus:

```
@FunctionalInterface
public interface WorkerC {

    public abstract void work(int value);
```

```
    public default WorkerC andThen(WorkerC other) {
        return v -> {
            work(v);
            other.work(v);
        };
    }
}
```

Was passiert in `andThen`? In `andThen` findet kein einziger Methodenaufruf statt – stattdessen wird ein neues Objekt erzeugt und zurückgeliefert.

Formulieren wir die obige Methode zunächst einmal etwas um:

```
    static void demoCSimple() {
        WorkerC w1 = v -> out.println("w1 : " + v);
        WorkerC w2 = v -> out.println("w2 : " + v);
        WorkerC w3 = w1.andThen(w2);
        w3.work(42);
    }
```

Wie leicht gezeigt werden könnte, wird in `andThen` ein neues `WorkerC`-Objekt erzeugt – auf welches dann `work` aufgerufen wird. Die eigentliche Arbeit (das `out.println`) verrichten aber natürlich die `w1`- und `w2`-Worker.

Um die Funktionsweise der `andThen`-Methode zu verstehen, definieren wir zunächst einmal ein Interface `WorkerA`, die ähnlich genutzt werden wie `WorkerC`:

```
public interface WorkerA {

    public abstract void work(int value);

    public default WorkerA andThen(final WorkerA other) {
        return new Combiner(this, other);
    }
}
```

Die `andThen`-Methode erzeugt einen neuen `Combiner` und liefert diesen als `WorkerA` zurück. `Combiner` muss also `WorkerA` implementieren:

```
public class Combiner implements WorkerA {
    private final WorkerA first;
    private final WorkerA second;
    public Combiner(WorkerA first, WorkerA second) {
        this.first = first;
        this.second = second;
```

```
    }
    @Override
    public void work(int value) {
        this.first.work(value);
        this.second.work(value);
    }
}
```

`Combiner` ist eine instantiierbare Klasse. Ein `Combiner` dient dazu, zwei `WorkerA` zu kombinieren. Dem Konstruktor werden die zu kombinierenden `WorkerA`-Objekte übergeben, deren Referenzen in den Instanzvariablen `first` und `second` gespeichert werden. In der `work`-Methode wird dann zunächst die `work`-Methode den ersten `WorkerA` und dann die `work`-Methode des zweiten `WorkerA` aufgerufen.

Wie können diese Klassen nun wie folgt nutzen:

```
    static void demoA() {
        WorkerA w1 = new WorkerA() {
            public void work(int v) {
                out.println("w1 : " + v);
            }
        };
        WorkerA w2 = new WorkerA() {
            public void work(int v) {
                out.println("w2 : " + v);
            }
        };
        WorkerA w3 = w1.andThen(w2);
        w3.work(42);
    }
```

Oder kürzer:

```
    static void demoA() {
        WorkerA w1 = v -> out.println("w1 : " + v);
        WorkerA w2 = v -> out.println("w2 : " + v);
        WorkerA w3 = w1.andThen(w2);
        w3.work(42);
    }
```

In der `andThen`-Methode von `WorkerA` wurde ein Objekt der globalen `Combiner`-Klasse erzeugt. Man könnte hier natürlich auch ein Objekte einer anonymen Klasse erzeugen.

Bauen wird also ein Interface `WorkerB`:

```
@FunctionalInterface
```

```
public interface WorkerB {

    public abstract void work(int value);

    public default WorkerB andThen(final WorkerB other) {
        return new WorkerB() {
            public void work(int v) {
                WorkerB.this.work(v);
                other.work(v);
            }
        };
    }
}
```

Die globale `Combiner`-Klasse ist nun migriert zu einer anonymen Klasse der `work`-Methode.

Wo sind die `first`- und `second`-Referenzen geblieben? Die Rolle der `first`-Referenz hat nun `Worker.this` übernommen; und der Wert des Parameters `other` ist in das erzeugte Objekt der anonymen Klasse hineinkopiert worden (weil `other` in `work` angesprochen wird). Die Rolle der `second`-Referenz hat also die `other`-Kopie übernommen.

Auch `WorkerB` funktioniert nun erwartungsgemäß.

Der letzte Schritt – die Klasse `WorkerC`.

Statt eines Objekts einer anonymen, von `WorkerB` abgeleiteten Klasse zu returnieren, wird ein Objekt returniert, welches über einen Lambda-Ausdruck erzeugt wird (und dessen Klasse natürlich `WorkerC` implementiert):

```
@FunctionalInterface
public interface WorkerC {

    public abstract void work(int value);

    public default WorkerC andThen(WorkerC other) {
        return v -> {
            this.work(v);
            other.work(v);
        };
    }
}
```

Die Rolle, die in der letzten Lösung `WorkerC.this` spielte, spielt hier `this`. (Wobei man auf die explizite `this`-Angabe natürlich auch hätte verzeichten können.) Der Wert des `other`-Parameters von `andThan` ist auch hier in das "Lambda-Objekt" hineinkopiert worden. Auch das hier erzeugte Objekt enthält somit Referenzen auf zwei `WorkerC`-Objekte.

Eine kurze Analyse dieser Lösung (der Name der umschließenden Klasse ist `Application`):

```
    static void demoC() {
        mlog();
        WorkerC w1 = v -> out.println("w1 : " + v);
        WorkerC w2 = v -> out.println("w2 : " + v);
        WorkerC w3 = w1.andThen(w2);
        out.println(w1);
        out.println(w2);
        out.println(w3);
        w3.work(42);
        // ...
    }
```

Die Ausgaben zeigen, dass drei verschiedene Objekte existieren – zwei davon sind in der umschließenden Klasse `Application` erzeugt worden und das dritte im Kontext des `WorkerC`-Interfaces:

```
appl.Application$$Lambda$1/20112757@9ee92
appl.Application$$Lambda$2/11505757@f39991
appl.WorkerC$$Lambda$3/15710278@12b3a41
```

Wir geben die Features von `w3` aus (von dem "Kombinations-Objekt"):

```
        Features.print(w3.getClass());
```

```
appl.WorkerC$$Lambda$3/... (null)
   Constructors
      private appl.WorkerC$$Lambda$3/...(appl.WorkerC,appl.WorkerC)
   Fields
      private final appl.WorkerC appl.WorkerC$$Lambda$3/...arg$1
      private final appl.WorkerC appl.WorkerC$$Lambda$3/...arg$2
   Methods
      public void appl.WorkerC$$Lambda$3/15710278.work(int)
      // ...
```

Man erkennt: der Konstruktor hat zwei `WorkerC`-Parameter, die an die Felder `arg$1` und `arg$2` zugewiesen werden.

Wir benutzen schließlich eine kleine Helper-Methode `readField`, um zu zeigen, dass `arg$1` tatsächlich auf das via `w1` referenzierte `WorkerC`-Objekt zeigt und `arg$2` auf das via `w2` referenzierte Objekt:

```
        out.println(readField(w3, "arg$1") == w1);
        out.println(readField(w3, "arg$2") == w2);
```

Die Ausgaben:

```
true
true
```

Hier die kleine Helper-Methode:

```
    static Object readField(Object obj, String name) {
        try {
            final Field field =
obj.getClass().getDeclaredField(name);
            field.setAccessible(true);
            return field.get(obj);
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

Der Mechanismus, der "fluent programming" ermöglicht, besteht also darin, bei jedem Punkt ein neues(!) Objekt zu erzeugen und dieses zurückzuliefern...

5.6 Default-Methoden und Dynamic Proxy

Können Dynamic-Proxies auch für solche Interfaces generiert werden, welche `default`-Methoden besitzen?

Wir verwenden zur Demonstration folgende `InvocationHandler`-Klasse:

```
package appl;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.util.Arrays;

public class TraceHandler implements InvocationHandler {
    final Object target;
    public TraceHandler(Object target) {
        this.target = target;
    }
    public Object invoke(Object proxy, Method method, Object[]
args)
                                                throws Throwable
{
        // quick and dirty...
        System.out.println("--> " + method.getName() + " " +
            Arrays.toString(args));
        final Object result = method.invoke(this.target, args);
        System.out.println("<-- " + method.getName() + " --> " +
            result);
        return result;
    }

};
```

Sei nun folgendes Interface gegeben:

```
package appl;

public interface MathService {
    public abstract double sum(double x, double y);
    public default double diff(double x, double y) {
        return this.sum(x, -y);
    }
}
```

Und folgende Implementierung:

```
package appl;

public class MathServiceImpl implements MathService {
    public double sum(double x, double y) {
        return x + y;
    }
}
```

Wir bauen eine kleine Demo-Applikation:

```
    static void demo() {

        final MathService m1 = new MathServiceImpl();
        out.println(m1.sum(40, 2));
        out.println(m1.diff(80, 3));
        out.println();

        final MathService m2 =
(MathService)Proxy.newProxyInstance(
                ClassLoader.getSystemClassLoader(),
                new Class<?>[] { MathService.class },
                new TraceHandler(m1));
        out.println(m2.sum(40, 2));
        out.println(m2.diff(80, 3));
        out.println();
    }
```

Alles funktioniert erwartungsgemäß:

```
42.0
77.0

--> sum [40.0, 2.0]
<-- sum --> 42.0
42.0
--> diff [80.0, 3.0]
<-- diff --> 77.0
77.0
```

5.7 Aufgaben

Interfaces - 1

Gegeben ist wieder folgende einfache `Array`-Klasse:

```
package ex1;

import java.util.Arrays;

public class Array<T> {
    @SuppressWarnings("unchecked")
    private T[] elements = (T[]) new Object[2];
    private int size;

    public void add(T element) {
        this.ensureCapcity();
        this.elements[this.size] = element;
        this.size++;
    }
    public int size() {
        return this.size;
    }

    public T get(int index) {
        if (index < 0 || index >= this.size)
            throw new IndexOutOfBoundsException();
        return this.elements[index];
    }
    private void ensureCapcity() {
        if (this.elements.length == size) {
            this.elements = Arrays.copyOf(elements, this.size *
2);
        }
    }
}
```

Erweitern Sie die Klasse derart, dass sie das Interface `Iterable` implementiert. Der Kopf der Klasse soll also wie folgt aussehen:

```
public class Array<T> implements Iterable<T>
```

Was fällt Ihnen am Interface `Iterator` auf?

Interfaces - 2

Es existiert folgende Klasse:

```
package ex2;
// ...
public abstract class Processor {
    public final void run(Reader reader) {
        try(final Reader r = reader) {
            this.begin();
            for (int ch = r.read(); ch != -1; ch = r.read())
                this.process((char)ch);
            this.end();
        }
        catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
    protected void begin() {
    }
    protected abstract void process(char ch);
    protected void end() {
    }
}
```

Die Klasse ist gemäß des Template-Method-Pattern aufgebaut. Die template-Methode `run` ruft die hook-Methoden `begin`, `process` und `end` auf. `begin` und `end` besitzen bereits eine Implementierung – `process` dagegen ist `abstract`. Der Grund für diesen Unterschied wird aus der folgenden Benutzung deutlich:

```
package ex2;
// ...
public class Application {
    static class PrintProcessor extends Processor {
        @Override
        protected void process(char ch) {
            System.out.print(ch);
        }
    }
    static class CharCountProcessor extends Processor {
        private int count = 0;
        @Override
        protected void begin() {
            this.count = 0;
        }
        @Override
```

```
        protected void process(char ch) {
            this.count++;
        }
        @Override
        protected void end() {
            System.out.println(this.count);
        }
    }
    public static void main(String[] args) {
        PrintProcessor p1 = new PrintProcessor();
        p1.run(new StringReader("hello"));
        System.out.println();

        CharCountProcessor p2 = new CharCountProcessor();
        p2.run(new StringReader("world"));
        System.out.println();
    }
}
```

Der `PrintProcessor` muss nur `process` implementieren (es wäre nervig, wenn er auch `begin` und `end` implementieren müsste). Der `CharCountProcessor` aber überschreibt auch `begin` und `end`.

Zerlegen Sie die obige `Processor`-Klasse in zwei Teile – in die Klasse `ProcessorRunner` und `Processor`. Die Klasse `ProcessorRunner` enthält nur die `run`-Methode; diese delegiert an ein Objekt, dessen Klasse das Interface `Processor` implementiert. Dieses hat die Methoden `begin`, `process` und `end`. Die Methoden `begin` und `end` sollten dann bereits defaultmäßig implementiert sein. Und bringen Sie natürlich die Anwendung wieder zum Laufen...

6 Neue funktionale Interfaces

Das Paket `java.util.function` enthält eine Reihe von neuen funktionalen Interfaces – Interfaces, die also als Target-Types von Lambdas genutzt werden können. Neben dem altbekannten Interface `java.lang.Runnable` existieren nun folgende grundlegende Interfaces:

- `Supplier`
- `Function`
- `BiFunction`
- `Consumer`

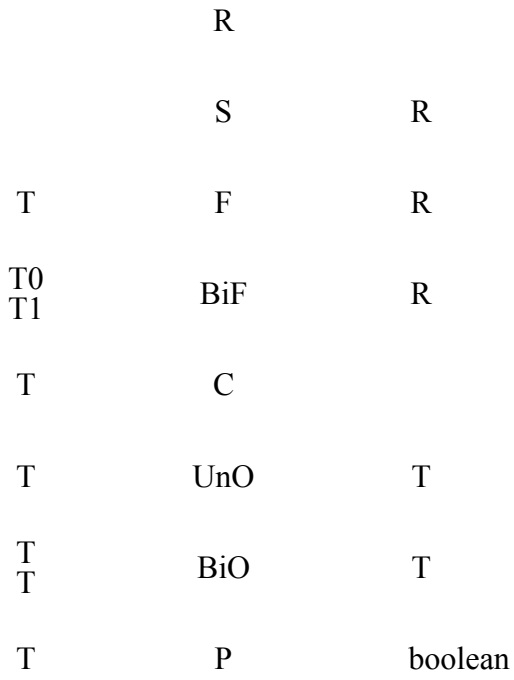
Von `Function` und `BiFunction` sind noch wiederum weitere Interfaces abgeleietet:

- `UnaryOperator` (abgeleitet von `Function`)
- `BinaryOperator` (abgeleiet von `BiFunction`)

Und schließlich gibt's noch das Interface

- `Predicate`.

Hier eine kleine schematische Übersicht zu der Bedeutung dieser Interfaces:



- Ein `Runnable` (`R`) hat weder einen Input noch einen Output.
- Ein `Supplier` (`S`) liefert einen Output.
- Eine `Function` (`F`) transformiert einen Input zu einen Output. Input und Output können unterschiedlichen Typs sein.
- Eine `BiFunction` (`BiF`) transformiert zwei Inputs zu einen Output. Die beiden Inputs können unterschiedlichen Typs sein.
- Ein `Consumer` (`C`) konsumiert einen Input (liefert aber keinen Output).
- Ein `UnaryOperator` (`UnO`) transformiert einen Input zu einem Output. Input und Output sind vom selben Typ.
- Ein `BinaryOperator` (`BiO`) transformiert zwei Inputs zu einem Output. Inputs und Output sind vom selben Typ.
- Ein `Perdicate` (`P`) schließlich transformiert einen Input zu einem Output vom Typ `boolean`.

(Natürlich gab es immer schon das Bedürfnis nach solchen Interfaces – vor Java 8 hat man sie allerdings selbst schreiben müssen...)

Im folgenden wird gezeigt werden, wie diese Interfaces definiert sind und wie sie genutzt werden können.

Insbesondere wird im letzten Abschnitt gezeigt, wie Implementierungen dieser Interfaces kombiniert werden können zur Spezifikation komplexer Abläufe – von Folgen von Berechnungen, von denen einige auch parallel ablaufen können (ein kleiner Vorgriff auf die Diskussion einer neuen Klasse im `concurrent`-Paket: `CompletableFuture`...)

Diese neuen Interfaces von Java 8 machen extensiv Gebrauch von Generics – insbesondere von Parametertypen der Form `X<? extends T>` und `X<? super T>`. Deshalb beginnen wir mit einem kleinen Exkurs zur Bedeutung solcher Parametertypen. (Dieser erste Abschnitt kann übersprungen werden, wenn diese Typen bereits sicher beherrscht werden.)

6.1 Exkurs: Typ-Parameter

Seien drei Klassen gegeben: `A`, `B` und `C` – wobei `B` von `A` und `C` von `B` abgeleitet sind:

```
public class A {
    public final int x;
    public A(int x) {
        this.x = x;
    }
}
```

```
public class B extends A {
    public final int y;
    public B(int x, int y) {
        super(x);
        this.y = y;
    }
}
```

```
public class C extends B {
    public int z;
    public C(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }
}
```

Ein `A` hat ein `x`; ein `B` hat zusätzlich ein `y`; und ein `C`-Objekt hat zusätzlich ein `z`.

Angenommen, wir wollen solche Objekte in einer `Box` verpacken. Morgen sollen in einer solchen `Box` natürlich noch andere Dinge verpackt werden. Dann bietet es sich an, eine generische Klasse zu definieren:

```
public class Box<T> {
    private T value;
    public Box(T value) {
        this.value = value;
    }
    public void set(T value) {
        this.value = value;
    }
    public T get() {
        return this.value;
    }
}
```

Bei Erzeugen einer `Box` muss bereits ein Inhalt (ein `T`) übergeben werden; der Inhalt kann ausgelesen werden (`get`); und eine `Box` kann einen neuen Inhalt bekommen (`set`).

Angenommen, wir erzeugen nun drei Schachteln – die erste enthält ein `A`, die zweite ein `B` und die dritte ein `C`:

```
        Box<A> ba = new Box<>(new A(1));
        Box<B> bb = new Box<>(new B(1, 2));
        Box<C> bc = new Box<>(new C(1, 2, 3));
```

Angenommen, wir möchten eine Methode schreiben, welcher sowohl eine `Box<A>`, eine `Box<B>` und eine `Box<C>` übergeben werden können. Ein Versuch:

```
    static void tuWas(Box<A> box) {
        // ...
    }
```

Man kann dieser Methode zwar eine `Box<A>` übergeben:

```
    tuWas(ba)
```

Nicht aber die `bb` und `bc`-Box:

```
    tuWas(bb);  // illegal
    tuWas(bc);  // illegal
```

M.a.W.: `B` ist zwar kompatibel (zuweisbar) zu `A`, `Box<B>` aber nicht zu `Box<A>`:

```
   A        <---     B
   A        <---     C
   Box<A>   <-/-     Box<B>
   Box<A>   <-/-     Box<C>
```

Man kann nun aber eine Methode mit einem Parameter des Typs `Box<? extends A>` schreiben:

```
    static void extendsA(Box<? extends A> box) {
        A a = box.get();
        out.println(a);
        //box.set(new A(10));         // illegal
        //box.set(new B(10,20));      // illegal
        //box.set(new C(10,20,30));   // illegal
        box.set(null);                // the only way to call
set...
    }
```

Frage: warum kann zwar die `get`-Methode problemlos aufgerufen werden, nicht aber die `set`-Methode (bzw. diese nur mit `null`)? (Das hat seinen Sinn!)

Dieser Methode können sowohl `ba`, `bb` als auch `bc` übergeben werden:

```
        extendsA(ba);
        extendsA(bb);
        extendsA(bc);
```

Es gilt also:

```
   Box<? extends A>     <---     Box<A>
   Box<? extends A>     <---     Box<B>
   Box<? extends A>     <---     Box<C>
```

Eine weitere Methode:

```
    static void extendsB(Box<? extends B> box) {
        B b = box.get();
        out.println(b);
        //box.set(new A(10));         // illegal
        //box.set(new B(10,20));      // illegal
        //box.set(new C(10,20,30));   // illegal
        box.set(null);                // the only way to call
set...
    }
```

(Auch hier ist ein "sinnvoller" Aufruf der `set`-Methode nicht zulässig.)

Dieser können die `bb`- und `bc`-Schachteln, nicht aber die `ba`-Schachtel übergeben werden:

```
        // extendsB(ba); // illgeal
        extendsB(bb);
        extendsB(bc);
```

Und eine letzte Methode:

```
    static void extendsC(Box<? extends C> box) {
        C c = box.get();
        out.println(c);
        //box.set(new A(10));         // illegal
        //box.set(new B(10,20));      // illegal
        //box.set(new C(10,20,30));   // illegal
```

```
        box.set(null);                // the only way to call
set...
    }
```

(Auch hier funktioniert `set` nicht.)

Dieser Methode kann nurmehr `bc` übergeben werden:

```
        //extendsC(ba);
        //extendsC(bb);
        extendsC(bc);
```

Die drei `exends...`-Methoden haben einen sog. "kovarianten" Parameter.

`super` ist invers zu `extends`. Wir bauen drei Methoden mit "kontravariantem" Parameter.

Hier die erste dieser Methoden:

```
    static void superA(Box<? super A> box) {
        Object o = box.get();
        out.println(o);
        box.set(new A(1));
        box.set(new B(1, 2));
        box.set(new C(1, 2, 3));
    }
```

Der Parameter ist vom Typ `Box<? super A>`. Der Aufruf von `get` liefert "maximal" `Object`. Die `set`-Methode ist aufrufbar mit einer `A`-, mit einer `B`- und mit einer `C`-Referenz.

Wie kann die Methode aufgerufen werden? Ihr kann nur eine `Box<A>` übergeben werden:

```
        superA(ba);
        //superA(bb);
        //superA(bc);
```

Eine zweite `super`-Methode:

```
    static void superB(Box<? super B> box) {
        Object o = box.get();
        out.println(o);
        //box.set(new A(1));
        box.set(new B(1, 2));
        box.set(new C(1, 2, 3));
```

```
    }
```

Diese Methode kann die `set`-Methode nur noch mit einem `B` oder einem `C` aufrufen. `get` liefert ebenfalls wieder "maximal" `Object`.

Wie kann `superB` aufgerufen werden?

```
        superB(ba);
        superB(bb);
        // superB(bc);
```

Beim Aufruf kann eine `Box<A>` oder eine `Box<B>` übergeben werden.

Und schließlich die letzte Methode:

```
    static void superC(Box<? super C> box) {
        Object o = box.get();
        out.println(o);
        //box.set(new A(1));
        //box.set(new B(1, 2));
        box.set(new C(1, 2, 3));
    }
```

Die Methode kann an `set` nur noch ein `C` übergeben. Und `get` liefert weiterhin nur `Object`.

An `superC` können alle Schachteln übergeben werden.

```
        superC(ba);
        superC(bb);
        superC(bc);
```

Man erkennt die "Symmetrie".

Regel:

Hat eine Methode einen Parameter `p` vom Typ `P<? extends X>`, so kann die Methode auf `p` nur solche Methoden aufrufen, die keinen Parameter vom Typ `X` verlangen (kann also keine "Setter" aufrufen). Sie kann aber Methoden aufrufen, die ein `X` zurückliefern (kann also "Getter" aufrufen).

Hat eine Methode einen Parameter `p` vom Typ `P<? super X>`, so kann sie Methoden auf `p` aufrufen, deren Parameter vom Typ `X` ist ("Setter").´Methoden von `P`, die ein `X` liefern, können zwar aufgerufen werden, liefern aber nur `Object`.

Kürzer (und ungenauer): Bei `extends` darf man nur lesen, aber nicht schreiben. Bei `super` darf man schreiben, aber nicht lesen.

Was muss der Aufrufer beachten?:

Einer Methode mit einem `extends`-Parameter kann man "mehr" übergeben, als sie verlangt – aber nicht "weniger". Einer Methode mit einem `super`-Parameter kann man "weniger" übergeben, als sie verlangt – aber nicht "mehr".

Nun zum eigentlichen Thema.

6.2 Supplier

Ein `Supplier` hat was anzubieten – stellt irgendetwas zur Verfügung:

```
package java.util.function;

@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

Ein `Supplier` könnte den Wert `42` zur Verfügung stellen.

Hier ein Beispiel mit einer anonymen Klasse:

```
    static void demoSupplier1() {
        Supplier<Integer> s = new Supplier<Integer>() {
            public Integer get() {
                return 42;
            }
        };
        int v = s.get();
        out.println(v);
    }
```

Als Ausgabe wird `42` erscheinen. Der `Supplier` ist natürlich nicht sonderlich intelligent. Bei jedem `get`-Aufruf wird er stets dieselbe magische Zahl liefern. Ein `Supplier` könnte aber natürlich auch intelligenter sein...

Hier ein äquivalenter `Supplier` in Form eines Lambda-Ausdrucks

```
    static void demoSupplier2() {
        Supplier<Integer> s = () -> 42;
        int v = s.get();
        out.println(v);
    }
```

Supplier ist mit `T` parametrisiert. `T` ist natürlich immer ein Referenztyp. Was bedeutet: in den obigen Lösungen wird immer geboxt (und un-geboxt): `int` zu `Integer`, `Integer` zu `int`. Deshalb gibt's für einige primitive Typen Spezialvarianten des Interfaces.

Z.B. das nicht-generische Interace `IntSupplier`, dessen `get`-Metode `int` liefert:

```
    static void demoIntSupplier() {
```

```
        IntSupplier s = new IntSupplier() {
            public int getAsInt() {
                return 42;
            }
        };
        int v = s.getAsInt();
        out.println(v);
    }
```

```
    static void demoIntSupplier() {
        IntSupplier s = () -> 42;
        int v = s.getAsInt();
        out.println(v);
    }
```

Man beachte den Namen der get-Methode: `getAsInt`.

Neben `IntSupplier` gibt's noch `DoubleSupplier` und `LongSupplier`. Alle anderen primitiven Typen können als Spezialfälle von `int`, `long` und `double` gelten (`byte`, `short` und `char` können auf `int` abgebildet werden, `float` auf `double`). Solche Typ-Konvertierungen nimmt der Compiler automatisch vor. Und beim `Boolean`–Typ gibt's kein Problem – es gibt nur zwei Werte (sollte nur diese beiden geben): `Boolean.TRUE` und `Boolean.FALSE`.

Hier ein etwas intelligenterer `IntSupplier`:

```
    static void demoIntRangeSupplier() {
        IntSupplier s = new IntSupplier() {
            int n = 0;
            public int getAsInt() {
                return n == 10 ? 0 : ++n;
            }
        };
        for (int v = s.getAsInt(); v != 0; v = s.getAsInt())
            System.out.print(v + " ");
        System.out.println();
    }
```

Die Ausgaben:

```
1 2 3 4 5 6 7 8 9 10
```

6.3 Consumer

Ein `Consumer` ist das genaue Gegenstück zu einem `Supplier`.

Ein `Cosnumer` kann konsumieren – er akzeptiert das, was ihm zur Konsumption vorgelegt wird. Wenn er etwas konsumiert hat, könnte er das, was übrigbleibt, an einen weiteren `Consumer` weiterreichen:

```
package java.util.function;

@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

Man beachte dass `andThen` bereits implementiert ist (trotzdem ist es ein funktionales Interface: es hat genau eine SAM).

Der folgende `Consumer` konsumiert, indem er das zu konsumierende Objekt (ein `Integer`-Objekt) ausspuckt. Wir füttern ihn mit `42`:

```
    static void demoConsumer() {
        Consumer<Integer> c = v -> System.out.println(v);
        c.accept(42);
    }
```

`Supplier` und `Consumer` sind invers zueinander – aber gerade deshalb können sie auch zusammenspielen:

```
    static void demoSupplierConsumer() {
        Supplier<Integer> supplier = () -> 42;
        Consumer<Integer> consumer = v -> System.out.println(v);
        consumer.accept(supplier.get());
    }
```

Was der `Supplier` anbietet (natürlich `42`), wird an den `Consumer` zur Konsumption weitergereicht. Letzterer wird dann die `42` ausgeben...

Das klingt schon sehr nach Pipeline-Verarbeitung...

Wir bauen also eine `pipe`-Methode:

```
    static <T> void pipe(Supplier<? extends T> s, Consumer<?
super T> c) {
        c.accept(s.get());
    }
```

Man beachte, dass ser `Supplier`-Parameter kovariant ist (`extends`), der `Consumer`-Parameter umgekehrt aber kontravariant ist (`super`).

Und rufen sie wie folgt auf:

```
    static void demoPipe() {
        Supplier<Integer> supplier = () -> 42;
        Consumer<Integer> consumer = v -> System.out.println(v);
        pipe(supplier, consumer);
    }
```

Seien wieder die in der Einleitung vorgestellten Klassen `A`, `B` und `C` gegeben (wobei `B` von `A` und `C` von `B` abgeleitet sind).

Wenn ein `Supplier` verspricht, ein `C` bereitzustellen, und wenn der `Consumer` ein solches `C` verlangt – dann können beide natürlich problemlos via `pipe` zusammengebracht werden (beide sind auf gleicher Augenhöhe):

```
    static void demoPipeCC() {
        Supplier<C> supplier = () -> new C(1, 2, 3);
        Consumer<C> consumer = c -> System.out.println(c.x + c.y
+ c.z);
        pipe(supplier, consumer);
    }
```

Angenommen, der `Supplier` stellt wieder ein `C` bereit – und der `Consumer` ist mit jedem `A` zufrieden (ein anspruchsloser `Consumer`) : auch dann ist die Sache problemlos:

```
    static void demoPipeCA() {
        Supplier<C> supplier = () -> new C(1, 2, 3);
        Consumer<A> consumer = a -> System.out.println(a.x);
        pipe(supplier, consumer);
    }
```

Der `Consumer` betrachtet das ihm übergebene `C`-Objekt natürlich nur als `a` – und gibt nur den Wert des `x`-Attributs aus (also 1).

Und der `Consumer` kann natürlich auch ein `B` verlangen:

```
    static void demoPipeCB() {
        Supplier<C> supplier = () -> new C(1, 2, 3);
        Consumer<B> consumer = b -> System.out.println(b.x +
b.y);
        pipe(supplier, consumer);
    }
```

Stellt der `Supplier` nun ein `B` zur Verfügung, darf der `Consumer` natürlich `A` verlangen:

```
    static void demoPipeBA() {
        Supplier<B> supplier = () -> new B(1, 2);
        Consumer<A> consumer = a -> System.out.println(a.x);
        pipe(supplier, consumer);
    }
```

Stellt aber der `Supplier` nur ein `B` zur Verfügung, kann ein `Consumer`, der ein `C` möchte (also ein anspruchsvoller `Consumer`), leider nicht bedient werden (und dafür, dass dies nicht funktioniert, sorgt der Compiler):

```
    static void demoPipeBC() {
        Supplier<B> supplier = () -> new B(1, 2);
        Consumer<C> consumer = c -> System.out.println(c.x + c.y
+ c.z);
        // pipe(supplier, consumer); // illegal
    }
```

(Man übersetzte `A` nach "Getränk", `B` nach "Wein" und `C` nach "Rotwein"....)

Ein `Consumer` kann das, was bei der Konsumption übrig bleibt, weiterreichen. Es geht also um die `andThen`-Methode des Interfaces:

```
    static void demoAndThen() {
        Consumer<Integer> c1 = v -> System.out.println("c1: " +
v);
        Consumer<Integer> c2 = v -> System.out.println("c2: " +
v);
        Consumer<Integer> c3 = v -> System.out.println("c3: " +
v);
        c1.andThen(c2).andThen(c3).accept(42);
    }
```

Die `42` wird dreimal ausgegeben werden: zuerst von `c1`, dann von `c2` und dann von `c3`. Hier die Ausgaben:

```
c1: 42
c2: 42
c3: 42
```

Arbeiten wir wieder mit `A`, `B` und `C`. Dabei ist zu beachten, dass der Parameter von `andThen` kontravariant ist.

Ein `Consumer` der ein `C` konsumiert, kann einen Nachfolger haben, der auch mit weniger zufrieden ist (er darf nur nicht mehr verlangen); und der Nachfolger dieses Nachfolgers kann sich wiederum mit weniger zufrieden geben...:

```
    static void demoAndThenCBA() {
        Consumer<C> c1 = c -> System.out.println("c1: " + (c.x +
c.y + c.z));
        Consumer<B> c2 = b -> System.out.println("c2: " + (b.x +
b.y));
        Consumer<A> c3 = a -> System.out.println("c3: " + (a.x));
        c1.andThen(c2).andThen(c3).accept(new C(1, 2, 3));
    }
```

Die umgekehrte Reihung würde vom Compiler als fehlerhaft zurückgewiesen:

```
            c3.andThen(c2).andThen(c1).accept(new A(1));  //
illegal
```

Auch hier gibt's Spezialvarianten des Interfaces: `IntConsumer`, `LongConsumer` und `DoubleConsumer`:

```
    static void demoIntConsumer() {
        IntConsumer c = x -> System.out.println(x);
        c.accept(42);
    }
```

`Consumer` (und auch natürlich `Supplier`) spielen in der Standardbibliothek eine wichtige Rolle. Hier ein Beispiel:

Das `Iterable`-Interface der Standardbibliothek wurde um eine `default`-Methode erweitert (eine, die nicht unbedingt die performanteste ist – aber immer funktioniert):

```
public interface Iterable<T> {

    Iterator<T> iterator();

    default void forEach(Consumer<? super T> action) {
        for (T t : this) {
            action.accept(t);
```

```
        }
    }
    // ...
}
```

`forEach` verlangt einen `Consumer` (der nicht mehr verlangen darf als `T` hergibt).

Da das Interface `List` von `Iterable` erbt, kann `forEach` auf `List`-Referenzen aufgerufen werden:

```
    static void demoListForEach() {
        List<Integer> list = Arrays.asList(10, 20, 30);
        list.forEach(element -> System.out.println(element));
    }
```

Die Ausgabe:

```
10
20
30
```

Die `forEach`-Methode ist übrigens in der Klasse `ArrayList` (die ihrerseits ja von `List` abgeleitet ist) überschrieben:

```
public class ArrayList<E> implements List<E> ... {
    // ...
    @Override
    public void forEach(Consumer<? super E> action) {
        final int expectedModCount = modCount;
        final E[] elementData = (E[]) this.elementData;
        final int size = this.size;
        for (int i=0; modCount == expectedModCount && i < size;
i++) {
            action.accept(elementData[i]);
        }
        if (modCount != expectedModCount) {
            throw new ConcurrentModificationException();
        }
    }
}
```

Wie man sieht, wird auf direkte Weise auf den Array zugegriffen, welcher einer `ArrayList` als Speicherfläche dient – und dies ist perfomanter als die default-Implementierung im `Iterable`-Interface.

BiConsumer

Manche `Consumer` brauchen nicht nur Wein, sondern auch Käse. Man benötigt also ein weiteres Interface:

```
@FunctionalInterface
public interface BiConsumer<T, U> {
    void accept(T t, U u);
    default BiConsumer<T, U> andThen(BiConsumer<? super T, ?
super U> after) {
        // ...
    }
}
```

Der folgende `BiConsumer` braucht einen Zahl und einen String:

```
    static void demoBiConsumer() {
        BiConsumer<Integer, String> c =
            (i, s) -> System.out.println(i + " " + s);
        c.accept(42, "Hello");
    }
```

6.4 Function

Eine Funktion bildet irgendetwas auf irgendetwas anderes ab (oder – ein Extremfall – ein irgendetwas auf dieses irgendetwas selbst – dann heißt die Funktion "Identitäts-Funktion").

Das `Function`-Interface hat zwei Typ-Parameter: `T` bezeichnet den Input-Typ, `R` den Output-Typ (den Resultat-Typ – daher `R`). Die Methode `apply` ist die einzige abstrakte Methode. Sie ist dazu gedacht, ein `T` auf ein `R` abzubilden:

```
package java.util.function;

@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);

    default <V> Function<V, R> compose(
            Function<? super V, ? extends T> before) {
        return (V v) -> apply(before.apply(v));
    }

    default <V> Function<T, V> andThen(
            Function<? super R, ? extends V> after) {
        return (T t) -> after.apply(apply(t));
    }

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

Die drei `default`-Methoden werden später besprochen. Zunächst zu `apply`. Hier eine `Function`, welche eine `String` bekommt und einen `Integer` liefert:

```
    static void demoFunction() {
        Function<String, Integer> f = v -> Integer.parseInt(v);
        int v = f.apply("42");
        System.out.println(v);
    }
```

Die `apply`-Methode liefert aufgrund der Zeichenkette `"42"` den Zahlenwert 42.

Verbinden wir wieder einen `Supplier` mit einem `Consumer` – schalten diesmal aber eine `Function` dazwischen:

```
    static void demoSupplierFunctionConsumer() {
        Supplier<String> supplier = () -> "42";
        Function<String, Integer> function = v ->
Integer.parseInt(v);
        Consumer<Integer> consumer = v -> System.out.println(v);
        consumer.accept(function.apply(supplier.get()));
    }
```

Kann die letzte Zeile allgemeingültig formuliert werden? Kann die bereits bekannte `pipe`-Methode um einen `Function`-Parameter erweitert werden? Hier die Lösung:

```
    static <S,T> void pipe(
            Supplier<? extends S> s,
            Function<S,T> f,
            Consumer<? super T> c) {
        c.accept(f.apply(s.get()));
    }
```

Die `Function` verlangt `S` und verspricht `T`. Dann muss der `Supplier` "mindestens" `S` liefern – und der `Consumer` darf "höchstens" `T` verlangen. Der `Supplier`-Parameter ist kovariant, der `Consumer`-Parameter kontravariant. Die `Function`-Parameter schließlich sind nonvariant.

Die erste Anwendung der `pipe`-Methode (die `Function` verlangt `String` und liefert `Integer`; der `Supplier` liefert `String`; der `Consumer` verlangt `Integer` – das passt genau):

```
    static void demoSupplierFunctionConsumerIntegerToString() {
        Supplier<String> supplier = () -> "42";
        Function<String, Integer> function = v ->
Integer.parseInt(v);
        Consumer<Integer> consumer = v -> System.out.println(v);
        pipe(supplier, function, consumer);
    }
```

Die zweite Anwendung benutzt wieder `A`, `B` und `C`. Die `Function` verlangt `C` und liefert `B`. Der `Supplier` liefert `C` – das passt genau. Der `Consumer` verlangt `A` – das ist zwar "weniger" als die `Function` liefert – aber passt:

```
    static void demoSupplierFunctionConsumerCToB() {
        Supplier<C> supplier = () -> new C(1, 2, 3);
        Function<C, B> function = (c) -> new B(c.x + 1, c.y + 1);
```

```
        Consumer<A> consumer = a -> System.out.println(a.x);
        pipe(supplier, function, consumer);
    }
```

Die Ausgabe lautet 2. (Von dem anfänglichen `C`-Objekt ist also wenig "übriggeblieben"...)

Wie kann die `default`-Methode `andThen` benutzt werden? Wir definieren drei `Functions` und verbinden sie mit `andThen`:

```
    static void demoAndThen() {
        Function<Integer, Integer> f1 = x -> x + 1;
        Function<Integer, Integer> f2 = x -> 2 * x;
        Function<Integer, Integer> f3 = x -> x * x;
        int v = f1.andThen(f2).andThen(f3).apply(3);
        System.out.println(v); // -> 64
    }
```

Woher kommt `64`? `3 + 1` ergibt `4`; `2 * 4` ergibt `8`; und `8 * 8` ergibt `64`. Wie der Name der Funktion sagt: zuerst wird `f1`, dann `f2` und dann `f3` angewandt.

Die `default`-Methode `compose` funktioniert genau anders herum:

```
    static void demoCompose() {
        Function<Integer, Integer> f1 = x -> x + 1;
        Function<Integer, Integer> f2 = x -> 2 * x;
        Function<Integer, Integer> f3 = x -> x * x;
        int v = f1.compose(f2).compose(f3).apply(3);
        System.out.println(v); // -> 19
    }
```

Und natürlich können `andThen` und `compose` kombiniert werden:

```
    static void demoAndThenCompose() {
        Function<Integer, Integer> f1 = x -> x + 1;
        Function<Integer, Integer> f2 = x -> 2 * x;
        Function<Integer, Integer> f3 = x -> x * x;
        int v = f1.andThen(f2).compose(f3).apply(3);
        System.out.println(v); // -> 20
    }
```

Am "Ende der Zeile" aber steht immer `apply`. "fluent programming"...

Die statische(!) Methode `identity` liefert eine `Function`, die sich keinerlei Mühe gibt: sie liefert jeweils genau das zurück, was man ihr gibt:

```
    static void demoIdentity() {
```

```
        Function<Integer, Integer> f = Function.identity();
        int v = f.apply(42);
        System.out.println(v); // -> 42
    }
```

`IntFunction` ist ein spezialisiertes Interface. Der Input ist `int` – der Output muss über einen Typ-Parameter spezifiziert werden:

```
    static void demoIntFunctionInteger() {
        IntFunction<Integer> f = x -> 2 * x;
        int v = f.apply(42);
        System.out.println(v); // -> 84
    }
```

Auch die `apply`-Methode der folgenden `IntFunction` verlangt einen `int`-Wert; sie liefert aber ein `Double`-Objekt:

```
    static void demoIntFunctionDouble() {
        IntFunction<Double> f = x -> Math.sqrt(x);
        double v = f.apply(2);
        System.out.println(v); // -> 1.41...
    }
```

Neben `IntFunction` gibt's natürlich auch `LongFunction` und `DoubleFunction`.

