

2 Neuerungen in JDK 8

Am 18. März 2014 war es endlich so weit: Das zuvor mehrfach verschobene und lang erwartete Java 8 ist erschienen. Dieses Release enthält diverse wegweisende Erweiterungen und mit Lambda-Ausdrücken ein neues Sprachkonstrukt, das die funktionale Programmierung in Java erlaubt. Durch ein sorgfältiges API-Design der Massoperationen für Collections, sogenannten Bulk Operations on Collections, wie Filterung, Transformation und Sortierung, lässt sich die funktionale Programmierung gut mit der bisherigen objektorientierten Programmierung verbinden. Neben diesen Neuerungen gibt es eine Vielzahl weiterer Funktionalitäten in Java 8 zu entdecken. Dieses Kapitel gibt einen Überblick über wesentliche mit JDK 8 neu eingeführte Sprachfeatures und Erweiterungen. Dies sind unter anderem folgende:

- Lambda-Ausdrücke, Default-Methoden und Methodenreferenzen
- Bulk Operations on Collections
- Date And Time API
- JavaFX 8

Die aufgelisteten Themen werden in eigenen Unterkapiteln behandelt. Von den diversen weiteren Änderungen in JDK 8 behandle ich stellvertretend und überblicksartig unter anderem die Themen Parallel Array Sorting, Überarbeitung der Garbage Collection und die Unterstützung von Base64. Den Abschluss dieses Kapitels bildet eine Beschreibung über Sprachfeatures, die es leider auch nicht in JDK 8 geschafft haben.

Hinweis: Umfang von JDK 8

Schon JDK 7 wurde einige Male verschoben, unter anderem weil die Fertigstellung verschiedener Features wie Lambda-Ausdrücke, Modularisierung, ein neues Date And Time API usw. viel länger dauerte als geplant. Schließlich wurden diverse Funktionalitäten aus Zeitmangel von JDK 7 auf JDK 8 verschoben.

Wer nun gedacht hätte, dass alle diese Funktionalitäten auch tatsächlich im JDK 8 erscheinen würden, wird enttäuscht. Die Geschichte des Verschiebens und Abkündigens von Features wiederholt sich auch für JDK 8. Trotz mehrfacher Verschiebungen wurden einige Features leider nicht realisiert, wie z. B. die Modularisierung. Somit verbleibt als bedeutendste Neuerung in JDK 8 die Einführung von Lambda-Ausdrücken – allerdings sind deren Auswirkungen wirklich umfangreich und ziehen sich durch verschiedenste Teile des JDKs.

2.1 Lambda-Ausdrücke

Mit Lambda-Ausdrücken (kurz *Lambdas*, zum Teil auch *Closures* genannt) wurde ein neues und von vielen Entwicklern heiß ersehntes Sprachkonstrukt in Java eingeführt, das bereits in ähnlicher Form in verschiedenen anderen Programmiersprachen wie C#, Groovy und Scala erfolgreich genutzt wird. Der Einsatz von Lambdas erfordert zum Teil eine andere Denkweise und führt zu einem neuen Programmierstil, der dem Paradigma der *funktionalen Programmierung* folgt. Mithilfe von Lambdas lassen sich einige Lösungen auf sehr elegante Art und Weise formulieren. Insbesondere im Bereich von Frameworks und zur Parallelverarbeitung kann man aus Lambdas einen enormen Gewinn ziehen. Diverse Funktionalitäten im Collections-Framework und an anderen Stellen des JDKs wurden auf Lambdas umgestellt. Bevor wir darauf zurückkommen, schauen wir zunächst einmal auf Lambdas an sich.