BiFunction

Die `apply`-Methode einer `BiFunction` verlangt zwei Argumente. `BiFunction` hat also drei Typ-Parameter: Input-1, Input-2 und Output:

```
package java.util.function;

@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);

    default <V> BiFunction<T, U, V> andThen(
            Function<? super R, ? extends V> after) {
        return (T t, U u) -> after.apply(apply(t, u));
    }
}
```

Pythagoras spielt mit einem rechtwinkligen Dreieck und berechnet aufgrund der beiden Katheten die Hypotenuse des Dreiecks:

```
    static void demoBiFunction() {
        BiFunction<Integer, Integer, Double> f =
            (x, y) -> Math.sqrt(x * x + y * y);
        double d = f.apply(3, 4);
        System.out.println(d);
    }
```

Und man könnte z.B. auch aus einem `A` und einem `B` ein `C` bauen:

```
    static void demoBiFunctionABC() {
        BiFunction<A, B, C> f = (a, b) -> new C(a.x, b.x, b.y);
        C c = f.apply(new A(1), new B(2, 3));
        System.out.println(c.x + " " + c.y + " " + c.z);
    }
```

6.5 UnaryOperator

`UnaryOperator` ist von `Function` abgeleitet – wobei der Output vom selben Typ ist wie der Input. Von `Function` erbt sie natürlich die `apply`-Methode.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
    static <T> UnaryOperator<T> identity() {
        return t -> t;
    }
}
```

Das folgende Objekt repräsentiert den unären Minus-Operator:

```
    static void demoUnaryOperator() {
        UnaryOperator<Integer> op = x -> -x;
        System.out.println(op.apply(42));
    }
```

Und `42` kann natürlich auch auf `42` abgebildet werden:

```
    static void demoIdentity() {
        UnaryOperator<Integer> op = UnaryOperator.identity();
        System.out.println(op.apply(42));
    }
```

Das `List`-Interface der Standardbibliothek enthält die `default`-Implementierung von `replaceAll`. Dieser Methode wird ein `UnaryOperator` übergeben, dessen `apply`-Methode für jedes Element der Liste aufgerufen wird. Ihr wird das bisherige Element der Liste übergeben – sie muss das neue Element zurückliefern (welches dann an die Stelle des alten Elements gesetzt wird):

```
public interface List<E> ... {
    // ...
    default void replaceAll(UnaryOperator<E> operator) {
        final ListIterator<E> li = this.listIterator();
        while (li.hasNext()) {
            li.set(operator.apply(li.next()));
        }
    }
}
```

Die Klasse `ArrayList` überschreibt diese Implementierung (natürlich zum Zwecke der Optimierung):

```
public class ArrayList<E> ... {
    // ...
    @Override
    void replaceAll(UnaryOperator<E> operator) { ... }
}
```

In der folgenden Anwendung wird jedes Element einer `Integer`-Liste durch einen Wert ersetzt, der das 10-fache des vorgefundenen Werts beträgt:

```
    static void demoListReplaceAll() {
        List<Integer> list = new ArrayList<>(Arrays.asList(10,
20, 30));
        list.replaceAll(x -> x * 10);
        list.forEach(x -> System.out.println(x)); // -> 100 200
300
    }
```

6.6 BinaryOperator

`BinaryOperator` ist abgeleitet von `BiFunction` – wobei die Inputs und der Output allesamt vom selben Typ sind. Von `BiFunction` erbt das Interface natürlich u.a. die `apply`-Methode.

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T> {

    public static <T> BinaryOperator<T> minBy(
            Comparator<? super T> comparator) {
        return (a, b) -> comparator.compare(a, b) <= 0 ? a : b;
    }

    public static <T> BinaryOperator<T> maxBy(
            Comparator<? super T> comparator) {
        return (a, b) -> comparator.compare(a, b) >= 0 ? a : b;
    }
}
```

Die statischen Methoden `minBy` und `maxBy` erzeugen einen neuen `BinaryOperator`, der einen `Comparator` benutzt (lassen also einen `Comparator` als `BinaryOprator` "erscheinen").

Der folgende `BinaryOperator` repräsentiert die Plus-Operation für `Integer`:

```
    static void demoBinaryOperator() {
        BinaryOperator<Integer> op = (x, y) -> x + y;
        System.out.println(op.apply(40, 2)); // -> 42
    }
```

Hier eine Anwendung von `minBy` und `maxBy`:

```
    static void demoMinByMaxBy() {
        BinaryOperator<String> min =
            BinaryOperator.minBy((x, y) -> x.compareTo(y));
        BinaryOperator<String> max =
            BinaryOperator.maxBy((x, y) -> x.compareTo(y));
        System.out.println(min.apply("Hello", "World")); // ->
"Hello"
        System.out.println(max.apply("Hello", "World")); // ->
"World"
    }
```

Neben dem generischen Interface gibt's auch hier spezialisierte Interfaces.

Um zwei `int`-Werte zu einem neuen zu verknüpfen, kann `IntBinaryOperator` verwendet werden:

```
    static void demoIntBinaryOperator() {
        IntBinaryOperator op = (x, y) -> x + y;
        System.out.println(op.applyAsInt(40, 2)); // 42
    }
```

Um `double`-Werte binär zu verknüpfen, kann `DoubleBinaryOperator` verwendet werden:

```
    static void demoDoubleBinaryOperator() {
        DoubleBinaryOperator op = (x, y) -> x + y;
        System.out.println(op.applyAsDouble(40, 2)); // -> 42.0
    }
```

6.7 Predicate

Ein `Predicate` bekommt einen Input. Es schaut sich diesen Input an, überlegt, und liefert dann entweder `true` oder `false`. Ein `Predicate` ist also ein Tester – und die Test-Methode heißt `test`:

```
package java.util.function;

@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> negate() {
        return (t) -> !test(t);
    }

    default Predicate<T> or(Predicate<? super T> other) {
        return (t) -> test(t) || other.test(t);
    }

    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
                ? Objects::isNull
                : object -> targetRef.equals(object);
    }
}
```

Das folgende `Predicate` liefert kann auf `Integer` angewandt werden. Es liefert `true`, wenn der Input gerade ist, ansonsten `false`:

```
    static void demoPredicate() {
        Predicate<Integer> p = v -> v % 2 == 0;
        System.out.println(p.test(3)); // -> false
        System.out.println(p.test(4)); // -> true
    }
```

Mittels der `default`-Methode `and` können zwei `Predicate`s zu einem neuen verknüpft werden – dessen `test`-Methode dann (und nur dann) `true` liefert, wenn die `test`-Methoden der beiden verknüpften `Predicate`s `true` liefern:

```
    static void demoAnd() {
        Predicate<Integer> p1 = v -> v > 10;
        Predicate<Integer> p2 = v -> v < 20;
        System.out.println(p1.and(p2).test(3)); // -> false
        System.out.println(p1.and(p2).test(30)); // -> false
        System.out.println(p1.and(p2).test(13)); // -> true
    }
```

Damit dürfte auch die Bedeutung der `or`-Methode klar sein.

Die Klasse `ArrayList` der Standardbibliothek definiert eine Methode `removeIf`. Dieser Methode wird ein `Predicate` übergeben, welches auf jedes Element der Liste angewandt wird. Liefert das `Predicate` den Wert `true`, wird das entsprechende Element aus der Liste entfernt:

```
public class ArrayList<E> ... {
    // ...
    @Override
    public boolean removeIf(Predicate<? super E> filter) { ... }
}
```

Im folgenden Beispiel werden alle geraden Zahlen aus einer `Integer`-Liste entfernt:

```
    static void demoListRemoveIf() {
        // List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);
        List<Integer> list = new ArrayList<>(
            Arrays.asList(1, 2, 3, 4, 5, 6));
        list.removeIf(x -> x % 2 == 0);
        list.forEach(x -> System.out.println(x)); // 1 3 5
    }
```

(Hinweis: Die Factory-Methode `Arrays.asList` liefert eine `List`, welche die `removeIf`-Methode nicht(!) unterstürzt. Deshalb ist die Benutzung dieser Methode im obigen Beispiel auskommentiert.)

6.8 Reader-Writer-Beispiel

Eine Eingabe soll zeichenweise gelesen werden; und eine Ausgabe mit Zeichen gefüllt werden.

Für das Lesen der Eingabe ist ein `CharacterReader` zuständig. `CharacterReader` implementiert `Supplier<Character>` - er kann Zeichen zur Verfügung stellen. Dem Konstruktor wird ein `Reader` übergeben:

```
public class CharacterReader implements Supplier<Character> {

    private final Reader reader;

    public CharacterReader(Reader reader) {
        this.reader = reader;
    }

    public Character get() {
        try {
            int ch = reader.read();
            return ch < 0 ? null : (char) ch;
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Die `get`-Methode liefert bei jedem Aufruf das jeweils nächste Zeichen – resp. `null`, wenn EOF erreicht ist.

Ein `CharacterReader` erlaubt es also, einen `Reader` als `Supplier<Character>` zu behandeln.

Ein `CharacterWriter` ist für das Schreiben in eine Ausgabe verantwortlich. Im wird ein `Writer` übergeben. Die Klasse implementiert das Interface `Consumer<Character>`:

```
public class CharacterWriter implements Consumer<Character> {

    private final Writer writer;

    public CharacterWriter(Writer writer) {
        this.writer = writer;
    }
```

```
    public void accept(Character ch) {
        try {
            writer.write(ch);
            writer.flush();
        }
        catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Ein `CharacterWriter` erlaubt es also, einen `Writer` wie einen `Consumer<Character>` zu behandeln.

Die folgende Beispiel-Applikation benutzt eine `Function` – eine `Function`, die jedes Zeichen auf die UpperCase-Form dieses Zeichens abbildet:

```
public class ToUpper implements Function<Character, Character> {
    public Character apply(Character ch) {
        return Character.toUpperCase(ch);
    }
}
```

Weiterhin existiert eine allgemeine `process`-Methode, der ein `Supplier`, eine `Function` und ein `Consumer` übergeben werden:

```
    static <T, R> void process(
            Supplier<T> supplier,
            Function<T, R> function,
            Consumer<R> consumer) {
        T t;
        while ((t = supplier.get()) != null) {
            R r = function.apply(t);
            consumer.accept(r);
        }
    }
```

Die `process`-Methode ruft wiederholt den `Supplier` auf (bis dieser `null` liefert). Der Output des `Suppliers` wird dann als Input an die `Function` übergeben; und schließlich wird der Output der `Function` als Input an den `Consumer` übergeben.

Hier eine Anwendung der obigen Klassen und Methoden:

```
    static void demo1() {
        String input = "abc\n";
```

```
        Supplier<Character> reader =
            new CharacterReader(new StringReader(input));
        Function<Character, Character> toUpper =
            new ToUpper();
        Consumer<Character> writer =
            new CharacterWriter(new PrintWriter(System.out));
        process(reader, toUpper, writer);
    }
```

Die Ausgabe:

```
ABC
```

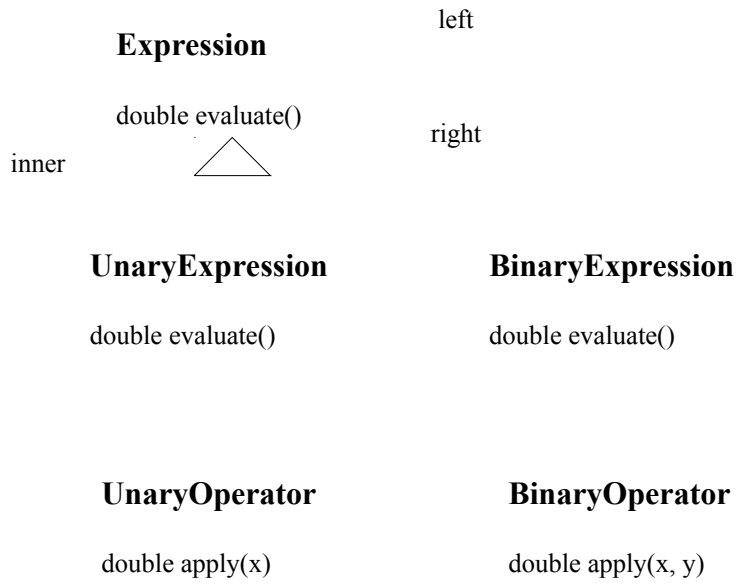
Hier eine etwa kompaktere Variante:

```
    static void demo2() {
        String input = "abc\n";
        process(
            new CharacterReader(new StringReader(input)),
            ch -> Character.toUpperCase(ch),
            ch -> System.out.print(ch));
    }
```

6.9 Expressions-Beispiel

Das folgende Beispiel verwendet `UnaryOperator`s und `BinaryOperator`s, um einen numerischen Ausdruck zu berechnen. Der numerische Ausdruck existiert in Form eines Baumes von `Expression`-Objekten.

Zunächst ein Klassendiagramm:



Die Basisklasse `Expression` spezifiziert eine parameterlose `evaluate`-Methode, die `double` liefern muss.

```
public abstract class Expression {
    public abstract double evaluate();
}
```

Eine `NumberExpression` hat einen `value` – eben dieser wird von `evaluate` geliefert:

```
public class NumberExpression extends Expression {

    private final double value;
    public NumberExpression(double value) {
        this.value = value;
    }
    public double evaluate() {
        return this.value;
```

```
    }
}
```

Eine `UnaryExpression` hat eine Referenz auf irgendeine andere `Expression` (namens `inner`) – und eine Referenz auf einen `UnarayOperator`:

```
public class UnaryExpression extends Expression {
    private final UnaryOperator<Double> op;
    private final Expression inner;
    public UnaryExpression(UnaryOperator<Double> op, Expression
inner) {
        this.op = op;
        this.inner = inner;
    }
    public double evaluate() {
        return op.apply(this.inner.evaluate());
    }
}
```

Die `evaluate`-Methode evaluiert zunächst die von `inner` referenzierte `Expression`. Das Ergebnis wird dann an die `apply`-Methode des `UnaryOperators` weitergereicht – um das Resultat dieser `apply`-Methode dann zurückzuliefern.

Elne `BinaryExpression` hat zwei `Expression`-Referenzen: `left` und `right` – und eine Referenz auf einen `BinaryOperator`:

```
public class BinaryExpression extends Expression {
    private final BinaryOperator<Double> op;
    private final Expression left;
    private final Expression right;
    public BinaryExpression(BinaryOperator<Double> op,
            Expression left, Expression right) {
        this.op = op;
        this.left = left;
        this.right = right;
    }
    public double evaluate() {
        return op.apply(this.left.evaluate(),
this.right.evaluate());
    }
}
```

`evaluate` benutzt den `BinaryOperator`, um den Wert der `left`- und der `right`-`Expression` zu einem neuen Wert zu verknüpfen, der dann zurückgeliefert wird.

Es werden vier `BinaryOperators` und ein `UnaryOperator` definiert:

```
public class Operators {
    public static final BinaryOperator<Double> PLUS = (x, y) -> x
+ y;
    public static final BinaryOperator<Double> MINUS = (x, y) ->
x - y;
    public static final BinaryOperator<Double> TIMES = (x, y) ->
x * y;
    public static final BinaryOperator<Double> DIV = (x, y) ->
x / y;
    public static final UnaryOperator<Double> UMINUS = x -> -x;
}
```

Dann kann ein beispielhafter `Expression`-Baum aufgebaut werden und schließlich evaluiert werden:

```
        Expression e1 = new NumberExpression(2);
        Expression e2 = new NumberExpression(10);
        UnaryExpression e3 = new UnaryExpression(UMINUS, e1);
        BinaryExpression e = new BinaryExpression(PLUS, e2, e3);
        out.println(e.evaluate());
```

Das Resulat:

```
8.0
```

Man hätte das Problem natürlich auch anders lösen können: Wir hätten von `BinaryExpression` die Klassen `PlusExpression`, `MinusExpression` etc. und von `UnaryExpression` die Klasse `UnaryMinusExpression` ableiten können. Die Berechnungs-Funktionalität könnte dann in diesen abgeleiteten `Expression`-Klassen implementiert sein – und wir benötigten keinerlei Operatoren. Worin liegt der Nachteil dieser möglichen Alternative?

6.10 Simulation harter Arbeit

Insbesondere beim Experimentieren mit Multithreading-Anwendungen kommt es häufig darauf an, "harte Arbeit" zu simulieren – Arbeit, die Zeit dauert (z.B. die Ausführung eines komplexen `SELECT`s oder einen Zugriff auf ein entferntes Objekt).

Im folgenden wird zunächst unabhängig vom Multithreading-Kontext ein einfaches Beispiel vorgestellt, welches bestimmte Utility-Klassen aus dem `shared`-Projekt benutzt – Utiliy-Klassen, mittels derer harte Arbeit vorgetäuscht werden kann.

Sei z.B. die folgende `execute`-Methode gegeben:

```
    static <T,R> void execute(
            Supplier<T> supplier,
            Function<T, R> function,
            Consumer<R> consumer,
            Runnable runnable) {
        T t = supplier.get();
        R r = function.apply(t);
        consumer.accept(r);
        runnable.run();
    }
```

Die Methode führt die `get`-Methode des ihr übergebenen `Supplier`s aus; das Ergebnis dieses Aufrufs (ein `T`) wird an die `apply`-Methode der übergebenen `Function` weitergereicht; das Ergebnis dieser `Function` (ein `R`) wird an die `accept`-Methode eines `Consumer`s weitergereicht; und schließlich wird noch die `run`-Methode eines `Runnable`s aufgerufen. Die wichtigsten funktionalen Interfaces sind hier also allesamt vertreten...

Hier ein beispielhafter Aufruf dieser `execute`-Methode:

```
    static void demo1() {
        execute(
            () -> 21,
            x -> 2 * x,
            x -> System.out.println(x),
            () -> System.out.println("FIN")
        );
    }
```

Hier die Ausgaben:

```
42
FIN
```

Wir wollen nun aber dem `Supplier`, der `Function`, dem `Consumer` und dem `Runnable` bei ihrer Arbeit "zuschauen" – und das soll einigermaßen "gemütlich" vor sich gehen: bei den Aufrufen soll jeweils ein frei zu bestimmender `Thread.sleep` stattfinden.

Die folgende Methode benutzt die statische `work`-Methode der Klasse `util.Work` (und den `enum`-Typ `Work.Type`):

```
    static void demo2() {
        execute(
            () -> work(Type.SUPPLIER, "s", 1000).thenReturn(21),
            x -> work(Type.FUNCTION, "f " + x, 2000).thenReturn(2
* x),
            x -> work(Type.CONSUMER, "c " + x, 1000),
            () -> work(Type.RUNNABLE, "r", 2000)
        );
    }
```

Die Ausgaben (ihre Produktion dauert so etwa 6 Sekunden):

```
[ 1 ] ->  (1000)  s {SUPPLIER}
[ 1 ] <-  (1000)  s
[ 1 ] ->  (2000)  f 21 {FUNCTION}
[ 1 ] <-  (2000)  f 21
[ 1 ] ->  (1000)  c 42 {CONSUMER}
[ 1 ] <-  (1000)  c 42
[ 1 ] ->  (2000)  r {RUNNABLE}
[ 1 ] <-  (2000)  r
```

Statt in den Lambdas direkt die "eigentliche" Operation zu hinterlegen, wird in ihnen eine Methode hinterlegt, welche jeweils ihrern Einsteig ausgibt, dann eine gewisse Zeit lang schläft und schließlich ihrern Ausstieg protokolliert – die `work`-Methode. Um einen `Consumer` resp. ein `Runnable` zu übergeben, reicht die Bereitstellung der `work`-Methode selbst; sofern aber ein `Supplier` oder eine `Function` übergeben werden soll, muss der der Aufruf von `work` mit dem Aufruf von `thenReturn` verbunden werden. In `thenReturn` wird dann das Ergebnis hinterlegt, welcher der `Supplier` zur Verfügung stellen soll bzw. welches die `Function` liefern soll. Der `work`-Methode wird ein `Work.Type` übergeben, der auszugebende Text und die `sleep`-Dauer (in Millisekunden). Die Protokollzeilen beginnen jeweils mit der ID desjenigen Threads, der die Anweisungen ausführt.

Hier ein Ausschnitt aus der `Work`-Klasse:

```
package util;
// ...
public class Work {
```

```
    public enum Type {
        SUPPLIER(new Color(255, 128, 128)),
        FUNCTION(new Color(128, 128, 255)),
        CONSUMER(new Color(128, 255, 128)),
        RUNNABLE(Color.lightGray),
        WAITER(Color.yellow);

        public final Color color;
        private Type(Color color) {
            this.color = color;
        }
    }

    // ...

    public static class W {
        public <T> T thenReturn(T value) {
            return value;
        }
    }

    public static W work(Type type, String text, int millis) {
        // ...
        tlog("->  (" + millis + ") \t" + text + " {" + type +
"}");
        xrun(() -> Thread.sleep(millis)); // simulating hard
work...
        tlog("<-  (" + millis + ") \t" + text);
        // ...
        return new W();
    }
    // ...
}
```

Die Bedeutung der Hilfsklasse `W` dürfte klar sein...

Neben der `work`-Methode enthält die Klasse `Work` noch einige Convenience-Methoden, welche allesamt auf die `work`-Methode abgebildet werden:

```
public class Work {
    // ...
    public static W swork(String text, int millis) {
        return work(WorkType.SUPPLIER, text, millis);
    }
    public static W fwork(String text, int millis) {
        return work(WorkType.FUNCTION, text, millis);
```

```
    }
    public static void cwork(String text, int millis) {
        work(WorkType.CONSUMER, text, millis);
    }
    public static void rwork(String text, int millis) {
        work(WorkType.RUNNABLE, text, millis);
    }
    // ...
}
```

Die Präfixierung ist hoffentlich einleuchtend. Man beachte, dass die ersten beiden Methoden `W` liefern, die letzten beiden Methoden aber einfach `void`.

Hier eine `demo3`-Methode, welche diese Convenience-Methode nutzt (und deren Aufruf dieselben Ausgaben produziert wie der Aufruf von `demo2`):

```
    static void demo3() {
        execute(
            () -> swork("s", 1000).thenReturn(21),
            x -> fwork("f", 2000).thenReturn(2 * x),
            x -> cwork("c", 1000),
            () -> rwork("r", 2000)
        );
    }
```

Neben der Console-Ausgabe können auch grafische Ausgaben erzeugt werden – es existiert ein grafischer `GuiViewer` (im Paket `util.workviewer`). Auf die Einzelheiten der Implementierung soll hier nicht näher eingegangen werden. Der Viewer wird von der `Work`-Klasse verwaltet (welche u.a. auch sein Interface spezifiziert: `Work.Viewer`):

```
public class Work {
    // ...

    public interface Viewer {
        public abstract void start();
        public abstract void beginWork(String text, Type type);
        public abstract void endWork();
        public abstract void stop();
        public abstract void terminate();
    }

    private static Viewer nullViewer = new Viewer() {
        public void start() { }
        public void beginWork(String text, Type type) { }
        public void endWork() {    }
        public void stop() { }
```

```
        public void terminate() { }
    };

    private static Viewer viewer = nullViewer;

    private static int millisBetweenStopStart = 0;

    public static void useViewer(int millisBetweenStopStart) {
        Work.millisBetweenStopStart = millisBetweenStopStart;
        Work.viewer = new GuiViewer();
    }

    public static Viewer getViewer() {
        return Work.viewer;
    }

    public static void call(XRunnable runnable) {
        Work.viewer.start();
        xrun(runnable);
        Work.viewer.stop();
        xrun(() -> Thread.sleep(Work.millisBetweenStopStart));
    }

    // ...
}
```

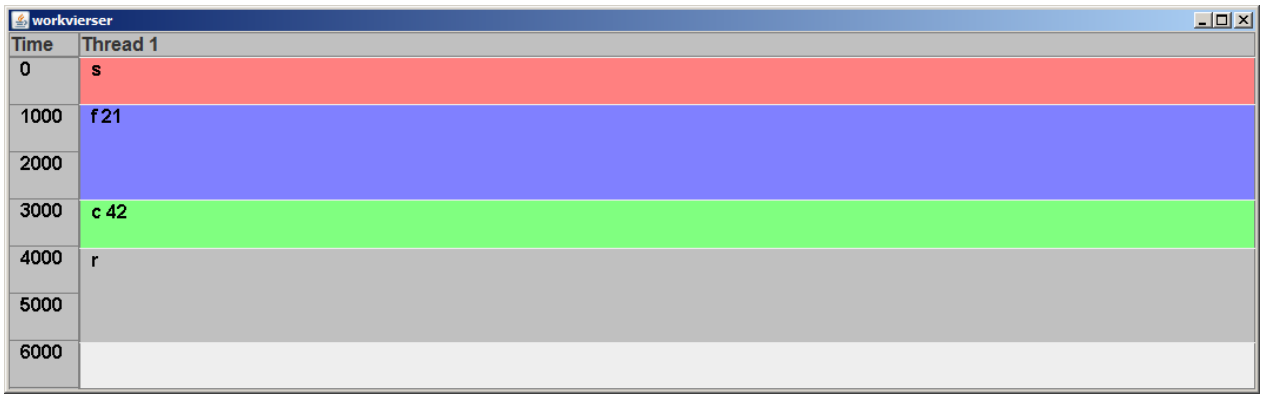
Die `call`-Methode startet den Viewer, führt das ihr übergebene `Runnable` aus und stoppt den Viewer.

Die `Application`-Klasse, welche die `demo`-Methoden enthält, wird wie folgt erweitert:

```
package appl;
// ...
import static util.Work.call;
import static util.Work.work;

public class Application {

    public static void main(String[] args) throws Exception {

        useViewer(2000);

        call(() -> demo1());
        call(() -> demo2());
        call(() -> demo3());
    }
```

```
    // Die execute-Methode...

    // Die demo-Methoden ...
}
```

Hier die Ausgaben des letzten Aufrufs (von `demo3`):



Hier geht noch alles sequentiell zu. Interessanter werden die Ausgaben, wenn Multithreading ins Spiel kommt... (Was die Farben angeht, siehe den `enum`-Typ `WorkType`: rot = `Supplier`, blau = `Function`, grün = `Consumer` und grau = `Runnable`.)

6.11 Multithreading

Die im letzten Abschnitt vorgestellten `Work`- und `Viewer`-Klassen sind eigentlich dazu gedacht, parallele Abläufe zu verdeutlichen (und Fehler in solchen Abläufen zu finden...!).

In diesem Abschnitt wird ein kleines "Mini-Framework" entwickelt, welches natürlich intensiv die neuen Mittel von Java 8 nutzt – insbesondere die neuen funktionalen Interfaces. Es handelt sich um ein "Framework", mittels dessen wir Schritte einer Berechnung spezifizieren können, die entweder sequentiell oder aber parallel ausgeführt werden können. (Genau dies ist auch der Sinn der in Java 8 neu eigeführten Klasse `CompletableFuture` – eine Klasse, die später im Multithreading-Kapitel ausführlich vorgestellt wird. Insofern sind die folgenden Erläuterungen auch als Einstieg in das Thema `CompletableFuture` brauchbar...)

Das Framework demonstriert, wie `Supplier`, `Function`s, `Consumer` und `Runnable`s zu einem "höheren" System kombiniert werden können. Die Interfaces werden natürlich sinnvollerweise als Lambdas implementiert. Man könnte sie natürlich auch als anonyme Klassen implementieren – mit dem Effekt allerdings, dass solche Implementierungen nur noch schwer lesbar wären.

Zunächst wird ein Framework gebaut, welches die Schritte einer Spezifikation nur sequentiell ausführen kann. Anschließend wird das Framework derart umgebaut, dass auch parallele Verarbeitung ermöglicht wird.

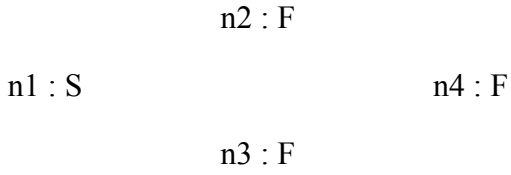
Hier eine `demo`-Methode:

```
static void demoSFFFSimple() {
    Node<Integer> n1 = Node.supply(() -> 4);
    Node<Integer> n2 = n1.apply(x -> x + 1);
    Node<Integer> n3 = n1.apply(x -> x - 1);
    Node<Integer> n4 = n2.combine(n3, (x, y) -> x * y);
    Integer result = n4.get();
    System.out.println(result);
}
```

Der Name deutet an, dass an den Schritten ein `Supplier` und drei `Functions` beteiligt sind.

Die ersten vier Zeilen können als "Spezifikation" verstanden werden: Ein `Supplier` stellt einen Wert bereit (hier: `4` – aber hier könnte natürlich auch eine Variable stehen). Dieser Wert wird an zwei `Functions` übergeben werden: die erste liefert einen Wert zurück, der um ein größer ist als der ihr übergebene Wert; die zweite liefert einen um 1 kleineren Wert zurück. Die Ergebnisse dieser beiden `Functions` werden dann an eine

weitere `Function` (an eine `BiFunction`) übergeben, welche sie multiplikativ verknüpft. Der `Supplier` und die `Functions` werden dabei jeweils in `Node`-Objekte eingehängt, welche zu einem gerichteten Grafen verbunden werden:



Es ist klar, dass die Schritte `n2` und `n3` parallel ausgeführt werden könnten. n4 kann natürlich erst dann ausgeführt werden, wenn sowohl `n2` als auch `n3` ihrer Ergebnisse geliefert haben. Und bevor nicht der `Supplier` seinen Wert geliefert hat, können natürlich auch `n2` und `n3` nicht starten.

Am Ende der obigen `demo`-Methode wird auf `n4` die Methode `get` aufgerufen. Erst der Aufruf von `get` setzt die Verarbeitung in Bewegung - und liefert das Endergebnis zurück. Diese sollte dann den Wert `15` liefern – also das Resultat der folgenden Berechnung: `(4-1)*(4+1)`.

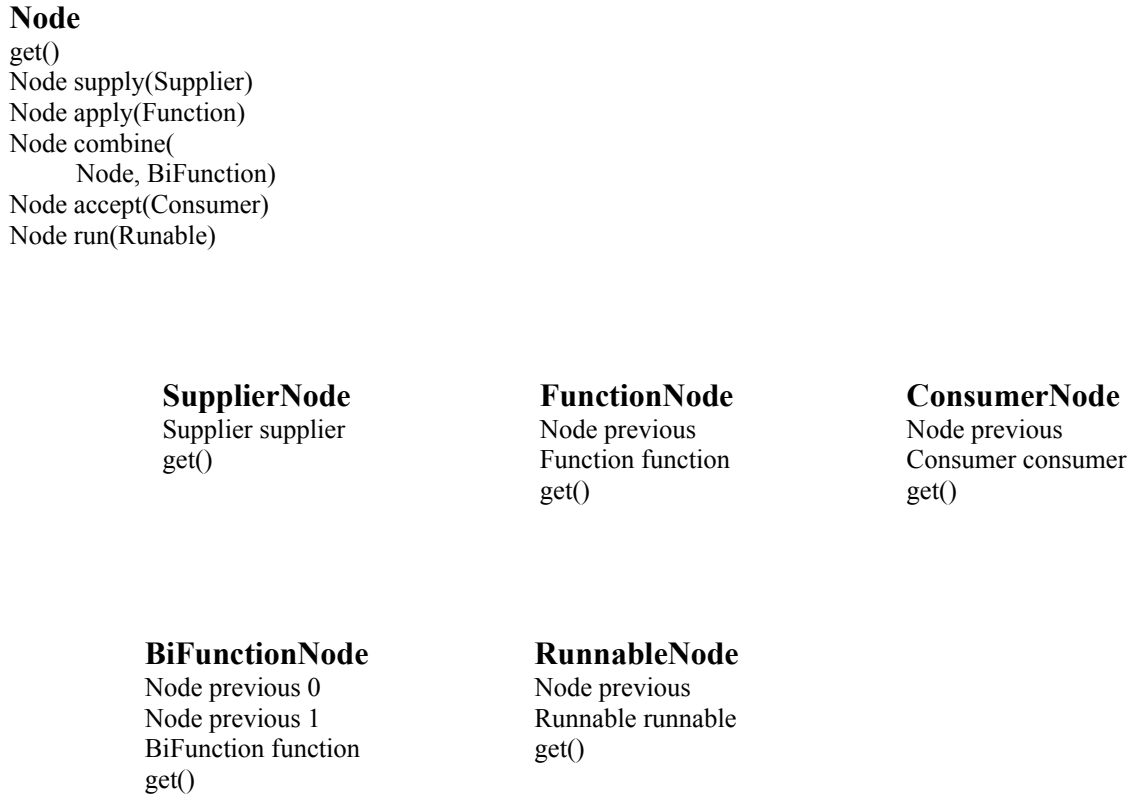
Das Studium der hier verwendeten `Node`-Klasse sei dem Leser / der Leserin überlassen – hier der komplette Code der Klasse:

```
package sequential;

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

public abstract class Node<T> {

    public abstract T get();

    private static class SupplierNode<T> extends Node<T> {
        public final Supplier<T> supplier;
        public SupplierNode(Supplier<T> supplier) {
            this.supplier = supplier;
        }
        private T result = null;
        public T get() {
            if (this.result == null)
                this.result = this.supplier.get();
            return this.result;
```

```
        }
    }

    private static class FunctionNode<T, R> extends Node<R> {
        private final Node<T> previous;
        public final Function<T, R> function;
        public FunctionNode(Function<T, R> function, Node<T>
previous) {
            this.previous = previous;
            this.function = function;
        }
        private R result = null;
        public R get() {
            if (result == null) {
                T value = this.previous.get();
                result = function.apply(value);
            }
            return result;
        }
    }

    private static class BiFunctionNode<T0, T1, R> extends
Node<R> {
        private final Node<T0> previous0;
        private final Node<T1> previous1;
        public final BiFunction<T0, T1, R> function;
        public BiFunctionNode(BiFunction<T0, T1, R> function,
                Node<T0> previous0, Node<T1> previous1) {
            this.previous0 = previous0;
            this.previous1 = previous1;
            this.function = function;
        }
        private R result = null;
        public R get() {
            if (result == null) {
                T0 value0 = this.previous0.get();
                T1 value1 = this.previous1.get();
                result = function.apply(value0, value1);
            }
            return result;
        }
    }

    private static class ConsumerNode<T> extends Node<Void> {
        private final Node<T> previous;
        public final Consumer<T> consumer;
```

```
        public ConsumerNode(Consumer<T> consumer, Node<T>
previous) {
            this.previous = previous;
            this.consumer = consumer;
        }
        public Void get() {
            T value = this.previous.get();
            consumer.accept(value);
            return null;
        }
    }

    private static class RunnableNode extends Node<Void> {
        private final Node<?> previous;
        public final Runnable runnable;
        public RunnableNode(Runnable runnable, Node<?> previous)
{
            this.previous = previous;
            this.runnable = runnable;
        }
        public Void get() {
            this.previous.get();
            runnable.run();
            return null;
        }
    }

    public static <T> Node<T> supply(Supplier<T> supplier) {
        return new SupplierNode<T>(supplier);
    }
    public <R> Node<R> apply(Function<T, R> function) {
        return new FunctionNode<T, R>(function, this);
    }
    public <T0, R> Node<R> combine(Node<T0> other,
            BiFunction<T, T0, R> function) {
        return new BiFunctionNode<T, T0, R>(function, this,
other);
    }
    public Node<Void> accept(Consumer<T> consumer) {
        return new ConsumerNode<T>(consumer, this);
    }
    public Node<Void> run(Runnable runnable) {
        return new RunnableNode(runnable, this);
    }
}
```

Ein kleines Klassendiagramm mag das Verständnis dieses Codes erleichtern:



Um den genauen Ablauf der Berechnungen zu protokollieren und die Berechnung nach menschlichen Maßstäben vernünftige Geschwindigkeit zu reduzieren, kann die obige `demo`-Methode wie folgt umgeschrieben werden:

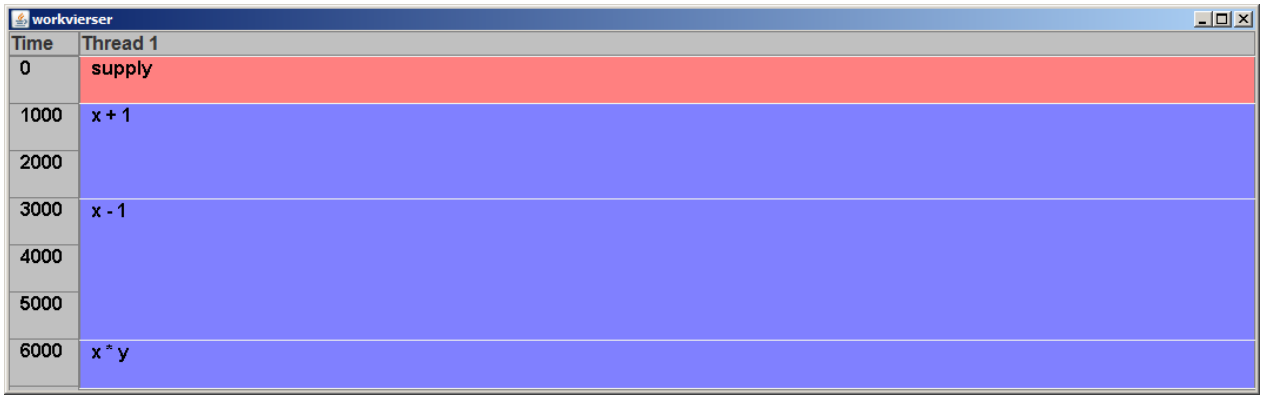
```
    static void demoSFFF() {
        Node<Integer> n1 = Node.supply(
            () -> swork("supply", 1000).thenReturn(4));
        Node<Integer> n2 = n1.apply(
            x -> fwork("x + 1", 2000).thenReturn(x + 1));
        Node<Integer> n3 = n1.apply(
            x -> fwork("x - 1", 3000).thenReturn(x - 1));
        Node<Integer> n4 = n2.combine(n3,
            (x, y) -> fwork("x * y", 1000).thenReturn(x * y));
        Integer result = n4.get();
        System.out.println(result);
    }
```

Das Protokoll:

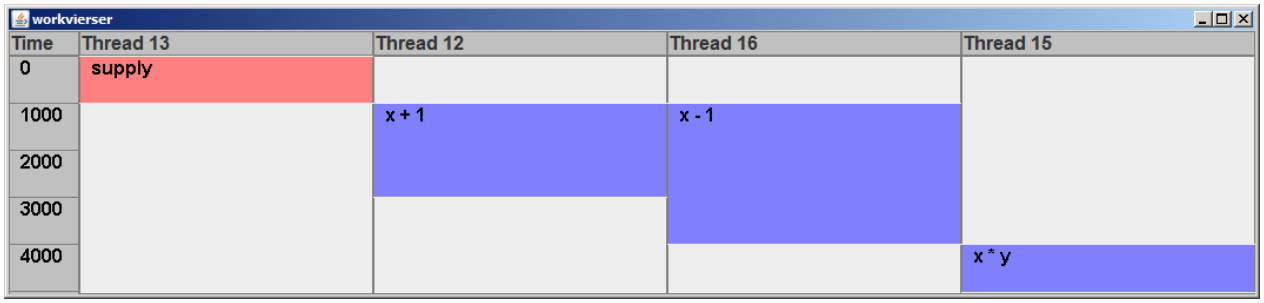
```
[ 1 ] ->  (1000)  supply {SUPPLIER}
[ 1 ] <-  (1000)  supply
```

```
[ 1 ] ->   (2000)   x + 1 {FUNCTION}
[ 1 ] <-   (2000)   x + 1
[ 1 ] ->   (3000)   x - 1 {FUNCTION}
[ 1 ] <-   (3000)   x - 1
[ 1 ] ->   (1000)   x * y {FUNCTION}
[ 1 ] <-   (1000)   x * y
15
```

Alle Berechnungen laufen im Hauptthread der Anwendung. Der Viewer zeigt folgendes Bild:



Neben der oben verwendeten Klasse `sequential.Node` existiert die Klasse `parallel.Node` – eine Klasse, die Parallelität ermöglicht. Bei der Verwendung dieser Klasse zeigt der Viewer folgendes Bild:



Die Berechnung ist offenbar 2 Sekunden schneller als die rein sequentielle. Thread 12 und Thread 18 führen parallel die beiden `Function`s aus.

Der Unterschied zur Klasse `sequential.Node` besteht nur in der Implementierung der `get`-Methoden – und in der Benutzung eines `ExecutorService`s:

```
package parallel;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
```

```
// ...

public abstract class Node<T> {

    public static final ExecutorService executor =
        Executors.Executors.newFixedThreadPool(10);

    public abstract T get();

    private static class SupplierNode<T> extends Node<T> {
        // ...
        synchronized public T get() {
            if (this.result == null) {
                Future<T> future = executor.submit(
                   () -> this.supplier.get());
                this.result = xcall(() -> future.get());
            }
            return this.result;
        }
    }

    private static class FunctionNode<T, R> extends Node<R> {
        // ...
        synchronized public R get() {
            if (this.result == null) {
                Future<R> future = executor.submit(() -> {
                    T value = this.parent.get();
                    return function.apply(value);
                });
                result = xcall(() -> future.get());
            }
            return result;
        }
    }

    private static class BiFunctionNode<T0, T1, R> extends
Node<R> {
        // ...
        synchronized public R get() {
            if (this.result == null) {
                Future<T0> future0 = executor.submit(() -> {
                    return this.previous0.get();
                });
                Future<T1> future1 = executor.submit(() -> {
                    return this.previous1.get();
                });
```

```
                T0 value0 = xcall(() -> future0.get());
                T1 value1 = xcall(() -> future1.get());
                Future<R> future = executor.submit(() -> {
                    return function.apply(value0, value1);
                });
                result = xcall(() -> future.get());
            }
            return result;
        }
    }

    private static class ConsumerNode<T> extends Node<Void> {
        // ...
        public Void get() {
            Future<Void> future = executor.submit(() -> {
                T value = this.previous.get();
                consumer.accept(value);
                return null;
            });
            return xcall(() -> future.get());
        }
    }

    private static class RunnableNode extends Node<Void> {
        // ...
        public Void get() {
            Future<Void> future = executor.submit(() -> {
                this.previous.get();
                runnable.run();
                return null;
            });
            return xcall(() -> future.get());
        }
    }

    // ...

}
```

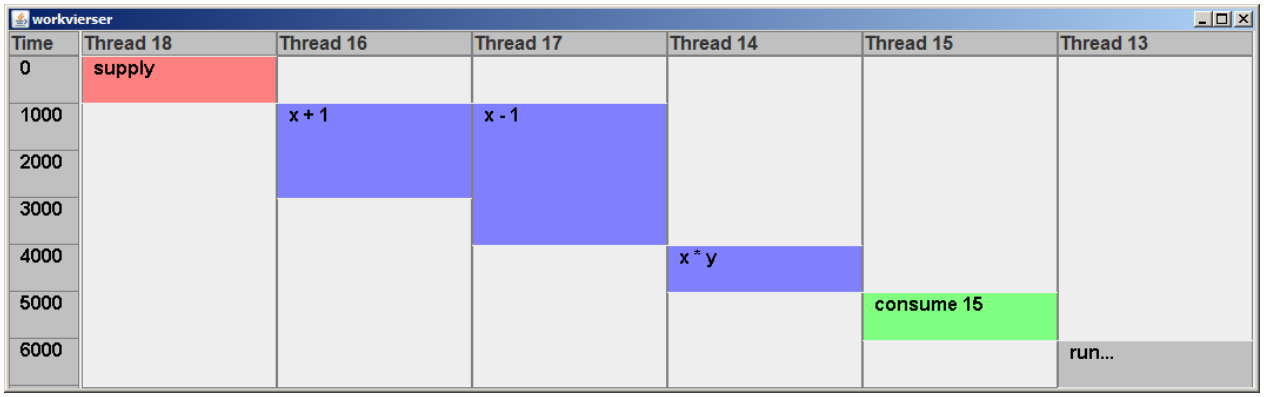
Die eigentliche Arbeit (`Supplier,get`, `Function.apply`, `Consumer.accept` und `Runnable.run`) wird nun jeweils in einem eigenen Thread ausgeführt. Man betrachte insbesondere die Klasse `BiFunctionNode`. Die `get`-Methode dieser Klasse führt die beiden Vorgänger-Schritte jeweils in einem eigenen Thread aus – so dass diese beiden Schritte parallel ausgeführt werden.

Hier ein erweiterter Ablauf, an dessen Ende ein `Consumer` und ein `Runnable` stehen:

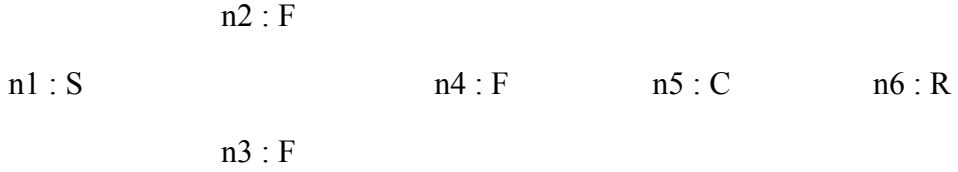
```
    static void demoSFFFCR() {
        Node<Integer> n1 = Node.supply(
            () -> swork("supply", 1000).thenReturn(4));
        Node<Integer> n2 = n1.apply(
            x -> fwork("x + 1", 2000).thenReturn(x + 1));
        Node<Integer> n3 = n1.apply(
            x -> fwork("x - 1", 3000).thenReturn(x - 1));
        Node<Integer> n4 = n2.combine(n3,
           (x, y) -> fwork("x * y", 1000).thenReturn(x * y));
        Node<Void> n5 = n4.accept(
           x -> cwork("consume " + x, 1000));
        Node<Void> n6 = n5.run(
           () -> rwork("run...", 1000));

        Void result = n6.get();
        System.out.println(result);
    }
```

Man beachte, dass das Result von Typ `Void` ist!

Hier der Ablauf im Viewer:



Das folgende Diagramm soll den Ablauf verdeutlichen:

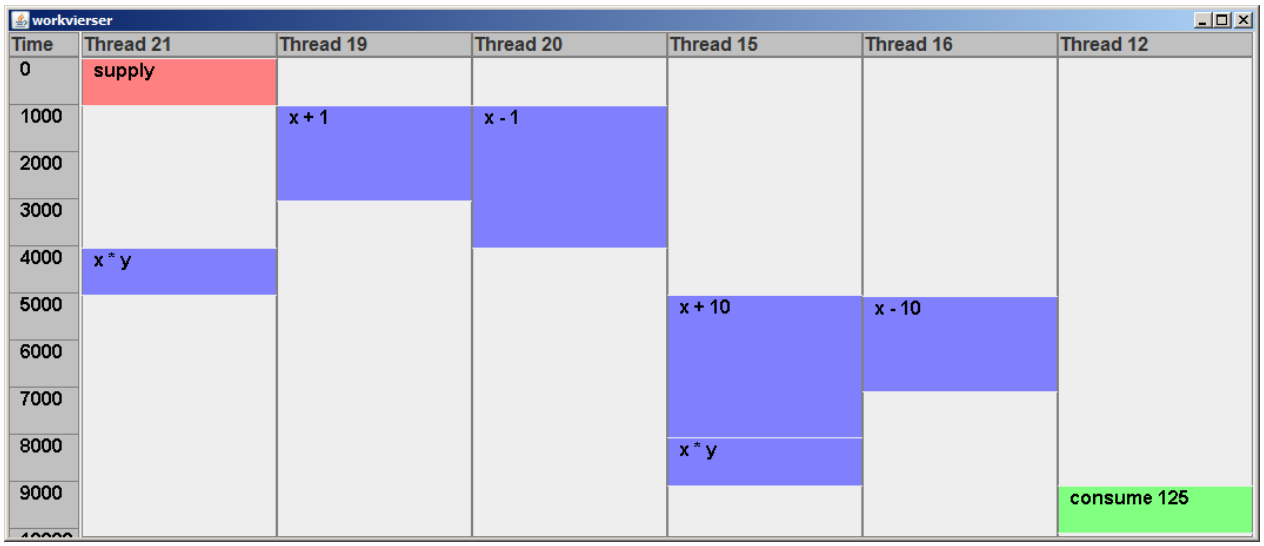


Und eine letzte Erweiterung:

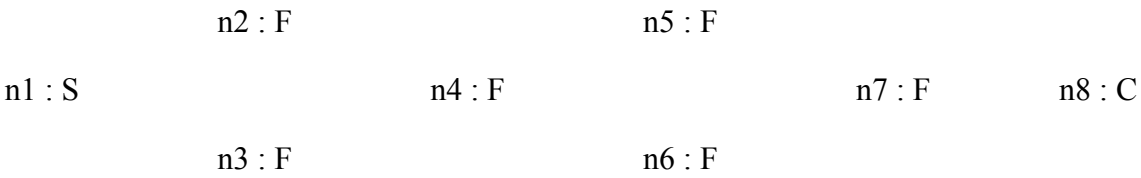
```
    static void demoSFFFFFC() {

        Node<Integer> n1 = Node.supply(
            () -> swork("supply", 1000).thenReturn(4));

        Node<Integer> n2 = n1.apply(
            x -> fwork("x + 1", 2000).thenReturn(x + 1));

        Node<Integer> n3 = n1.apply(
            x -> fwork("x - 1", 3000).thenReturn(x - 1));

        Node<Integer> n4 = n2.combine(n3,
            (x, y) -> fwork("x * y", 1000).thenReturn(x * y));

        Node<Integer> n5 = n4.apply(
            x -> fwork("x + 10", 3000).thenReturn(x + 10));

        Node<Integer> n6 = n4.apply(
            x -> fwork("x - 10", 2000).thenReturn(x - 10));

        Node<Integer> n7 = n5.combine(n6,
           (x, y) -> fwork("x * y", 1000).thenReturn(x * y));

        Node<Void> n8 = n7.accept(
            x -> cwork("consume " + x, 1000));

        Void result = n8.get();
        System.out.println(result);
    }
```

Natürlich liefert auch ein `Consumer` (wie auch ein `Runnable`) nur noch `Void`...

Der Ablauf im Viewer:



Das Diagramm:



Abschließend sei angemerkt, dass die Typ-Parametrisierung der `Node`-Methoden noch verbesserungswürdig ist (`? extends`, `? super`)...

6.12 Aufgaben

Functional - 1

Sei folgende Klasse gegeben:

```
    static class Foo {
        public final int i;
        public final String s;
        public final String data;
        public Foo(int i, String s, String data) {
            this.i = i;
            this.s = s;
            this.data = data;
        }
        @Override
        public String toString() {
            return "Foo [i=" + i + ", s=" + s + ", data=" + data
+ "]";
        }
    }
```

`Foo`-Objekte sollen in einer `BiMap` gespeichert werden können. Die `BiMap` soll wie folgt genutzt werden können:

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

        final BiMap<Integer, String, Foo> map = new BiMap<>(
            (i, s) -> new Foo(i, s, i + ", " + s.toUpperCase()));

        Foo f1 = map.get(1, "one");
        System.out.println(f1);
        // Foo [i=1, s=one, data=1, ONE]

        Foo f2 = map.get(1, "one");
        System.out.println(f1 == f2);
        // true

        map.get(1, "two");
        map.get(5, "red");
        map.get(5, "blue");
        map.get(5, "green");

        Map<String, Foo> m1 = map.get(1);
        for(Map.Entry<String,Foo> e : m1.entrySet())
```

```
            System.out.println(e.getKey() + " ==> " +
e.getValue());
        // one ==> Foo [i=1, s=one, data=1, ONE]
        // two ==> Foo [i=1, s=two, data=1, TWO]

        Map<String, Foo> m2 = map.get(5);
        for(Map.Entry<String,Foo> e : m2.entrySet())
            System.out.println(e.getKey() + " ==> " +
e.getValue());
        // red ==> Foo [i=2, s=red, data=2, RED]
        // blue ==> Foo [i=2, s=blue, data=2, BLUE]
        // green ==> Foo [i=2, s=green, data=2, GREEN]

        System.out.println();
        for (Integer i : map) {
            Map<String, Foo> m = map.get(i);
            for(Map.Entry<String,Foo> e : m.entrySet())
                System.out.println(e.getKey() + " ==> " +
e.getValue());
        }
        // one ==> Foo [i=1, s=one, data=1, ONE]
        // two ==> Foo [i=1, s=two, data=1, TWO]
        // red ==> Foo [i=5, s=red, data=5, RED]
        // blue ==> Foo [i=5, s=blue, data=5, BLUE]
        // green ==> Foo [i=5, s=green, data=5, GREEN]
    }
```

Functional - 2

Studieren Sie folgende Klasse:

```
package ex2;

import java.util.function.DoubleBinaryOperator;

public abstract class Value {
    static private class SimpleValue extends Value {
        final double value;
        SimpleValue(double value) {
            this.value = value;
        }
        public double eval() {
            return this.value;
        }
    }
```

```
    static private class ValuePair extends Value {
        final Value left;
        final Value right;
        final DoubleBinaryOperator op;
        public ValuePair(Value left, Value right,
DoubleBinaryOperator op) {
            this.left = left;
            this.right = right;
            this.op = op;
        }
        public double eval() {
            return this.op.applyAsDouble(left.eval(),
right.eval());
        }
    }

    public Value plus(double v) {
        return plus($(v));
    }
    public Value minus(double v) {
        return minus($(v));
    }
    public Value times(double v) {
        return times($(v));
    }
    public Value div(double v) {
        return div($(v));
    }

    public Value plus(Value v) {
        return new ValuePair(this, v, (x, y) -> x + y);
    }
    public Value minus(Value v) {
        return new ValuePair(this, v, (x, y) -> x - y);
    }
    public Value times(Value v) {
        return new ValuePair(this, v, (x, y) -> x * y);
    }
    public Value div(Value v) {
        return new ValuePair(this, v, (x, y) -> x / y);
    }

    public static Value $(double v) {
        return new SimpleValue(v);
    }
```

```
    public abstract double eval();
}
```

Es existiert auch bereits ein Hauptprogramm:

```
package ex2;

import static ex2.Value.$;

public class Application {
    public static void main(String[] args) {
        Value v = $(42);
        System.out.println(v.eval());
    }
}
```

Die Ausgabe ist `42`. Führen Sie mittels der Value-Klasse etwas komplexere Berechnungen durch (Strichrechnung gemischt mit Punktrechnung etc.)! Programmieren Sie fluent!

7 Erweiterungen der Standardbibliothek

Die Standardbibliothek ist natürlich von den neuen Möglichkeiten in Java 8 nicht unverschont geblieben. In den folgenden Abschnitten werden einige wichtige dieser Erweiterungen vorgestellt.

Eine Übersicht:

- Die Klasse `Arrays` ist insbesondere um die Möglichkeit erweitert worden, bestimme Aufgaben in parallel arbeitenden Threads auszuführen
- Das Interface `Iterable` ist um `forEach` erweitert worden; auch `Collection` und `List` wurden erweitert.
- Das `Map`-Interface hat einige Convenience-Methoden bekommen, welche den Umgang mit `Map`s wesentlich erleichtern.
- `Comparator`-Objekte können nun u.a. miteinander verknüpft werden.
- `Optional` ist eine neue Klasse, deren Objekte optionale Referenzen repräsentieren.
- Per Reflection können nun u.a. die Namen der formalen Parameter einer Methode ermittelt werden.
- `Spliterator` ist ein neues Interface, welches für Split-Join-Zwecke einsetzbar ist.

In den folgenden Abschnitten werden diese Erweiterungen detailliert vorgestellt.

7.1 Arrays

`Arrays` ist u.a. wie folgt erweitern worden:

```
class Arrays {

    public static <T> void parallelSort(T[] a,
        Comparator<? super T> cmp)

    public static void parallelSort(byte[] a)
    public static void parallelSort(short[] a)
    // ...
    public static void parallelSort(double[] a)


    public static <T> void parallelSetAll(
        T[] array, IntFunction<? extends T> generator)

    public static void parallelSetAll(
        int[] array, IntUnaryOperator generator)
    public static void parallelSetAll(
        long[] array, IntToLongFunction generator)
    public static void parallelSetAll(
        double[] array, IntToDoubleFunction generator)


    public static <T> void setAll(
        T[] array, IntFunction<? extends T> generator)

    public static void setAll(
        int[] array, IntUnaryOperator generator)
    public static void setAll(
        long[] array, IntToLongFunction generator)
    public static void setAll(
        double[] array, IntToDoubleFunction generator)

    // ...
}
```

parallelSort

Zusätzlich zur `sort`-Methode gibt's nun die Methode `parallelSort`.

```
    static void demoParallelSort() {
```

```
        final Integer[] array = new Integer[] { 40, 10, 70, 30,
50 } ;
        Arrays.parallelSort(array, (v1, v2) -> {
            tlog("sort " + v1 + " " + v2);
            return v1.compareTo(v2);
        });
    }
```

Die Ausgaben sind (auf den ersten Blick) überraschend:

```
[  1 ] sort 10 40
[  1 ] sort 70 10
[  1 ] sort 70 40
[  1 ] sort 30 40
[  1 ] sort 30 10
[  1 ] sort 50 40
[  1 ] sort 50 70
```

Alles wird in einem einzigen Thread erledigt – aber nur deshalb, weil die Größe des zu sortierenden Arrays sehr klein ist. Mehrere Threads würden sich hier noch nicht lohnen. Weiter unten wird das Performance-Verhalten der alten `sort`- und der neuen `parallelSort`-Methode etwas genauer untersucht.

parallelSetAll

Neben `setAll` gibt's nun `parallelSetAll` (hier demonstriert an der `parallelSetAll(int[])`-Methode). Im Gegensatz zu `parallelSort` verwendet `parallelSetAll` auch bei einem kleinen Array bereits mehrere Threads:

```
    static void demoParallelSetAll() {
        final int[] array = new int[10];
        Arrays.parallelSetAll(array, index -> {
            tlog("setAll " + index);
            return index * 2;
        });
        for (int value : array)
            out.print(value + " ");
        out.println();
    }
```

Die Ausgaben:

```
[  1 ] setAll 6
[  1 ] setAll 5
[ 12 ] setAll 7
[ 12 ] setAll 4
```

```
[ 12 ] setAll 3
[ 12 ] setAll 0
[ 12 ] setAll 9
[ 11 ] setAll 8
[ 10 ] setAll 2
[  1 ] setAll 1
0 2 4 6 8 10 12 14 16 18
```

parallelPrefix

Was mag `parallelPrefix` bewirken?:

```
    static void demoParallelPrefix() {
        final int[] array = new int[10];
        Arrays.setAll(array, index -> index + 1);
        Arrays.parallelPrefix(array, (left, right) -> {
            tlog("parallelPrefix: " + left + " " + right);
            return left + right;
        });
        for (int value : array)
            out.print(value + " ");
        out.println();
    }
```

Die Ausgaben:

```
[  1 ] parallelPrefix: 1 2
[  1 ] parallelPrefix: 3 3
[  1 ] parallelPrefix: 6 4
[  1 ] parallelPrefix: 10 5
[  1 ] parallelPrefix: 15 6
[  1 ] parallelPrefix: 21 7
[  1 ] parallelPrefix: 28 8
[  1 ] parallelPrefix: 36 9
[  1 ] parallelPrefix: 45 10
1 3 6 10 15 21 28 36 45 55
```

Die Bedeutung und Funktionsweise von `parallelPrefix` sollte nun klar sein...

Performance von parallelSet und parallelSort

Abschließend soll die Performance `parallelSetAll` und `parallelSort` näher untersucht werden – jeweils im Vergleich zu `setAll` resp. `sort`:

Zunächst eine Test-Methode zu `setAll` / `parallelSetAll`:

```
    static void demoSetAllPerformance(int arraySize,int loops) {
        final int[] array = new int[arraySize];
        new PerformanceRunner().run("
                setAll " + array.length + " " + loops,
                loops,
                () -> Arrays.setAll(array, index -> 2 * index));
        new PerformanceRunner().run(
                "parallelSetAll " + array.length + " " + loops,
                loops,
                () -> Arrays.parallelSetAll(array, index -> 2 *
index));
    }
```

Zwei Aufrufe:

```
        demoSetAllPerformance(10, 10_000_000);
        demoSetAllPerformance(10_000_000, 10);
```

Die Resultate:

```
setAll 10 10000000             :    95
parallelSetAll 10 10000000     : 18553

setAll 10000000 10             :    86
parallelSetAll 10000000 10     :    85
```

Nur bei sehr großen Arrays könnte sich die Verwendung von `parallelSetAll` lohnen.

Nun zu `sort` / `parallelSort`:

Die folgende kleine Hilfsmethode initialisiert einen `int`-Array – und zwar derart, dass die Elemente möglichst unsortiert sind:

```
    private static void initArray(int[] array) {
        for (int i = 0; i < array.length; i++) {
            array[i] = (int) (array.length * Math.random());
        }
    }
```

Hier die Test-Methode:

```
    static void demoSortPerformance(int arraySize, int loops) {
        mlog();
        final int[] array = new int[arraySize];
        new PerformanceRunner().run(
```

```
                "sort " + array.length + " " + loops,
                loops,
                () -> initArray(array),
                () -> Arrays.sort(array));
        new PerformanceRunner().run(
                "parallelSort " + array.length + " " + loops,
                loops,
                () -> initArray(array),
                () -> Arrays.parallelSort(array));
    }
```

Auch hier zwei Aufrufe:

```
        demoSortPerformance(10, 10_000_000);
        demoSortPerformance(10_000_000, 10);
```

Das Result zeigt, dass sich die parallele Variante nur bei sehr großen Arrays lohnt:

```
sort 10 10000000              :  1803
parallelSort 10 10000000      :  1798

sort 10000000 10              : 16036
parallelSort 10000000 10      :  7587
```

7.2 Iterable, Collection und List

Die Interfaces sind u.a. wie folgt erweitert worden:

```
public interface Iterable<T> {
    // ...
    default void forEach(Consumer<? super T> action)
}
```

```
public interface Collection<T> extends Iterable<T> {
    // ...
    default boolean removeIf(Predicate<? super E> filter)
}
```

```
public interface List<T> extends Collection<T> {
    // ...
    default void sort(Comparator<? super E> c)
    default void replaceAll(UnaryOperator<E> operator)
}
```

forEach

```
    static void demoForEach() {
        Iterable<Integer> list = Arrays.asList(20, 40, 10, 30);
        list.forEach(elem -> out.print(elem + " "));
        out.println();
    }
```

```
20 40 10 30
```

`list` ist deshalb als `Iterable` definiert, um zu zeigen, dass `forEach` für alle `Iterables` funktioniert.

removeIf

```
    static void demoRemoveIf() {
        // Collection<Integer> list = Arrays.asList(20, 40, 10,
30);
        Collection<Integer> list =
            new ArrayList<>(Arrays.asList(20, 40, 10, 30));
        list.removeIf(elem -> elem >= 30);
        list.forEach(s -> out.print(s + " "));
```

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

```
20 10
```

`list` ist deshalb als `Collection` definiert, um zu zeigen, dass `removeIf` für alle `Collections` funktioniert.

(Die erste Zeile würde allerdings eine Exception werfen: "Operation not supported".

sort

```
    static void demoSort() {
        List<Integer> list = Arrays.asList(20, 40, 10, 30);
        // list.sort((i0, i1) -> i0.compareTo(i1)); // List
        list.sort(Comparator.naturalOrder()); // List
        list.forEach(s -> out.print(s + " "));
        out.println();
    }
```

```
10 20 30 40
```

(Die auskommentierte Zeile würde natürlich auch funktionieren.)

replaceAll

```
    static void demoReplaceAll() {
        List<Integer> list = Arrays.asList(20, 40, 10, 30);
        list.replaceAll(elem -> 2 * elem); // List
        list.forEach(s -> out.print(s + " "));
        out.println();
    }
```

```
40 80 20 60
```

7.3 Map

Das `Map`-Interface ist u.a. wie folgt erweitert worden:

```
public interface Map<K,V> {
    // ...
    default V getOrDefault(Object key, V defaultValue) {

    default V putIfAbsent(K key, V value) {
    default V replace(K key, V value) {

    default V computeIfAbsent(K key,
            Function<? super K, ? extends V> mappingFunction) {
    default V computeIfPresent(K key,
            BiFunction<? super K, ? super V, ? extends V>
remappingFunction) {
}
```

Zur Demonstration dieser Methoden wird ein Verfahren für das Ermitteln der Häufigkeit eines Wortes in einer Liste entwickelt. Dieses Verfahren wird in unterschiedlichen Varianten vorgestellt.

Hier die Liste der Wörter, deren Häufigkeit ermittelt werden soll (ein Gedicht von Gertrude Stein):

```
    static final List<String> words = Arrays.asList(
            "eine", "Rose", "ist", "eine", "Rose", "ist", "eine",
"Rose");
```

Die erste, traditionelle Variante:

```
    static void demoOldFashion() {
        Map<String, Integer> counts = new HashMap<>();
        for (String word : words) {
            Integer count = counts.get(word);
            if (count == null) {
                count = 0;
                counts.put(word, count);
            }
            counts.put(word, count + 1);
        }
        out.println(counts);
    }
```

Die Ausgabe:

```
{eine=3, Rose=3, ist=2}
```

Man beachte, dass in einer Multithreading-Umgebung natürlich synchronisiert werden müsste.

getOrDefault

```
    static void demoGetOrDefault() {
        Map<String, Integer> counts = new HashMap<>();
        for (String word : words) {
            int count = counts.getOrDefault(word, 0);
            counts.put(word, count + 1);
        }
        out.println(counts);
    }
```

An `getOrDefault` wird ein Schlüssel und ein Default-Wert übergeben. Wird ein Eintrag für den Schlüssel gefunden, wird der Wert dieses Eintrags zurückgeliefert – ansonsten der an `getOrDefault` übergebene default-Wert.

putIfAbsent / replace

```
    static void demoPutIfAbsentAndReplace() {
        Map<String, Integer> counts = new HashMap<>();
        for (String word : words) {
            counts.putIfAbsent(word, 0);
            int count = counts.get(word);
            counts.replace(word, count + 1);
        }
        out.println(counts);
    }
```

An `putIfAbsent` wird ein Schlüssel und ein Wert übergeben. Falls zu dem Schlüssel noch kein Eintrag existiert, wird ein neuer Eintrag mit dem übergebenen Schlüssel und dem übergebenen Wert eingefügt.

An `replace` wird ebenfalls ein Schlüssel und ein Wert übergeben. Wird zu dem Schlüssel ein Eintrag gefunden, wird der Wert dieses Eintrags durch den neuen Wert ersetzt. Ansonsten gibt's eine Exception. (Man beachte also den Unterschied zur `put`-Methode.)

compute...

```
    static void demoCompute() {
        mlog();
        Map<String, Integer> counts = new HashMap<>();
        for (String word : words) {
            counts.computeIfAbsent(word, (String k) -> 0);
            counts.computeIfPresent(word, (String k, Integer v) -
> v + 1);
        }
        out.println(counts);
    }
```

An `computeIfAbsent` wird ein Schlüssel und ein "Wert-Generator" übergeben. Ist zu dem Schlüssel noch kein Eintrag vorhanden, so wird der Wert-Generator aufgerufen. Ihm wird der Schlüssel als Parameter übergeben – und seine Aufgabe besteht darin, einen zu diesem Schlüssel passenden Wert zu liefern. Dieser Wert wird dann unter dem neuen Schlüssel eingefügt.

An `ComputeIfPresent` wird ebenfalls ein Schlüssel und ein Wert-Generator übergeben. Dem Wert-Generator wird aber neben dem Schlüssel noch der alte Wert übergeben. Der Wert-Generator muss dann einen neuen Wert zurückliefern, der aus dem Schlüssel oder/und dem alten Wert berechnet werden können muss.

Das Resultat ist in allen Fällen dasselbe:

```
{eine=3, Rose=3, ist=2}
```

7.4 Comparator

Das `Comparator`-Interface ist u.a. wie folgt erweitert worden:

```
@FunctionalInterface
public interface Comparator<T> {

    public static <T, U extends Comparable<? super U>>
Comparator<T>
            comparing(Function<? super T, ? extends U>
keyExtractor)

    public static <T> Comparator<T> comparingInt(
            ToIntFunction<? super T> keyExtractor)
    public static <T> Comparator<T> comparingLong(
            ToLongFunction<? super T> keyExtractor)
    public static <T> Comparator<T> comparingLong(
            ToDoubleFunction<? super T> keyExtractor)

    default Comparator<T> thenComparingInt(
            ToIntFunction<? super T> keyExtractor) {
    default Comparator<T> thenComparingLong(
            ToLongFunction<? super T> keyExtractor) {
    default Comparator<T> thenComparingDouble(
            ToDoubleFunction<? super T> keyExtractor) {

    default Comparator<T> reversed()

    default Comparator<T> thenComparing(Comparator<? super T>
other)

    public static <T extends Comparable<? super T>> Comparator<T>
           reverseOrder()
    public static <T extends Comparable<? super T>> Comparator<T>
           naturalOrder()

    public static <T> Comparator<T> nullsFirst(
           Comparator<? super T> comparator)
    public static <T> Comparator<T> nullsLast(
           Comparator<? super T> comparator)

    // ....
}
```

Folgende Klasse wird als Demo-Klasse verwendet:

```
public class Book {
    private String isbn;
    private String title;
    private int price;
    // Konstruktor, getter und setter...
}
```

Es gibt drei Bücher:

```
    static final Book book1 = new Book("1111", "Pascal", 10);
    static final Book book2 = new Book("3333", "Modula", 30);
    static final Book book3 = new Book("5555", "Pascal", 30);
```

Man beachte, dass zwei Bücher denselben Titel haben.

comparing

Wir beginnen mit einem einfachen, traditionellen `Comparator`:

```
    static void demoCompareToWithIsbn() {
        Comparator<Book> c = (b1, b2) ->
b1.getIsbn().compareTo(b2.getIsbn());
        out.println(c.compare(book1, book2)); // -> -2
    }
```

Das geht nun einfacher – mittels des Aufrufs einer statischen `comparing`-Methode, welcher ein "Key-Extractor" übergeben wird:

```
    static void demoComparingWithIsbn() {
        Comparator<Book> c = Comparator.comparing(b ->
b.getIsbn());
        out.println(c.compare(book1, book2)); // -> -2
    }
```

Hier ein weiterer traditioneller `Comparator`, der `int`-Werte vergleicht:

```
    static void demoCompareToWithPrice() {
        Comparator<Book> c = (b1, b2) -> {
            if (b1.getPrice() > b2.getPrice())
                return 1;
            if (b1.getPrice() < b2.getPrice())
                return -1;
            return 0;
        };
```

```
        out.println(c.compare(book1, book2)); // -> -1
    }
```

Das geht mit `comparing` wesentlich einfacher (hier wird geboxt):

```
    static void demoComparingWithPrice() {
        Comparator<Book> c = Comparator.comparing(b ->
b.getPrice());
        out.println(c.compare(book1, book2)); // -> -1
    }
```

Für primitive Datentypen kann man auch `comparingInt` (etc.) verwenden:

```
    static void demoComparingInt() {
        Comparator<Book> c = Comparator.comparingInt(b ->
b.getPrice());
        out.println(c.compare(book1, book2)); // -> -1
    }
```

reversed

Mittels der Instanzmethode `reversed` kann ein neuer `Comparator` erzeugt werden:

```
    static void demoReversed1() {
        mlog();
        Comparator<Book> c = Comparator.comparingInt(b ->
b.getPrice());
        c = c.reversed();
        out.println(c.compare(book1, book2)); // -> 1
    }
```

Dasselbe in einer einzigen Zeile (hier muss aber der Lambda-Parameter explizit typisiert sein):

```
    static void demoReversed2() {
        mlog();
        Comparator<Book> c = Comparator.comparingInt(
            (Book b) -> b.getPrice()).reversed();
        out.println(c.compare(book1, book2)); // -> 1
    }
```

thenComparing

Mittels `thenComparing` kann der Vergleich bei einem zweiten Kriterium fortgesetzt werden, wenn der Vergleich aufgrund des ersten Kriteriums 0 ergibt.

```
    static void demoThenComparing1() {
        Comparator<Book> c = Comparator.comparing(b ->
b.getTitle());
        c = c.thenComparingInt(b -> b.getPrice());
        out.println(c.compare(book1, book3)); // -> -1
    }
```

Dasselbe in einer einzigen Zeile (Lambda-Parameter explizit typisiert):

```
    static void demoThenComparing2() {
        Comparator<Book> c = Comparator.comparing(
            (Book b) -> b.getTitle()).thenComparing(b ->
b.getIsbn());
        out.println(c.compare(book1, book3)); // -> -4
    }
```

Für primitive Typen gibt's `thenComparingInt`, etc.:

```
    static void demoThenComparing3() {
        Comparator<Book> c = Comparator.comparing(b ->
b.getTitle());
        c = c.thenComparingInt(b -> b.getPrice());
        out.println(c.compare(book1, book3)); // -> -1
    }
```

Ein letztes Beispiel:

```
    static void demoThenComparing4() {
        Comparator<Book> c1 =
            (b1, b2) -> b1.getTitle().compareTo(b2.getTitle());
        Comparator<Book> c2 =
            (b1, b2) -> b1.getIsbn().compareTo(b2.getIsbn());
        Comparator<Book> c = c1.thenComparing(c2);
        out.println(c1.compare(book2, book3));
        out.println(c.compare(book2, book3));
    }
```

naturalOrder / reverseOrder

```
    static void demoNaturalOrderReversedOrder() {
        Comparator<Integer> c1 = Comparator.naturalOrder();
        out.println(c1.compare(20, 30)); // -1
```

```
        Comparator<Integer> c2 = Comparator.reverseOrder();
        out.println(c2.compare(20, 30)); // 1
    }
```

nullsFirst / nullsLast

```
    static void demoNullsFirst() {
        Comparator<Integer> c = Comparator.naturalOrder();
        c = Comparator.nullsFirst(c);
        out.println(c.compare(20, 30)); // -1
        out.println(c.compare(null, 30)); // -1
        out.println(c.compare(30, null)); // 1
    }
```

Sortierung mit null-Elementen

Und hier schließlich zwei `Comparator`s im konkreten Einsatz:

```
    static void demoSortNullsFirst() {
        List<Integer> list = Arrays.asList(10, null, 30, null,
20);
        Comparator<Integer> c =
            Comparator.nullsFirst(Comparator.naturalOrder());
        list.sort(c);
        for (Integer v : list)
            out.print(v + " ");
        out.println();
    }
```

Die Ausgaben: `null null 10 20 30`

```
    static void demoSortNullsLast() {
        List<Integer> list = Arrays.asList(10, null, 30, null,
20);
        Comparator<Integer> c =
            Comparator.nullsLast(Comparator.reverseOrder());
        list.sort(c);
        for (Integer v : list)
            out.print(v + " ");
        out.println();
    }
```

Die Ausgaben: `30 20 10 null null`

7.5 Optional

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

Die konsequente Verwendung von `Optional`-Objekten kann `NullPointerException`s nahezu ausschließen.

```
package java.util;

public final class Optional<T>

    public static<T> Optional<T> empty()
    public static <T> Optional<T> of(T value)
    public static <T> Optional<T> ofNullable(T value)

    public T get()
    public boolean isPresent()
    public T orElse(T other)

    @Override public boolean equals(Object obj)
    @Override public int hashCode()
    @Override public String toString()
    // ...
}
```

empty / get

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

```
    static void demoEmtpyGet() {
        Optional<Integer> o = Optional.empty();
        System.out.println(o);
        try {
            o.get();
        }
        catch (NoSuchElementException e) {
            System.out.println("Expected: " + e.getMessage());
        }
    }
```

Die Ausgaben:

```
Optional.empty
Expected: No value present
```

of / get

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

```
    static void demoOfGet() {
        Optional<Integer> o = Optional.of(42);
        System.out.println(o);
        Integer v = o.get();
        System.out.println(v);
    }
```

Die Ausgaben:

```
Optional[42]
42
```

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

```
    static void demoOfException() {
        Integer i = null;
        try {
            Optional<Integer> o = Optional.of(i);
        }
        catch (NullPointerException e) {
            System.out.println("Expected");
        }
    }
```

isPresent

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

```
    static void demoIsPresent1() {
        Optional<Integer> o = Optional.empty();
        if (o.isPresent())
            System.out.println(o.get());
```

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

Die Ausgabe: `not present`

```
    static void demoIsPresent2() {
        Optional<Integer> o = Optional.of(42);
        if (o.isPresent())
            System.out.println(o.get());
        else
            System.out.println("not present");
    }
```

Die Ausgabe: `42`

orElse

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

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

Die Ausgabe: `42`

```
    static void demoOrElse2() {
        Optional<Integer> o = Optional.empty();
        Integer v = o.orElse(77);
        System.out.println(v);
    }
```

Die Ausgabe: `77`

ofNullable

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

```
    static void demoNullable1() {
```

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

Die Ausgabe: `77`

```
    static void demoNullable2() {
        Integer i = 42;
        Optional<Integer> o = Optional.ofNullable(i);
        Integer v = o.orElse(77);
        System.out.println(v);
    }
```

Die Ausgabe: `42`

Wie bereits in der Einleitung zu diesem Abschnitt erwähnt, kann die konsequente Verwendung von `Optional`s die Häufigkeit von `NullPointerException`s radikal reduzieren.

Allerdings ist die Verwendung von `Optional` relativ verbose – und natürlich nicht zwingend. In der Sprache Ceylon z.B. kann man mit "Optionals" wesentlich einfacher arbeiten:

```
Point? p1 = null;

if (exists p1) {
    print(f1.x);
}

Point? p2 = Point(3, 4);

if (exists p2) {
    print(p2.x);
}

Point p3 = null; // illegal!
```

Das `?` und das `if(exist...)`-Konstrukt sind Elemente der Sprache. Resultat: Ceylon kennt natürlich `null`-Pointer – aber keine `NullPointerException`s!

7.6 Reflection

Per Reflection können nun nicht nur die Typen der Parameter einer Methode oder eines Konstruktors ermittelt werden, sondern auch deren Namen (ein Bedürfnis, das man immer schon hatte...). Allerdings müssen die Klassen, die auf diese Weise inspiziert werden sollen, mittels eines speziellen Compilerschalters übersetzt werden:

```
javac -parameters
```

Als Demonstrationsklasse wird `Foo` verwendet:

```
public class Foo {
    public void f(int x, String y, double z) {
    }
    public int g(int hello, int world) {
        return hello + world;
    }
}
```

Im "alten" Java konnten nur die Typen der Parameter ausgegeben werden:

```
    static void demoInspectOld() {
        for (Method m : cls.getDeclaredMethods()) {
            out.print(m.getReturnType().getSimpleName() + " ");
            out.print(m.getName()+ "(");
            Class<?>[] types = m.getParameterTypes();
            for (int i = 0; i < types.length; i++) {
                if (i > 0)
                    out.print(", ");
                out.print(types[i].getSimpleName() + " p" + i);
            }
            out.println(")");
        }
    }
```

Ein Aufruf und seine Ausgaben:

```
    demoInspectOld(Foo.class)
```

```
int g(int p0, int p1)
void f(int p0, String p1, double p2)
```

Ab Java 8 ist ein neuer Reflection-Typ eingeführt worden: der Typ `Parameter`.

Eine Klasse kann nun wie folgt inspiziert werden:

```
    static void demoInspectNew(Class<?> cls) {
        for (Method m : cls.getDeclaredMethods()) {
            out.print(m.getReturnType().getSimpleName() + " ");
            out.print(m.getName() + "(");
            Parameter[] parameters = m.getParameters();
            for (int i = 0; i < parameters.length; i++) {
                Parameter p = parameters[i];
                if (i > 0)
                    out.print(", ");
                out.print(p.getType().getSimpleName() + " " +
p.getName());
            }
            out.println(")");
        }
    }
```

Ein Aufruf und seine Ausgaben:

```
    demoInspectNew(Foo.class)
```

```
int g(int hello, int world)
void f(int x, String y, double z)
```

Die Parameternamen müssen nun also nicht mehr künstlich generiert werden (`p0`, `p1` etc.)

Die meisten Klasse der Standardbibliothek sind allerdings nicht mit "`-parameters`" übersetzt worden. Trotzdem kann man auch dann mit `Method.getParameters` arbeiten (dann werden nun zumindest automatisch die künstlichen Namen generiert).

Auch die Klasse `Object` ist nicht mit "`-parameters`" übersetzt worden:

```
    demoInspectNew(Object.class)
```

```
void finalize()
void wait()
void wait(long arg0, int arg1)
void wait(long arg0)
boolean equals(Object arg0)
...
```

7.7 Spliterator

Angenommen, wir wollen die Summe aller Elemente eines riesengroßen Arrays von Zahlen berechnen. Wir können die Summe in einem einzigen Thread berechnen - oder aber wird splitten den Array in 2 Teile auf. Die Summe der Elemente der "linken" Hälfte berechnen wir einem anderen Thread als die Summe der Elemente der "rechten" Hälfte. Wenn beide Threads ihre Arbeit getan haben, addieren wir die Summen, die beide Threads berechnet haben und erhalten somit die Gesamtsumme aller Array-Elemente.

Natürlich können wir auch vier Threads verwenden: wir können die "linke" Hälfte in zwei weitere Hälften aufsplitten – und ebenso die "rechte" Hälfte. Etc. Irgendwann wird die Splitterei natürlich sinnlos (wenn die Array-Größe bei 1 angelangt ist, ist ein weiteres Splitten gar unmöglich geworden...)

Solche Split-Verfahren werden vom Interface `Spliterator` spezifiziert:

```
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<? super T> action);
    default void forEachRemaining(Consumer<? super T> action);
    Spliterator<T> trySplit();
    long estimateSize();
    // ....
}
```

Ein `Spliterator` ist gleichzeitig auch ein Iterator – der aber in anderer Gestalt daherkommt als der `java.util.Iterator`. Mittels `tryAdvance` kann er jeweils zum nächsten Element einer Menge schreiten und dieses Element an den dieser Methode übergebenen `Consumer` übergeben. Falls kein weiteres Voranschreiten möglich ist, liefert die Methode `false`. Per `forEachRemaining` können alle noch nicht verarbeiteten Elemente einer Menge verarbeitet werden – jedes dieser Elemente wird ebenfalls an einen `Consumer` übergeben.

Neben dieser Iterator-Funktionalität bietet ein `Spliterator` aber auch die Möglichkeit, sich selbst aufzusplitten – die "linke" Hälfte, die bei diesem Split übrigbleibt, wird einem neu erzeugten `Spliterator` übergeben; die "rechte" Hälfte behält der splittende `Spliterator` für sich. Genau dies bewirkt der Aufruf von `trySplit`. Falls weiteres Splitten als sinnlos erscheint, liefert `trySplit` statt eines neuen `Spliterator`s den Wert `null` zurück – derjenige `Spliterator`, auf welchen `trySplit` aufgerufen wird, behält dann alles für sich.

Neben dem Interface `Spliterator` existiert eine Klasse `Spliterators` (Plural!) mit ausschließlich statischen Methoden:

```
public final class Spliterators {
```

```
    public static <T> Spliterator<T> spliterator(Object[] array,
                                     int
additionalCharacteristics)

    public static Spliterator.OfInt spliterator(int[] array,
                                     int
additionalCharacteristics)
    public static Spliterator.OfLong spliterator(long[] array,
                                     int
additionalCharacteristics)
    public static Spliterator.OfDouble spliterator(double[]
array,
                                     int
additionalCharacteristics)
    // ...
}
```

Alle Methoden dieser Klassen liefern jeweils einen `Spliterator` zurück, welcher einen Array splitten kann.

Bevor wir überlegen, wie das Interface `Spliterator` implementiert werden könnte, sollen hier zunächst einige Anwendungen der `Spliterators`-Klasse gezeigt werden.

tryAdvance

```
    static void demoTryAdvance() {
        final Integer[] array = new Integer[] { 10, 20, 30, 40,
50, 60 };
        final Spliterator<Integer> s =
Spliterators.spliterator(array, 0);
        while (s.tryAdvance((Integer v) -> out.print(v + " ")))
            ;
        out.println();
    }
```

```
10 20 30 40 50 60
```

forEachRemaining

```
    static void demoForEachRemaining() {
        final Integer[] array = new Integer[] { 10, 20, 30, 40,
50, 60 };
        final Spliterator<Integer> s =
Spliterators.spliterator(array, 0);
```

```
        s.forEachRemaining((Integer v) -> out.print(v + " "));
        out.println();
    }
```