Beispiel: Sortierung nach Länge und kommaseparierte Aufbereitung

Um die Vorteile von Lambdas und auch später von den sogenannten Bulk Operations on Collections besser nachvollziehen zu können, betrachten wir ein praxisnahes Beispiel einer Liste von Namen. Diese Namen wollen wir zunächst nach deren Länge sortieren und die Längen danach kommasepariert ausgeben. Dazu würden wir bis einschließlich JDK 7 in etwa folgenden Sourcecode schreiben:

```
// Sortierung mit Comparator
final List<String> names = Arrays.asList("Andy", "Michael", "Max", "Stefan");
Collections.sort(names, new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        return Integer.compare(str1.length(), str2.length());
    }
});

// Iteration und Ausgabe
final Iterator<String> it = names.iterator();
while (it.hasNext())
{
    System.out.print(it.next().length() + ", ");
}

// 3, 4, 6, 7, // Letztes Komma als kleine Unschönheit in der Ausgabe
```

Beim Betrachten dieser Funktionalität kann man sich fragen, ob das nicht kürzer und einfacher gehen sollte? Die Antwort lautet: Ja, mit JDK 8 kann man dazu Lambdas nutzen. Beginnen wir also nun die Entdeckungsreise ins JDK 8.

2.1.1 Einstieg in Lambdas

Das Sprachkonstrukt Lambda kommt aus der funktionalen Programmierung. Ein *Lambda* ist ein Behälter für Sourcecode ähnlich einer Methode, allerdings ohne Na-

men und ohne die explizite Angabe eines Rückgabetyps oder ausgelöster Exceptions. Vereinfacht ausgedrückt kann man einen Lambda-Ausdruck am ehesten als anonyme Methode mit folgender Syntax und spezieller Kurzschreibweise auffassen:

```
(Parameter-Liste) -> { Ausdruck oder Anweisungen }
```

Syntax von Lambdas am Beispiel

Ein paar recht einfache Beispiele für Lambdas sind die Addition von zwei Zahlen vom Typ `int`, die Multiplikation eines `long`-Werts mit 2 oder die Ausgabe eines Textes, die keinen Eingabeparameter besitzt. Diese drei Aktionen kann man als Lambdas wie folgt schreiben:

```
(int x, int y) -> { return x + y; }
(long x) -> { return x * 2; }
() -> { String msg = "Lambda"; System.out.println("Hello " + msg); }
```

Das Ganze sieht recht unspektakulär aus und insbesondere wird klar, dass ein Lambda lediglich ein Stück ausführbarer Sourcecode ist, der

- keinen Namen besitzt, sondern lediglich Funktionalität, und dabei
- keine explizite Angabe eines Rückgabetyps und
- keine Deklaration von Exceptions erfordert und erlaubt.

Lambdas im Java-Typsystem

Wir haben bisher gesehen, dass wir einfache Berechnungen mithilfe von Lambdas ausdrücken können. Wie können wir diese aber nutzen und aufrufen? Versuchen wir zunächst, einen Lambda einer `Object`-Referenz zuzuweisen, so wie wir es mit jedem anderen Objekt in Java auch tun können:

```
// Compile-Error: incompatible types: Object is not a functional interface
Object greeter = () -> { String msg = "Lambda";
    System.out.println("Hello " + msg); }
```

Die gezeigte Zuweisung ist nicht erlaubt und führt zu einem Kompilierfehler. Die Fehlermeldung gibt einen Hinweis auf inkompatible Typen und verweist darauf, dass `Object` kein Functional Interface ist. Jetzt stellt sich gleich die nächste Frage, was denn ein Functional Interface ist?

Besonderheit: Lambdas im Java-Typsystem

Bis zur Einführung von Lambdas konnte alles in Java (zumindest indirekt) auf den Basistyp `Object` abgebildet werden. Mit Lambdas existiert nun ein Sprachelement, das nicht (ohne weiteres) dem Basistyp `Object` zugewiesen werden kann.