```
10 20 30 40 50 60
```

tryAdvance / forEachRemaining

```
    static void demoTryAdvanceAndForEachRemaining() {
        final Integer[] array = new Integer[] { 10, 20, 30, 40,
50, 60 };
        final Spliterator<Integer> s =
Spliterators.spliterator(array, 0);
        s.tryAdvance((Integer v) -> out.print(v + " "));
        s.forEachRemaining((Integer v) -> out.print(v + " "));
        out.println();
    }
```

```
10 20 30 40 50 60
```

trySplit

```
    static void demoTrySlit() {
        final Integer[] array = new Integer[] { 10, 20, 30, 40,
50, 60 };
        final Spliterator<Integer> s1 =
Spliterators.spliterator(array, 0);
        final Spliterator<Integer> s2 = s1.trySplit();
        System.out.println(s1);
        System.out.println(s2);
        s1.forEachRemaining(v -> out.print(v + " "));
        s2.forEachRemaining(v -> out.print(v + " "));
        out.println();
    }
```

```
java.util.Spliterators$ArraySpliterator@baf30c
java.util.Spliterators$ArraySpliterator@81197d
40 50 60 10 20 30
```

trySplit - parallel

```
    static void demoTrySplitParallel() {
```

```
        final Integer[] array = new Integer[] { 10, 20, 30, 40,
50, 60 };
        final Spliterator<Integer> s1 =
Spliterators.spliterator(array, 0);
        final Spliterator<Integer> s2 = s1.trySplit();
        final Thread t1 = new Thread(() -> {
            s1.forEachRemaining(v -> {
                xrun(() -> Thread.sleep(100));
                out.print(v + " ");
            });
        });
        t1.start();
        xrun(() -> Thread.sleep(50));
        final Thread t2 = new Thread(() -> {
            s2.forEachRemaining(v -> {
                xrun(() -> Thread.sleep(100));
                out.print(v + " ");
            });
        });
        t2.start();
        xrun(() -> t1.join());
        xrun(() -> t2.join());
        out.println();
    }
```

```
40 10 50 20 60 30
```

Spliterator.OfInt

```
    static void demoIntSpliterator() {
        final Spliterator.OfInt s1 = Spliterators.spliterator(
            new int[] { 10, 20, 30, 40, 50, 60 }, 0);
        final Spliterator.OfInt s2 = s1.trySplit();
        System.out.println(s1);
        System.out.println(s2);
        s1.forEachRemaining((int v) -> out.print(v + " "));
        s2.forEachRemaining((Integer v) -> out.print(v + " "));
        out.println();
    }
```

```
java.util.Spliterators$IntArraySpliterator@1f4acd0
java.util.Spliterators$IntArraySpliterator@bedef2
40 50 60 10 20 30
```

Wie kann das Interface `Spliterator` implementiert werden? Wir definieren eine einfache `Array`-Klasse (eine "Schmalspur-Version" von `ArrayList`), welche einen `Spliterator` zurückliefern kann:

```
public class Array<T> {

    private final T[] elements;

    public Array(T[] elements) {
        final int n = elements.length;
        this.elements = (T[])new Object[n];
        for (int i = 0; i < n; i++)
            this.elements[i] = elements[i];
    }

    public Spliterator<T> spliterator() {
        return new ArraySpliterator<>(this.elements, 0,
this.elements.length);
    }

    static class ArraySpliterator<T> implements Spliterator<T> {

        private final T[] array;
        private int origin;
        private final int fence; // max-index + 1

        ArraySpliterator(T[] array, int origin, int fence) {
            this.array = array;
            this.origin = origin;
            this.fence = fence;
        }

        public boolean tryAdvance(Consumer<? super T> action) {
            if (this.origin >= this.fence)
                return false;
            action.accept((T) this.array[this.origin]);
            this.origin++;
            return true;
        }

        public Spliterator<T> trySplit() {
            int start = this.origin;
            int middle = (start + this.fence) / 2;
            if (start >= middle)
                return null;
            this.origin = middle;
```

```
            return new ArraySpliterator<>(this.array, start,
middle);
        }

        public long estimateSize() {
            return this.fence - this.origin;
        }

        public int characteristics() {
            return 0;
        }
    }
}
```

`spliterator` liefert ein Objekt der Klasse `ArraySpliterator` zurück. Diese Klasse ist als statische innere Klasse von `Array` implementiert. Sie implementiert `Spliterator`. Man studiere insbesondere die Methode `trySplit`...

Einige Anwendungen der `Array`-Klasse:

```
    static void demoArraySpliteratorForEachRemaining() {
        final Array<Integer> array =
            new Array<>(new Integer[] { 10, 20, 30, 40, 50,
60 });
        final Spliterator<Integer> s = array.spliterator();
        s.forEachRemaining(v -> out.print(v + " "));
        out.println();
    }
```

```
10 20 30 40 50 60
```

```
    static void demoArraySpliteratorTryAdvance() {
        final Array<Integer> array =
            new Array<>(new Integer[] { 10, 20, 30, 40, 50,
60 });
        final Spliterator<Integer> s = array.spliterator();
        while (s.tryAdvance(v -> out.print(v + " ")))
            ;
        out.println();
    }
```

```
10 20 30 40 50 60
```

```
    static void demoArraySplitAndEstimateSize() {
        final Array<Integer> array =
```

```
            new Array<>(new Integer[] { 10, 20, 30, 40, 50,
60 });
        final Spliterator<Integer> s1 = array.spliterator();
        System.out.println(
            "s1   : " + s1.estimateSize());
        final Spliterator<Integer> s2 = s1.trySplit();
        System.out.println(
            "s1,s2: " + s1.estimateSize() + " " +
s2.estimateSize());
        while (s2.tryAdvance(v -> out.print(v + " ")))
            ;
        while (s1.tryAdvance(v -> out.print(v + " ")))
            ;
        out.println();
    }
```

```
s1   : 6
s1,s2: 3 3
10 20 30 40 50 60
```

Ein letzter Hinweis: Für das `Spliterator`-Interface existiert bereits eine abstrakte Implementierung: `Spliterators.AbstractSpliterator`. Statt das Interface direkt zu implementieren, könnte man die `ArraySpliterator`-Klasse auch von dieser partiellen Implementierung ableiten:

```
    static class ArraySpliterator<T>
            extends Spliterators.AbstractSpliterator<T> {
        private final T[] array;
        private int origin;
        private final int fence; // max-index + 1

        ArraySpliterator(T[] array, int origin, int fence) {
            super(fence-origin, 0);
            this.array = array;
            this.origin = origin;
            this.fence = fence;
        }

        public boolean tryAdvance(Consumer<? super T> action) {
            if (this.origin >= this.fence)
                return false;
            action.accept((T) this.array[this.origin]);
            this.origin++;
            return true;
        }
```

```
    }
```

7.8 Aufgaben

Mit den Mitteln von Java 7 ist folgende Klasse definiert worden:

```
package ex1;

import java.util.concurrent.RecursiveTask;
import java.util.concurrent.atomic.AtomicInteger;

public class NumberAddingTask extends RecursiveTask<Integer> {

    public static AtomicInteger addCount = new AtomicInteger();

    final int[] array;
    final int start;
    final int end;

    public NumberAddingTask(int[] array) {
        this(array, 0, array.length);
    }

    private NumberAddingTask(int[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        final int result;
        if (this.end - this.start <= 10) {
            int sum = 0;
            for (int i = this.start; i < this.end; i++) {
                sum += this.array[i];
                addCount.incrementAndGet();
            }
            result = sum;
        }
        else {
            final int center = (this.end + this.start) / 2;
            final NumberAddingTask t0 =
                new NumberAddingTask(this.array, this.start,
center);
            final NumberAddingTask t1 =
```

```
                new NumberAddingTask(this.array, center,
this.end);
            result = t0.invoke() + t1.invoke();
        }
        return result;
    }
}
```

Es existiert folgende `Application`:

```
package ex1;

import java.util.concurrent.ForkJoinPool;

public class Application {

    public static void main(String[] args) {
        final int N = 100;
        final int[] array = new int[N];
        for (int i = 0; i < N; i++)
            array[i] = i + 1;
        final NumberAddingTask task = new
NumberAddingTask(array);
        final ForkJoinPool pool = new ForkJoinPool();
        final int sum = pool.invoke(task);
        System.out.println(sum);
    }
}
```

Könnte die Implementierung der Klasse `NumberAddingTask` mit einem `Spliterator` vereinfacht werden?

8 Streams

Java 8 definiert ein neues Stream-API.

Streams können genutzt werden, um Pipelines zu bauen. Eine Pipeline bezieht die zu verarbeitenden Daten aus irgendeiner Quelle und schickt diese Daten dann an verschiedenen Stationen entlang, an denen sie verarbeitet werden können. Die Verarbeitung wird beschrieben wird durch `Function`s, `Predicate`s etc. Sie kann sequentiell oder möglicherweise auch parallel erfolgen. Am Ende der Pipeline steht ein Resultat, welches nicht mehr als Eingabe für eine weitere Station genutz werden kann.

Anstatt eine komplette Verarbeitung prozedural mittels Schleifen und Abfragen zu programmieren, müssen bei der Verwendung von Streams nur noch einzelne Aspekte "ausprogrammiert" werden – die eigentliche Kontrolle über die Verarbeitung liegt bei den Streams. Somit wird der "prozedurale" Stil von einem "deklarativen" Stil abgelöst.

Die zentrale Klasse des neuen API ist `Stream` – sorry: nicht die zentrale Klasse, sondern das zentrale Interface. Der Entwicker muss die eigentlichen Implementierungsklassen überhaupt nicht kennen.

Streams werden mittels Factories erzeugt. Es gibt eine Vielzahl solcher Factories:

- `Arrays.stream`
- `Collection.stream`
- `Stream.of`
- `Stream.empty`
- `Stream.iterate`
- `Stream.generate`
- `Stream.concat`

Die möglichen Verarbeitungsschritte sind durch sog. "Intermediate Operations" festgelegt (die im `Stream`-Interface definiert sind):

- `map`
- `flatMap`
- `filter`
- `peek`
- `distinct`
- `sorted`
- `skip`
- `limit`

Am Ende stehen sog. "terminal Operations":

- `forEach`
- `toArray`

- `reduce`
- `collect`
- `min / max / count`
- `match`
- `find`

Nach einer kleinen Einführung werden diese Factories, die intermediate und die terminal Operations im einzelnen vorgestellt werden. Weiterhin werden die sog. `Collector`s vorgestellt (ein `Collector` dient dazu, die Ergebnisse, die ein Stream liefert, zu sammeln).

Anschließend entwickeln wir ein kleines "Mini-Framework", welches es ermöglicht, einen Stream bei seiner Verarbeitung "zuzuschauen" – sich also in diese Verarbeitung irgenwie einzuklinken. Die hier entwickelten Klassen können insbesondere dann nützlich sein, wenn es darum geht, Performance-Aspekte zu untersuchen.

Abschließend gehen wir etwas genauer auf die Implementierung von Streams ein. U.a. werden wird dabei eine kleine "Mini-Implementierung" des Stream-Konzepts entwickeln.

8.1 Start

Ein erstes kleines Beispiel soll einen Eindruck von der Mächtigkeit des `Stream`-Konzepts vermitteln.

Ein Stream kann erzeugt auf Grundlage einer `List`:

```
static Stream<Integer> createStream() {
    final List<Integer> list = new ArrayList<>();
    for (int i = 0; i < 10; i++)
        list.add(i);
    return list.stream();
}
```

Auf einen solchen Stream kann `forEach` aufgerufen werden – eine Methode, welche einen `Consumer` verlangt. Dieser `Consumer` wird für jedes vom `Stream` gelieferte Element aufgerufen:

```
static void demoForEach() {
    Stream<Integer> stream = createStream();
    stream.forEach(x -> System.out.print(x + " "));
}
```

Die Ausgaben: `0 1 2 3 4 5 6 7 8 9`

Die eigentliche Verarbeitung beginnt hier noch nicht beim Aufruf von `list.stream` – sondern erst dann, wenn `forEach` aufgerufen wird.

Anstatt direkt `forEach` aufzurufen, können wir zunächst zweimal `filter` aufrufen:

```
static void demoFilter() {
    Stream<Integer> stream = createStream();
    stream
        .filter(x -> x > 5)
        .filter(x -> x % 2 == 0)
        .forEach(x -> System.out.print(x + " "));
}
```

Die Ausgaben: `6 8`

An `filter` wird ein `Predicate` übergeben. Elemente, mit denen das `Predicate` nicht einverstanden ist, werden von der weiteren Verarbeitung jeweils ausgeschlossen. Der erste Filter läßt also nur Werte durch, die größer sind als 5; der zweite nur ganzzahlige Werte.

Auf einen `Stream` kann neben `filter` und `forEach` auch die Methode `map` aufgerufen werden. `map` verlangt als Argument eine `Function`:

```
    static void demoMap() {
        Stream<Integer> stream = createStream();
        stream
            .map(x -> x * 2)
            .map(x -> x + 1)
            .forEach(x -> System.out.print(x + " "));
    }
```

Der erste "Mapper" ersetzt jede Zahl des `Stream`s durch das Doppelte dieser Zahl; der zweite Mapper ersetzt jede Zahl durch eine um 1 höhere Zahl.

Die Ausgaben: `1 3 5 7 9 11 13 15 17 19`

Natürlich können `map` und `filter` beliebig miteinander kombiniert werden:

```
    static void demoMapFilter() {
        Stream<Integer> stream = createStream();
        stream
            .map(x -> x * 3)
            .filter(x -> x % 2 == 0)
            .forEach(x -> System.out.print(x + " "));
    }
```

Die Ausgaben: `0 6 12 18 24`

Ein Stream schließlich, dessen Elemente bereits verarbeitet wurden, ist "verbraucht" – er kann nicht erneut "gestartet" werden:

```
    static void demoException() {
        Stream<Integer> stream = createStream();
        stream.forEach(x -> System.out.print(x + " "));
        System.out.println();
        try {
            stream.forEach(x -> System.out.print(x + " "));
        }
        catch (IllegalStateException e) {
            System.out.println("Expected: " + e);
        }
    }
```

Die Meldung:

```
Expected: stream has already been operated upon or closed
```

8.2 Stream-Creation

Bevor ein Stream arbeiten kann, muss er erzeugt werden. In verschiedenen Klassen existieren eine Reihe von Factory-Methoden.

Das Interface `Collection` bietet zwei Instanz-Methoden an:

```
public interface Collection<E> ... {
    // ...
    default Stream<E> stream() { ... }
    default Stream<E> parallelStream() { ... }
}
```

Die Klasse `Arrays` enthält eine Vielzahl von statischen `Stream`-Factory-Methoden:

```
public class Arrays {
    // ...
    public static <T> Stream<T> stream(T[] array)
    public static <T> Stream<T> stream(T[] array,
        int startInclusive, int endExclusive)

    public static IntStream stream(int[] array)
    public static IntStream stream(int[] array,
        int startInclusive, int endExclusive)

    public static LongStream stream(long[] array)
    public static LongStream stream(long[] array,
        int startInclusive, int endExclusive)

    public static DoubleStream stream(double[] array)
    public static DoubleStream stream(double[] array,
        int startInclusive, int endExclusive)
}
```

Und auch das Interface `Stream` selbst enthält statische Factory-Methoden:

```
public interface Stream<T> ... {
    public static<T> Stream<T> empty()
    public static<T> Stream<T> of(T t)
    public static<T> Stream<T> of(T... values)
    public static<T> Stream<T> iterate(final T seed, final
UnaryOperator<T> f)
    public static<T> Stream<T> generate(Supplier<T> s)
    public static <T> Stream<T> concat(
        Stream<? extends T> a, Stream<? extends T> b)
```

```
}
```

Im folgenden wird für jede Factory-Methode ein kleines Beispiel vorgestellt. Die Ausgaben werden i.d.R. nicht weiter dokumentiert – sie sind selbstverständlich.

Collection.stream

Die `steam`-Methode von `Collection<T>` liefert `Stream<T>`:

```
    static void demoCollectionStream() {
        final List<Integer> list = Arrays.asList(1, 2, 3);
        Stream<Integer> stream = list.stream();
        System.out.println(stream.getClass().getName());
        stream.forEach(x -> System.out.print(x + " "));
    }
```

Arrays.stream

An `Arrays.stream` könnte ein `Integer`-Array übergeben werden:

```
    static void demoArraysStream() {
        final Integer[] array = new Integer[] { 1, 2, 3 };
        Stream<Integer> stream = Arrays.stream(array);
        System.out.println(stream.getClass().getName());
        stream.forEach(x -> System.out.print(x + " "));
    }
```

Die letzte Zeile könnte ausführlicher dargestellt werden (der Typ von `x` ist `Integer`):

```
        stream.forEach((Integer x) -> out.print(x + " "));
```

`Arrays.stream` ist überladen – damit primitive Typen performanter behandelt werden können:

```
    static void demoIntStream() {
        final int[] array = new int[] { 1, 2, 3 };
        IntStream stream = Arrays.stream(array);
        System.out.println(stream.getClass().getName());
        stream.forEach(x -> System.out.print(x + " "));
    }
```

Hier liefert `Arrays.stream` nicht `Stream<Integer>`, sondern `IntStream`. Und `x` ist vom Typ `int`. Die letzte Zeile hätte man also auch wie folgt schreiben können:

```
        stream.forEach((int x) -> out.print(x + " "));
```

Neben `IntStream` gibt's natürlich auch `LongStream` und `DoubleStream`:

```
    static void demoLongSream() {
        final long[] array = new long[] { 1, 2, 3 };
        LongStream stream = Arrays.stream(array);
        System.out.println(stream.getClass().getName());
        stream.forEach(x -> System.out.print(x + " "));
    }
```

```
    static void demoDoubleStream() {
        final double[] array = new double[] { 1.0, 2.0, 3.0 };
        DoubleStream stream = Arrays.stream(array);
        System.out.println(stream.getClass().getName());
        stream.forEach(x -> System.out.print(x + " "));
    }
```

Stream.of

An die statische `of`-Methode von `Stream` können beliebig viele Elemente eines Typs `T` übergeben werden:

```
    static void demoStreamOf() {
        Stream<Integer> stream = Stream.of(1, 2, 3);
        System.out.println(stream.getClass().getName());
        stream.forEach(x -> System.out.print(x + " "));
    }
```

Die Ausgaben: `1 2 3`

IntStream.range

An `range` von `IntStream` kann ein Interval übergeben (einschließlich Untergrenze, ausschließlich Obergrenze):

```
    static void demoIntStreamRange() {
        IntStream stream = IntStream.range(1, 4);
        stream.forEach(x -> System.out.print(x + " "));
    }
```

Die Ausgaben: `1 2 3`

IntStream.iterate

An `iterate` von `IntStream` wird ein Anfangswert und eine `Function` übergeben, welche aufgrund des Vorgänger-Werts jeweils den nächsten Wert berechnet. Die Anzahl der Berechnungen (und damit die Menge der gelieferten Ergebnisse) sollte natürlich mittels Aufrufs der `limit`-Methode begrenzt werden:

```
    static void demoIntStreamIterate() {
        final IntStream stream = IntStream.iterate(10, x -> x +
2).limit(5);
        stream.forEach((int x) -> System.out.print(x + " "));
    }
```

Die Ausgaben: `10 12 14 16 18`

Stream.iterate

`iterate` kann auch auf `Stream` aufgerufen werden:

```
    static void demoStreamIterate() {
        final IntStream stream = IntStream.iterate(10, x -> x +
2).limit(5);
        stream.forEach((int x) -> System.out.print(x + " "));
    }
```

Stream.empty

`Stream` enthält schließlich eine Methode `empty`. Sie erzeugt einen leeren `Stream`:

```
    static void demoStreamEmpty() {
        Stream<Integer> stream = Stream.empty();
        stream.forEach((Integer x) -> System.out.print(x + " "));
    }
```

Die Ausgabe ist so leer wie der `Stream`...

Ein Hinweis zur Performance:

Wenn möglich, sollte statt `Stream<Integer>` die Klasse `IntStream` verwendet werden (und das gilt natürlich auch für `Stream<Long>` (Alternative: `LongStream`) und

`Stream<Double>` (Alternative: `DoubleStream`). Das zeigt der folgende Performance-Vergleich (wir benutzen wieder den `PerformanceRunner` aus dem `shared`-Projekt).

Wir verwenden eine kleine Hilfsklasse:

```
static class IntHolder {
    public int value;
}
```

Und folgende Test-Parameter:

```
static final int size = 1_000_000;
static final int loops = 1_000;
```

Zunächst der Performance-Test für `Stream<Integer>`:

```
static void demoPerformanceStream() {
    final PerformanceRunner runner = new PerformanceRunner();
    final Integer[] array = new Integer[size];
    for (int i = 0; i < size; i++)
        array[i] = i;
    final IntHolder h = new IntHolder();
    runner.run("Integer", loops, () -> {
        final Stream<Integer> stream = Arrays.stream(array);
        stream.forEach((Integer x) -> h.value += x);
    });
    System.out.println(h.value);
}
```

(Der Wert vom `IntHolder` wird nur deshalb inkrementiert, um definitiv auszuschließen, dass zur Laufzeit der `forEach` komplett wegoptimiert wird.)

Im Client-Modus benötigt der Test etwa 5,8 Sekunden; im Server-Modus etwa 2,5 Sekunden (bei einer 32-Bit-JVM – die 64-Bit-JVM läuft immer im Server-Modus).

Dann der Test von `IntStream`:

```
static void demoPerformanceIntStream() {
    final PerformanceRunner runner = new PerformanceRunner();
    final int[] array = new int[size];
    for (int i = 0; i < size; i++)
        array[i] = i;
    final IntHolder h = new IntHolder();
    runner.run("int", loops, () -> {
        final IntStream stream = Arrays.stream(array);
        stream.forEach((int x) -> h.value += x);
```

```
        });
        System.out.println(h.value);
    }
```

Im Client-Modus werden 4,8 Sekunden benötigt; im Server-Modus nur 0,5 Sekunden.

Die absoluten Zeiten sind natürlich nicht verallgemeinerbar – die Verhältnisse dieser Zeiten zueinander aber sehr wohl. Zwar ist die Maschine im Client-Modus nur unwesentlich schneller, wenn statt `Stream<Integer>` die Klasse `IntStream` verwendet wird – dafür aber ist der Unterschied im Server-Modus um so größer (um den Faktor 5).

8.3 Intermediate Operations

Im folgenden werden die Intermediate-Operationen vorgestellt – wiederum anhand kleiner Beispiele. Alle Intermediate-Operationen werden auf `Stream`s aufgerufen (sind daher auch im `Stream`-Interface als `default`-Methoden implementiert) – und liefern jeweils ein neues(!) `Stream`-Objekt zurück.

Hier eine Übersicht zu den im folgenden vorgestellten Intermediate Operations:

- `map`
- `flatMap`
- `filter`
- `peek`
- `distinct`
- `sorted`
- `skip`
- `limit`

Wir beginnen mit `map`.

map

An alle `map`-Operationen wird jeweils eine `Function` (resp. eine `ToIntFunction`, `ToLongFunction` oder `ToDoubleFunction`) übergeben

```
public interface Stream<T> ... {
    // ...
    <R> Stream<R> map(
        Function<? super T, ? extends R> mapper);

    IntStream mapToInt(
        ToIntFunction<? super T> mapper);
    LongStream mapToLong(
        ToLongFunction<? super T> mapper);
    DoubleStream mapToDouble(
        ToDoubleFunction<? super T> mapper);
}
```

Der folgende `Stream` transformiert eine Liste von `String`s in einen `Stream` von `int`-Werten (die Länge der Strings):

```
    static void demoMap() {
        final List<String> list = Arrays.asList("red", "green",
"blue");
```

```
        Stream<Integer> stream = list.stream().map(s ->
s.length());
        stream.forEach(x -> out.print(x + " "));
    }
```

Die Ausgaben: `3 5 4`

Statt `map` kann auch `mapToInt` (resp. `mapToLong`, `mapToDouble`) verwendet werden – wir erhalten als Resultat einen `IntStream` (resp. `LongStream`, `DoubleStream`):

```
    static void demoMapToInt() {
        final List<String> list = Arrays.asList("red", "green",
"blue");
        IntStream stream = list.stream().mapToInt(s ->
s.length());
        stream.forEach(x -> out.print(x + " "));
    }
```

Der folgende `Stream` transformiert `String`s zu `double`s:

```
    static void demoMapToDouble() {
        final List<String> list = Arrays.asList("red", "green",
"blue");
        DoubleStream stream = list.stream().mapToDouble(
            s -> s.length() * 0.5);
        stream.forEach(x -> out.print(x + " "));
    }
```

Die Ausgaben: `1.5 2.5 2.0`

flatMap

`flatMap` kann genutzt werden, um aus einer Liste von Listen eine Liste zu fabrizieren. Die an `flatMap` übergebene `Function` muss aus einem Element der "äußeren" Liste einen Stream erzeugen und diesen zurückliefern:

```
public interface Stream<T> ... {
    // ...
    <R> Stream<R> flatMap(
        Function<? super T, ? extends Stream<? extends R>>
mapper);

    IntStream flatMapToInt(
        Function<? super T, ? extends IntStream> mapper);
    LongStream flatMapToLong(
```

```
        Function<? super T, ? extends LongStream> mapper);
    DoubleStream flatMapToDouble(
        Function<? super T, ? extends DoubleStream> mapper);
}
```

Im folgenden werden zwei Listen von `String`s erzeugt. Diese Listen werden zu einer dritten Liste hinzugefügt – einer `List<List<String>>`. `flatMap` "klopft" dann diese Listen "falch":

```
    static void demoFlatMap() {
        final List<String> list1 = Arrays.asList("red", "green",
"blue");
        final List<String> list2 = Arrays.asList("rot", "gruen",
"blau");
        final List<List<String>> list = Arrays.asList(list1,
list2);

        Stream<String> stream =
            list.stream().flatMap((List<String> l) ->
l.stream());
        stream.forEach(s -> out.print(s + " "));
    }
```

Die Ausgaben: `red green blue rot gruen blau`

Auch Arrays von Arrays von `int`s lassen sich flachklopfen (man beachte: wir benutzen `flatMapToInt`):

```
    static void demoFlatMapToInt() {
        final int[] array1 = new int[] { 10, 20, 30 };
        final int[] array2 = new int[] { 40, 50 };
        final int[] array3 = new int[] { 60 };
        final int[][] array = new int[][] { array1, array2,
array3 };

        IntStream stream = Arrays.stream(array).flatMapToInt(
            (int[] a) -> Arrays.stream(a));
        stream.forEach((int i) -> out.print(i + " "));
    }
```

Die Ausgaben: `10 20 30 40 50 60`

filter

An die `filter`-Methode wird ein `Predicate` übergeben:

```
public interface Stream<T> ... {
    // ...
    Stream<T> filter(Predicate<? super T> predicate);
}
```

Der folgende Stream filtert alle Strings aus, deren Länge kleiner oder gleich `3` ist:

```
    static void demoFilter() {
        Stream<String> stream =
            Arrays.asList("red", "green", "blue").stream();
        stream
            .filter(s -> s.length() > 3)
            .forEach(s -> out.print(s + " "));
    }
```

Ausgaben: `green blue`

peek

An `peek` wird ein `Consumer` übergeben. Der `Consumer` wird für jedes Element des Streams aufgerufen:

```
public interface Stream<T> ... {
    // ...
    Stream<T> peek(Consumer<? super T> action);
}
```

```
    static void demoPeek() {
        Stream<String> stream =
            Arrays.asList("red", "green", "blue").stream();
        stream
            .peek(s -> out.print(s.length() + ":"))
            .forEach(s -> out.print(s + " "));
    }
```

Die Ausgaben: `3:red 5:green 4:blue`

distinct

`distinct` sorgt dafür, dass jedes Element "nur einmal" geliefert wird. Sind also mehrere Objekte per `equals` gleich, wird nur eines diese Elemente jeweils durchgelassen. Die Methode ist parameterlos:

```
public interface Stream<T> ... {
    // ...
    Stream<T> distinct();
}
```

Der folgende Input enthält mehrfach die Elemente `"red"` und `"green"`:

```
    static void demoSort() {
        Stream<String> stream = Arrays.asList(
            "red", "green", "red", "blue", "green").stream();
        stream.distinct().forEach(s -> out.print(s + " "));
    }
```

In der Ausgabe erscheinen `"red"` und `"green"` jeweils nur einmal: `red green blue`

sorted

Die Methode `sorted` sorgt für die Sortierung der Elemente. `sorted` ist überladen:

```
public interface Stream<T> ... {
    // ...
    Stream<T> sorted();
    Stream<T> sorted(Comparator<? super T> comparator);
}
```

Hier eine Anwendung der ersten der beiden `sorted`-Methoden:

```
    static void demoSort() {
        Stream<String> stream =
             Arrays.asList("red", "green", "blue").stream();
        stream.sorted().forEach(s -> out.print(s + " "));
```

Die Ausgaben sind lexikalisch sortiert: `blue green red`

skip

Mittels `skip` können `n-1` Elemente des Inputs übersprungen werden:

```
public interface Stream<T> ... {
```

```
    // ...
    Stream<T> skip(long n);
}
```

Der folgende `skip`-Aufruf sorgt dafür, dass jeweils ein(!) Element übersprungen wird:

```
    static void demoSkip() {
        Stream<Integer> stream = Arrays.asList(10, 20, 30,
40).stream();
        stream.skip(2).forEach(s -> out.print(s + " "));
    }
```

Die Ausgaben: `30 40`

limit

`limit` sorgt dafür, dass maximal `maxSize` Elemente des Inputs durchgelassen werden:

```
public interface Stream<T> ... {
    // ...
    Stream<T> limit(long maxSize);
}
```

```
    static void demoLimit() {
        Stream<Integer> stream = Arrays.asList(10, 20, 30,
40).stream();
        stream.limit(3).forEach(s -> out.print(s + " "));
    }
```

Die Ausgaben: `10 20 30`

Verketten von intermediate operations

Natürlich können intermediate Operationen zu einer Kette verknüpft werden:

```
    static void demoCombination1() {
        Stream<Integer> stream = Stream.of(33, 55, 44, 11, 22,
66);
        stream
            .skip(1)
            .limit(4)
            .filter(x -> x % 2 == 0)
            .sorted()
            .forEach(s -> out.print(s + " "));
```

```
    }
```

Die Ausgaben: `22 44`

In der folgenden Methode aber gibt's eine Exception:

```
    static void demoCombination2() {
        try [
            Stream<Integer> stream = Stream.of(33, 55, 44, 11,
22, 66);
            stream
               .skip(1)
               .limit(4);
            stream
               .filter(x -> x % 2 == 0)
               .sorted()
               .forEach(s -> out.print(s + " "));
        }
        catch(Exception e) {
            System.out.println("Expected: " + e.getMessage();
        }
    }
```

Die Ausgabe: `Expected: stream has already been operated upon or closed`

Aber so geht's:

```
    static void demoCombination3() {
        Stream<Integer> stream = Stream.of(33, 55, 44, 11, 22,
66);
        stream = stream
           .skip(1)
           .limit(4);
        stream
           .filter(x -> x % 2 == 0)
           .sorted()
           .forEach(s -> out.print(s + " "));
    }
```

Was liefern die z.B. die `map`- und die `filter`-Methoden zurück? Natürlich ein Objekt, dessen Klasse das Interface `Stream` implementiert. Aber von welchem genauen Typ sind diese Objekte? Wir benutzen `Features.printInheritance`, um diese Typen und ihrer Ableitungsbeziehungen ausgeben zu lassen:

```
    static void demoInternal() {
        mlog();
```

```
        Stream<Integer> stream = Stream.of(33, 55, 44, 11, 22,
66);
        Features.printInheritance(stream);
        stream = stream.map(x -> x + 1);
        Features.printInheritance(stream);
        stream = stream.filter(x -> x % 2 == 0);
        Features.printInheritance(stream);
        stream.forEach(s -> System.out.print(s + " "));
        System.out.println();
    }
```

Die Ausgabe zeigt, dass es sich jeweils um unterschiedliche Klassen handelt – Klassen aber, die allesamt von `ReferencePipeline` abgeleitet sind:

```
java.util.stream.ReferencePipeline$Head
   java.util.stream.ReferencePipeline
      java.util.stream.AbstractPipeline
         java.util.stream.PipelineHelper

java.util.stream.ReferencePipeline$3
   java.util.stream.ReferencePipeline$StatelessOp
      java.util.stream.ReferencePipeline
         java.util.stream.AbstractPipeline
            java.util.stream.PipelineHelper

java.util.stream.ReferencePipeline$2
   java.util.stream.ReferencePipeline$StatelessOp
      java.util.stream.ReferencePipeline
         java.util.stream.AbstractPipeline
            java.util.stream.PipelineHelper
```

8.4 Terminal Operations

Erst der Aufruf einer terminalen Operation stößt die gesamte Verarbeitung eines Streams an. Eine terminale Operation "verbraucht" die Elemente. Wenn eine terminale Operation noch etwas zurückliefert, dann auf jeden Fall keinen `Stream` mehr

Hier eine Übersicht zu den im folgenden erläuterten terminalen Operationen:

- `forEach`
- `toArray`
- `reduce`
- `collect`
- `min / max / count`
- `match`
- `find`

Wie beginnen mit `forEach`.

forEach

```
public interface Stream<T> ... {
    // ...
    void forEach(Consumer<? super T> action);
    void forEachOrdered(Consumer<? super T> action);
}
```

`forEach` wurde schon immer in den bisherigen Beispielen verwendet. Hier werden einige zusätzlich Varianten vorgestellt.

`forEach` arbeitet per default im sequential-Modus; dies kann auch explizit angefordert werden:

```
   static void demoForEachSequential() {
      Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50)
            .sequential();
        stream.forEach(s -> tlog("forEach: " + s));
   }
```

`util.Util.tlog` gibt u.a. die Thread-Id aus.

Hier die Ausgaben:

```
[ 1 ] forEach: 10
```

```
[ 1 ] forEach: 20
[ 1 ] forEach: 30
[ 1 ] forEach: 40
[ 1 ] forEach: 50
```

`forEach` kann aber auch im parallelen Modus arbeiten:

```
   static void demoForEachParallel() {
      Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50)
            .parallel();
        stream.forEach(s -> tlog("forEach: " + s));
   }
```

Die Ausgaben:

```
[ 1 ] forEach: 30
[ 1 ] forEach: 50
[ 11 ]forEach: 40
[ 10 ]forEach: 20
[ 11 ]forEach: 10
```

Wie man sieht, werden die Ausgaben in drei verschiedenen Thread erzeugt (das ist natürlich rein zufällig). Sie erscheinen daher auch ungeordnet.

Hier schließlich eine Anwendung für `forEachOrdered`:

```
   static void demoForEachOrdered() {
      Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50)
            .parallel();
      stream.forEachOrdered(s -> System.out.print(s + " "));
   }
```

Die Ausgaben:

```
[ 12 ]forEach: 10
[ 12 ]forEach: 20
[ 12 ]forEach: 30
[ 12 ]forEach: 40
[ 12 ]forEach: 50
```

Für die Produktion der Ausgaben wird ein neuer Thread benutzt – aber ein einziger. Nur deshalb können die Ausgaben natürlich auch geordnet sein.

toArray

```
public interface Stream<T> ... {
```

```
    // ...
    Object[] toArray();
    <A> A[] toArray(IntFunction<A[]> generator);
}
```

Die erste Variante von `toArray` liefert den Output eines Streams als `Object`-Array zurück:

```
   static void demoToObjectArray() {
      Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
      Object[] array = stream.toArray();
      for (Object v : array)
         System.out.print(v + " ");
   }
```

Die Ausgaben: `10 20 30 40 50`

Mittels der zweiten kann ein `Integer`-Array erzeugt werden:

```
   static void demoToIntegerArray() {
      Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
      Integer[] array = stream.toArray(n -> new Integer[n]);
      for (Integer v : array)
         System.out.print(v + " ");
   }
```

Die Ausgaben sind natürlich dieselben wie bei der letzten `demo`-Methode.

Und von einem `IntStream` kann ein `int`-Array erzeugt werden

```
   static void demoToIntArray() {
      IntStream stream = IntStream.of(10, 20, 30, 40, 50);
      int[] array = stream.toArray();
      for (int v : array)
         System.out.print(v + " ");
   }
```

reduce

Reduktion bedeutet, dass die Werte eines Streams zu einem einzigen Wert kumuliert werden.

```
public interface Stream<T> ... {
    // ...
    T reduce(T identity, BinaryOperator<T> accumulator);
```

```
    Optional<T> reduce(BinaryOperator<T> accumulator);
    <U> U reduce(U identity,
                 BiFunction<U, ? super T, U> accumulator,
                 BinaryOperator<U> combiner);
}
```

Allen drei `reduce`-Methoden wird ein `accumulator` übergeben – dieser ist entweder ein `BinaryOperator` oder (im letzten Falle) eine `BiFunction`.

Ein erstes Beispiel – welches die Summe aller Werte des Streams berechnet:

```
   static void demoReduce() {
      Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
      Integer sum = stream.reduce(0, (x, y) -> {
         System.out.println(x + " " + y);
         return x + y;
      });
      System.out.println("sum = " + sum);
   }
```

Die Ausgaben zeigen anschaulich die Schritte der Reduktion:

```
0 10
10 20
30 30
60 40
100 50
sum = 150
```

Die `0` in der ersten Zeile ist die an `reduce` übergebene "Identität" – bei additiven Operationen muss diese natürlich `0` sein (und bei multiplikativen Operationen `1`).

Wird auf die Trace-Ausgaben verzichtet, kann man das Ganze natürlich etwas knapper formulieren:

```
        System.out.println("sum = " +
            Stream.of(10, 20, 30, 40, 50).reduce(0, (x, y) -> x +
y));
```

Auch ein leerer Stream kann reduziert werden:

```
    static void demoReduceEmpty() {
        Stream<Integer> stream = Stream.empty();
        Integer sum = stream.reduce(0, (x, y) -> x + y);
        System.out.println(sum);
    }
```

Die Ausgabe ist `0` (der Wert der an `reduce` übergebenen Identiät.

Die zweite der oben aufgelisteten `reduce`-Methoden verlangt nur einen `BinaryOperator` – und keine Identität. Sie liefert daher ein `Optional` zurück (der bei einem leeren Stream inhaltslos wäre):

```
    static void demoReduceOptional() {
        Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
        Optional<Integer> result = stream.reduce((x, y) -> x +
y);
        int sum = result.get();
        System.out.println(sum);
    }
```

(Der Aufruf von `result.get` würde bei einem leeren Stream natürlich eine Exception werfen.)

Man kann Reduktion auch parallel betreiben:

```
    static void demoReduceParallel() {
        Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50)
            .parallel();
        Integer product = stream.reduce(1, (x, y) -> x * y);
        System.out.println(product);
    }
```

(Da hier multiplikativ verknüpft wird, wir an `reduce` die Identität `1` übergeben.)

collect

Mittels der `collect`–Methoden können die Elemente eines Streams z.B. zu einer Liste zusammengefaßt werden.

```
public interface Stream<T> ... {
    // ...
    <R> R collect(Supplier<R> supplier,
                  BiConsumer<R, ? super T> accumulator,
                  BiConsumer<R, R> combiner);
    <R, A> R collect(Collector<? super T, A, R> collector);
}
```

Wir beginnen mit der ersten `collect`-Methode. Der an sie zu übergebene `Supplier` muss eine Datenstruktur (z.B. eine `Collection`) zur Verfügung stellen können, welche

die Elemente des `Stream`s aufnehmen wird. Als `accumulator` wird ein `BiConsumer` übergeben, welchem als erster Parameter genau diejenige Datenstrukutur übergeben wird, die der `Supplier` zur Verfügung gestellt hat; als zweiter Parameter wird das aktuelle Element des `Stream`s übergeben. Der `BiConsumer` hat somit die Aufgabe, die Datenstruktur, die der `Supplier` zur Verfügung gestellt hat, um das jeweils aktuelle Element des `Stream`s zu bereichern. Der als letzer Parameter übergebene `BiConsumer` (der `combiner`) muss möglicherweise die Elemente einer zweiten erzeugten Datenstruktur zu einer anderen erzeugten Datenstruktur hinzufügen (er wird im parallelen Betrieb benötigt).

Im folgenden Beispiel stellt der `Supplier` eine `ArrayList` zur Verfügung. Für jedes Element des `Stream`s wird dann der als zweiter Parameter übergebene `BiConsumer` aufgerufen – welcher das aktuelle `Stream`-Element der `ArrayList` hinzufügt. Der `BiConsumer`, der als letzter Parameter übergeben wird, fügt alle Elemente einer zweiten `List` zur ersten `List` hinzu:

```
    static void demoCollectSequential() {
        Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
        List<Integer> list = stream.collect(
            () -> new ArrayList<Integer>(),
            (List<Integer> l, Integer v) -> l.add(v),
            (List<Integer> l1, List<Integer> l2) ->
l1.addAll(l2));
        System.out.println(list);
    }
```

Die Ausgaben: `[10, 20, 30, 40, 50]`

`collect` kann auch parallel operieren:

```
    static void demoCollectParallel() {
        Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50)
            .parallel();
        List<Integer> list = stream.collect(
            () -> {
                tlog("supply");
                return new ArrayList<Integer>();
            }.
            (List<Integer> l, Integer v) -> l.add(v),
            (List<Integer> l1, List<Integer> l2) -> {
                tlog("combine " + l1 + " and " + l2);
                l1.addAll(l2);
            }
        );
        System.out.println(list);
```

```
    }
```

Folgende Ausgaben könnten produziert werden:

```
[ 11 ]supply
[ 11 ]supply
[ 11 ]combine [10] and [20]
[ 12 ]supply
[ 1 ] supply
[ 10 ]supply
[ 10 ]combine [40] and [50]
[ 10 ]combine [30] and [40, 50]
[ 10 ]combine [10, 20] and [30, 40, 50]

[10, 20, 30, 40, 50]
```

Man sieht: beim obigen Ablauf ist der `supplier` offenbar fünf mal aufgerufen worden. Daher muss dann natürlich auchder `combiner` viermal aufgerufen werden.

Nun zur zweiten `collect`-Methode. Diese verlangt statt drei Parametern nur einen einzigen: einen `Colloctor`.

`Collector` ist ein Interface, welches Methoden spezifiziert, welche den benötigten `supplier`, den benötigten `accumulator` und `combiner` zur Verfüng stellen. Zudem existiert eine weitere Methode, welche einen `finisher` bereitstellt.

Im folgenden Beispiel wird dieses Interface in Form einer anonymen Klasse implementiert:

```
    static void demoCollector1() {
        Collector<Integer, List<Integer>, Integer[]> collector =
                new Collector<Integer, List<Integer>,
Integer[]>() {
            public Supplier<List<Integer>> supplier() {
                return () -> new ArrayList<Integer>();
            }

            public BiConsumer<List<Integer>, Integer>
accumulator() {
                return (l, v) -> l.add(v);
            }

            public BinaryOperator<List<Integer>> combiner() {
                return (l1, l2) -> {
                    l1.addAll(l2);
                    return l1;
                };
```

```
            }

            public Function<List<Integer>, Integer[]> finisher()
{
                return l -> l.toArray(new Integer[l.size()]);
            }

            public Set<Characteristics> characteristics() {
                return new HashSet<Characteristics>();
            }
        };

        Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
        Integer[] array = stream.collect(collector);
        System.out.println(Arrays.toString(array));
        Collectors.toList();
    }
```

Der von `supplier` gelieferte `Supplier` stellt hier eine `ArrayList` zur Verfügung. Der von `accumulator` gelieferte `BiConsumer` fügt das aktuelle Element des Streams zu dieser `ArrayList` hinzu. Etc. Die von `finisher` gelieferte `Function` transformiert die `ArrayList` in einen Array – der dann als endgültiges Resultat von `collect` zurückgeliefert werden wird.

Die Ausgaben: `[10, 20, 30, 40, 50]`

Soll aber einfach genau diejenige Liste zurückgeliefert werden, welche der `Supplier` zur Verfügung stellt, so kann die `finisher`-Methode eine `Function` liefern, die ihren Input unverändert als Output returniert:

```
    static void demoCollector2() {
        Collector<Integer, List<Integer>, List<Integer>>
collector =
                new Collector<Integer, List<Integer>,
List<Integer>>() {

            // ...
            public Function<List<Integer>, List<Integer>>
finisher() {
                return l -> l;
            }
       };

        Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
        List<Integer> list = stream.collect(collector);
        System.out.println(list);
```

```
    }
```

Um die Elemente eines `Streams` in einer `List` zu sammeln, kann eine Default-Implementierung von `Collector` verwendet werden:

```
    static void demoCollectorsToList() {
        Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
        List<Integer> list = stream.collect(Collectors.toList());
        System.out.println(list);
    }
```

Auch hier die Ausgaben: `[10, 20, 30, 40, 50]`

min, max, count

Mittels `min` kann das kleinste Element eines `Stream`s, mittels `max` das größte Element und mittels `count` die Anzahl der Elemente bestimmt werden. `min` und `max` liefern `Optional`:

```
public interface Stream<T> ... {
    // ...
    Optional<T> min(Comparator<? super T> comparator);
    Optional<T> max(Comparator<? super T> comparator);
    long count();
}
```

Eine Beispiel:

```
    static void demoMinMaxCount() {
        Stream<Integer> stream1 = Stream.of(10, 20, 30, 40,
50).parallel();
        System.out.println(stream1.min((x, y) ->
x.compareTo(y)));

        Stream<Integer> stream2 = Stream.of(10, 20, 30, 40,
50).parallel();
        System.out.println(stream2.max((x, y) ->
x.compareTo(y)));

        Stream<Integer> stream3 = Stream.of(10, 20, 30, 40,
50).parallel();
        System.out.println(stream3.count());
    }
```

Die Ausgaben:

```
Optional[10]
Optional[50]
5
```

match

An alle `match`-Methoden wird ein Tester in Form eines `Predicate`s übergeben. Alle `match`-Methode liefern `boolean`.

Mittels `anyMatch` kann ermittelt werden, ob in einem `Stream` ein Element enthalten ist, welches dem übergebenen `Predicate` genügt; mittels `allMatch` wird geprüft, ob alle Elemente dem Predicate genügen; und über `noneMatch`, ob keines der Elemente dem `Predicate` genügt:

```
public interface Stream<T> ... {
    // ...
    boolean anyMatch(Predicate<? super T> predicate);
    boolean allMatch(Predicate<? super T> predicate);
    boolean noneMatch(Predicate<? super T> predicate);
}
```

Ein Beispiel:

```
    static void demoMatch() {
        Stream<Integer> stream1 = Stream.of(10, 20, 30, 40,
50).parallel();
        System.out.println(stream1.anyMatch(x -> x == 20));

        Stream<Integer> stream2 = Stream.of(10, 20, 30, 40,
50).parallel();
        System.out.println(stream2.allMatch(x -> x <= 50));

        Stream<Integer> stream3 = Stream.of(10, 20, 30, 40,
50).parallel();
        System.out.println(stream3.noneMatch(x -> x == 42));
    }
```

Die Ausgaben: dreimal `true`.

find

Mittels `findFirst` kann das erste Element eines `Stream`s ermittelt werden; mittels `findAny` irgendeines. Beide Methoden liefern natürlich `Optional` (der Stream könnte leer sein):

```
public interface Stream<T> ... {
    // ...
    Optional<T> findFirst();
    Optional<T> findAny();
}
```

Ein Beispiel:

```
    static void demoFind() {
        Stream<Integer> stream1 = Stream.of(10, 20, 30, 40,
50).parallel();
        System.out.println(stream1.findAny()); // mal so, mal
so....

        Stream<Integer> stream2 = Stream.of(10, 20, 30, 40,
50).parallel();
        System.out.println(stream2.findFirst());
    }
```

Die Ausgaben:

```
Optional[40]
Optional[10]
```

8.5 Collectors

Die Klasse `Collectors` enthält einige statische Factory-Methoden, welche spezielle `Collector`-Objekte zurückliefern:

- `joining`
- `counting`
- `groupingBy`
- `partitionigBy`

joining

Die Methode `joining` liefert einen `Collector`, der seinerseits einen String mit allen Elementen eines `Stream`s zurückliefert – getrennt durch den an die Methode übergebenen Separator:

```
static void demoJoining() {
   List<String> list = Arrays.asList("red", "green", "blue");
   String s = list.stream().collect(Collectors.joining(", "));
   System.out.println(s);
}
```

Die Ausgaben: `red, green, blue`

Das obige Resultat läßt sich allerdings auch etwas einfacher produzieren – Java 8 hat die `String`-Klasse um eine `join`-Methode bereichert:

```
static void demoStringJoin() {
   List<String> list = Arrays.asList("red", "green", "blue");
   String s = String.join(", ", list);
   System.out.println(s);
}
```

Die Ausgaben: `red, green, blue`

Natürlich kann auch ein String produziert werden, welche die Elemente einer `Integer`-Liste enthält – wiederum getrennt durch einen Separator:

```
static void demoJoiningInts() {
   List<Integer> list = Arrays.asList(10, 20, 30);
   String s = list.stream()
         .map(i -> String.valueOf(i))
         .collect(Collectors.joining(", "));
```

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

Die Ausgaben: `10, 20, 30`

counting

`counting` kann benutzt werden, um die Anzahl der Elemente eines Stream zu ermitteln:

```
    static void demoCounting() {
        List<String> list =
            Arrays.asList("a", "bb", "ccc", "ddd", "ee", "f");
        long count =
list.stream().collect(Collectors.counting());
        System.out.println(count);
    }
```

Die Ausgabe: `6`

groupingBy

`groupingBy` liefert einen `Collector`, der seinerseits eine `Map` zurückliefert:

```
   static void demoGroupingBy() {
      List<String> list =
           Arrays.asList("a", "bb", "ccc", "ddd", "ee", "f");
      Map<Integer, List<String>> map =
           list.stream().collect(Collectors.groupingBy(s ->
s.length()));
      System.out.println(map);
   }
```

Die Ausgaben: `{1=[a, f], 2=[bb, ee], 3=[ccc, ddd]}`

Die produzierte `Map` enthält Gruppen von Elementen des `Stream`s. An `groupingBy` wird eine `Function` übergeben, die für jedes Element des Streams aufgerufen wird. Diese `Function` fungiert als Key-Erzeuger: der Output dieser `Function` wird als Key einer Gruppe benutzt. Eine Gruppe hat die Form einer `List`.

An `groupingBy` kann neben der Key-Erzeuger-`Function` ein weiterer `Collector` übergeben werden – z.B. `Collectors.counting`:

```
   static void demoGroupingByCounting() {
```

```
    List<String> list =
         Arrays.asList("a", "bb", "ccc", "ddd", "ee", "f");
    Map<Object, Long> map = list.stream()
         .collect(Collectors.groupingBy(
             s -> s.length(), Collectors.counting()));
    System.out.println(map);
  }
```

Die Ausgaben: `{1=2, 2=2, 3=2}`

partitionigBy

Die Methode `partitionigBy` erzeugt eine `Map` mit zwei Einträgen: mit zwei Listen, die über die Keys `true` und `false` angesprochen werden. Jedes Element des Streams wird zu einer diesen beiden Listen hinzugefügt – abhängig von den Resultat des `Predicate`s, welches an `partitionigBy` übergeben wird. Diejenigen Elemente, für welche das `Predicate` den Wert `true` liefert, landen in der `true`-Liste, die anderen in der `false`-Liste:

```
            List<String> list =
                Arrays.asList("a", "bb", "ccc", "ddd", "ee",
"f");
            Map<Boolean, List<String>> map = list.stream()
                .collect(Collectors.partitioningBy(s ->
s.length() > 2));
            out.println(map);
```

Die Ausgaben: `{false=[a, bb, ee, f], true=[ccc, ddd]}`

8.6 Parallelität

Wie kann das parallele Verhalten von Streams beeinflußt werden? Im folgenden werden zwei Varianten vorgestellt.

Die folgende Methode zeigt zunächst das Standardverhalten:

```
   static void demoStandard(int size, int sleepTime) {
      System.out.println("availableProcessors = " +
            Runtime.getRuntime().availableProcessors());
      final Set<Thread> threads = new HashSet<>();
      IntStream.range(1, size)
         .parallel()
         .peek(x -> xrun(() -> Thread.sleep(sleepTime)))
         .forEach(x -> threads.add(Thread.currentThread()));
      System.out.println("Used Threads: " + threads.size());
   }
```

In `peek` wird `sleepTime` Millisekunden geschlafen; in `forEach` wird der aktuelle Thread zu einem `Set` hinzugefügt (natürlich nur dann, wenn er noch nicht im `Set` existiert), um am Ende die Anzahl der verwendeten Threads ausgeben zu können. Es werden `size` viele Elemente verarbeitet.

Angenommen, als `size` wird `1000` und als `sleepTime` der Wert `2` übergeben. Die verwendete Testmaschine hat 4 Prozessoren.

Die Ausgabe: `Used Threads: 4`

Per default werden also i.d.R. genau so viele Threads erzeugt, wie Prozessoren vorhanden sind (das muss aber nicht exakt übereinstimmen – manchmal werden auch 3 oder 5 Threads genutzt).

Die Ausführung der obigen Methode dauert auf der verwendeten Testmaschine ca. 600 Millisekunden.

Intern verwendet der Stream-Mechanismus einen `ForkJoinPool`. Man kann das Verhalten dieses Pools über eine System-Property einstellen:

```
   static void demoThreadPoolProperty(int size, int sleepTime) {
      final Set<Thread> threads = new HashSet<>();
      System.setProperty(

"java.util.concurrent.ForkJoinPool.common.parallelism",
            "20");
```

```
      IntStream.range(1, size)
         .parallel()
         .peek(x -> xrun(() -> Thread.sleep(sleepTime)))
         .forEach(x -> threads.add(Thread.currentThread()));
      System.out.println("Used Threads: " + threads.size());
   }
```

Auf der Testmaschine werden dann etwa 25 - 30 Threads genutzt. Die Ausführung dauert dann nur ca. 200 Millisekunden (das ist nur etwa ein Drittel der "normalen" Ausführungsdauer).

Die zweite Variante besteht darin, einen eigenen `ForkJoinPool` bereitzustellen – und die Stream an diesen Pool zu übergeben:

```
   static void demoForkJoinPool(int size, int sleepTime) {
      final ForkJoinPool forkJoinPool = new ForkJoinPool(10);
      final Set<Thread> threads = new HashSet<>();
      final Future<?> future = forkJoinPool.submit(() ->
         IntStream.range(1, size)
            .parallel()
            .peek(x -> xrun(() -> Thread.sleep(sleepTime)))
            .forEach(x -> threads.add(Thread.currentThread())));
      xrun(() -> future.get());
      System.out.println("Used Threads: " + threads.size());
   }
```

Es werden dann 10 Thread genutzt – die Ausführungsdauer beträgt 250 Millisekunden.

Zusammemgefaßt (Anzahl Prozessoren, Ausführungsdauer):

```
Standard          4        600
System-Property   25-30    200
Eigener Pool      10       250
```

Sofern die `sleepTime` auf 0 gesetzt wird, verhalten sich die drei Lösungen ganz anders als bei einer `sleepTime` von 2:

```
Standard          4        100
System-Property   25-30    5
Eigener Pool      10       100
```

Man sieht: allgemeine Aussagen sind schwierig. Die Standardvariante schneidet anscheinend immer relativ schlecht ab. Je nach Kontext wird es sich entweder anbieten, entweder einen eigenen `ForkJoinPool` zu verwenden oder aber die System-Property zu setzen.

8.7 Interceptor

Manchmal ist es für das Verständnis von Stream sinnvoll, dem Stream bei seiner Arbeit zuzuschauen – also die Reihenfolge der Schritte zu studieren, die innerhalb des Streams ausgeführt werden. Die gilt natürlich insbesondere dann, wenn Parallelität genutzt wird.

Im folgenden wird ein kleines Tool vorgestellt, mittels dessen man die Arbeit eines Streams recht einfach beobachten kann.

Zur Produktion eines Streams wird stets folgende Methode benutzt:

```
    static Stream<Integer> createStream(int size, boolean
parallel) {
        final List<Integer> list = new ArrayList<>();
        for (int i = 0; i < size; i++)
            list.add(size - i);
        Stream<Integer> stream = list.stream();
        if (parallel)
            stream = stream.parallel();
        return stream;
    }
```

Die Methode produziert einen Stream, der aus `size` vielen Elementen besteht und entweder auf den sequentiellen oder auf den parallelen Operationsmodus eingestellt ist.

Angenommen, wird bauen eine Pipe, an welcher ein Filter und einer Mapper beteiligt sind – und die Arbeit dieser beiden soll protokolliert werden. Hier eine Lösung:

```
        Stream<Integer> stream = createStream(size, parallel);
        stream = stream.filter(v -> {
             System.out.println("--> filter(" + v + ")");
             boolean result = v % 2 == 0);
             System.out.println("<-- filter(" + result + ")");
             return result;
        });
        stream = stream.map(v -> {
             System.out.println("--> map(" + v + ")");
             int result = v + 1);
             System.out.println("<--map(" + result + ")");
             return result;
        });
        stream.forEach(v -> System.out.println(v));
```

Das ist natürlich mühselig zu implementieren – das Ganze wird noch aufwendiger, wenn der Trace je nach Situation ein- oder ausgeschaltet sein soll...

Wir brauchen ein allgemeines Verfahren, mittels dessen wir in den Ablauf der einzelnen Schritte eines Streams einzugreifen können.

Im `shared`-Projekt existiert die Klasse `Interceptor`:

```
package util.streams;
// ...
public abstract class Interceptor {

    protected void before(Stream<?> stream, Object[] args) { }
    protected void after(Stream<?> stream, Object result) { }

    private boolean isOverridden(String methodName, Class<?>...
paramTypes) {
        Class<?> cls = this.getClass();
        while (cls != Interceptor.class) {
            try {
                cls.getDeclaredMethod(methodName, paramTypes);
                return true;
            }
            catch (Exception e) {
                cls = cls.getSuperclass();
            }
        }
        return false;
    }

    private boolean callBefore =
        isOverridden("before", Stream.class, Object[].class);
    private boolean callAfter =
        isOverridden("after", Stream.class, Object.class);

    public final <T, R> Stream<R> map(
            Stream<T> stream, Function<T, R> function) {
        return stream.map(v -> {
            if (callBefore)
                this.before(stream, new Object[] { v });
            R result = function.apply(v);
            if (callAfter)
                this.after(stream, result);
            return result;
        });
    }
```

```
public final <T> Stream<T> filter(
        Stream<T> stream, Predicate<T> predicate) {
    return stream.filter(v -> {
        if (callBefore)
            this.before(stream, new Object[] { v });
        boolean result = predicate.test(v);
        if (callAfter)
            this.after(stream, result);
        return result;
    });
}

public final <T> Stream<T> sorted(
        Stream<T> stream, Comparator<T> comparator) {
    return stream.sorted((v1, v2) -> {
        if (callBefore)
            this.before(stream, new Object[] { v1, v2 });
        int result = comparator.compare(v1, v2);
        if (callAfter)
            this.after(stream, result);
        return result;
    });
}

public final <T> Stream<T> peek(
        Stream<T> stream, Consumer<T> consumer) {
    return stream.peek(v -> {
        if (callBefore)
            this.before(stream, new Object[] { v });
        consumer.accept(v);
        if (callAfter)
            this.after(stream, null);
    });
}

public final <T> Stream<Void> forEach(
        Stream<T> stream, Consumer<T> consumer) {
    stream.forEach(v -> {
        if (callBefore)
            this.before(stream, new Object[] { v });
        consumer.accept(v);
        if (callAfter)
            this.after(stream, null);
    });
    return null;
```

```
    }

    // must be completed!
}
```

Für jede Stream-Methode (`filter`, `map` etc.) enthält diese Klasse eine Methode gleichen Namens. Jeder dieser Methoden wird ein `Stream` übergeben; und jede dieser Methoden liefert einen `Stream` zurück.

Betrachten wir z.B. die `map`-Methode etwas genauer:

```
    public final <T, R> Stream<R> map(
            Stream<T> stream, Function<T, R> function) {
        return stream.map(v -> {
            if (callBefore)
                this.before(stream, new Object[] { v });
            R result = function.apply(v);
            if (callAfter)
                this.after(stream, result);
            return result;
        });
    }
```

Die `map`-Methode erzeugt aufgrund des ihr übergebenen `Stream`s einen neuen `Stream` – indem auf den übergebenen `Stream` dessen `map`-Methode aufgerufen wird. An diesen wird eine `Function` übergeben. Diese `ruft` zunächst eine `before`-Methode auf (hierzu gleich mehr). Dann ruft sie auf diejenige `Function`, welcher der `map`-Methode des `Interceptor`s übergeben wurde, deren `apply`-Methode auf. Dann wird eine Methode namens `after` aufgerufen. Und schließlich wird das von `apply` gelieferte Resultat des an `stream.map` übergebenen Lambdas returniert.

Die Idee besteht also darin: anstatt direkt auf einen `Stream` die `map`-Methode aufzurufen (und somit einen neuen `Stream` zu erzeugen), wird die `map`-Methode eines `Interceptors` aufgerufen – wobei der `Stream` und eine `Function` übergeben wird. Diese `map`-Methode erzeugt ein Lambda, welches seinerseits dann die `map`-Methode auf den `Stream` aufruft – und den von dieser Methode erzeugten `Stream` zurückliefert (wir "verschachteln" also die `Function` in einer weiteren `Function`). Dabei wird mittels `before` und `after` eingegriffen.

Natürlich wird an die `filter`-Methode statt einer `Function` ein `Predicate` übergeben; und an die `peek`-Methode ein `Consumer`. Etc.

`before` und `after` sind Methoden, die zum Überscheiben gedacht sind. Sie haben allerdings bereits eine (leere) Implementierung. An beide Methoden wird der `Stream`

übergeben. An before werden zusätzlich die Argumente der aufgerufenen Methode übergeben; an `after` zusätzlich deren Resultat.

Da die Produktion eines `Object`-Arrays, in welchem die Parameter verpackt werden, relativ aufwendig ist, findet der Aufruf von `before` (und damit auch das Verpacken der Parameter) nur dann statt, wenn die Methode tatsächlich in einer abgeleiteten Klasse überschrieben ist. Dasselbe gibt für `after`: auch diese Methode wird nur dann aufgerufen, wenn eine abgeleitete Klasse sie überschrieben hat. Um zu testen, ob die Methode überschrieben ist, wird bei der Erzeugung eines `Intercpetors` die Hilfsmethode `isOverridden` aufgerufen – sowohl für die `before`- als auch für die `after`-Methode.

Sofern also `before` und `after` nicht überschrieben sind, hat ein `Interceptor` kaum negativen Einfluß auf die Performance.

Von dieser Klasse kann eine weitere Klasse abgeleitet werden, welche zum Tracen verwendet werden kann:

```
package util.streams;
// ...
public class TraceInterceptor extends Interceptor {
    private final String name;
    private final int sleepMillis;
    private final static Object lock = new Object();

    public TraceInterceptor(String name, int sleepMillis) {
        this.name = name;
        this.sleepMillis = sleepMillis;
    }

    @Override
    protected void before(Stream<?> stream, Object[] args) {
        synchronized (lock) {
            tlog("--> %010x %-10s %s",
                stream.hashCode(), this.name,
Arrays.toString(args));
            if (this.sleepMillis > 0)
                xrun(() -> Thread.sleep(this.sleepMillis));
        }
    }

    @Override
    protected void after(Stream<?> stream, Object result) {
        synchronized (lock) {
            String sResult = result == null ? "" : "returns " +
result;
```

```
            tlog("<-- %010x %-10s %s",
                stream.hashCode(), this.name, sResult);
        }
    }
}
```

Ein `TraceInterceptor` hat einen Namen. Sowohl `before` als auch `after` sind überschrieben. In beiden Methoden werden Trace-Ausgaben produziert. Die `before`-Methode schäft evtl. etwas...

Und hier schließlich einige mögliche Verwendungen.

Wir definieren zunächst eine kleine Helper-Methode (nur deshalb, um weniger schreiben zu müssen):

```
    static TraceInterceptor intercept(String name, int
sleepMillis) {
        return new TraceInterceptor(name, sleepMillis);
    }
```

Hier die erste Demo-Methode:

```
    static void demoFilterMapForEach(
            int size, boolean parallel, int sleepMillis) {
        Stream<Integer> stream = createStream(size, parallel);

        stream = intercept("filter", sleepMillis).filter(stream,
            v -> v % 2 == 0);
        stream = intercept("map", sleepMillis).map(stream,
            v -> v + 1);
        intercept("forEach", sleepMillis).forEach(stream,
            v -> System.out.println(v));
    }
```

Die Ausgaben (bei `size` = 5, `parallel` = true und `sleepMillis` = 100):

```
[  1 ] --> 0001e6f5c3 filter     [3]
[  1 ] <-- 0001e6f5c3 filter     returns false
[  1 ] --> 0001e6f5c3 filter     [2]
[ 10 ] --> 0001e6f5c3 filter     [1]
[ 10 ] <-- 0001e6f5c3 filter     returns false
[  9 ] --> 0001e6f5c3 filter     [5]
[  8 ] --> 0001e6f5c3 filter     [4]
[  8 ] <-- 0001e6f5c3 filter     returns true
[  8 ] --> 00018aaa99 map        [4]
[  9 ] <-- 0001e6f5c3 filter     returns false
[  1 ] <-- 0001e6f5c3 filter     returns true
```

```
[  1 ] --> 00018aaa99 map        [2]
[  8 ] <-- 00018aaa99 map        returns 5
[  8 ] --> 00014d4cd8 forEach    [5]
[  1 ] <-- 00018aaa99 map        returns 3
[  1 ] --> 00014d4cd8 forEach    [3]
5
[  8 ] <-- 00014d4cd8 forEach
3
[  1 ] <-- 00014d4cd8 forEach
```

Wie man erkennt, beginnt der Mapper bereits dann, wenn der Filter noch filtert – die verschiedenen Phasen können sich also überlagern. Auch `forEach` beginnt bereits, obwohl der Filter alle Elemente gefiltert hat.

Das gilt auch dann, wenn der sequentielle Modus eingestellt ist:

```
[  1 ] --> 0000574795 filter     [5]
[  1 ] <-- 0000574795 filter     returns false
[  1 ] --> 0000574795 filter     [4]
[  1 ] <-- 0000574795 filter     returns true
[  1 ] --> 0000f49f1c map        [4]
[  1 ] <-- 0000f49f1c map        returns 5
[  1 ] --> 0001469ea2 forEach    [5]
5
[  1 ] <-- 0001469ea2 forEach
[  1 ] --> 0000574795 filter     [3]
[  1 ] <-- 0000574795 filter     returns false
[  1 ] --> 0000574795 filter     [2]
[  1 ] <-- 0000574795 filter     returns true
[  1 ] --> 0000f49f1c map        [2]
[  1 ] <-- 0000f49f1c map        returns 3
[  1 ] --> 0001469ea2 forEach    [3]
3
[  1 ] <-- 0001469ea2 forEach
[  1 ] --> 0000574795 filter     [1]
[  1 ] <-- 0000574795 filter     returns false
```

Hier ist es allerdings nur ein einziger Thread, welcher die gesamten Arbeiten ausführt.

Die folgende Demo-Methode baut eine pipe, in welcher zusätzlich ein Sorter eingehängt ist:

```
    static void demoFilterMapSortedForEach(
            int size, boolean parallel, int sleepMillis) {
        Stream<Integer> stream = createStream(size, parallel);

        stream = intercept("filter", sleepMillis).filter(stream,
            v -> v % 2 == 0);
        stream = intercept("map", sleepMillis).map(stream,
```

```
            v -> v + 1);
        stream = intercept("sorted", sleepMillis).sorted(stream,
            (v1, v2) -> v1.compareTo(v2));
        intercept("forEach", sleepMillis).forEach(stream,
            v -> System.out.println(v));
    }
```

Die Ausgaben im parallelen Modus (`size` = 10):

```
[  9 ] --> 00004441cf filter     [2]
[  9 ] <-- 00004441cf filter     returns true
[  9 ] --> 0001f87043 map        [2]
[ 10 ] --> 00004441cf filter     [3]
[ 10 ] <-- 00004441cf filter     returns false
[  1 ] --> 00004441cf filter     [4]
[  1 ] <-- 00004441cf filter     returns true
[  8 ] --> 00004441cf filter     [8]
[  8 ] <-- 00004441cf filter     returns true
[  1 ] --> 0001f87043 map        [4]
[ 10 ] --> 00004441cf filter     [9]
[ 10 ] <-- 00004441cf filter     returns false
[  9 ] <-- 0001f87043 map        returns 3
[ 10 ] --> 00004441cf filter     [10]
[  1 ] <-- 0001f87043 map        returns 5
[  1 ] --> 00004441cf filter     [5]
[  8 ] --> 0001f87043 map        [8]
[  1 ] <-- 00004441cf filter     returns false
[ 10 ] <-- 00004441cf filter     returns true
[ 10 ] --> 0001f87043 map        [10]
[ 10 ] <-- 0001f87043 map        returns 11
[  9 ] --> 00004441cf filter     [1]
[  9 ] <-- 00004441cf filter     returns false
[ 10 ] --> 00004441cf filter     [7]
[  1 ] --> 00004441cf filter     [6]
[  1 ] <-- 00004441cf filter     returns true
[  1 ] --> 0001f87043 map        [6]
[  1 ] <-- 0001f87043 map        returns 7
[  8 ] <-- 0001f87043 map        returns 9
[ 10 ] <-- 00004441cf filter     returns false
[  1 ] --> 0000e253f1 sorted     [9, 11]
[  1 ] <-- 0000e253f1 sorted     returns -1
[  1 ] --> 0000e253f1 sorted     [7, 9]
[  1 ] <-- 0000e253f1 sorted     returns -1
[  1 ] --> 0000e253f1 sorted     [5, 7]
[  1 ] <-- 0000e253f1 sorted     returns -1
[  1 ] --> 0000e253f1 sorted     [3, 5]
[  1 ] <-- 0000e253f1 sorted     returns -1
[  8 ] --> 0000ddb982 forEach    [5]
[  1 ] --> 0000ddb982 forEach    [7]
5
```

```
7
[ 10 ] --> 0000ddb982 forEach    [11]
11
[  9 ] --> 0000ddb982 forEach    [3]
3
[ 10 ] <-- 0000ddb982 forEach
[ 10 ] --> 0000ddb982 forEach    [9]
9
[  1 ] <-- 0000ddb982 forEach
[  8 ] <-- 0000ddb982 forEach
[ 10 ] <-- 0000ddb982 forEach
[  9 ] <-- 0000ddb982 forEach
```

Auch hier überlagern sich wieder `map` und `filter`. Allerdings kann `ForEach` hier natürlich erst dann beginnen, wenn `sorted` komplett abgeschlossen ist.

8.8 Stage

Manchmal ist es sinnvoll, die Einzelteile eines Streams "generisch" nach Belieben zusammenstecken zu können. Die Einzelteile sollten zunächst unabhängig von einem Stream z.B. in einem Array abgestellt werden können – und aufgrund eines solchen Arrays (oder einer `List`) sollte dann der Stream erzeugt und ausgeführt werden können. Natürlich sollte auch weiterhin Interception möglich sein.

Normalerweise werden `Stream`s festverdrahtet erzeugt und dabei miteinander verbunden: mal wird die `map`-Methode, mal die `filter`-Methode, mal die `peek`-Methode. All diese Methoden verlangen zudem i.d.R. verschiedene Argumente (`Function`, `Consumer`, ...). Und dies ist das zu lösende Problem: eine Array-förmige Sicht ist natürlich etwas ganz anderes...

Ein Problem ist dazu da, um gelöst zu werden – und bei der Lösung benutzen wir wieder den bereits bekannten `Interceptor` (diese Klasse ist also offenbar wiederverwendbar...):

Von `Interceptor` ist die Klasse `Stage` abgeleitet (ebenfalls im `shared`-Paket definiert):

```
package util.streams;

import java.util.HashSet;
import java.util.Objects;
import java.util.Set;
import java.util.function.BiFunction;
import java.util.stream.Stream;

public class Stage<I, O> extends Interceptor {

    public interface StreamCreator<I,O>
        extends BiFunction<Stage<I,O>, Stream<I>, Stream<O>> {
    }

    private final String name;
    private final StreamCreator<I,O> streamCreator;

    public Stage(String name, StreamCreator<I,O> streamCreator) {
        Objects.requireNonNull(name);
        Objects.requireNonNull(streamCreator);
        this.name = name;
        this.streamCreator = streamCreator;
    }
```

```
    public final String getName() {
        return this.name;
    }

    public Stream<O> createStream(Stream<I> s) {
        return this.streamCreator.apply(this, s);
    }

    @Override
    public int hashCode() {
        return this.name.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == null || this.getClass() != obj.getClass())
            return false;
        return name.equals(((Stage<?,?>) obj).name);
    }

    private static void checkStages(Stage<?, ?>... stages) {
        Set<Stage<?, ?>> set = new HashSet<>();
        for (Stage<?, ?> stage : stages)
            if (!set.add(stage))
                throw new IllegalArgumentException(
                    "duplicate stage-name: '" + stage.getName());
    }

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static void run(Stream<?> stream, Stage<?, ?>...
stages) {
        checkStages();
        for (Stage stage : stages)
            stream = stage.createStream(stream);
    }
}
```

`Stage` definiert zunächst ein neues funktionales Interface: `StreamCreator`. Das Interface ist abgeleitet von `BiFunction`. Der `apply`-Methode des Interfaces muss eine `Stage` und ein `Stream` übergeben werden – die Methode liefert ein `Stream` zurück. Die `apply`-Methode hat also die Aufgabe, unter Zuhilfenahme eines `Stage`-Objekts aufgrund eines `Stream`s einen neuen `Stream` zu erzeugen.

`Stage` ist eine instantiierbare Klasse. Dem Konstruktor wird ein Name und ein `StreamCreator` übergeben.

Die Methode `createStream` benutzt den `StreamCreator`, um aufgrund des ihr übergebenen `Stream`s einen neuen `Stream` zu erzeugen.

Die Klasse überschreibt schließlich `equals` und `hashCode` – `Factory`-Objekte sind daher auch als Schlüssel einer `Map` verwendbar.

`Stage` definiert zudem eine statische Methode namens `run`. Dieser wird der Anfangs-`Stream` übergeben und beliebig viele `Stage`-Objekte. Mittels einer `for`-Schleife können dann aufgrund dieser `Stage`-Objekte die eigentlichen `Stream`s erzeugt und miteinander verkettet werden.

Wir könnten nun folgende `Stage`-Objekte erzeugen (hier werden noch keine `Stream`s erzeugt!):

```
    static final Stage<Integer, Integer> MAP =
        new Stage<Integer, Integer>("map", (stage, stream) ->
            stage.map(stream, v -> v / 10));

    static final Stage<Integer, Integer> FILTER =
        new Stage<Integer, Integer>("filter", (stage, stream) ->
             stage.filter(stream, v -> v % 2 == 0));

    static final Stage<Integer, Integer> SORTED =
        new Stage<Integer, Integer>("filter", (stage, stream) ->
            stage.sorted(stream, (v1, v2) -> v1.compareTo(v2)));

    static final Stage<Integer, Integer> PEEK =
        new Stage<Integer, Integer>("peek", (stage, stream) ->
            stage.peek(stream, v -> { }));

    static final Stage<Integer, Void> FOREACH =
        new Stage<Integer, Void>("forEach", (stage, stream) ->
            stage.forEach(stream, v -> System.out.println(v)));
```

Um den Anfangs-`Stream` zu erzeugen, verwenden wir folgende Methode:

```
    static Stream<Integer> createStream() {
        final Integer[] array= new Integer[] { 40, 30, 20, 10 };
        return Arrays.stream(array);
    }
```

In den folgende Demo-Methoden wird jeweils aufgrund der oben definierten `Stage`-Objekte ein `Stream` gebaut und gestartet.

Zunächst ein Stream, der nur eine `forEach`-Operation enhält:

```
    static void demoForEach() {
        Stage.run(createStream(), FOREACH);
    }
```

Die Ausgaben:

```
40
30
20
10
```

Die zweite Methode erzeugt einen `Stream`, in welchen gemappt und gefiltert wird:

```
    static void demoMapFilterForEach() {
        Stage.run(createStream(), MAP, FILTER, FOREACH);
    }
```

Die Ausgaben:

```
4
2
```

Die dritte Methode enthält eine `map`-, eine `sorted`- und eine `peek`-Operation (abgeschlossen wieder via `forEach`):

```
    static void demoMapSortedPeekForEach() {
        mlog();
        Stage.run(createStream(), MAP, SORTED, PEEK, FOREACH);
    }
```

Die Ausgaben:

```
1
2
3
4
```

8.9 Performance

Mittels des Stage-Konzepts können wir nun sehr einfache Methoden schreiben, welche das Performance-Verhalten beliebiger Kombinationen beliebiger Streams verdeutlichen.

In den folgenden Demo-Methoden verwenden wird wieder die bereits bekannte Methode `createStream`:

```
   static Stream<Integer> createStream(int size, boolean
parallel) {
      final List<Integer> list = new ArrayList<>();
      for (int i = 0; i < size; i++)
         list.add(i);
      Stream<Integer> stream = list.stream();
      if (parallel)
         stream.parallel();
      return stream;
   }
```

Wir leiten von `Stage` die Klasse `SleepingStage` ab:

```
    static class SleepingStage<I, O> extends Stage<I, O> {
        public SleepingStage(String name,
                Stage.StreamCreator<I,O>streamCreator) {
            super(name, streamCreator);
        }
        @Override
        protected void after(Stream<?> stream, Object result) {
            xrun(() -> Thread.sleep(SLEEPTIME));
        }
    }
```

Der `Thread.sleep` wird benötigt, um die Ausführung von Operationen simulieren zu können, deren Ausführung eine Zeitlang dauert...

Wir können dann, ohne bereits irgendwelche `Stream`s zu bauen, folgende `SleepingStage`s erzeugen:

```
    static final Stage<Integer, Integer> MAP =
        new SleepingStage<Integer, Integer>("map", (stage,
stream) ->
            stage.map(stream, v -> v + 1));

    static final Stage<Integer, Integer> FILTER =
```

```
        new SleepingStage<Integer, Integer>("filter", (stage,
stream) ->
            stage.filter(stream, v -> v % 2 == 0));

    static final Stage<Integer, Integer> SORTED =
        new SleepingStage<Integer, Integer>("sorted", (stage,
stream) ->
            stage.sorted(stream, (v1, v2) -> v1.compareTo(v2)));

    static final Stage<Integer, Integer> PEEK =
        new SleepingStage<Integer, Integer>("peek", (stage,
stream) ->
            stage.peek(stream, v -> { }));

    static final Stage<Integer, Void> FOREACH =
        new SleepingStage<Integer, Void>("forEach", (stage,
stream) ->
            stage.forEach(stream, v -> { }));
```

Dann könnten wird folgende Demo-Methode definieren:

```
   static void demoSortedForEach(int size, int times) {
      new PerformanceRunner().run("parallel", times, () ->
         Stage.run(createStream(size, true), SORTED, FOREACH));
      new PerformanceRunner().run("sequential", times, () ->
         Stage.run(createStream(size, false), SORTED, FOREACH));
   }
```

Wir rufen zweimal `Stage.run` auf (einmal für einen sequentiell, einmal für einen parallel arbeitenden `Stream`). An `run` wird jeweils der Start-`Stream` übergeben und beliegig viele weitere `Stage`s. Die `run`-Methode erzeuge aufgrund dieser `Stage`s die benötigten `Stream`s und führt den Ende-`Stream` aus. (Man beachte, dass am Ende der Parameterliste eine `Stage` übergeben wird, welche einen Stream mit einer terminalen Operation erzeugt – wie z.B. `FOREACH`).

`run` baut also einen `Stream` der folgenden Form:

```
stream.sorted(...).forEach
```

Die `Stage.run`-Methode wird `times`-mal vom `PerformanceRunner` aufgerufen werden.

Angenommen, es seien folgende Parameter gesetzt:

```
   final static int SIZE = 100000;
   final static int TIMES = 100;
   final static int SLEEPTIME = 0;
```

Dann könnten etwa folgende die Ausgaben produziert werden:

```
parallel                    :  4670
sequential                  :  6556
```

Wir verändern die Parameter:

```
   final static int SIZE = 1000;
   final static int TIMES = 1;
   final static int SLEEPTIME = 1;
```

```
parallel                    :  1266
sequential                  :  2000
```

Die parallele Ausführung schneidet offenbar umso besser ab, je länger die einzelnen Operationen dauern.

Ein weiterer Test:

```
   static void demoMapSortedPeekForEach(int size, int times) {
      new PerformanceRunner().run("parallel", times, () ->
         Stage.run(createStream(size, true),
               MAP, SORTED, PEEK, FOREACH));
      new PerformanceRunner().run("sequential", times, () ->
         Stage.run(createStream(size, false),
               MAP, SORTED, PEEK, FOREACH));
   }
```

Hier wird jeweils `Stage`s `MAP`, `SORTED`, `PEEK` und `FOREACH` übergeben – also jeweils ein `Stream` der folgenden Form aufgebaut:

```
stream.map(...).sorted(...).peek(...).forEach
```

Zwei Messungen:

```
   final static int SIZE = 100000;
   final static int TIMES = 100;
   final static int SLEEPTIME = 0;
```

```
parallel                    :  7943
sequential                  : 12993
```

```
   final static int SIZE = 1000;
   final static int TIMES = 1;
   final static int SLEEPTIME = 1;
```

```
parallel                        :  1866
sequential                      :  4007
```

Der Unterschied zwischen sequentiellem und parallelem Modus ist auch diesmal umso größer, je länger die einzelnen Operationen dauern.

8.10 Stateless

Die bei der Stream-Verarbeitung benutzten Lambdas sollten - wenn eben möglich - zustandslos sein.

Im folgenden wird ein Lambda benutzt, welches den Zustand einer Datenstruktur ändert:

```
    static void demoSequential() {
        List<Integer> source = Arrays.asList(10, 11, 12, 13, 14,
15);
        List<Integer> result = new ArrayList<>();
        source.stream()
            .filter(x -> x % 2 == 0)
            .forEach(x -> result.add(x));
        result.stream()
            .forEach(x -> out.print(x + " "));
    }
```

Sofern nur ein einziger Thread läuft, ist das zwar problemlos. Wenn aber eine parallele Verarbeitung stattfindet, muss auf jeden Fall synchronisiert werden. Im folgenden wird der Stream via `parallelStream` angefordert:

```
    static void demoParallel() {
        List<Integer> source = Arrays.asList(10, 11, 12, 13, 14,
15);
        List<Integer> result = new ArrayList<Integer>() {
            @Override
            public boolean add(Integer value) {
                xrun(() -> Thread.sleep(100));
                tlog("in add von List");
                synchronized(this) {
                    return super.add(value);
                }
            }
        };
        source.parallelStream()
            .filter(x -> x % 2 == 0)
            .forEach(x -> result.add(x));
        result.stream()
            .forEach(x -> out.print(x + " "));
    }
```

Die Ausgaben:

```
[  9 ] in add von List
```

```
[   8 ] in add von List
[  10 ] in add von List
14 12 10
```

8.11 Non-Interfering

Während der Bearbeitung einer Datenquelle durch einen Stream sollten an der Datenquelle keine Änderungen vorgenommen werden. Das folgende Fragment ist offensichtlich sehr "problematisch":

```
static void demoProblem() {
    try {
        List<Point> points = new ArrayList<>();
        points.add(new Point(1, 1));
        points.add(new Point(2, 2));
        points.stream()
            .forEach(point -> points.add(new Point(0, 0)));
    }
    catch(Exception e) {
        System.out.println("Expected: " + e);
    }
}
```

Hier wird natürlich eine `ConcurrentModificationException` geworfen...

Die folgenden Zeilen aber sind problemlos:

```
static void demoOkay() {
    mlog();
    List<Point> points = new ArrayList<>();
    points.add(new Point(1, 1));
    points.add(new Point(2, 2));
    points.stream()
        .forEach(point -> point.x += 1);
    points.stream()
         .forEach(point -> out.println(point));
}
```

Der Zustand der Elemente kann also geändert werden.

8.12 Account-Beispiel

Gönnen wir uns – nur zur Abwechslung und zum Entspannen - ein einfaches Beispiel: ein Bespiel endlich aber einmal mit "realen", mit "richtigen" Objekten: mit `Customer`s und `Account`s

Die Klasse `Customer` ist trivial:

```
public class Customer {
    private String name;
    public Customer(String name) { ... }
    // etc.
}
```

Ein `Account` hat eine Referenz auf einen `Customer`:

```
public class Account {
    private final int number;
    private final Customer customer;
    private int balance;
    public Account(int number, Customer customer, int balance)
{ ... }
    // etc.
}
```

Wir bauen eine kleine (Daten-)Bank:

```
    static final Customer c1 = new Customer("Nowak");
    static final Customer c2 = new Customer("Rueschenpoehler");
    static final List<Account> accounts = new ArrayList<>();
    static {
        accounts.add(new Account(4711, c1, 100));
        accounts.add(new Account(4712, c2, 200));
        accounts.add(new Account(4713, c1, 300));
        accounts.add(new Account(4714, c2, 400));
    }
```

Jeder der beiden `Customer`s hat zwei `Account`s.

Wir berechnen die Summe aller Guthaben des ersten Kunden (`c1`) – zunächst auf traditionelle Weise:

```
    static void demoSumOfBalancesOldFashion() {
        int sum = 0;
```

```
        for (Account a : accounts) {
            if (a.getCustomer() == c1)
                sum += a.getBalance();
        }
        System.out.println(sum);
    }
```

Die Ausgabe: `400`

Und dasselbe noch einmal – aber ganz anders, ganz modern:

```
    static void demoSumOfBalancesNewFashion() {
        int sum = accounts.stream()
                .filter(a -> a.getCustomer() == c1)
                .map(a -> a.getBalance())
                .reduce(0, (v1, v2) -> v1 + v2);
        System.out.println(sum);
    }
```

Anschließend drucken wird alle Konten des ersten Kunden – zunächst wieder auf traditionelle Weise:

```
    static void demoPrintAccountsOfCustomerOldFashion()    {
        for (int i = 0; i < accounts.size(); i++) {
            Account a = accounts.get(i);
            if (a.getCustomer() == c1)
                System.out.println(a);
        }
    }
```

Das Ganze noch einmal, aber schöner:

```
    static void demoPrintAccountsOfCustomerNewFashion() {
        accounts.stream()
            .filter(a -> a.getCustomer() == c1)
            .forEach(a -> System.out.println(a));
    }
```

Und noch einmal – aber etwas komplizierter (warum einfach, wenn's auch kompliziert geht):

```
    static void demoPrintAccountsOfCustomerNewFashionDifficult()
{
        mlog();
        IntStream.range(0, accounts.size())
            .mapToObj(i -> accounts.get(i))
```

```
            .filter(a -> a.getCustomer() == c1)
            .forEach(a -> System.out.println(a));
    }
```

8.13 Eine einfache Implementierung des Stream-Konzepts

Um zumindest einige elementare Grundlagen der realen Implementierung der Stream-Bibliothek verstehen (erahnen) zu können, bietet es sich natürlich an, eine eigene einfache Implementierung der Grundkonzepte zu erstellen.

Wir definieren unser eigenes, recht spärliches Interface (welches allerdings recht einfach um weitere Operationen wir `reduce`, `peek` etc. erweitert werden könnte – und auch die Implementierung dieser zusätzlichen Operation wäre einfach):

```
package utils;
// ...
import utils.Pipeline.Head;

public interface Stream<T> {

    public static <OUT> Stream<OUT> create(Iterator<OUT>
iterator) {
        return new Head<OUT>(iterator);
    }

    Stream<T> filter(Predicate<? super T> predicate);
    <R> Stream<R> map(Function<? super T, ? extends R> mapper);
    void forEach(Consumer<? super T> action);
}
```

Mittels des Aufrufs der statischen `create`-Methode kann ein Anfangs-`Stream` erzeugt werden. Der Methode wird ein `Iterator` übergeben.

Auf einen `Stream` können dann die intermediate-Operationen `filter` und `map` aufgerufen werden. An `filter` wird ein `Predicate` übergeben, an `map` eine `Function`. Beide liefern jeweils einen neuen `Stream` zurück.

Als terminale Operation kann `forEach` aufgerufen werden. An `onEach` wird ein `Consumer` übergeben; sie liefert `void`.

Zum Verständnis des Konzepts kann man sich die Typ-Parameter zunächst einmal wegdenken – und gedanklich jedes Vorkommen von `T`, `? super T` oder `? extends T` durch z.B. `Integer` ersetzten...

Zwei kleine Demo-Methoden sollen zunächst die Verwendung dieses Interfaces zeigen.

Die erste ist verbose formuliert:

```
    static void demo1() {
        List<String> list = new ArrayList<>();
        list.add("red");
        list.add("green");
        list.add("blue");

        Stream<String> s1 = Stream.create(list.iterator());
        Stream<String> s2 = s1.filter(s -> s.length() > 3);
        Stream<String> s3 = s2.map(s -> s.substring(2));
        Stream<Integer> s4 = s3.map(s -> s.length());
        Stream<Integer> s5 = s4.filter(i -> i <= 2);
        s5.forEach(v -> System.out.println(v));
    }
```

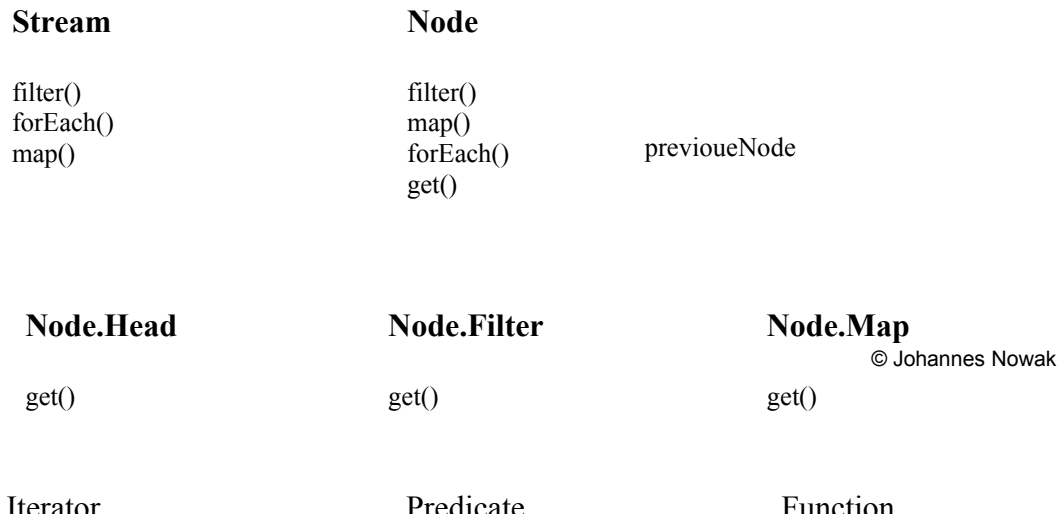
Die Ausgabe: `2`

Die zweite fluent:

```
    static void demo2() {
        mlog();
        List<String> list = new ArrayList<>();
        list.add("red");
        list.add("green");
        list.add("blue");

        Stream.create(list.iterator())
            .filter(s -> s.length() > 3)
            .map(s -> s.substring(2))
            .map(s -> s.length())
            .filter(i -> i <= 2)
            .forEach(v -> System.out.println(v));
    }
```

Man sieht: das Interface kann ganz ähnlich verwendet werden wie das "richtige" Stream-Interface.

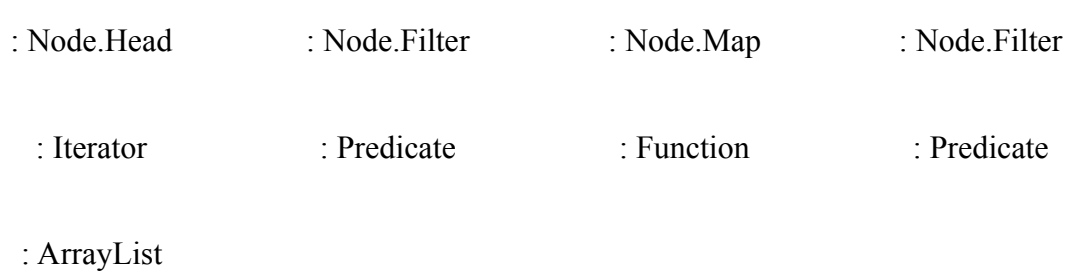
Nun zur Implementierung. Zunächst ein kleines Klassendiagramm:



`Node` ist eine abstrakte Basisklasse, welche alle Methoden von `Stream` bereits implementiert. Sie führt aber eine zusätzliche abstrakte Methode namens `get` ein. Jeder `Node` hat eine Referenz auf seinen Vorgänger-`Node`: `previousStage`. Beim Erzeugen eines `Node`-Objekts muss diese Referenz dem Konstruktor übergeben werden. `Node`-Objekte sind immutable. `Node`-Objekt werden also bei ihrem Erzeugen in Richtung des `Head`-Knotens miteinander verknüpft.

Von `Node` sind drei statische innere Klassen abgeleitet: `Head`, `Filter` und `Map`. Diese Implementieren jeweils die `get`-Methode. Ein `Head`-Objekt referenziert einen Iterator; ein `Filter` ein `Predicat`; und ein `Map`-Objekt referenziert eine `Function`. Man beachte, dass es keinen `Node.ForEach` gibt!

Ein beispielhaftes Objektdiagramm:



Bei der Implementierung ist natürlich zu beachten, dass die Verarbeitung erst dann angestoßen wird, wenn die terminale Operation aufgerufen wird.

`forEach` implementiert eine Schleife, innerhalb derer die `get`-Methode des zuletzt erzeugten `Node`s aufgerufen wird. Diese delegiert an die `get`-Methode des previousNodes – bis der `Head` erreicht ist. Dieser versucht, dem `Iterator` das nächste Element der Eingabe zu entlocken – und returniert dieses Element (sofern kein weiteres Element mehr vorhanden ist, wird `null` returniert. Die `get`-Methoden kehren in der umgekehrten Reihenfolge zurück, in der sich aufgerufen wurden: sie wurden von "rechts" nach "links" aufgerufen, sie kehren von "links" nach "rechts" zurück. Bevor sie aber zu ihrem rechten Nachbar zurückkehren, wird natürlich die mit den `Node` verknüpfte Operation aufgerufen. (Die `get`-Methode eines `Filter`-Nodes muss möglicherweise erst mehere Elemente verwerfen, bevor sie zurückkehrt.)

Nach diesen Erläuterungen sollte die Interpretation der Implementierung kein Problem mehr sein:

```
package utils;
// ...
public abstract class Node<IN,OUT> implements Stream<OUT> {

    public static class Head<OUT> extends Node<Object,OUT> {
        protected Iterator<?> sourceIterator;
        public Head(Iterator<?> sourceIterator) {
            super(null)
            this.sourceIterator = sourceIterator;
        }
        protected OUT get() {
            if (this.sourceIterator.hasNext())
                return (OUT)this.sourceIterator.next();
            return null;
        }
    }
    static class Filter<R> extends Node <R,R> {
        final Predicate<? super R> predicate;
        public Filter(Node<?, R> previousNode,
                Predicate<? super R> predicate) {
            super(previousNode);
            this.predicate = predicate;
        }
        protected R get() {
            R elem = (R)this.previousNode.get();
            while (elem != null && ! predicate.test(elem))
                elem = (R)this.previousNode.get();
            return elem;
        }
    }
    static class Map<IN,OUT> extends Node<IN,OUT> {
        final Function<? super IN, ? extends OUT> mapper;
        public Map(Node<?,IN> previousNode,
                Function<? super IN, ? extends OUT> mapper) {
            super(previousNode);
            this.mapper = mapper;
        }
        protected OUT get() {
            IN elem = (IN)this.previousNode.get();
            return elem == null ? null : mapper.apply(elem);
        }
    }
```

```
    protected final Node previousNode;

    private Node(Node <?,IN> previousNode) {
        this.previousNode = previousNode;
    }

    protected abstract OUT get();

    @Override
    public Stream<OUT> filter(Predicate<? super OUT> predicate) {
        return new Filter<OUT>(this, predicate);
    }

    @Override
    public <R> Stream<R> map(Function<? super OUT, ? extends R>
mapper) {
        return new Map<OUT,R>(this, mapper);
    }

    @Override
    public void forEach(Consumer<? super OUT> action) {
        for(OUT elem = this.get(); elem != null; elem =
this.get()) {
            action.accept(elem);
        }
    }
}
```

Natürlich unterstützt diese Implementierung keine Parallelität...

Aber ihr Verständnis mag für das Verständnis der "realen" Implementierung ein wenig hilfreich sein.

8.14 Hinweise zur realen Implementierung

Im folgenden werden einige Einblicke in die "reale" Implementierung vermittelt.

Wir benutzten eine `readField`-Methode, welche den Wert eines Attributs auslesen kann – wobei alle (auch die privaten) Attribute einer gesamten Vererbungshierarchie berücksichtigt werden:

```
    static Object readField(Object obj, String name) {
        Class<?> cls = obj.getClass();
        while (cls != Object.class) {
            try {
                final Field field = cls.getDeclaredField(name);
                field.setAccessible(true);
                return field.get(obj);
            }
            catch (Exception e) {
                cls = cls.getSuperclass();
            }
        }
        return null;
    }
```

Diese Methode wird in `inspectStream` genutzt, um die Werte einiger privater Variablen eines Stream-Objekts auszulesen (genauer: eines Objekts, dessen Klasse das Interface Stream implementiert):

```
    static void inspectStream(Stream<?> stream) {
        out.println("Stream " + stream); // + " == " +
            stream.getClass().getSuperclass().getName());
        out.println("\tsource            " +
            readField(stream, "sourceStage"));
        out.println("\tprevious          " +
            readField(stream, "previousStage"));
        out.println("\tnext              " +
            readField(stream, "nextStage"));
        out.println("\tsourceSpliterator " +
            readField(stream, "sourceSpliterator"));
        out.println("\tval$predicate     " +
            readField(stream, "val$predicate"));
        out.println("\tval$mapper        " +
            readField(stream, "val$mapper"));
    }
```

Wie man sieht, hat auch ein "richtiger" Stream eine Variable previousStage (vergleichbar mit `previousNode` der im letzten Abschnitt vorgestellten simplen Implementierung). Zusätzlich hat aber jeder Stream noch einen Verweis auf seinen Nachfolger: nextStage. Und auf den Kopf des Streams: `sourceStage`. Knoten können einen Verweis auf ein `Predicate` haben (`val$predicate`), auf eine Mapper-`Function` (`val$mapper`) etc.

Das Hauptprogramm erzeugt einen `Stream`, dessen Quelle eine `ArrayList` ist. An diese Quelle ist ein Filter und ein Mapper angeschlossen. Der `Stream` wird ausgeführt, indem `forEach` aufgerufen wird. Alle drei `Stream`s werden mittels `inspectStream` untersucht:

```
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Hello");
        list.add("World");
        Stream<String> s1 = list.stream();
        Stream<String> s2 = s1.filter(s -> s.startsWith("H"));
        Stream<Integer> s3 = s2.map(s -> s.length());
        s3.forEach(i -> out.println(i));
        inspectStream(s1);
        inspectStream(s2);
        inspectStream(s3);
    }
```

Hier die Ausgaben (der `package`-Name `java.util.stream` ist der Übersichtlichkeit halber jeweils durch `...` ersetzt worden):

```
Stream ...ReferencePipeline$Head@132e575
   source            ...ReferencePipeline$Head@132e575
   previous          null
   next              ...ReferencePipeline$2@af905d
   sourceSpliterator null
   val$predicate     null
   val$mapper        null
Stream ...ReferencePipeline$2@af905d
   source            ...ReferencePipeline$Head@132e575
   previous          ...ReferencePipeline$Head@132e575
   next              ...ReferencePipeline$3@9ee92
   sourceSpliterator null
   val$predicate     appl.Application$$Lambda$1/8460669@f39991
   val$mapper        null
Stream ...stream.ReferencePipeline$3@9ee92
   source            ...ReferencePipeline$Head@132e575
   previous          ...stream.ReferencePipeline$2@af905d
   next              null
   sourceSpliterator null
```

```
val$predicate     null
val$mapper        appl.Application$$Lambda$2/26403060@574795
```

Die Interpretation sei dem Leser / der Leserin überlassen...

8.15 Aufgaben

Hans im Glück (aus Wikipedia)

Hans erhält als Lohn für sieben Jahre Arbeit einen kopfgroßen Klumpen Gold. Diesen tauscht er gegen ein Pferd, das Pferd gegen eine Kuh, die Kuh gegen ein Schwein, das Schwein gegen eine Gans, und die Gans gibt er für einen Schleifstein mitsamt einem einfachen Feldstein her. Er glaubt, jeweils richtig zu handeln, da man ihm sagt, ein gutes Geschäft zu machen. Von Stück zu Stück hat er auf seinem Heimweg scheinbar weniger Schwierigkeiten. Zuletzt fallen ihm noch, als er trinken will, die beiden schweren Steine in einen Brunnen.

"So glücklich wie ich, rief er aus, gibt es keinen Menschen unter der Sonne‘. Mit leichtem Herzen und frei von aller Last ging er nun fort, bis er daheim bei seiner Mutter angekommen war."

– Fassung der Brüder Grimm

Implementieren Sie einen Stream, an dessen Anfang ein Klumpen Gold (oder der Lohnherr von Hans?) und an dessen Ende nichts steht: `void`! Und schauen Sie Hans bei seinem Heimweg zu – mittels `peek`. (Die Klassen `Gold`, `Pferd`, `Kuh`, `Schwein`, `Gans` und `Stein` existieren bereits.)

9 Das Date And Time API

Das neue API ersetzt `Date` und `Calendar`. Es ist an das joda-time-API angelehnt.

Die wichtigsten Klassen im Überblick:

- `Instant` (Zeitpunkt in Nanos)
- `Duration` (Zeitdauer in Nanos)
- `DayOfWeek` / `Month` (Enum-Typen)
- `LocalDateTime`
- `ZonedDateTime`
- `Year`, `YearMonth`, `MonthDay` ("verständlicher" Zeitpunkt)
- `Period` ("verständliche" Zeitdauer)
- `DateTimeFormatter` (Formatierung)

Wir dokumentieren im folgenden nur die Demo-Methoden und deren Ausgaben – und ersparen uns nähere Erläuterungen. Die Demo-Methoden sprechen für sich...

9.1 ChronoUnit

Die Demo-Methode nutzt die folgende Klasse:

```
import java.time.temporal.ChronoUnit;
```

`ChronoUnit` ist ein `enum`-Typ, der Maßeinheiten für Zeitdauern enthält:

```
   static void demoChronoUnit() {
      for (ChronoUnit u : ChronoUnit.values()) {
         out.println(u);
      }

   out.println(ChronoUnit.SECONDS.compareTo(ChronoUnit.YEARS));
      out.println(ChronoUnit.SECONDS.compareTo(ChronoUnit.DAYS));

   out.println(ChronoUnit.SECONDS.compareTo(ChronoUnit.MINUTES));
   }
```

```
Nanos
Micros
Millis
Seconds
Minutes
Hours
HalfDays
Days
Weeks
Months
Years
Decades
Centuries
Millennia
Eras
Forever

-7
-4
-1
```

9.2 Instant

Die Demo-Methoden nutzen die folgenden Klassen:

```
import java.time.Instant;
import java.time.format.DateTimeParseException;
import java.time.temporal.ChronoUnit;
```

Ein `Instant`-Objekt repräsentiert einen Zeitpunkt in Nanosekunden. Der Referenz-Zeitpunkt ist der 1.1.1970. `Instant`-Objekte sind immutable (und also threadsafe). Die Erzeugung geschieht ausschließlich über Factory-Methoden.

Erzeugung

```
    static void demoCreation() {
        Instant d1 = Instant.ofEpochMilli(10);
        out.println(d1);
        Instant d2 = Instant.ofEpochSecond(10);
        out.println(d2);
        Instant d3 = Instant.ofEpochSecond(10, 20);
        out.println(d3);
        Instant d4 = Instant.now();
        out.println(d4);
    }
```

```
1970-01-01T00:00:00.010Z
1970-01-01T00:00:10Z
1970-01-01T00:00:10.000000020Z
2015-02-26T12:20:26.099Z
```

plus / minus

```
    static void demoPlusMinus() {
        Instant now = Instant.now();
        out.println(now);
        Instant d1 = now.plus(10, ChronoUnit.SECONDS);
        out.println(d1);
        Instant d2 = now.plus(10, ChronoUnit.MINUTES);
        out.println(d2);
        Instant d3 = now.minus(10, ChronoUnit.DAYS);
        out.println(d3);
    }
```

```
2015-02-26T12:20:26.099Z
2015-02-26T12:20:36.099Z
2015-02-26T12:30:26.099Z
2015-02-16T12:20:26.099Z
```

isAfter / isBefore

```
    static void demoAfterBefore() {
        Instant now = Instant.now();
        Instant later = now.plus(10, ChronoUnit.SECONDS);
        out.println(now.isAfter(later));
        out.println(now.isBefore(later));
    }
```

```
false
true
```

truncatedTo

```
    static void demoTruncated() {
        Instant now = Instant.now();
        Instant result = now.truncatedTo(ChronoUnit.DAYS);
        out.println(result);
    }
```

```
2015-02-26T00:00:00Z
```

compareTo

```
    static void demoCompareTo() {
        Instant d1 = Instant.ofEpochSecond(10);
        Instant d2 = Instant.ofEpochSecond(20);
        out.println(d1.compareTo(d2));
        out.println(d2.compareTo(d1));
    }
```

```
-1
1
```

parse

```
    static void demoParse() {
```

```
        Instant d1 = Instant.parse("2015-01-20T13:10:05.429Z");
        out.println(d1);
        Instant d2 = Instant.parse("2015-01-20T13:10:05Z");
        out.println(d2);
        // weniger geht nicht...
        try {
            Instant.parse("2015-01-20T13:10Z");
        }
        catch (DateTimeParseException e) {
            out.println("Expected: " + e.getMessage());
        }

    }
```

```
2015-01-20T13:10:05.429Z
2015-01-20T13:10:05Z
Expected: Text '2015-01-20T13:10Z' could not be parsed at index 16
```

9.3 Duration

Die Demo-Methoden nutzen die folgenden Klassen:

```
import java.time.Duration;
import java.time.Instant;
import java.time.format.DateTimeParseException;
import java.time.temporal.ChronoUnit;
```

`Duration`-Objekte repräsentieren eine Zeitspanne (die auf Nanos abgebildet wird). `Duration`-Objekte sind immutable. `Duration`-Objekte können nur mittels Factory-Methoden erzeugt werden.

Erzeugung

```
   static void demoCreation1() {
      Duration d = Duration.of(10, ChronoUnit.MINUTES);
      out.println(d);
      out.println(Duration.of(10, ChronoUnit.SECONDS));
      out.println(Duration.of(10, ChronoUnit.MILLIS));
      out.println(Duration.ofDays(10));
      out.println(Duration.ofHours(10));
      out.println(Duration.ofMinutes(10));
   }
```

```
PT10M
PT10S
PT0.01S
PT240H
PT10H
PT10M
```

```
   static void demoCreation2() {
      Duration d = ChronoUnit.MINUTES.getDuration();
      out.println(d);
      out.println(ChronoUnit.SECONDS.getDuration());
      out.println(ChronoUnit.FOREVER.getDuration());
      Duration d1 = Duration.ofDays(10);
      Duration d2 = Duration.ofDays(10);
      out.println(d1.equals(d2));
      out.println(d1 == d2);
   }
```

```
PT1M
```

```
PT1S
PT2562047788015215H30M7.999999999S
true
false
```

Getter

```
   static void demoGetter() {
      Duration d = Duration.of(10_007, ChronoUnit.MILLIS);
      out.println(d.getSeconds());
      out.println(d.getNano());
   }
```

```
10
7000000
```

plus / minus / between

```
   static void demoPlusMinusBetween() {
      Duration d = Duration.of(10, ChronoUnit.MINUTES);
      Duration d1 = d.plus(Duration.of(5, ChronoUnit.MINUTES));
      out.println(d1);
      Duration d2 = d.plus(5, ChronoUnit.MINUTES);
      out.println(d2);
      Duration d3 = Duration.between(
           Instant.now(), Instant.now().plus(10,
ChronoUnit.MINUTES));
      out.println(d3);
   }
```

```
PT15M
PT15M
PT10M
```

parse

```
   static void demoParse() {
      Duration d = Duration.parse("PT15M");
      out.println(d);
      try {
         Duration.parse("PT15");
      }
      catch (DateTimeParseException e) {
```

```
        out.println("Expected: " + e.getMessage());
    }
  }
```

```
PT15M
Expected: Text cannot be parsed to a Duration
```

9.4 DayOfWeek / Month

Die Demo-Methoden nutzen folgende Klassen:

```
import java.time.DayOfWeek;
import java.time.Month;
```

DayOfWeek

`DayOfWeek` ist ein `enum`, der die Wochentage aufzählt:

```
   static void demoDayOfWeekEnum() {
      for (DayOfWeek d : DayOfWeek.values()) {
         out.print(d + " ");
      }
      out.println();
   }
```

```
MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY SATURDAY SUNDAY
```

```
   static void demoDayOfWeek() {
      out.println(DayOfWeek.of(1) == DayOfWeek.MONDAY);
      out.println(DayOfWeek.of(7) == DayOfWeek.SUNDAY);

      out.println(DayOfWeek.SUNDAY.getValue() == 7);
      out.println(DayOfWeek.MONDAY.getValue() == 1);

      out.println(DayOfWeek.MONDAY.plus(2) ==
DayOfWeek.WEDNESDAY);
      out.println(DayOfWeek.WEDNESDAY.minus(2) ==
DayOfWeek.MONDAY);

      out.println(DayOfWeek.MONDAY.compareTo(DayOfWeek.WEDNESDAY)
== -2);
      out.println(DayOfWeek.WEDNESDAY.compareTo(DayOfWeek.MONDAY)
== 2);
   }
```

Alle Zeilen geben `true` aus.

Month

`Month` ist ein `enum`, der die Monate aufzählt:

```
   static void demoMonthEnum() {
      for (Month m : Month.values()) {
         out.print(m + " ");
      }
      out.println();
   }
```

```
JANUARY FEBRUARY MARCH APRIL MAY JUNE
JULY AUGUST SEPTEMBER OCTOBER NOVEMBER DECEMBER
```

```
   static void demoMonth() {
      out.println(Month.of(1) == Month.JANUARY);
      out.println(Month.of(12) == Month.DECEMBER);

      out.println(Month.JANUARY.getValue() == 1);
      out.println(Month.DECEMBER.getValue() == 12);

      out.println(Month.DECEMBER.minus(11) == Month.JANUARY);
      out.println(Month.JANUARY.plus(11) == Month.DECEMBER);

      out.println(Month.JANUARY.compareTo(Month.MARCH) == -2);
      out.println(Month.MARCH.compareTo(Month.JANUARY) == 2);

      out.println(Month.FEBRUARY.firstMonthOfQuarter() ==
Month.JANUARY);
      out.println(Month.DECEMBER.firstMonthOfQuarter() ==
Month.OCTOBER);

      out.println(Month.MARCH.firstDayOfYear(true) == 61);
      out.println(Month.MARCH.firstDayOfYear(false) == 60);

      out.println(Month.FEBRUARY.length(true) == 29);
      out.println(Month.FEBRUARY.length(false) == 28);
   }
```

Auch hier geben alle Zeilen `true` aus.

```
   static void demoMonthMinMax() {
      for (Month m : Month.values()) {
         out.print(m.minLength() + " ");
      }
      out.println();
      for (Month m : Month.values()) {
         out.print(m.maxLength() + " ");
```

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

```
31 28 31 30 31 30 31 31 30 31 30 31
31 29 31 30 31 30 31 31 30 31 30 31
```

9.5 LocalDate, LocalTime und LocalDateTime

Die Demo-Methoden nutzen folgende Klassen:

```
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
import java.time.ZoneOffset;
```

`LocalDate`-Objekte repräsentieren ein Datum; `LocalTime`-Objekte eine Uhrzeit; und `LocalDateTime`-Objekte repräsentieren sowohl ein Datum als auch eine Zeit.

Alle Objekte sind immutable und nur mittels Factory-Methoden erzeugt werden.

LocalDate

```
    static void demoLocalDate() {
        LocalDate d1 = LocalDate.of(2015, 1, 20);
        LocalDate d2 = LocalDate.of(2015, Month.JANUARY, 20);
        LocalDate d3 = LocalDate.now();
        out.println(d3);
        out.println(d1.equals(d2));
        out.println(d1);
        out.println(d1.getDayOfMonth());
        out.println(d1.getDayOfWeek());
        out.println(d1.getDayOfYear());
        out.println(d1.getMonth());
        out.println(d1.getYear());
        LocalDate d4 = LocalDate.parse("2015-01-20");
        out.println(d4);
    }
```

```
2015-02-26
true
2015-01-20
20
TUESDAY
20
JANUARY
2015
2015-01-20
```

LocalTime

```
    static void demoLocalTime() {
        LocalTime t1 = LocalTime.of(12, 30);
        LocalTime t2 = LocalTime.of(12, 30, 5);
        LocalTime t3 = LocalTime.of(12, 30, 5, 500 * 1000_000);
        LocalTime t4 = LocalTime.now();
        out.println(t1);
        out.println(t2);
        out.println(t3);
        out.println(t4);
        out.println(t3.getHour());
        out.println(t3.getMinute());
        out.println(t3.getSecond());
        out.println(t3.getNano());
        LocalTime t5 = LocalTime.parse("12:30:05");
        out.println(t5);
        LocalTime t6 = LocalTime.parse("12:30");
        out.println(t6);
    }
```

```
12:30
12:30:05
12:30:05.500
12:48:26.812
12
30
5
500000000
12:30:05
12:30
```

LocalDateTime

```
    static void demoLocalDateTime() {
        LocalDate d = LocalDate.of(2015, 1, 20);
        LocalTime t = LocalTime.of(12, 30, 5);
        LocalDateTime dt = LocalDateTime.of(d, t);
        out.println(dt);
        out.println(dt.getYear());
        // ...
        out.println(dt.getSecond());
        out.println(dt.getNano());
        LocalDate d1 = dt.toLocalDate();
        out.println(d == d1);
```

```
        LocalTime t1 = dt.toLocalTime();
        out.println(t == t1);
        out.println(LocalDateTime.parse("2015-01-20T12:30:05"));
    }
```

```
2015-01-20T12:30:05
2015
5
0
true
true
2015-01-20T12:30:05
```

LocalDateTime und Instant

```
    static void demoLocalDateTimeToInstant() {
        Instant instant =
LocalDateTime.now().toInstant(ZoneOffset.UTC);
        out.println(instant);
    }
```

```
2015-02-26T12:48:26.812Z
```

9.6 ZonedDateTime

Die Demo-Methoden benutzen folgende Klassen:

```
import java.time.Clock;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
```

`ZonedDateTime` hat eine zugeordnete Zeitzone.

Zunächst zwei weitere Klassen: `ZoneId` und `Clock`.

ZoneId

Zeitzonen werden durch `Strings` resp. durch `ZoneId`-Objekte repräsentiert. Die Klasse `ZoneId` hat u.a. statische Helper, mittels derer sich diese Strings ermitteln lassen.

```
    static void demoZoneIds() {
        for (String s : ZoneId.getAvailableZoneIds()) {
            out.print(s + " ");
        }
        ZoneId zid1 = ZoneId.systemDefault();
        out.println(zid1);
        ZoneId zid2 = ZoneId.of("Europe/Berlin");
        out.println(zid1.equals(zid2));
    }
```

```
Asia/Aden America/Cuiaba Etc/GMT+9 Etc/GMT+8 Africa/Nairobi ...
true
```

Clock

```
    static void demoClock() {
        Clock c1 = Clock.systemUTC();
        Clock c2 = Clock.systemDefaultZone();
        Clock c3 = Clock.system(ZoneId.systemDefault());
        out.println(c1);
        out.println(c1.millis());
        out.println(c2);
        out.println(c2.millis());
        out.println(c3);
        out.println(c3.millis());
```

```
        out.println(c1.equals(c2));
        out.println(c2.equals(c3));
        ZoneId zid = c2.getZone();
        System.out.println(zid);
    }
```

```
SystemClock[Z]
1421835279923
SystemClock[Europe/Berlin]
1421835279923
SystemClock[Europe/Berlin]
1421835279923
false
true
Europe/Berlin
```

ZonedDateTime

```
    static void demoZonedDateTime() {
        ZonedDateTime dt1 = ZonedDateTime.now();
        out.println(dt1);
        ZonedDateTime dt2 =
ZonedDateTime.now(ZoneId.systemDefault());
        out.println(dt1.equals(dt2));
        LocalDateTime ldt = dt1.toLocalDateTime();
        out.println(ldt);
        out.println(ZonedDateTime.parse(
            "2015-01-21T11:01:00.314+01:00[Europe/Berlin]"));
    }
```

```
2015-01-21T11:02:29.390+01:00[Europe/Berlin]
true
2015-01-21T11:02:29.390
2015-01-21T11:01:00.314+01:00[Europe/Berlin]
```

9.7 YearMonth, MonthDay und Year

Die Demo-Methoden nutzen folgende Klassen:

```
import java.time.Month;
import java.time.MonthDay;
import java.time.Year;
import java.time.YearMonth;
```

"Im Januar 2015 mache ich Urlaub."
"Mein Geburtstag ist am 28. November".
"1974 habe ich Abitur gemacht".

Solche Angelegenheiten sollte man nicht mit `LocalDate`-Objekten erschlagen...

YearMonth

```
   static void demoYearMonth() {
      YearMonth ym1 = YearMonth.of(2015, 1);
      YearMonth ym2 = YearMonth.of(2015, Month.JANUARY);
      YearMonth ym3 = YearMonth.now();
      out.println(ym1);
      out.println(ym1.equals(ym2));
      out.println(ym1 == ym2);
      out.println(ym3.getYear());
      out.println(ym3.getMonth());
      out.println(ym3.lengthOfMonth());
      out.println(ym3.lengthOfYear());
      out.println(ym3.isAfter(ym1));
      out.println(ym3.isBefore(ym1));
   }
```

```
2015-01
true
false
2015
FEBRUARY
28
365
true
false
```

MonthDay

```
   static void demoMonthDay() {
      MonthDay md1 = MonthDay.of(1, 20);
      MonthDay md2 = MonthDay.of(Month.JANUARY, 20);
      MonthDay md3 = MonthDay.now();
      out.println(md1);
      out.println(md1.equals(md2));
      out.println(md1 == md2);
      out.println(md3.getMonth());
      out.println(md3.getDayOfMonth());
      // ...
   }
```

```
01-20
true
false
FEBRUARY
26
```

Year

```
   static void demoYear() {
      Year y1 = Year.of(2014);
      Year y2 = Year.of(2014);
      Year y3 = Year.now();
      out.println(y1);
      out.println(y1.equals(y2));
      out.println(y1 == y2);
      out.println(y3.getValue());
      out.println(y3.isLeap());
      out.println(y3.length());
      // ...
   }
```

```
2014
true
false
2015
false
365
```

9.8 Period

Ein `Period`-Objekt (`java.time.Period`) besteht aus Jahr, Monat und Tag (ist also "verständlicher" als ein `Duration`-Objekt).

```
    static void demoCreation() {
        Period p1 = Period.ofYears(1);
        Period p2 = Period.ofYears(1).withMonths(6);
        Period p3 = Period.ofYears(1).withMonths(6).withDays(15);
        out.println(p1);
        out.println(p2);
        out.println(p3);
    }
```

```
P1Y
P1Y6M
P1Y6M15D
```

```
    static void demoGetMethods() {
        Period p = Period.ofYears(1).withMonths(6).withDays(15);
        out.println(p.getYears());
        out.println(p.getMonths());
        out.println(p.getDays());
    }
```

```
1
6
15
```

```
    static void demoPlusMinus() {
        Period p = Period.ofYears(1).withMonths(6).withDays(15);
        out.println(p);
        p = p.plusMonths(10);
        p = p.plusDays(10);
        p = p.minusDays(3);
        out.println(p);
    }
```

```
P1Y6M15D
P1Y16M22D
```

```
    static void demoParse() {
        Period p = Period.parse("P1Y6M15D");
        out.println(p);
```

```
    }
```

```
P1Y6M15D
```

9.9 Formatter

Die Demo-Methoden benutzen folgende Klassen:

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.format.FormatStyle;
import java.time.temporal.ChronoField;
import java.time.temporal.TemporalAccessor;
```

Wie können Datums und Zeiten formatiert werden? Und wie kann das Ergebnis einer solchen Formatierung wieder geparst werden?

DateTimeFormatter.ofLocalizedDate

`demoDate` zeigt, wie ein Datum formatiert und geparst werden kann. Beim Aufruf muss ein `FormatStyle` übergeben werden:

```
    static void demoDate(FormatStyle style) {

        DateTimeFormatter f =
DateTimeFormatter.ofLocalizedDate(style);
        String s = f.format(LocalDateTime.now());
        System.out.println(s);

        TemporalAccessor ta = f.parse(s);
        int day = ta.get(ChronoField.DAY_OF_MONTH);
        int month = ta.get(ChronoField.MONTH_OF_YEAR);
        int year = ta.get(ChronoField.YEAR);
        System.out.println(day + " " + month + " " + year);
    }
```

Hier zwei Aufrufe dieser Methode und deren Resultate:

```
        demoDate(FormatStyle.SHORT);
```

```
26.02.15
26 2 2015
```

```
        demoDate(FormatStyle.LONG);
```

```
26. Februar 2015
26 2 2015
```

DateTimeFormatter.ofLocalizedTime

`demoTime` zeigt, wie eine Uhrzeit formatiert und geparst werden kann. Beim Aufruf muss wiederum ein `FormatStyle` übergeben werden:

```
    static void demoTime(FormatStyle style) {

        DateTimeFormatter f =
DateTimeFormatter.ofLocalizedTime(style);
        String s = f.format(LocalDateTime.now());
        System.out.println(s);

        TemporalAccessor ta = f.parse(s);
        int hour = ta.get(ChronoField.HOUR_OF_DAY);
        int minute = ta.get(ChronoField.MINUTE_OF_HOUR);
        int second = ta.get(ChronoField.SECOND_OF_MINUTE);
        System.out.println(hour + " " + minute + " " + second);
    }
```

Zwei Aufrufe dieser Methode und deren Resultate:

```
        demoTime(FormatStyle.SHORT);
```

```
11:59
11 59 0
```

```
        demoTime(FormatStyle.MEDIUM);
```

```
11:59:02
11 59 2
```

DateTimeFormatter.ofLocalizedDateTime

Die folgende Methode zeigt schließlich, wie die Kombination Datum / Uhrzeit formatiert und geparst werden kann. Auch hier muss ein `FormatStyle` übergeben werden:

```
    static void demoDateTime(FormatStyle style) {

        DateTimeFormatter f =
DateTimeFormatter.ofLocalizedDateTime(style);
        String s = f.format(LocalDateTime.now());
        System.out.println(s);

```

```
        TemporalAccessor ta = f.parse(s);
        int day = ta.get(ChronoField.DAY_OF_MONTH);
        int month = ta.get(ChronoField.MONTH_OF_YEAR);
        int year = ta.get(ChronoField.YEAR);
        int hour = ta.get(ChronoField.HOUR_OF_DAY);
        int minute = ta.get(ChronoField.MINUTE_OF_HOUR);
        int second = ta.get(ChronoField.SECOND_OF_MINUTE);
        System.out.println(day + " " + month + " " + year + " " +
            hour + " " + minute + " " + second);
    }
```

Zwei Aufrufe dieser Methode und deren Resultate:

```
        demoDateTime(FormatStyle.SHORT);
```

```
26.02.15 11:59
26 2 2015 11 59 0
```

```
        demoDateTime(FormatStyle.MEDIUM);
```

```
26.02.2015 11:59:02
26 2 2015 11 59 2
```

9.10 Interoperablilität mit Date und Calendar

Wie verhalten sich die "alten" Klassen zu den "neuen" Klassen?

Date / Instant

```
    static void demoDateToInstant() {
        Date date1 = new Date();
        Instant instant = date1.toInstant();
        out.println(instant);
        instant = instant.plus(Duration.of(1,
ChronoUnit.SECONDS));
        Date date2 = Date.from(instant);
        out.println(date2.getTime() - date1.getTime());
    }
```

```
2015-02-26T11:16:32.408Z
1000
```

Calendar / ZonedDateTime

```
    static void demoCalendarToZonedDateTime() {
        GregorianCalendar calendar1 =
            (GregorianCalendar) GregorianCalendar.getInstance();
        ZonedDateTime zdt = calendar1.toZonedDateTime();
        out.println(zdt);
        zdt = zdt.plus(Duration.of(1, ChronoUnit.SECONDS));
        GregorianCalendar calendar2 =
GregorianCalendar.from(zdt);
        out.println(
            calendar2.getTime().getTime() -
calendar1.getTime().getTime());
    }
```

```
2015-02-26T12:16:32.486+01:00[Europe/Berlin]
1000
```

Calendar / Instant

```
    static void demoCalendarToInstant() {
        Calendar calendar = GregorianCalendar.getInstance();
        Instant instant = calendar.toInstant();
        out.println(instant);
    }
```

```
2015-02-26T11:16:32.517Z
```

9.11 Aufgaben

Die `PropertyEditor`en von `java.beans` können z.B. wie folgt angewendet werden:

```
package ex1;

import java.beans.PropertyEditor;
import java.beans.PropertyEditorManager;

public class Application {

    public static void main(String[] args) {
        demo1();
        demo2();
    }
    static void demo1() {
        PropertyEditor e =
PropertyEditorManager.findEditor(int.class);
        e.setValue(42);
        String s = e.getAsText();
        System.out.println(s);
    }
    static void demo2() {
        PropertyEditor e =
PropertyEditorManager.findEditor(int.class);
        e.setAsText("42");
        Integer v = (Integer)e.getValue();
        System.out.println(v);
    }
}
```

Registrieren Sie beim `PropertyEditorManager` auch einen PropertyEditor für `DateTime`! Benutzen Sie in Ihrer Implementierung die Klasse `DateTimeFormatter`.