Functional Interfaces und SAM-Typen

Ein **Functional Interface** ist eine neue Art von Typ, die mit JDK 8 eingeführt wurde und steht für ein Interface mit genau einer abstrakten Methode. Ein solches wird auch **SAM-Typ** genannt, wobei SAM für Single Abstract Method steht. Solcherlei Interfaces finden sich vielfach im JDK, besaßen aber früher keinen Namen. Bekannte Vertreter der SAM-Typen und Functional Interfaces sind etwa `Runnable`, `Callable`, `Comparator`, `FileFilter`, `FilenameFilter`, `ActionListener`, `EventHandler` usw.

```
@FunctionalInterface
public interface Runnable
{
    public abstract void run();
}

@FunctionalInterface
public interface Comparator<T>
{
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```

Um ein Interface explizit als Functional Interface zu kennzeichnen, wurde die im obigen Listing gezeigte Annotation `@FunctionalInterface` neu in Java 8 eingeführt – deren Angabe ist jedoch optional, wichtig für ein Functional Interface ist nur, dass im Sourcecode lediglich eine abstrakte Methode definiert ist. Das wird vom Compiler sichergestellt, wenn die Annotation `@FunctionalInterface` genutzt wird. Außerdem stellt jeder SAM-Typ, also ein Interface mit nur einer abstrakten Methode, auch ohne explizite Kennzeichnung ein Functional Interface dar.

Tipp: Besondere Methoden in Functional Interfaces

Wenn wir im obigen Listing genauer hinsehen, könnten wir uns fragen, wieso denn `Comparator<T>` ein Functional Interface ist, wo es doch zwei Methoden enthält und keine davon abstrakt ist, oder?

Als Besonderheit gilt in Functional Interfaces folgende Ausnahme für die Definition von abstrakten Methoden: Alle im Typ `Object` definierten Methoden können zusätzlich zu der abstrakten Methode in einem Functional Interface angegeben werden.

Verbleibt noch die Frage, warum wir in der Definition des `Comparator<T>`s keine abstrakte Methode sehen. Mit ein wenig Java-Basiswissen oder nach einem Blick in die Java Language Specification (JLS) erinnern wir uns daran, dass alle Methoden in Interfaces automatisch `public` und `abstract` sind, auch wenn dies nicht explizit über Schlüsselwörter angegeben ist.

Implementierung von Functional Interfaces

Herkömmlicherweise wird ein SAM-Typ bzw. Functional Interface durch eine anonyme innere Klasse implementiert. Seit JDK 8 kann man alternativ zu dessen Implementierung auch Lambdas nutzen. Voraussetzung dafür ist, dass das Lambda die abstrakte Methode des Functional Interface »erfüllen« kann, d. h. dass die Anzahl der Parameter übereinstimmt sowie deren Typen und der Rückgabotyp kompatibel sind. Schauen wir

zur Verdeutlichung zunächst auf ein allgemeines, etwas abstraktes Modell zur Transformation von bisherigen Realisierungen eines SAM-Typ mithilfe einer anonymen inneren Klasse in einen Lambda-Ausdruck:

```
// SAM-Typ als anonyme innere Klasse
new SAMTypeAnonymousClass()
{
    public void samMethod(METHOD-PARAMETERS)
    {
        METHOD-BODY
    }
}

// Kurzschreibweise mit Lambda
(METHOD-PARAMETERS) -> { METHOD-BODY }
```

Bei kurzen Methodenimplementierungen wie sie für SAM-Typen häufig vorkommen, ist das Verhältnis von Nutzcode zu Boiler-Plate-Code (oder Noise) recht schlecht. Wenn man für derartige Realisierungen Lambdas einsetzt, so kann man mit einer Zeile das ausdrücken, was sonst fünf Zeilen benötigt. Nachfolgend wird dies für das Interface `Runnable` verdeutlicht.

Besipiel 1: Runnable Konkretisieren wir die allgemeine Transformation anhand eines `Runnable` mit einer trivialen Implementierung einer Konsolenausgabe:

```
Runnable runnableAsNormalMethod = new Runnable()
{
    @Override
    public void run()
    {
        System.out.println("runnable as normal method");
    }
}
```

Als Lambda schreibt man dies deutlich kürzer wie folgt:

```
Runnable runnableAsLambda = () -> System.out.println("runnable as lambda");
```

Natürlich ist das keine wirklich sinnvolle Funktionalität in diesem `Runnable`, sondern dient mehr der ersten Verdeutlichung.

Besipiel 2: Comparator<T> Für das Functional Interface `Comparator<T>` lassen sich prägnantere Beispiele konstruieren. Ein Komparator dient bekanntlich dazu, zwei Instanzen vom Typ `T` miteinander zu vergleichen, wozu die abstrakte Methode `int compare(T, T)` passend zu realisieren ist. Wollte man zwei Strings nach deren Länge sortieren, so entsteht herkömmlicherweise recht viel Sourcecode:

```

Comparator<String> compareByLength = new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        final int length1 = str1.length();
        final int length2 = str2.length();

        if (length1 < length2)
            return 1;
        if (length1 > length2)
            return -1;

        return 0;
    }
}

```

Mit JDK 7 wurde die Klasse `Integer`, um eine Methode `compare(int, int)` erweitert, die einen Komparator-konformen Rückgabewert (vgl. nachfolgenden Praxistipp) liefert und so die Implementierung deutlich vereinfacht und verkürzt:

```

Comparator<String> compareByLength = new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        return Integer.compare(str1.length(), str2.length());
    }
}

```

Wenn man Lambdas nutzt, lässt sich der Komparator knackig wie folgt schreiben:

```

Comparator<String> compareByLength = (final String str1, final String str2) ->
{
    Integer.compare(str1.length(), str2.length());
}

```

Hinweis: Realisierung von Komparatoren und Rückgabewerte

Der Rückgabewert der `compare(T, T)`-Methode bestimmt den Ausgang des Vergleichs der beiden Objekte: Ein Wert > 0 besagt, dass das erste Objekt größer ist, 0 beschreibt Gleichheit und < 0 signalisiert, dass das erste Objekt kleiner als das zweite ist. Einen `Comparator<T>` zu implementieren, ist nicht besonders schwierig, erfordert aber häufig einige Fallunterscheidungen und wird dadurch recht schnell unübersichtlich.

Im obigen Beispiel sehen wir zunächst eine `if`-Anweisungen. Es scheint einfacher, die beiden Werte voneinander zu subtrahieren. Teilweise sieht man solche Lösungen. Allerdings besteht dabei die Gefahr, dass es zu einem Überlauf des Wertebereichs des `int` kommt und damit zu Berechnungsfehlern.

Type Inference und Besonderheiten der Syntax

Beim Einsatz von Lambdas möchte man den Sourcecode recht knapp formulieren können. Dazu existieren verschiedene Besonderheiten der Syntax und die sogenannte **Type Inference** oder **Typableitung**. Ähnlich wie beim Diamond Operator bei der Definition von generischen Klassen ist es für Lambdas möglich, auf die Typangaben für die Parameter im Sourcecode verzichten. Durch die Type Inference ermittelt der Compiler die passenden Typen aus dem Einsatzkontext. Den obigen Komparator schreibt man ohne Typangabe im Lambda wie folgt:

```
Comparator<String> compareByLength = (str1, str2) ->
    Integer.compare(str1.length(), str2.length());
```

Eine weitere Verkürzung in der Schreibweise eines Lambdas kann man durch folgende Regeln erzielen: Falls das auszuführende Stück Sourcecode nur ein Ausdruck ist, können auch die geschweiften Klammern um die Anweisungen entfallen. Existiert nur ein Eingabeparameter, so sind auch die runden Klammern um den Parameter optional. Damit ergibt sich für die Ausdrücke

```
(int x, int y) -> { return x + y; }
(long x) -> { return x * 2; }
```

folgende Kurzschreibweise:

```
(x, y) -> x + y
x -> x * 2
```

Neben dem offensichtlichen Vorteil, eine recht kompakte Schreibweise zu ermöglichen, ist etwas anderes viel entscheidender: Lambdas können flexibler als streng typisierte Methoden genutzt werden: Für die gezeigten Berechnungen ist ein Einsatz überall dort möglich, wo für die Parameter die Operatoren + bzw. * definiert sind, also etwa für die Typen int, float, double usw. Anders formuliert: **Alles, was hergeleitet werden kann, darf in der Syntax weggelassen werden**. Zur Verdeutlichung betrachten wir als Beispiel folgende ActionListener-Implementierung:

```
// Alter Stil
button.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(final ActionEvent e)
    {
        System.out.println("button clicked (old way)");
    }
});
```

Diese lässt sich als Lambda und mit Type Inference deutlich kürzer schreiben:

```
// Lambda-Variante mit Type Inference
button.addActionListener( (e) -> { System.out.println("button clicked!"); } );
```

Nutzt man die obigen Regel zur Schreibweisenabkürzung entsteht Folgendes:

```
// Lambda-Kurzschreibweise
button.addActionListener( e -> System.out.println("button clicked!") );
```

Besonderheiten: this, Zugriff auf Variablen und Erweiterungen

Zwar kann man Lambdas prinzipiell überall dort einsetzen, wo man bisher eine anonyme innere Klasse genutzt hat. Allerdings gibt es dabei doch ein paar kleine Unterschiede u. a. die Bedeutung von `this` sowie den Zugriff auf Variablen, die außerhalb des Lambdas bzw. der anonymen inneren Klasse definiert sind. Schauen wir zur Verdeutlichung auf ein kleines Beispielprogramm:

```
public class LambdaVsInnerClassExample
{
    private String outerAttribute = "fromOutside";

    public static void main(final String[] args)
    {
        new LambdaVsInnerClassExample().executeMethodAndLambda();
    }

    private void executeMethodAndLambda()
    {
        int effectivelyFinal = 4711;

        Runnable asNormalMethod = new Runnable()
        {
            @Override
            public void run()
            {
                System.out.println(this);
                // Nicht finale Variable war bis JDK 7 nicht referenzierbar
                System.out.println("effectivelyFinal = " + effectivelyFinal);
                // Spezielle Syntax zum Zugriff auf Attribute der äußeren Klasse
                System.out.println("outerAttribute = " +
                                   LambdaVsInnerClassExample.this.outerAttribute);
            }
        };

        public String anotherMethod()
        {
            return "Anonymous Runnable";
        }
    };

    // Man kann keine weiteren Methoden in Lambdas definieren
    Runnable asLambda = () ->
    {
        System.out.println(this);
        // Nicht finale Variable war bis JDK 7 nicht referenzierbar
        System.out.println("effectivelyFinal = " + effectivelyFinal);
        System.out.println("outerAttribute = " + outerAttribute);
    };

    asNormalMethod.run();
    asLambda.run();
}
```


Führen wir das Programm `LAMBDAVSINNERCLASSEXAMPLE` aus, so kommt es zu folgenden Ausgaben:

```
jdk8.LambdaVsInnerClassExample$1@4617c264    // $1 => Innere Klasse
effectivelyFinal = 4711
outerAttribute = fromOutside

jdk8.LambdaVsInnerClassExample@36baf30c
effectivelyFinal = 4711
outerAttribute = fromOutside
```

Zwar sieht man bei der Ausgabe nur einen kleinen Unterschied, jedoch zeigt der Sourcecode durchaus Unterschiede und Besonderheiten, die oben fett hervorgehoben sind und auf die ich nun explizit eingehe.

Bedeutung von `this` Lambdas repräsentieren lediglich ein Stück Funktionalität, ist die Bedeutung von `this` innerhalb eines Lambdas identisch mit der direkt außerhalb davon. Für innere Klassen referenziert `this` die innere Klasse selbst. Das hat insbesondere Einfluss darauf, wie man auf das Attribut `outerAttribute` der äußeren Klasse zugreifen kann. Dazu schreibt man für die innere Klasse etwas umständlich `LambdaVsInnerClassExample.class.outerAttribute`. Für den Lambda nutzt man nur den Variablennamen, also hier `outerAttribute`.

Zugriff auf Variablen Kommen wir zum Zugriff auf Variablen, die außerhalb des Lambdas bzw. der anonymen inneren Klasse definiert sind. Bis JDK 7 konnte man auf derartige Variablen nur dann zugreifen, wenn diese explizit `final` definiert waren. Mit JDK 8 wurde das Ganze etwas gelockert: Es reicht nun sowohl für Lambdas als auch für anonyme innere Klassen, wenn die Variablen »effectively« `final` sind. Darunter versteht man, dass die Variablen nicht mehr explizit `final` deklariert werden müssen, sondern es reicht, wenn diese ihren Wert zur Programmlaufzeit nicht ändern. Dieser Sachverhalt wird vom Compiler geprüft und Verstöße werden als Fehler angemahnt.

Erweiterungen des SAM-Typs Innerhalb von anonymen inneren Klassen kann man beliebige weitere Methoden definieren, wie dies im Listing mit der Methode `anotherMethod()` gezeigt ist. Für Lambdas ist das nicht möglich. Sie können zwar stellvertretend für SAM-Typen genutzt werden, erlauben aber keine Erweiterung um Methoden, da sie eher anonymen Methoden als anonymen Klassen entsprechen.

Lambdas als Methodenparameter und als Rückgabewerte

Wir haben mittlerweile ein wenig Gespür für Lambdas gewonnen und schauen nun, wie man damit Aufrufe lesbarer gestalten kann. Dazu greifen wir das Beispiel aus der Einleitung wieder auf und betrachten die Sortierung einer Liste von Namen per `Comparator<T>`. Hierfür können wir mit JDK 8 folgende Varianten mit Lambdas schreiben, wobei der Lambda einmal in Form eines Methodenparameters und einmal als Rückgabewert genutzt wird:

```

public class LambdaExample
{
    public static void main(final String[] args)
    {
        final List<String> names = Arrays.asList("Andy", "Michael",
                                                "Max", "Stefan");

        // Lambda als Methodenparameter
        Collections.sort(names, (str1, str2) -> Integer.compare(str1.length(),
                                                                str2.length()));

        // Alternative mit Lambda als Rückgabe einer Methode
        names.sort(stringLengthCompare());
    }

    public static Comparator<String> stringLengthCompare()
    {
        return (str1, str2) -> Integer.compare(str1.length(), str2.length());
    }
}

```

Wenn Sie im Listing ganz genau hingeschaut haben, dann könnte Ihnen aufgefallen sein, dass wir bei der zweiten Variante gar nicht `Collections.sort()` aufrufen wird, sondern `names.sort()`, also direkt auf einer Instanz von `List<String>`. Wie geht das denn? Diese Methode existiert doch gar nicht im Interface `java.util.List<T>`, oder? Das ist bis JDK 7 korrekt. Doch mit JDK 8 wurde das Interface `List<T>` sowie viele weitere Interfaces mithilfe des neuen Sprachkonstrukts Default-Methoden angepasst.

2.1.2 Default-Methoden

Beim Entwurf von Lambdas und deren Integration in das JDK stellte sich heraus, dass für eine sinnvolle Nutzbarkeit auch die bestehenden Klassen und Interfaces erweitert werden mussten. Bis zur Einführung von JDK 8 war es allerdings nicht möglich, ein Interface nach seiner Veröffentlichung zu verändern, ohne dass dies Auswirkungen bei allen einsetzenden Klassen gehabt hätte. Vielmehr führt die Erweiterung eines Interface bis inklusive JDK 7 immer zu einem Kompatibilitätsproblem: Wenn eine Methode neu in ein Interface hinzugefügt wurde, musste diese in allen Klassen realisiert werden, die das Interface implementieren. Ansonsten kompilierten einige Klasse solange nicht mehr, bis die Implementierung im Nachhinein bereitgestellt wurde.¹ Dieses Dilemma war von den Oracle-Entwicklern zu lösen.

Interface-Erweiterungen

Mit JDK 8 kann man Inkompatibilitätsprobleme bei Interface-Erweiterungen vermeiden, indem man im Soucecode des Interfaces eine sogenannte Default-Implementierung

¹Für Binaries kam es dann zu einem `AbstractMethodError`, der ausdrückt, dass in der `.class`-Datei eine Methode nicht bereitgestellt wird.

vorgibt. Dazu nutzt man das neue Sprach-Feature der *Default-Methoden*. Das sind *spezielle Implementierungen von Methoden, die in Interfaces definiert werden können*. Um sie von den normalen abstrakten Methoden in Interfaces zu unterscheiden, werden Default-Methoden mit dem Schlüsselwort `default` eingeleitet. Die Default-Methoden sind eine wichtige Neuerung, um Lambdas für bestehende Funktionalitäten des JDK gewinnbringend nutzen zu können.

Die Default-Methoden `sort()` und `forEach()`

Schauen wir auf zwei Erweiterungen. Zum einen eine, die den bereits zuvor genutzten Aufruf von `sort()` direkt auf der Instanz einer `List<E>` möglich macht und zum anderen eine, die ein Bearbeiten aller Elemente einer Collection ermöglicht.

Beginnen wir mit der Erweiterung im Interface `List<E>`. Dort findet sich nun die Definition von `sort()` wie folgt:

```
public interface List<E> extends Collection<E>
{
    // ...
    default void sort(Comparator<? super E> c)
    {
        Collections.sort(this, c);
    }
    // ...
}
```

Das Sortieren ist zwar praktisch, aber eine recht spezielle Funktionalität. Deutlich gebräuchlicher und allgemeiner sind Iterationen über die Elemente einer Collection. Dazu steht nun die Methode `forEach(Consumer<? super T>)` im Interface `java.lang.Iterable<T>` bereit, was die Basis von `java.util.Collection<T>` und `List<T>` ist. Die Default-Methode `forEach(Consumer<? super T>)` ist dort folgendermaßen implementiert:

```
public interface Iterable<T>
{
    // ...
    default void forEach(Consumer<? super T> action)
    {
        Objects.requireNonNull(action);
        for (T t : this)
        {
            action.accept(t);
        }
    }
    // ...
}
```

Im Listing sehen wir das Functional Interface `Consumer<T>`. Zur Ergänzung und Flexibilisierung des JDKs wurde eine Vielzahl solcher Interfaces in JDK 8 integriert (vgl. folgenden Praxistipp). Bei der täglichen Arbeit wird man meistens Lambdas nutzen. Das wollen wir uns nun anschauen und kommen dazu auf das einleitende Beispiel zurück.

Hintergrundwissen: Functional Interfaces

Zum Einsatz von Lambdas musste das JDK um diverse Functional Interfaces (also SAM-Typen) erweitert werden. Im Package `java.util.function` finden sich um die 40 Functional Interfaces, oftmals mit sprechendem Namen, unter anderem folgende, die wir im Verlauf unserer Entdeckungsreise durch JDK 8 näher betrachten werden:

- `Consumer<T>` – beschreibt eine Aktion auf einem Element vom Typ `T`.
- `Predicate<T>` – erhält eine Eingabe vom Typ `T` und berechnet einen booleschen Rückgabewert (z. B. `olderThan()`).
- `Function<T, R>` – definiert eine Funktion mit dem Eingabetyp `T` und der Rückgabe vom Typ `R`.

Lambdas und Default-Methoden im Einsatz

Im einleitenden Beispiel zur Motivation von Lambdas habe ich gezeigt, wie man eine Liste von Namen ihrer Länge nach sortiert und die Längen dann kommasepariert ausgibt. Dazu waren übersichtlich formatiert fast 15 Zeilen Sourcecode nötig – bei platzsparender Anordnung immer noch über 10.

Wir werden nun zwei Varianten zur Implementierung des Functional Interface `Consumer<T>` kennenlernen, um weiter mit Lambdas vertraut zu werden. Das Functional Interface `Consumer<T>` deklariert die abstrakte Methode `accept(T)` und ermöglicht die Hintereinanderausführung mehrerer `Consumer<T>`-Instanzen mithilfe der Default-Methode `andThen()`:

```
@FunctionalInterface
public interface Consumer<T>
{
    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after)
    {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

Zunächst schauen wir auf eine Realisierung mithilfe eines Lambda, der eine Konsolenausgabe vornimmt:

```
public class ConsumerAndLambdaExample
{
    public static void main(final String[] args)
    {
        final List<String> names = Arrays.asList("Andy", "Michael",
                                                "Max", "Stefan");

        names.sort(stringLengthCompare());
    }
}
```

```

        names.forEach(it -> System.out.print(it.length() + ", "));
    }

    public static Comparator<String> stringLengthCompare()
    {
        return (str1, str2) -> Integer.compare(str1.length(), str2.length());
    }
}

```

Vielleicht fragen Sie sich, wofür `it` steht. Gemäss der Transformation bzw. Ersetzung von Parametern und Anweisungen zwischen anonymer innerer Klasse und Lambda entspricht der im Lambda genutzte Parameter `it`, dem Parameter der abstrakten Methode `accept(T)` im Functional Interface `Consumer<T>`. `it` ist eine gebräuchliche Abkürzung für Parameter beliebigen Typs bei Iterationen. In diesem Fall wäre `name` eine besser lesbare Alternativen gewesen.