10 Multithreading

Dieses Kapitel stellt zwei neue Klassen des `java.util.concurrent`-Pakets vor: `CompletableFuture` und `StampedLock`.

`CompletableFutures` erlauben die Beschreibung eines Netzes von Einzelschritten, die erst in Zukunft im Kontext zusätzlicher Threads (und möglicherweise parallel) ausgeführt werden. Jeder dieser Einzelschritte (jeder Netzknoten) wird ausgestattet mit einer Implementierung eines funktionalen Interfaces (`Supplier`, `Function`, `Consumer` etc.). Wir können also eine komplexe Funktionalität zurechtbasteln, ohne diese sofort auszuführen. Irgendwann können wir die Ausführung dieser Funktionalität starten und auf das Ergebnis der Berechnung warten.

`CompletableFuture`s haben also wenig mit den "gewöhnlichen" `Future`s gemein – außer dass beide Klassen eine `get`-Methode besitzen, mittels derer wir auf das Ergebnis einer abgespaltenen Berechnung warten können.

`StampedLock` stellt einen neuen performanten Lock-Mechanismus zur Verfügung, der allerdings nur mit äußerster Vorsicht genutzt werden sollte.

10.1 CompletableFuture - Beispiel

Einen kleinen Vorgeschmack auf `CompletableFuture` lieferte bereits der Multithreading-Abschnitt des Functional-Interfaces-Kapitels – dort wurden mittels einer Klasse `Node` Schritte einer Berechnung spezifiziert, wobei einiger dieser Schritte parallel ablaufen konnten.

Hier zunächst einige kleine Beispiele, welche die Benutzung von `CompletableFuture` demonstrieren sollen. Wir benutzen wieder die bereits bekannte Klasse `Work` und den `GuiViewer`.

Die Klasse `Application` nutzt u.a. die folgenden Importe:

```
import static util.Work.call;
import static util.Work.fwork;
import static util.Work.swork;
import static util.Work.useViewer;

import java.util.concurrent.CompletableFuture;

import util.Work;
import util.Work.Type;
```

Alle der folgenden Methoden werden von der `main`-Methode der Klasse `Application` aufgerufen.

Die erste Methode berechnet aufgrund eines `value`-Parameters folgenden Wert:

```
(value + 1) * (value - 1)
```

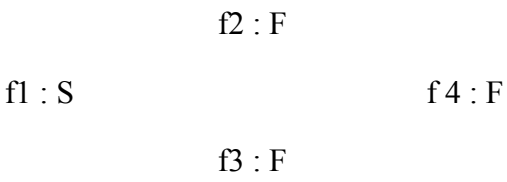
Hier die Implementierung:

```
   static int productOfSumAndDiff1Simple(int value) throws
Exception {

     final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
          () -> value);

     final CompletableFuture<Integer> f2 = f1.thenApplyAsync(
          x -> x + 1);

     final CompletableFuture<Integer> f3 = f1.thenApplyAsync(
          x -> x - 1);
```

```
    final CompletableFuture<Integer> f4 = f2.thenCombine(f3,
         (v1, v2) -> v1 * v2);

    return f4.get();
  }
```

Die Methode erzeugt einen Grafen von vier `CompletableFuture`-Objekten. Das erste Objekt wird mittels einer statischen Methode von `CompletableFuture` erzeugt: `supplyAsync`. Die folgenden drei `CompletableFutures` werden mittels Instanzmethoden der Klasse erzeugt: mit den Methoden `thenApplyAsync` und `thenCombine`.

Hier zunächst ein Diagramm (das bereits vertraut erscheinen sollte):



An die `supplyAsync`–Methode wird ein `Supplier<T>` übergeben; und sie liefert ein `CompletableFuture<T>` zurück. Man beachte, dass der übergebene `Supplier` hier allerdings noch nicht ausgeführt wird - dass seine `get`-Methode also noch nicht aufgerufen wird!

Auch `thenApplyAsync` liefert ein neues `CompletableFuture`-Objekt zurück. An die Methode wird eine `Function` übergeben. Auch hier wird die `Function` noch nicht ausgeführt! Im obigen Beispiel werden mittels `thenApplyAsync` zwei `CompletableFuture`s erzeugt.

`thenCombine` liefert das letzte `CompletableFuture`-Objekt zurück. An `thenCombine` wird als Parameter ein weiterers `CompletableFuture` übergeben – und eine `BiFunction`.

Hier die Spezifikation der drei Methoden:

```
    public static <U> CompletableFuture<U> supplyAsync(
        Supplier<U> supplier)
```

```
    public <U> CompletableFuture<U> thenApplyAsync(
        Function<? super T,? extends U> function)
```

```
    public <U,V> CompletableFuture<V> thenCombine(
```

```
        CompletionStage<? extends U> other,
        BiFunction<? super T,? super U,? extends V> function)
```

Man beachte die Typ-Parametrisierung!

Auf das zuletzt produzierte `CompletableFuture` wird dann die `get`-Methode aufgerufen. Diese setzt die Berechnung in Bewegung – und blockiert solange, bis das Ergebnis der Berechnung vorliegt:

Zunächst wird die `get`-Methode des an das erste `CompletableFuture`-Objekt übergebenen `Supplier`s ausgeführt – in einem neuen Thread. Dann werden in zwei seperaten Threads parallel die `apply`-Methoden der an die beiden "mittleren" `CompletableFuture`s übergebenen `Function`s ausgeführt – ihnen wird jeweils das vom `Supplier` bereitgestellte Ergebnis als Argument übergeben; wenn beide dieser `Function`s zurückgekehrt sind, wird schließlich die `BiFunction` ausgeführt, welche an das letzte `CompletableFuture`-Objekt übergeben wurde – welcher als Argumente die Ergebnisse der beiden zuvor aufgerufen `Function`s übergeben werden.

Das Ergebnis dieser `BiFunction` wird schließlich von `get` zurückgeliefert.

Wir erweitern die oben vorgestellte Methode um die Simulation harter Arbeit (die dann mittels des `GuiViewer`s visualisiert werden kann):

```
   static int productOfSumAndDiff1(int value) throws Exception {

      final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
          () -> swork("supply " + value, N).thenReturn(value));

      final CompletableFuture<Integer> f2 = f1.thenApplyAsync(
          x -> fwork(x + " + 1", N).thenReturn(x + 1));

      final CompletableFuture<Integer> f3 = f1.thenApplyAsync(
          x -> fwork(x + " - 1", N).thenReturn(x - 1));

      final CompletableFuture<Integer> f4 = f2.thenCombine(f3,
          (v1, v2) -> fwork(v1 + " * " + v2, N).thenReturn(v1 *
v2));

      return get("f4.get", f4);
   }
```

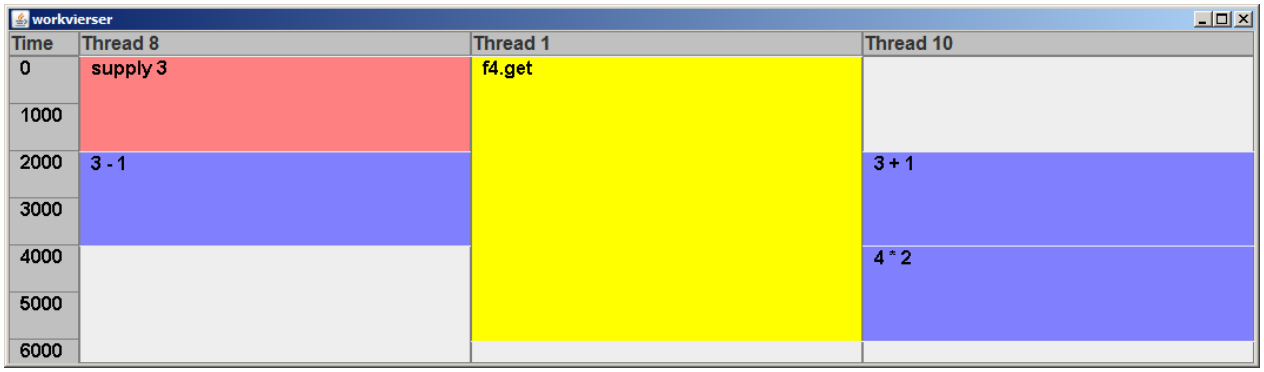
`N` steht z.B. für `2000` Millisekunden.

Die in der letzten Zeile aufgerufene `get`-Methode sieht wie folgt aus:

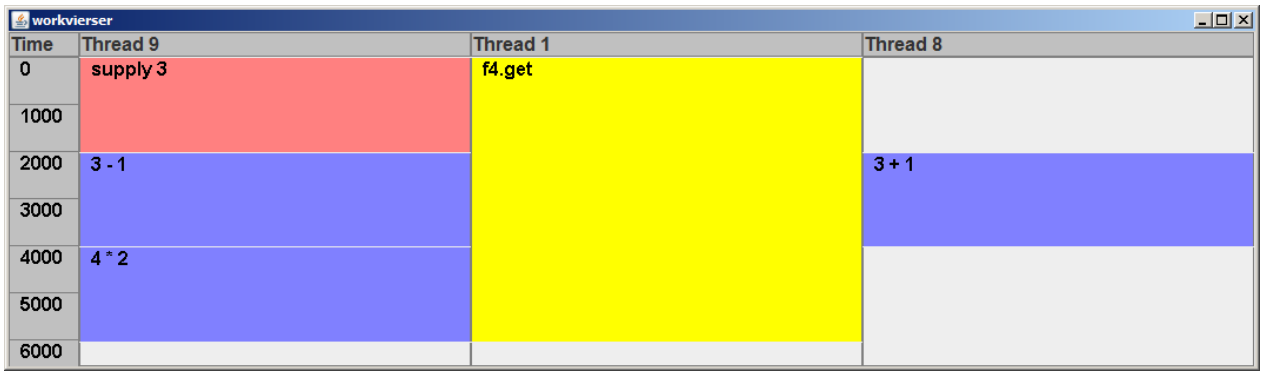
```
   static <T> T get(String text, CompletableFuture<T> f) throws
Exception {
      Work.getViewer().beginWork(text, Type.WAITER);
      final long start = System.nanoTime();
      tlog("-> " + text);
      final T result = f.get();
      final long end = System.nanoTime();
      tlog("<- " + text + " --> " + result +
           " (" + ((end - start) / 1_000_000) + ")");
      Work.getViewer().endWork();
      return result;
   }
```

Dass `CompletableFuture.get` solange blockiert, bis die Berechnung erfolgt ist (und dann den Wert dieser Berechnung zurückliefert), wird nun auch im `GuiViewer` durch ein gelbes Band visualisiert werden.

Hier ein möglicher Ablauf einer Berechnung:



Natürlich könnte die Ausführung exakt derselben Methode auch in einem anderen Ablauf resultieren:



Immer aber werden die Summen- und die Differenz-Berechnung parallel ausgeführt.

Anstatt bei der Erzeugung des ersten `CompletableFutures` einen `Supplier` anzugeben, kann der "Startwert" auch auf eine andere Weise vorgegeben werden:

```
   static int productOfSumAndDiff2Simple(int value) throws
Exception {

      final CompletableFuture<Integer> f1 = new
CompletableFuture<>();

      final CompletableFuture<Integer> f2 = f1.thenApplyAsync(
            x -> x + 1);

      final CompletableFuture<Integer> f3 = f1.thenApplyAsync(
            x -> x - 1);

      final CompletableFuture<Integer> f4 = f2.thenCombine(f3,
            (v1, v2) -> v1 * v2);
      f1.complete(value);

      return f4.get();
   }
```

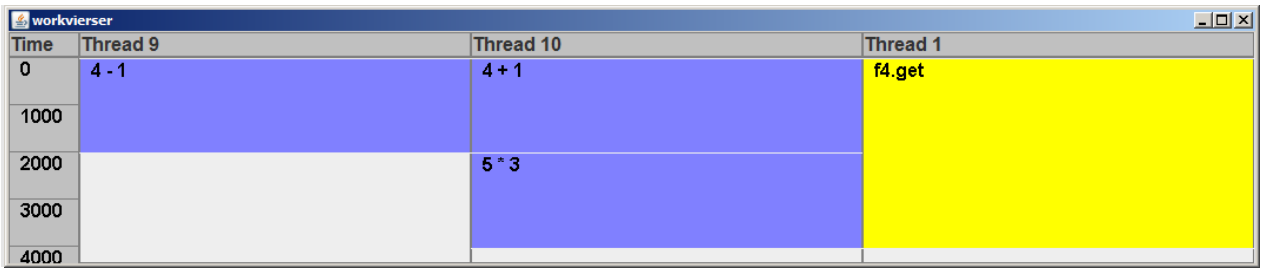
Das erste `CompletableFuture` wird einfach über den parameterlosen Konstruktor erzeugt. Aber es kann natürlich erst "ausgeführt" werden, wenn ein Startwert vorliegt. Ein solcher Startwert wird mittels der `complete`-Methode vorgegeben, die auf das erste `CompletableFuture` aufgerufen wird und welcher der Startwert übergeben wird.

Wir bereichern die obige Methode um die Möglichkeit, den Ablauf im Viewer zu verfolgen:

```
    static int productOfSumAndDiff2(int value) throws Exception {

        final CompletableFuture<Integer> f1 = new
CompletableFuture<>();

        final CompletableFuture<Integer> f2 = f1.thenApplyAsync(
            x -> fwork(x + " + 1", N).thenReturn(x + 1));

        final CompletableFuture<Integer> f3 = f1.thenApplyAsync(
            x -> fwork(x + " - 1", N).thenReturn(x - 1));

        final CompletableFuture<Integer> f4 = f2.thenCombine(f3,
           (v1, v2) -> fwork(v1 + " * " + v2, N).thenReturn(v1 *
v2));
```

```
        xrun("f1.complete(" + value + ")", () ->
f1.complete(value));

        return get("f4.get", f4);
    }
```

Ein möglicher Ablauf:



Hier gibt's nun natürlich keinen asynchronen `Supplier`-Aufruf.

Die bislang durchgeführte Berechnung hatte nur einen einzigen "Parameter". Wie kann eine Berechnung ablaufen, die mehrere Parameter verlangt?

Pythagoras berechnet `c = Math.sqrt(a * a + b * b)`. Die Berechnung hat zwei Parameter: `a` und `b`. Sie kann auf verschiedene Weise ausgeführt werden.

Hier die erste Variante:

```
    static double pythagoras1Simple(double a, double b) throws
Exception {
        final CompletableFuture<Double> f1 =
CompletableFuture.supplyAsync(
            () -> a);
        final CompletableFuture<Double> f2 =
CompletableFuture.supplyAsync(
            () -> b);
        final CompletableFuture<Double> f3 = f1.thenApplyAsync(
            x -> x * x);
        final CompletableFuture<Double> f4 = f2.thenApplyAsync(
            x -> x * x);
        final CompletableFuture<Double> f5 = f3.thenCombine(f4,
            (v1, v2) -> v1 + v2);
        final CompletableFuture<Double> f6 = f5.thenApply(
            x -> Math.sqrt(x));
        return f6.get();
    }
```

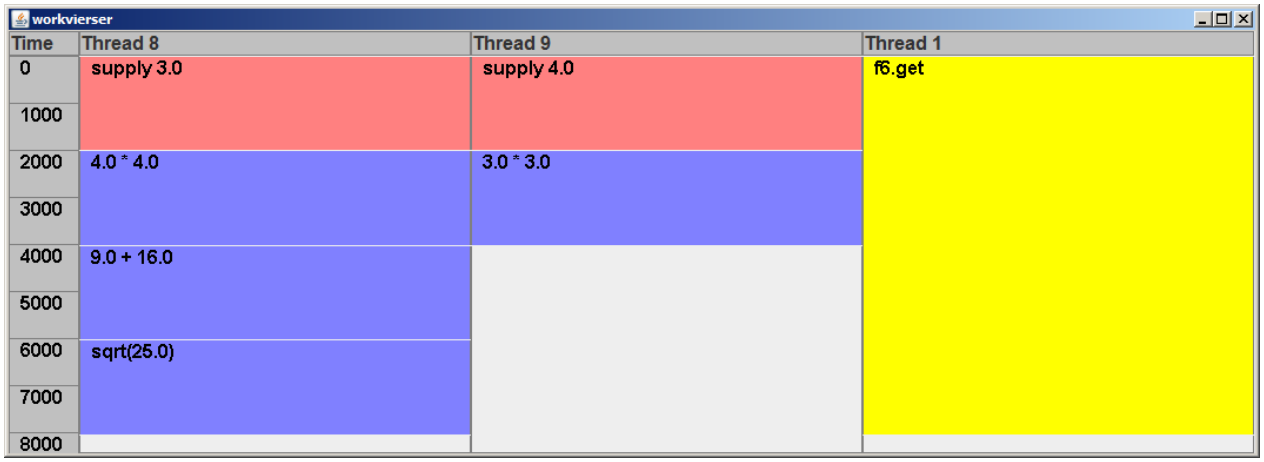
Hier das Ablaufdiagramm:

f1 : S f3 : F

f5 : F f5 : F

f2 : S f4 : F

Der Ablauf könnte sich im Viewer wie folgt präsentieren (sofern die obige Methode um entsprechende `work`-Aufrufe erweitert würde):



Eine zweite Variante (eine ohne `Supplier`):

```
    static double pythagoras2Simple(double a, double b) throws
Exception {
        final CompletableFuture<Double> f1 = new
CompletableFuture<>();
        final CompletableFuture<Double> f2 = new
CompletableFuture<>();
        final CompletableFuture<Double> f3 = f1.thenApplyAsync(
            x -> x * x);
        final CompletableFuture<Double> f4 = f2.thenApplyAsync(
            x -> x * x);
        final CompletableFuture<Double> f5 = f3.thenCombine(f4,
            (v1, v2) -> v1 + v2);
        final CompletableFuture<Double> f6 = f5.thenApply(
            x -> Math.sqrt(x));
        f1.complete(a);
        f2.complete(b);
        return f6.get();
    }
```

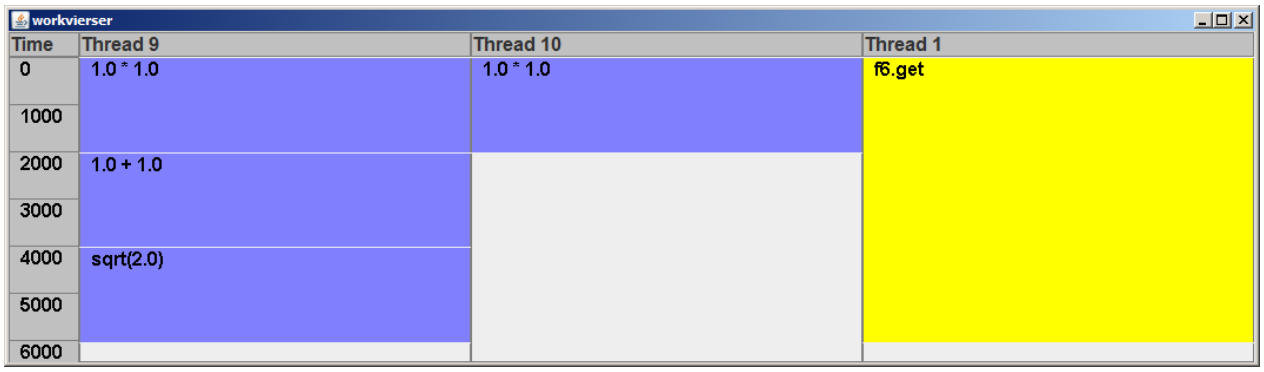
`f1` und `f2` müssen nun via `complete` "von außen" mit Werten versorgt werden:

f1 f3 : F

f5 : F f5 : F

f2 f4 : F

Ein möglicher Ablauf:



Die dritte Variante nutzt eine Helper-Klasse `Pair`:

```
    static class Pair<X, Y> {
        public final X x;
        public final Y y;
        public Pair(X x, Y y) {
            this.x = x;
            this.y = y;
        }
        @Override
        public String toString() {
            return "Pair [x=" + x + ", y=" + y + "]";
        }
    }
```

Die zwei Input-Parameter können nun zu einem einzigen zusammengefaßt werden:

```
    static double pythagoras3Simple(double a, double b) throws
Exception {

        final CompletableFuture<Pair<Double, Double>> f1 =
            new CompletableFuture<>();
        final CompletableFuture<Double> f2 = f1.thenApplyAsync(
            p -> p.x * p.x);
        final CompletableFuture<Double> f3 = f1.thenApplyAsync(
```

```
            p -> p.y * p.y);
        final CompletableFuture<Double> f4 = f2.thenCombine(f3,
            (v1, v2) -> v1 + v2);
        final CompletableFuture<Double> f5 = f4.thenApply(
            x -> Math.sqrt(x));

        Pair<Double, Double> pair = new Pair<>(a, b);
        f1.complete(pair);

        return f5.get();
    }
```

Natürlich könnte man hier statt mit `complete` auch wieder mit einem (einzigen!) `Supplier` arbeiten.

Ein letztes Beispiel soll zeigen, dass ein Netz von Einzelschritten spezifiziert werden kann, welches erst später dann ausgeführt wird.

Ein `Context` referenziert zwei `CompletableFuture`s – und erlaubt ein `complete` auf das erste `CompletableFuture` und ein `get` auf das zweite:

```
    static class Context<S,E> {
        public final CompletableFuture<S> start;
        public final CompletableFuture<E> end;
        public Context(CompletableFuture<S> start,
CompletableFuture<E> end) {
            this.start = start;
            this.end = end;
        }
        public Context<S,E> complete(S value) {
            this.start.complete(value);
            return this;
        }
        public E get() throws Exception {
            return this.end.get();
        }
    }
```

Die Methode `buildContext` erzeugt ein Netz – und liefert den Startknoten und den Endknoten dieses Netzes in Form eines `Context`-Objekts zurück (ohne aber bereits die Ausführung des Netzes zu starten):

```
    static Context<Integer, Integer> buildContext() {
        final CompletableFuture<Integer> f1 = new
CompletableFuture<>();
        final CompletableFuture<Integer> f2 = f1.thenApplyAsync(
```

```
            x -> x + 1);
        final CompletableFuture<Integer> f3 = f1.thenApplyAsync(
            x -> x - 1);
        final CompletableFuture<Integer> f4 = f2.thenCombine(f3,
            (x1, x2) -> x1 * x2);
        return new Context<>(f1, f4);
    }
```

Die folgende Methode führt drei Berechnungen aus (man beachte, dass das Netz stets neu erzeugt wird!):

```
    static void demoBuildAndCalculate() throws Exception {
        System.out.println(buildContext().complete(3).get());
        System.out.println(buildContext().complete(4).get());
        System.out.println(buildContext().complete(5).get());
    }
```

Die Ausgaben:

```
8
15
24
```

10.2 CompletableFuture - Details

Dieser Abschnitt stellt einige weitere Details der Klasse `CompletableFuture` vor.

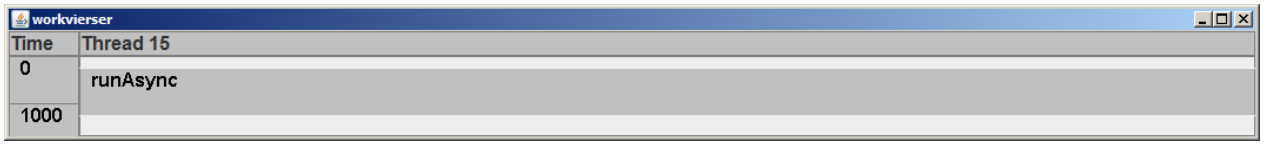
Wir benutzen zur Darstellung der Abläufe wieder den Viewer und die folgende, aus dem letzten Abschnitt bereits bekannte Helper-Methode (die auf das Ergebnis wartet und dieses Warten im Viewer anzeigt):

```
    static <T> T get(String text, CompletableFuture<T> f) throws
Exception {
        Work.getViewer().beginWork(text, Type.WAITER);
        final long start = System.nanoTime();
        tlog("-> " + text);
        final T result = f.get();
        final long end = System.nanoTime();
        tlog("<- " + text + " --> " +
            result + " (" + ((end - start) / 1_000_000) + ")");
        Work.getViewer().endWork();
        return result;
    }
```

Zunächst werden einige statische Methoden von `CompletableFuture` vorgestellt, welche allesamt als Factory-Methoden für `CompletableFuture`s fungieren.

runAsync

```
    static void demoRunAsync() throws Exception {
        final CompletableFuture<Void> f =
CompletableFuture.runAsync(
                () -> rwork("runAsync", 1000));
        tlog("f.isDone = " + f.isDone());
        final Void result = f.get();
        tlog("after f.get: " + result);
        tlog("f.isDone = " + f.isDone());
    }
```



An `runAync` wir ein `Runnable` übergeben. (Der Status eines `CompletableFutures` kann mittels der Instanzmethode `isDone` ermittelt werden – beim ersten der beiden obigen Aufrufe wird `false`, beim Aufruf wird `true` ausgegeben.)

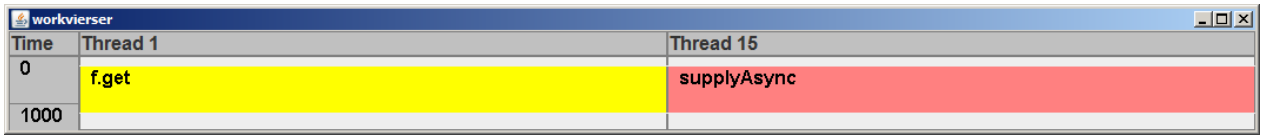
An eine überladene Variante von `runAsync` kann auch zusätzlich ein `Executor` übergeben werden:

```
    static void demoRunAsyncExecutor() throws Exception {
        final ExecutorService executor =
Executors.newFixedThreadPool(2);
        final CompletableFuture<Void> f =
CompletableFuture.runAsync(
                () -> rwork("runAsync", 1000), executor);
        final Void result = get("f.get", f);;
        executor.shutdown();
    }
```

Falls kein expliziter `Executor` übergeben wird, wird der `ForkJoinPool.commonPool` verwendet.

supplyAsync

```
    static void demoSupplyAsync() throws Exception {
        final CompletableFuture<Integer> f =
CompletableFuture.supplyAsync(
                () -> swork("supplyAsync", 1000).thenReturn(42));
        final Integer result = get("f.get", f);;
    }
```



Die `supplyAsync`-Methode wurde bereits im letzten Abschnitt verwendet.

Auch `supplyAsync` kann mit einem expliziten `Executor` aufgerufen werden:

```
    static void demoSupplyAsyncExecutor() throws Exception {
        final ExecutorService executor =
Executors.newFixedThreadPool(2);
        final CompletableFuture<Integer> f =
CompletableFuture.supplyAsync(
                () -> swork("supplyAsync", 1000).thenReturn(42),
executor);
```

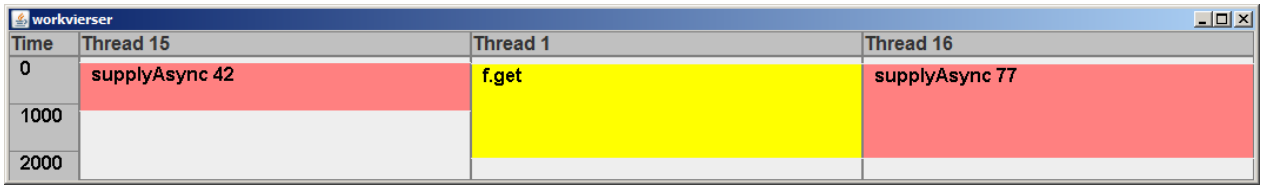
```
        final Integer result = get("f.get", f);;
        executor.shutdown();
    }
```

allOf

An `allOf` können beliebig viele `CompletableFutures` übergeben werden. `allOf` wartet, bis alle diese Objekte ihre Arbeit getan haben:

```
    static void demoAllOf() throws Exception {
        final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
                () -> swork("supplyAsync 42",
1000).thenReturn(42));
        final CompletableFuture<Integer> f2 =
CompletableFuture.supplyAsync(
                () -> swork("supplyAsync 77",
2000).thenReturn(77));
        CompletableFuture<Void> f = CompletableFuture.allOf(f1,
f2);
        final Void result = get("f.get", f);;
    }
```

`allOf` liefert ein `CompletableFuture<Void>` (auf welches dann im obigen Beispiel die `get`-Methode aufgerufen wird – welche dann ihrerseits natürlich auch ein `Void`-Resulat liefert (und damit den einzig möglichen Wert `null`). Der Grund ist klar: beliebig viele Resultate (möglicherweise unterschiedlichen Typs) können nicht auf einen vernünftigen Resultat-Typ abgebildet werden (oder?...)



anyOf

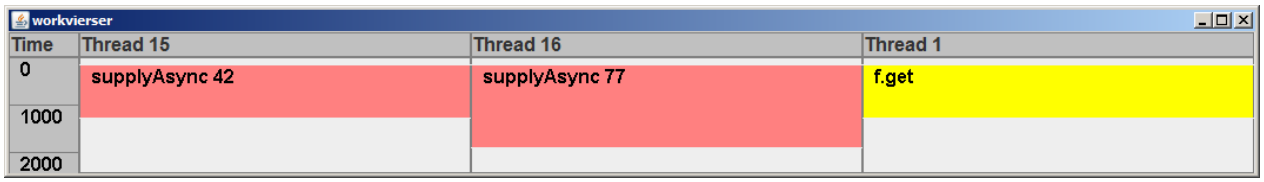
Auch an `anyOf` können beliebig viele `CompletableFuture`s übergeben werden. Sie kehrt zurück, wenn das erste dieser `CompletableFuture`s seine Arbeit erledigt hat. `anyOf` liefert im Unterschied zu `allOf` ein `CompletableFuture<Object>` zurück – da nur ein Resultat weiterverarbeitet wird, kann die `get`-Methode `Object` liefern (statt `Void`).

```
    static void demoAnyOf() throws Exception {
        final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
                () -> swork("supplyAsync 42", 1000,
2000).thenReturn(42));
        final CompletableFuture<Integer> f2 =
CompletableFuture.supplyAsync(
                () -> swork("supplyAsync 77", 1000,
2000).thenReturn(77));
        CompletableFuture<Object> f = CompletableFuture.anyOf(f1,
f2);
        final Object result = get("f.get", f);
        f1.get();
        f2.get();
    }
```

Der oben aufgerufen `swork`-Methode werden zwei Millisekunden-Werte übergeben: min und max. Es wird dann eine Zufallszahl zwischen min und max berechnet.

Mittels `f1.get()` und `f2.get()` warten wir darauf, dass beide Supplier ihrer Arbeit getan haben.

Hier ein möglicher Ablauf:



Das Resultat von `f` wäre dann `42`.

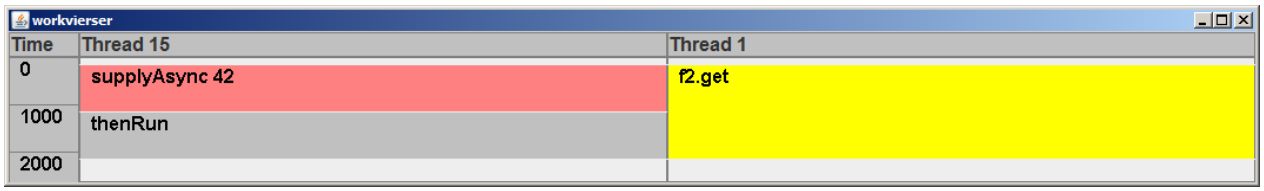
Im folgenden werden die verschiedenen `then...`-Methoden vorgestellt. All diese Methoden sind Instanzmethoden.

thenRun

```
    static void demoThenRun() throws Exception {
        final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
                () -> swork("supplyAsync 42",
1000).thenReturn(42));
        final CompletableFuture<Void> f2 = f1.thenRun(
                () -> rwork("thenRun", 1000));
```

```
        final Void result = get("f2.get", f2);
    }
```

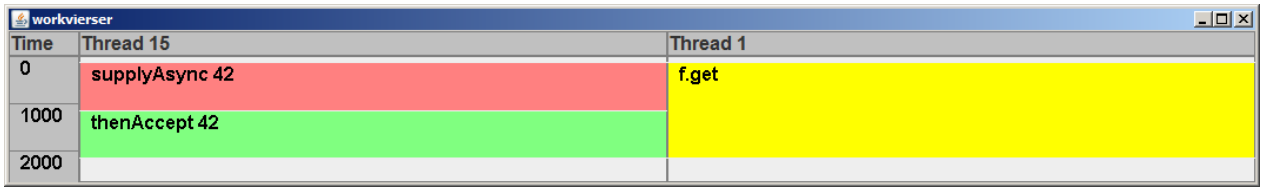
An `thenRun` wird ein `Runnable` übergeben; sie liefert `CompletableFuture<Void>` zurück.



thenAccept

An `thenAccept` wird ein `Consumer` übergeben. Sie liefert `CompletableFuture<Void>` zurück.

```
    static void demoThenAccept() throws Exception {
        final CompletableFuture<Void> f =
CompletableFuture.supplyAsync(
                () -> swork("supplyAsync 42",
1000).thenReturn(42))
            .thenAccept(
                x -> cwork("thenAccept " + x, 1000));
        final Void result = get("f.get", f);;
    }
```

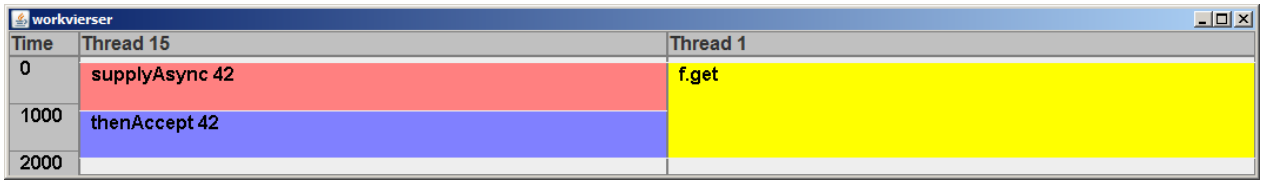


thenApply

Die Methode `thenApply` ist bereits aus dem letzten Abschnitt bekannt:

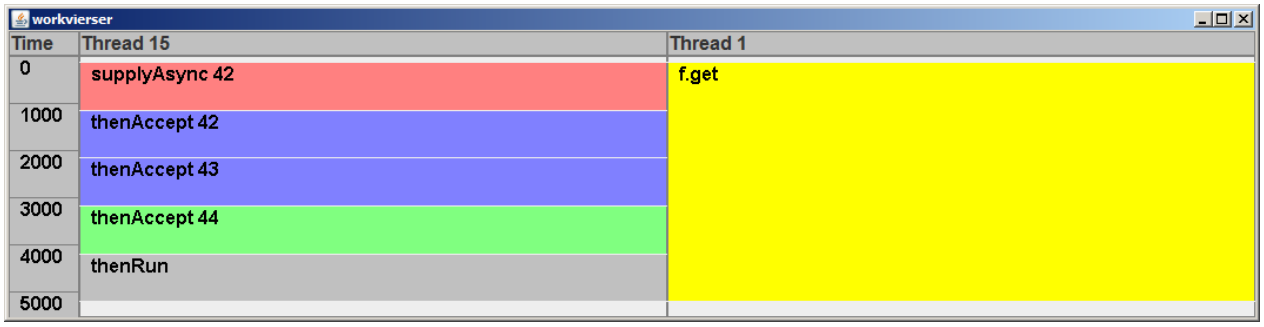
```
    static void demoThenApply() throws Exception {
        final CompletableFuture<Integer> f =
CompletableFuture.supplyAsync(
                () -> swork("supplyAsync 42",
1000).thenReturn(42))
            .thenApply(
```

```
            x -> fwork("thenAccept " + x, 1000).thenReturn(x +
1));
        final Integer result = get("f.get", f);;
    }
```



Eine then-then-then-Folge

```
    static void demoThenApplyThenAcceptThenRun() throws Exception
{
        final CompletableFuture<Void> f =
CompletableFuture.supplyAsync(
                () -> swork("supplyAsync 42",
1000).thenReturn(42))
            .thenApply(
                x -> fwork("thenAccept " + x, 1000).thenReturn(x
+ 1))
            .thenApply(
                x -> fwork("thenAccept " + x, 1000).thenReturn(x
+ 1))
            .thenAccept(
                x -> cwork("thenAccept " + x, 1000))
            .thenRun(
                () -> rwork("thenRun", 1000));
        final Void result = get("f.get", f);;
    }
```



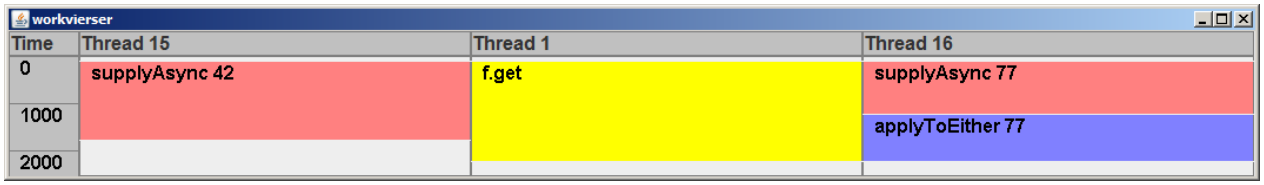
Im folgenden werden `...either`- und `...both`-Methoden vorgestellt.

applyToEither

An `applyToEither` wird ein weiteres `CompletableFuture` und eine `Function` übergeben. Je nachdem, welches der beiden `CompletableFutures` eher zurückkehrt, wird dessen Output als Input für das neu erzeugte `CompletableFuture` verwendet. Im folgenden Beispiel wird wieder diejenige `swork`-Methode verwendet, die eine Zufalls-Sleeptime benutzt:

```
    static void demoApplyToEither() throws Exception {
        final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 42", 1000,
3000).thenReturn(42));
        final CompletableFuture<Integer> f2 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 77", 1000,
3000).thenReturn(77));
        final CompletableFuture<Integer> f = f1.applyToEither(f2,
            x -> fwork("applyToEither " + x, 1000).thenReturn(x +
1));
        final Integer result = get("f.get", f);
        f1.get();
        f2.get();
    }
```

Ein möglicher Ablauf:

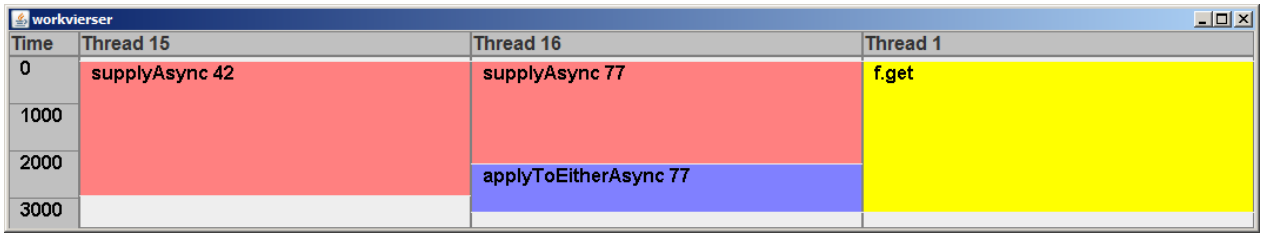


Neben `applyToEither` gibt's auch `applyToEitherAsync`:

```
    static void demoApplyToEitherAsync() throws Exception {
        final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 42", 1000,
3000).thenReturn(42));
        final CompletableFuture<Integer> f2 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 77", 1000,
3000).thenReturn(77));
```

```
        final CompletableFuture<Integer> f =
f1.applyToEitherAsync(f2,
            x -> fwork("applyToEitherAsync " + x,
1000).thenReturn(x + 1));
        final Integer result = get("f.get", f);;
    }
```

Ein möglicher Ablauf:

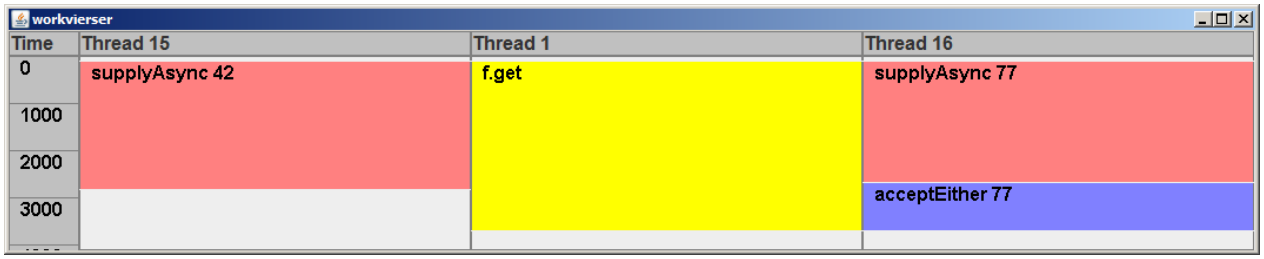


acceptEither

An `acceptEither` wird statt einer `Function` (wie bei `applyEither`) ein `Consumer` übergeben:

```
    static void demoAcceptEither() throws Exception {
        final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 42", 1000,
3000).thenReturn(42));
        final CompletableFuture<Integer> f2 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 77", 1000,
3000).thenReturn(77));
        final CompletableFuture<Void> f = f1.acceptEither(f2,
            x -> fwork("acceptEither " + x, 1000));
        final Void result = get("f.get", f);;
    }
```

Ein möglicher Ablauf:



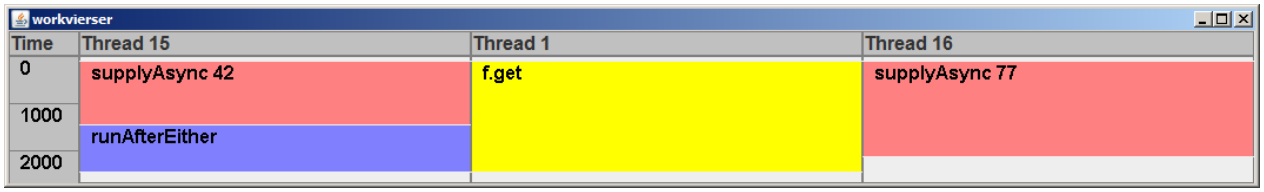
Neben `acceptEither` gibt's auch `acceptEitherAsync`.

runAfterEither

An `runAfterEither` schließlich wird ein `Runnable` übergeben. (Auch hier gibt's `runAfterEitherAsync`).

```
    static void demoRunAfterEither() throws Exception {
        final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 42", 1000,
3000).thenReturn(42));
        final CompletableFuture<Integer> f2 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 77", 1000,
3000).thenReturn(77));
        final CompletableFuture<Void> f = f1.runAfterEither(f2,
            () -> fwork("runAfterEither", 1000));
        final Void result = get("f.get", f);;
    }
```

Ein möglicher Ablauf:



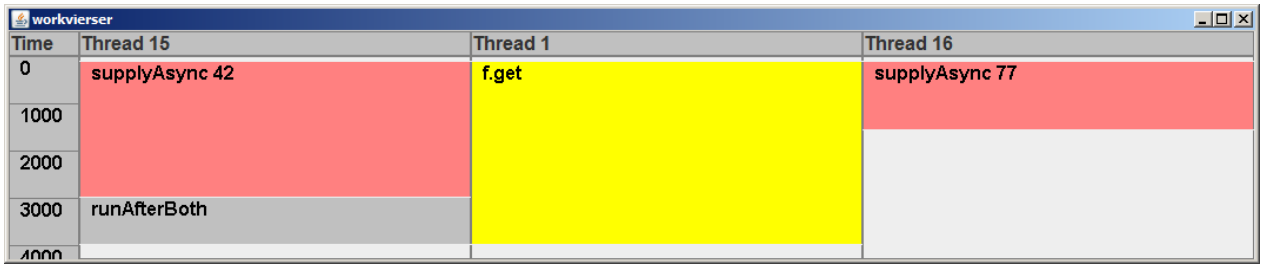
runAfterBoth

`runAfterBoth` startet erst dann, wenn von zwei `CompletableFutures` auch das letzte seine Arbeit erledigt hat:

```
    static void demoRunAfterBoth() throws Exception {
        final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 42", 1000,
3000).thenReturn(42));
        final CompletableFuture<Integer> f2 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 77", 1000,
3000).thenReturn(77));
        final CompletableFuture<Void> f = f1.runAfterBoth(f2,
```

```
            () -> rwork("runAfterBoth", 1000));
        final Void result = get("f.get", f);;
    }
```

Ein möglicher Ablauf:

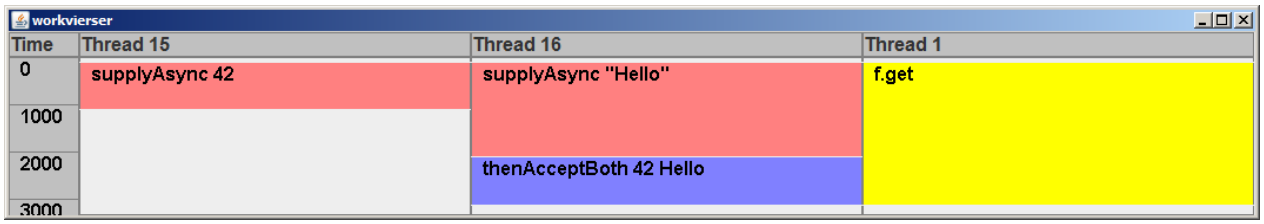


thenAcceptBoth

An `thenAcceptBoth` wird neben dem anderen `CompletableFuture` ein `BiConsumer` übergeben – der die Ergebnisse von zwei `CompletableFuture`s konsumiert:

```
    static void demoThenAcceptBoth() throws Exception {
        final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 42", 1000).thenReturn(42));
        final CompletableFuture<String> f2 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync \"Hello\"",
2000).thenReturn("Hello"));
        final CompletableFuture<Void> f = f1.thenAcceptBoth(f2,
            (i, s) -> fwork("thenAcceptBoth " + i + " " + s,
1000));
        final Void result = get("f.get", f);;
    }
```

Ein möglicher Ablauf:



Im folgenden geht's um einige weitere Methoden von `CallableFuture`.

complete

Die `complete`-Methode ist bereits aus dem letzten Abschnitt bekannt.

`complete` kann auf ein `CompletableFuture`-Objekt in demjenigen Thread aufgerufen werden, in welchem das `CompletableFuture` erzeugt wurde:

```
    static void demoComplete1() throws Exception {
        final CompletableFuture<Integer> f = new
CompletableFuture<>();
        f.complete(42);
        final Integer result = get("f.get", f);;
    }
```

Als Resultat wird `42` geliefert.

`complete` kann auch in einem anderen Thread aufgerufen werden:

```
    static void demoComplete2() throws Exception {
        final CompletableFuture<Integer> f = new
CompletableFuture<>();
        new Thread() {
            public void run() {
                rwork("run" , 1000);
                f.complete(42);
            }
        }.start();
        final Integer result = get("f.get", f);
    }
```

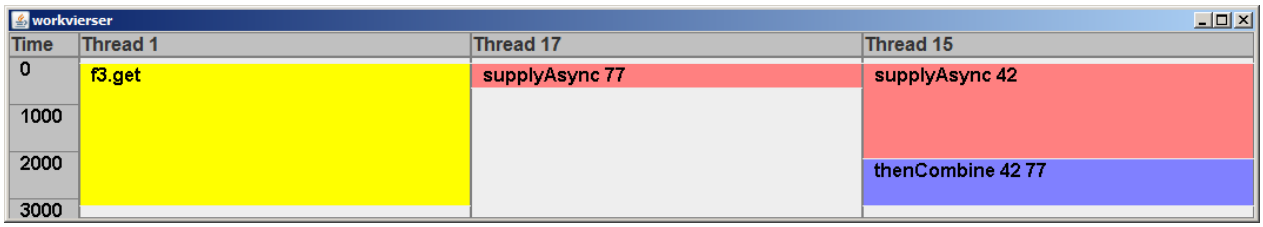
combine

`combine` wurde bereits im letzten Abschnitt verwendet. An die Methode wird neben dem "anderen" `CompletableFuture` eine `BiFunction` übergeben:

```
    static void demoCombine() throws Exception {
        final CompletableFuture<Integer> f1 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 42", 2000).thenReturn(42));
        final CompletableFuture<Integer> f2 =
CompletableFuture.supplyAsync(
            () -> swork("supplyAsync 77", 500).thenReturn(77));
        final CompletableFuture<Integer> f3 = f1.thenCombine(f2,
```

```
                (x, y) -> fwork("thenCombine " + x + " " + y,
1000)
                        .thenReturn(x + y));
        final Integer result = get("f3.get", f3);
    }
```

Ein möglicher Ablauf:



Warten mit get

Mehrere Threads können gleichzeitig mittels des Aufrufs von `get` auf das Ergebnis eines `CompletableFutures` warten. Alle `get`-Aufrufe kehren zurück, wenn das Ergebnis vorliegt:

```
    static void demoWait() throws Exception {
        final CompletableFuture<Integer> f = new
CompletableFuture<>();
        class MyThread extends Thread {
            public void run() {
                try {
                    final int value = get("f.get", f);
                }
                catch (Exception e) {
                    throw new RuntimeException(e);
                }
            }
        }
        for (int i = 0; i < 5; i++)
            new MyThread().start();
        rwork("wait", 1000);
        f.complete(42);
        final int value = get("f.get", f);
    }
```

Man beachte, dass neben der parameterlosen `get`-Methode noch eine parametrisierte existiert: als Parameter wird ihr die maximale Zeit, die gewartet werden soll, übergeben. In der Regel sollte diese parametrisierte Methode genutzt werden...

CallableFutures mit verschiedenen Typ-Parametern

Z.B. kann das erste `CallableFuture` einen `String` liefern; das zweite aus diesem `String` einen `Integer` bauen; und das letzte aus dem `Integer` einen `Double` produzieren:

```
    static void demoDifferentTypes1() throws Exception {
        final CompletableFuture<String> f1 =
CompletableFuture.supplyAsync(
            () -> "10");
        final CompletableFuture<Integer> f2 = f1.thenApply(
            Integer::parseInt);
        final CompletableFuture<Double> f3 = f2.thenApply(
            r -> r * r * Math.PI);
        final Double result = get("f3.get", f3);
    }
```

Natürlich kann das Ganze auch fluent programmiert werden:

```
    static void demoDifferentTypes2() throws Exception {
        CompletableFuture<Double> f =
CompletableFuture.supplyAsync(
            () -> "10")
            .thenApply(Integer::parseInt)
            .thenApply(r -> r * r * Math.PI);
        Double result = get("f.get", f);;
    }
```

Exceptions

```
    static void demoExceptions() throws Exception {
        CompletableFuture<Void> f =
CompletableFuture.supplyAsync(
            () -> 42
        ).thenApplyAsync((i) -> {
            tlog("thenApply");
            throw new RuntimeException("abc");
        }).exceptionally(e -> {
            tlog("exceptionally");
            return -1;
        }).thenAcceptAsync((i) -> {
            tlog("i = " + i);
        });
        Void result = get("f.get", f);;
```

```
    }
```

Da das Parsen von `"abc"` zu einer Exception führt, wird der `exceptionally`-Zweig ausgeführt. Es wird der Text `"exceptionally"` ausgegeben. Da `exceptionally` nun den Wert `-1` liefert, wird eben dieser Wert als Input für `thenAcceptAsync` verwendet – dessen Consumer dann diesen Wert (`-1`) ausgibt.

CompletableFutures und Streams

`CompletableFuture`s scheinen irgendetwas mit `Stream`s gemeinsam zu haben. Die Gemeinsamkeiten beschränken sich aber darauf, dass sowohl bei `Stream`s als auch bei `CompletableFuture`s Parallelität unterstützt wird – und dass es "Abschnitte" gibt, die aufeinander folgen.

Die Unterschiede:

Streams:

- Die Quelle eines Streams ist in irgendeiner Art und Weise eine Kollektion von Werten – von Werten, die i.d.R. individuell weitergereicht und verarbeitet werden.
- Jeder Abschnitt hat genau einen einzigen Nachfolger (oder keinen) – jeder genau einen (oder keinen) Vorgänger
- Am Ende steht genau ein einziges Resultat – welches von einer terminalen Operation geliefert wird (oder: void – wie etwa bei forEach).
- Verschiedene Element können sich zu einem bestimmten Zeitpunkt in verschiedenen Abschnitten befinden.

CompletableFutures:

- Jeder Abschnitt baut genau ein einzigen (womöglich komplexes!) Element auf – auf welches der Nachfolger (oder: die Nachfolger) warten muß (müssen).
- Ein Abschnitt kann mehrere Vorgänger haben – und mehrere Nachfolger.
- Ein Abschnitt muss komplett "fertig" sein, bevor Nachfolger ihre Arbeit beginnen können.
- Am Ende können mehrere Resultate stehen.

10.3 StampedLock

`StampedLock` ist nicht reentrant. Ein Read-Lock, den man von einem `StampedLock` erhalten hat, kann in einen WriteLock konvertiert werden. `StampedLock` ermöglicht optimistisches Lesen.

Wir benutzen zur Demonstration verschiedene Klassen, die allesamt das folgende Interface implementieren:

```
package appl;

public interface Account {
    public abstract void withdraw(int amount);
    public abstract int getAvailable();

    public static void check(int amount) {
        if (amount < 0)
            throw new IllegalArgumentException();
    }
}
```

Von einem Konto kann man etwas abheben – aber nur soviel, wie verfügbar ist. getAvailable liefert den verfügbaren Bestand. Dieser Bestand wird sich jeweils zusammensetzen aus dem tatsächlichen Bestand und einem Kreditlimit.

ReentrantReadWriteLock

Um die Eigenschaften von `StampedLock` darstellen zu können, beginnen wir mit der seit Java 5 existierenden Klasse `ReentrantReadWriteLock` — also mit einer Implementierung des `Account`-Interfaces, welche einen `ReentrantReadWriteLock` nutzt.

```
package appl;
// ...
public class Account1 implements Account {

    private int balance;
    private int credit;

    private final ReadWriteLock lock = new
ReentrantReadWriteLock();

    public Account1(int balance, int credit) {
```

```
        Account.check(balance);
        Account.check(credit);
        this.balance = balance;
        this.credit = credit;
    }

    public void withdraw(int amount) {
        Account.check(amount);
        final Lock l = this.lock.writeLock();
        l.lock();
        tlog("\t\twithdraw: after writeLock");
        try {
            if (amount > this.getAvailable())
                throw new IllegalArgumentException();
            xrun(() -> Thread.sleep(1000));
            this.balance -= amount;
        }
        finally {
            tlog("\t\twithdraw: before unlock");
            l.unlock();
        }
    }

    public int getAvailable() {
        final Lock l = this.lock.readLock();
        l.lock();
        tlog("\t\tgetAvailable: after readLock");
        try {
            return this.balance + this.credit;
        }
        finally {
            tlog("\t\tgetAvailable: before unlock");
            l.unlock();
        }
    }
}
```

`withdraw` forder einen Write-Lock an; `getAvailable` einen Read-Lock. `withdraw` legt sich zwischen der Anforderung des Write-Locks und seiner Freigabe eine Sekunde schlafen (der Bankangestellte muss das Geld erst aus dem Tresor holen).

Man beachte, dass in dem kritischen Abschnitt von `withdraw` die `getAvailable`-Methode aufgerufen wird. Damit soll demonstriert werden, dass der Lock reentrant ist (im Unterschied zum `StampedLock`).

Alle Ausgaben, die in der Klasse produziert werden, werden eingerückt dargestellt.

Hier die erste Demo-Methode:

```
    static void demoConcurrentWriteRead(Account account) {
        mlog(account.getClass().getSimpleName());
        Thread t1 = new Thread(() -> {
            tlog(">> deposit");
            account.withdraw(4000);
            tlog("<< deposit");
        });
        t1.start();
        xrun(() -> Thread.sleep(500));
        Thread t2 = new Thread(() -> {
            tlog(">> getAvailable");
            tlog("available = " + account.getAvailable());
            tlog("<< getAvailable");
        });
        t2.start();
        xrun(() -> t1.join());
        xrun(() -> t2.join());
    }
```

Die Methode startet einen Thread, in welchem `deposit` aufgerufen wird, Kurze Zeit später – der Bankangestelle ist noch im Tresor – wird die `getAvailable`-Methode aufgerufen.

Angenommen, die Methode wird mit einer `Account1`-Instanz aufgerufen. Dann werden folgende Ausgaben produziert:

```
+-----------------------------------------------
| demoConcurrentWriteRead [Account1]
+-----------------------------------------------
[  8 ] >> deposit
[  8 ]      withdraw: after writeLock
[  8 ]      getAvailable: after readLock
[  8 ]      getAvailable: before unlock
[  9 ] >> getAvailable
[  8 ]      withdraw: before unlock
[  9 ]      getAvailable: after readLock
[  9 ]      getAvailable: before unlock
[  9 ] available = 2000
[  9 ] << getAvailable
[  8 ] << deposit
```

Von `deposit` konnte `getAvailable` aufgerufen werden; die von der Anwendung aufgerufene `getAvailable`-Methode erhält den Read-Lock erst dann, wenn `deposit` seinen Write-Lock freigegeben hat.

StampedLock – eine einfache Verwendung

`Account2` implmentiert das `Account`-Interface mittels `StampedLock`. Wie auch im letzten Beispiel wird in `deposit` wird ein Write-Lock, in `getAvailable` ein Read-Lock angefordert:

```
package appl;
// ...
public class Account2 implements Account {
    // ...
    private final StampedLock lock = new StampedLock();

    public void withdraw(int amount) {
        Account.check(amount);
        final long stamp = this.lock.writeLock();
        tlog("\t\twithdraw: after writeLock : " + stamp);
        try {
            if (amount > this.balance + this.credit)
            // if (amount > this.getAvailable())  // this will
not work...
                throw new IllegalArgumentException();
            xrun(() -> Thread.sleep(1000));
            this.balance -= amount;
        }
        finally {
            tlog("\t\twithdraw: before unlock");
            this.lock.unlockWrite(stamp);
        }
    }
    public int getAvailable() {
        final long stamp = this.lock.readLock();
        tlog("\t\tgetAvailable: after readLock : " + stamp);
        try {
            return this.balance + this.credit;
        }
        finally {
            tlog("\t\tgetAvailable: before unlock");
            this.lock.unlockRead(stamp);
        }
    }
}
```

Anders als bei der Klasse `ReentrantReadWriteLock` liefern die Methoden `readLock` und `writeLock` nun einen `long`-Wert zurück, der den Lock identifiziert. Um einen Lock

freizugeben, wird `unlockRead` resp. `unlockWrite` aufgerufen, wobei der "Stamp" als Parameter übergeben wird.

Wird nun ein `Account2` an die `demoConcurrentWriteRead`-Methode übergeben, sieht die Ausgabe wie folgt aus:

```
+---------------------------------------------
| demoConcurrentWriteRead [Account2]
+---------------------------------------------
[ 10 ] >> deposit
[ 10 ]      withdraw: after writeLock : 384
[ 11 ] >> getAvailable
[ 10 ]      withdraw: before unlock
[ 10 ] << deposit
[ 11 ]      getAvailable: after readLock : 513
[ 11 ]      getAvailable: before unlock
[ 11 ] available = 2000
[ 11 ] << getAvailable
```

Der entscheidende Unterschied in der Implementierung liegt in der ersten der beiden folgenden Zeilen der deposit-Methode:

```
            if (amount > this.balance + this.credit)
                throw new IllegalArgumentException();
```

Hier wird – anders als in `Account1` – nicht die `getAvailable`-Methode aufgerufen! Würde man `getAvailable` aufrufen, würde ein Deadlock entstehen: `StampedLock`s sind nicht reentrant.

StampedLock – Weiterreichen des Stamps

Um in `withdraw` dennoch die `getAvailable`-Methode aufrufen zu können, könnte man sich eines Tricks bedienen – man muss den Stamp an `getAvailable` weiterreichen. Da die Schnittstelle dieser Methode natürlich nicht geändert werden kann, nutzen wir `ThreadLocal`:

```
package appl;
// ...
public class Account3 implements Account {
    // ...
    private final StampedLock lock = new StampedLock();
    private final ThreadLocal<Long> stamps = new ThreadLocal<>();

    public void withdraw(int amount) {
        Account.check(amount);
        stamps.set(this.lock.writeLock());
```

```
        tlog("\t\twithdraw: after writeLock : " + stamps.get());
        try {
            if (amount > this.getAvailable())
                throw new IllegalArgumentException();
            xrun(() -> Thread.sleep(1000));
            this.balance -= amount;
        }
        finally {
            tlog("\t\twithdraw: before unlock");
            this.lock.unlockWrite(stamps.get());
            stamps.remove();
        }
    }
    public int getAvailable() {
        long stamp = 0;
        Long s = stamps.get();
        if (s == null)
            stamp = this.lock.readLock();
        tlog("\t\tgetAvailable: after readLock : " + stamp);
        try {
            return this.balance + this.credit;
        }
        finally {
            tlog("\t\tgetAvailable: before unlock");
            if (s == null)
                this.lock.unlockRead(stamp);
        }
    }
}
```

`withdraw` stellt den Write-Stamp in den `stamps-ThreadLocal` ab – und entfernt ihn natürlich in dem `finally`-Zweig. `getAvailable` schaut nach, ob mit dem aktuellen Thread bereits ein Stamp assoziiert ist – und fordert den Read-Lock nur dann an, wenn der `ThreadLocal` nichts hergibt (und gibt ihn natürlich auch nur dann frei).

Wird also `getAvailable` von "außen" aufgerufen, wird ein Read-Lock angefordert; wird die Methode aber von `withdraw` aufgerufen, wird kein Lock angerfordert.

Wird ein `Account3` an `demoConcurrentWriteRead` übergeben, wird folgende Ausgabe produziert:

```
+------------------------------------------------
| demoConcurrentWriteRead [Account3]
+------------------------------------------------
[ 12 ] >> deposit
[ 12 ]      withdraw: after writeLock : 384
```

```
[ 12 ]      getAvailable: after readLock : 0
[ 12 ]      getAvailable: before unlock
[ 13 ] >> getAvailable
[ 12 ]      withdraw: before unlock
[ 13 ]      getAvailable: after readLock : 513
[ 13 ]      getAvailable: before unlock
[ 13 ] available = 2000
[ 13 ] << getAvailable
[ 12 ] << deposit
```

StampedLock – optimistisches Lesen

Die folgende Klasse `Account4` demonstriert optimistisches Lesen:

```
package appl;
// ...
public class Account4 implements Account {
    // ...

    private final StampedLock lock = new StampedLock();

    public void withdraw(int amount) {
        Account.check(amount);
        final long stamp = this.lock.writeLock();
        tlog("\t\twithdraw: after writeLock : " + stamp);
        try {
            if (amount > this.balance + this.credit)
                throw new IllegalArgumentException();
            xrun(() -> Thread.sleep(1000));
            this.balance -= amount;
        }
        finally {
            tlog("\t\twithdraw: before unlock");
            this.lock.unlockWrite(stamp);
        }
    }
    public int getAvailable() {
        long stamp = this.lock.tryOptimisticRead();
        int result = this.balance + this.credit;
        if (this.lock.validate(stamp)) {
            tlog("\t\tvalidate okay");
            return result;
        }
        stamp = lock.readLock();
        tlog("\t\tgetAvailable: after readLock : " + stamp);
        try {
```

```
            return this.balance + this.credit;
        }
        finally {
            tlog("\t\tgetAvailable: before unlock");
            this.lock.unlockRead(stamp);
        }
    }
}
```

In `withdraw` wird der verfügbare Bestand wieder an Ort und Stelle berechnet – ohne also auf das Reentrant-Problem zu stoßen.

`tryOptimisticRead` liefert einen Stamp zurück – ohne allerdings bereits eine reale Sperre zu setzten. Dann kann in aller Ruhe ein Ergebnis berechnet werden (`balance + credit`). Dieses Ergebnis kann natürlich falsch sein – ein Schreiber könnte dazwischen funken (oder ein solcher Schreiber könnte bereits existieren). Nach der Berechnung wird `validate` aufgerufen – und diese Methode liefert `false`, wenn tatsächlich ein Schreiber dazwischen gefunkt hat. Dann muss die Berechnung natürlich erneut ausgeführt werden – mittels einer normalen Lesesperre. Liefert `validate` dagegen `true`, dann ist das Ergebnis der ersten Berechnung korrekt – und kann zurückgeliefert werden.

Wird ein `Account4` an `demoConcurrentWriteRead` übergeben, wird sich natürlich zeigen, dass der Optimismus fehl am Platz war (`validate` liefert `false`, weil bereits ein Schreiber existiert: `deposit` läuft gerade...):

```
+-----------------------------------------------
| demoConcurrentWriteRead [Account4]
+-----------------------------------------------
[ 14 ] >> deposit
[ 14 ]      withdraw: after writeLock : 384
[ 15 ] >> getAvailable
[ 14 ]      withdraw: before unlock
[ 15 ]      getAvailable: after readLock : 513
[ 15 ]      getAvailable: before unlock
[ 15 ] available = 2000
[ 15 ] << getAvailable
[ 14 ] << deposit
```

Angenommen aber, die Zugriffe (die Aufrufe von deposit und getAvailable) finden nicht parallel, sondern sequentiell statt:

```
    static void demoSequentialWriteRead(Account account) {
        mlog(account.getClass().getSimpleName());
        Thread t1 = new Thread(() -> {
            tlog(">> deposit");
            account.withdraw(4000);
            tlog("<< deposit");
```

```
        });
        t1.start();
        xrun(() -> t1.join());
        Thread t2 = new Thread(() -> {
            tlog(">> getAvailable");
            tlog("available = " + account.getAvailable());
            tlog("<< getAvailable");
        });
        t2.start();
        xrun(() -> t2.join());
    }
```

Dann werden, sofern dieser Methode ein `Account4` übergeben wird, folgende Ausgaben produziert:

```
+-----------------------------------------------
| demoSequentialWriteRead [Account4]
+-----------------------------------------------
[ 18 ] >> deposit
[ 18 ]      withdraw: after writeLock : 384
[ 18 ]      withdraw: before unlock
[ 18 ] << deposit
[ 19 ] >> getAvailable
[ 19 ]      validate okay
[ 19 ] available = 2000
[ 19 ] << getAvailable
```

`validate` würde dann `true` liefern – eine Lese-Sperre ist also nicht erforderlich.

StampedLock – Konvertierung eines Locks

Ein Read-Lock kann bei Bedarf in einen Write-Lock konvertiert werden. Angenommen, es ist sehr wahrscheinlich, dass die Verfügbarkeitsprüfung in withdraw negativ ausfällt. In solchen Situationen ist dann der Write-Lock eigentlich zuviel des Guten – ein Read-Lock genügt. In denjenigen Fällen, in denen die Prüfung aber positiv ausfällt, muss der Read-Lock in einen Write-Lock konvertiert werden können – unter dessen Kontrolle dann die Dekrementierung des Bestands ausgeführt wird.

Hier die Lösung:

```
package appl;
// ...
public class Account5 implements Account {
   // ...

    private final StampedLock lock = new StampedLock();
```

```
    public void withdraw(int amount) {
        Account.check(amount);
        long stamp = this.lock.readLock();
        tlog("\t\twithdraw: after readLock : " + stamp);
        try {
            if (amount > this.balance + this.credit)
                throw new IllegalArgumentException();
            long writeStamp =
this.lock.tryConvertToWriteLock(stamp);
            if (writeStamp == 0) {
                this.lock.unlock(stamp);
                tlog("\t\twithdraw: unlock readLock : " + stamp);
                stamp = this.lock.writeLock();
                tlog("\t\twithdraw: after writeLock");
            }
            else {
                tlog("\t\twithdraw: convert done");
                stamp = writeStamp;
            }
            xrun(() -> Thread.sleep(1000));
            this.balance -= amount;
        }
        finally {
            tlog("\t\twithdraw: before unlock");
            this.lock.unlock(stamp);
        }
    }
    public int getAvailable() {
        final long stamp = this.lock.readLock();
        tlog("\t\tgetAvailable: after readLock : " + stamp);
        try {
            return this.balance + this.credit;
        }
        finally {
            tlog("\t\tgetAvailable: before unlock");
            this.lock.unlockRead(stamp);
        }
    }
}
```

Hier eine letzte Demo-Methode:

```
    static void demoPositiveNegativeTest(Account account) {
        mlog(account.getClass().getSimpleName());
        Thread t1 = new Thread(() -> {
```

```
            tlog(">> deposit");
            account.withdraw(4000);
            tlog("<< deposit");
        });
        t1.start();
        xrun(() -> t1.join());
        Thread t2 = new Thread(() -> {
            try {
                tlog(">> deposit");
                account.withdraw(1000000);
                tlog("<< deposit");
            }
            catch(Exception e) {
                tlog("expected exception");
            }
        });
        t2.start();
        xrun(() -> t2.join());
    }
```

Die beiden `deposit`-Threads werden nacheinander ausgeführt. Beim ersten `deposit`-Aufruf fällt die Prüfung positiv aus; beim zweiten negativ.

Hier die Ausgaben:

```
+-----------------------------------------------
| demoPositiveNegativeTest [Account5]
+-----------------------------------------------
[ 22 ] >> deposit
[ 22 ]      withdraw: after readLock : 257
[ 22 ]      withdraw: convert done
[ 22 ]      withdraw: before unlock
[ 22 ] << deposit
[ 23 ] >> deposit
[ 23 ]      withdraw: after readLock : 513
[ 23 ]      withdraw: before unlock
[ 23 ] expected exception
```

Man sieht: der letzte Thread hat keinerlei Schreibsperre angefordert. Der erste konnte seine Lese- in eine Schreibsperre konvertieren.

Resultat

`StampedLock` erlaubt eine sehr feingranulare Steuerung – ist aber eben deshalb auch nicht einfach zu handhaben.

10.4 Aufgaben

CompletableFuture 1

Implementieren Sie folgende Funktion (ohne Helper-Klasse!) - achten Sie dabei auf größtmögliche Parallelität!:

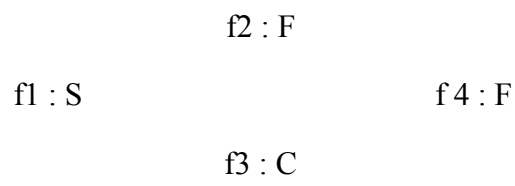
f(a, b, c) = (a + 1) * (b + 2) * (c + 3)

CompletableFuture 2

Implementieren Sie obige Funktion mit einer Helper-Klasse namens `Triple`!

CompletableFuture 3

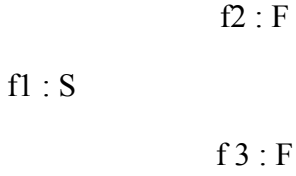
Programmieren Sie folgendes Netz:



`f1` (ein `Supplier`) soll `Integer` liefern; `f2` (eine `Function`) soll `Integer` verlangen und einen neuen `Integer` liefern; `f3` (ein `Consumer`!) soll einen `Integer` verlangen (und kann, weil `Consumer`, natürlich nichts liefern). `f4` soll erst dann starten werden, wenn sowohl `f2` als auch `f3` ihre Arbeit erledigt haben. Was ist das Resultat?

CompletableFuture 4

Programmieren Sie folgendes Netz:



11 Nashorn

Nashorn ist die neue JavaScript-Engine von Java. Mittels dieser Engine können JavaScript-Programme interpretativ ausgeführt werden.

Der Nashorn-Interpreter löst den "alten" Rhino ab.

Hier eine Übersicht zu den folgenden Abschnitten:

- Im folgenden wird zunächst gezeigt, wie ein Nashorn erzeugt wird – und dieses mittels der Methode `eval` dazu veranlasst werden kann, ein Sript auszuführen.
- Dann wird gezeigt, wie aus einem Java-Programm JavaSript-Funktionen aufgerufen werden können (mittels der Methode `invokeFunction`).
- Im dritten Abschnitt wird gezeigt, wie ein Skript geladen werden kann, welches auf mehrere `js`-Dateien verteilt ist.
- Der letzte Abschnitt zeigt schließlich, wie aus JavaScript heraus Methoden auf Java-Objekte aufgerufen werden können.

Natürlich gäbe es zu der Verbindung von Java und JavaScript weitaus mehr zu sagen...

Die folgenden Projekte verwenden u.a. folgende Importe:

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
```

11.1 Start

Zunächst benötigen wir eine `SriptEngine`. Die besorgen wir uns mittels eines `ScriptEngineManager`s. Die Engine wird mittels eines Namens angefordert: entweder `"nashorn"` oder allgemeiner: `"JavaScript"`. Egal, welchen dieser beiden Namen wir verwenden, wir bekommen jeweils eine `NashorhnScriptEngine` (`ScriptEngine` ist natürlich nur ein Interface):

```
    static void demoCreateEngine() throws ScriptException {
        ScriptEngine engine1 =
            new ScriptEngineManager().getEngineByName("nashorn");
        ScriptEngine engine2 =
            new
ScriptEngineManager().getEngineByName("JavaScript");
        System.out.println(engine1.getClass().getName());
        System.out.println(engine2.getClass().getName());
        System.out.println(engine1 == engine2);
    }
```

Die Ausgaben:

```
jdk.nashorn.api.scripting.NashornScriptEngine
jdk.nashorn.api.scripting.NashornScriptEngine
false
```

Wie können JavaScript-Anweisungen (also ein Skript) an die ScriptEngine übergeben werden? Im folgenden werden drei Varianten vorgestellt. Dabei produzieren die `demo`-Methoden allesamt dieselbe Ausgabe:

```
Hello World!
```

Die erste Variante ruft die `eval`-Methode der `ScriptEngine` auf und übergibt ihr einen String, der das JavaScript-Statement enthält:

```
    static void demoEvalString() throws ScriptException {
        ScriptEngine engine =
            new ScriptEngineManager().getEngineByName("nashorn");
        engine.eval("print('Hello World!');");
    }
```

Hier wird eine `print`-Methode aufgerufen. Diese `print`-Methode ist natürlich Nashorn-spezifisch. Sie wird von Nashorn in das globale JavaScript-Objekt eingehängt.

Die `eval`-Methode ist überladen. Statt eines `Strings`, welcher das Script enthält, kann auch ein `Reader` übergeben werden. Im folgenden Beispiel wird ein `StringReader` übergeben:

```
    static void demoEvalStringReader() throws ScriptException {
        Reader reader = new StringReader("print('Hello
World!')");
        ScriptEngine engine =
            new ScriptEngineManager().getEngineByName("nashorn");
        engine.eval(reader);
    }
```

In der letzten Variante wird das Script aus einer Datei gelesen – mittels eines `FileReader`s, welcher dann als `Reader` an `eval` übergeben wird:

```
    static void demoEvalFileReader()
            throws ScriptException, FileNotFoundException {
        Reader reader = new FileReader("src/hello.js");
        ScriptEngine engine =
            new ScriptEngineManager().getEngineByName("nashorn");
        engine.eval(reader);
    }
}
```

11.2 Invocable

Wie kann via Java eine JavaScript-Funktion aufgerufen werden?

Das Beispiel-Script (`plus.js`) definiert eine `plus`-Funktion:

```
var plus = function(x, y) {
    print('plus(' + x + ', ' + y + ')');
    return x + y;
};
```

Diese kann von Java wie folgt aufgerufen werden:

```
    public static void main(String[] args) throws Exception {
        ScriptEngine engine =
            new ScriptEngineManager().getEngineByName("nashorn");
        engine.eval(new FileReader("src/plus.js"));
        Invocable invocable = (Invocable) engine;
        int sum = (int)invocable.invokeFunction("plus", 40, 2);
        System.out.println(sum);
    }
```

Die Ausgaben:

```
plus(40, 2)
42
```

(Die erste Ausgabe kommt von JavaScript, die zweite von Java.)

Zunächst wird das Skript mittels `eval` geladen. (Da das Skript keine Anweisungen enthält, werden beim `eval` auch noch keine Anweisungen ausgeführt – sondern nur die darin enthaltene Funktion registriert).

Um via Java nun die `plus`-Funktion aufzurufen, muss die `ScriptEngine` nach `Invocable` gecastet werden. Über das `Invocable`-Interface kann dann die Methode `invokeFunction` aufgerufen werden, welcher der Name der JavaScript-Funktion und die beim Aufruf dieser Funktion zu übergebenden Parameter übergeben werden (in Form von varArgs). `invokeFunction` liefert dann das von der ausgeführten JavaScript-Funktion returnierte Ergebnis zurück (als `Object`-Referenz). Wir wissen, dass ein `Integer` geliefert wird...

11.3 Multiple Files

Das Projekt enthält drei Skript-Dateien: `calculator.js`, `pythagoras.js` und `main.js`. Die ersten beiden Dateien befinden sich im Verzeichnis `js`, die zweite im Verzeichnis `src`.

Das Skript `js/calculator.js` definiert die Funktionen `sqr` und `sqrt`:

```
var sqr = function(x) {
    return x * x;
};

var sqrt = function(x) {
    return Math.sqrt(x);
};
```

Das Skript `js/pythagoras.js` definiert die Funktionen `c` (Berechnung der Hypothenuse) und `a` (Berechnung einer Kathete) – und nutzt dabei die in Funktionen `sqr` und `sqrt`:

```
var c = function(a, b) {
    return sqrt(sqr(a) + sqr(b));
};

var a = function(c, b) {
    return sqrt(sqr(c) - sqr(b));
};
```

Das dritte Skript (`src/main.js`) definiert eine `main`-Funktion, welche die Pythagoras-Funktionen `c` und `a` aufruft:

```
function main() {
    print(c(3.0, 4.0));
    print(a(5.0, 3.0));
    print(a(5.0, 4.0));
};
```

Um die drei Skripts zu laden, wird eine allgemein verwendbare Utility-Methode genutzt: util.JSLoader:

```
package util;

import java.io.File;
import java.io.FileReader;
```

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class JSLoader {

    public static ScriptEngine load(String... files) {
        final ScriptEngine engine =
            new ScriptEngineManager().getEngineByName("nashorn");
        for (String file : files) {
            try {
                engine.eval(new FileReader(file));
            }
            catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
        return engine;
    }
}
```

Der `load`-Methode werden beliebig viele Dateinamen übergeben. Für jede dieser Dateien `eval` aufgerufen. Auf diese Weise wird der Code aller `js`-Dateien sukzessive geladen.

Die `main`-Methode von Java ruft die `main`-Funktion aus `src/main.js` auf:

```
package appl;

import javax.script.ScriptEngine;
import util.JSLoader;

public class Application {

    public static void main(String[] args) throws Exception {
        final ScriptEngine engine = JSLoader.load(
            "js/calculator.js", "js/pythagoras.js", "src/main.js"
        );
        engine.eval("main()");
    }
}
```

Die Ausgaben:

```
5
4
```

3

11.4 Calling Java Methods

Wie können in einem JavaScript-Programm Java-Methoden aufgerufen werden?

Es existiert folgende Java-Klasse:

```
package appl;

public class Calculator {

    public static int plus(int x, int y) {
        System.out.println("Calculator.plus(" + x + ", " + y +
")");
        return x + y;
    }

    public int minus(int x, int y) {
        System.out.println(this + ".minus(" + x + ", " + y +
")");
        return x - y;
    }

    public int times(int x, int y) {
        System.out.println(this + ".times(" + x + ", " + y +
")");
        return x * y;
    }
}
```

`plus` ist eine statische Methode, `minus` und `times` sind Instanzmethoden.

In `calulator.js` sind drei gleichnamige Funktionen vereinbart – Funktionen, welche die entsprechenden Java-Methoden aufrufen:

```
var plus = function(x, y) {
    print('plus(' + x + ', ' + y + ')');
    var cls = Java.type('appl.Calculator');
    return cls.plus(x, y);
};

var minus = function(obj, x, y) {
    print('minus(' + obj + ', ' + x + ', ' + y + ')');
    return obj.minus(x, y);
};
```

```
var times = function(x, y) {
    print('times(' + x + ', ' + y + ')');
    var cls = Java.type('appl.Calculator');
    var obj = new cls();
    return obj.times(x, y);
};
```

Der `plus`- und der `times`-Funktion werden nur die Parameter `x` und `y` übergeben; der `minus`-Funktion wird zusätzlich ein `obj`-Parameter übergeben (über welchen jeweils eine Instanz der Java-eigenen `Calculator`-Klasse referenziert werden wird).

Um in der `plus`-Funktion die statische `Calculator.plus`-Methode aufzurufen, muss das `Class`-Objekt ermittelt werden, welches die `Calculator`-Klasse beschreibt:

```
    var cls = Java.type('appl.Calculator');
```

Und über die so erhaltene `cls`-Variable kann dann die `plus`-Methode aufgerufen werden (und ihr Resultat zurückgeliefert werden):

```
    return cls.plus(x, y);
```

Die `minus`-Funktion ruft die `minus`-Methode eines `Calculator`s einfach über die ihr übergebene `obj`-Referenz auf (die hoffentlich tatsächlich ein `Calculator`-Objekt referenziert...):

```
    return obj.minus(x, y);
```

Die `times`-Funktion ermittelt zunächst das `Class`-Objekt, welches die `Calculator`-Klasse beschreibt:

```
    var cls = Java.type('appl.Calculator');
```

Dann wird ein Objekt dieser Klasse erzeugt:

```
    var obj = new cls();
```

Und auf dieses mittels JavaScript erzeugte `Calculator`-Objekt wird dann dessen `times`-Methode aufgerufen:

```
    return obj.times(x, y);
```

Hier schließlich die `main`-Methode, welche mittels der bereits vorgestellten `invokeFunction`-Methode die JavaScript-Funktionen `plus`, `minus` und `times` aufruft:

```
package appl;
// ...
public class Application {

    public static void main(String[] args) throws Exception {

        ScriptEngine engine =
            new
ScriptEngineManager().getEngineByName("JavaScript");
        engine.eval(new FileReader("src/calculator.js"));
        Invocable invocable = (Invocable) engine;

        int sum = (int)invocable.invokeFunction("plus", 40, 2);
        System.out.println(sum);

        Calculator calculator  = new Calculator();
        int diff = (int)invocable.invokeFunction("minus",
calculator , 40, 2);
        System.out.println(diff);

        int product = (int)invocable.invokeFunction("times", 40,
2);
        System.out.println(product);
    }
}
```

Die Ausgaben:

```
plus(40, 2)
Calculator.plus(40, 2)
42

minus(appl.Calculator@1083826, 40, 2)
appl.Calculator@1083826.minus(40, 2)
38

times(40, 2)
appl.Calculator@168ef40.times(40, 2)
80
```

11.5 Aufgaben

Wie können in Java Methoden Objekte aufgerufen werden, die im JavaScript-Kontext definiert sind?

12 Literatur

Michael Inden: Java – Die Neuerungen (DPunkt-Verlag)

Christian Ullebohm: Java SE 8 Standard-Bibliothek (Galileo Computing)