Standardverhalten vorgeben

Neben der Erweiterung eines Interfaces besteht ein weiterer Anwendungsfall von Default-Methoden darin, für bereits existierende Methoden ein Standardverhalten vorgeben zu können, das für die meisten Implementierer adäquat ist und somit vermeidet, dass diese Methode ständig gleich implementiert wird.

Das war beispielsweise für eigene Realisierungen des Interface `Iterator<E>` und dessen Methode `remove()` sehr häufig der Fall. Fast immer wurde diese so implementiert, dass dort eine `UnsupportedOperationException` ausgelöst wird. Nun ist dieses Standardverhalten im JDK durch folgende Default-Methode umgesetzt, wodurch sich eigene Spezialisierungen von `Iterator<E>` auf die Implementierung der Iteration fokussieren können.

```

public interface Iterator<E>
{
    boolean hasNext();
    E next();

    default void remove()
    {
        throw new UnsupportedOperationException("remove");
    }

    // ...
}

```

Erweiterte Möglichkeiten durch Default-Methoden

Wir haben bereits gesehen, dass man mithilfe von Default-Methoden ein gewünschtes Standardverhalten für Subtypen des Interfaces vorgeben kann. Spezialisierungen können selbstverständlich eigene Realisierungen für Default-Methoden vornehmen. Das kann in Form komplett eigener Realisierungen geschehen oder aber indem eine eigene Implementierung einer Default-Methode bereitgestellt wird. Auf diese Weise kann

man in einem Basisinterface eine allgemein gültige Realisierung vorgeben und in einem Subtyp eine Spezialbehandlung durchführen. Das mag noch etwas merkwürdig klingen, wird aber sofort klar, wenn wir auf ein konkretes Beispiel schauen, nämlich auf das Sortieren von Listen. Im Collections-Framework gibt es dazu im Interface `List<E>` die bereits vorgestellte Default-Methode `sort(Comparator<? super E>)`. Darüber hinaus wird in der Klasse `ArrayList<E>` die Definition einer Default-Methode genutzt, um eine performantere Sortierung zu realisieren.

Beispiel: Sortieren von Listen Die im Interface `List<E>` zum Sortieren definierte Default-Methode nutzt die Utility-Klasse `Collections`, in der das Sortieren von Listen wie folgt realisiert ist:

```
// Collections
public static <T> sort(List<T> list, Comparator<? super T> c)
{
    // Schritt 1: Liste in ein temporäres Array übertragen
    Object[] a = list.toArray();
    // Schritt 2: Array sortieren
    Arrays.sort(a, (Comparator)c);
    // Schritt 3: Array zurück in die Liste übertragen
    ListIterator<T> i = list.listIterator();
    for (int j=0; j<a.length; j++)
    {
        i.next();
        i.set((T)a[j]);
    }
}
```

Anhand des Listings erkennen wir, dass im Wesentlichen drei Schritte ausgeführt werden: Zunächst wird die Liste in ein temporäres Array kopiert, dann wird dieses sortiert und anschließend wieder zurück in die Liste übertragen. Es sind also mehrere Kopier- bzw. Einfügeoperationen in und aus einem Array zusätzlich zur eigentlichen Sortierung erforderlich. Diese Schritte verbrauchen Laufzeit und auch zusätzlichen Speicher. Beides wirkt sich bei zunehmender Anzahl an gespeicherten Elementen negativ aus. Einen solchen Preis muss man oftmals zahlen, wenn man eine allgemein gültige Lösung realisiert. Der Grund dafür ist, dass diese Art von Lösungen verschiedene Einschränkungen besitzt und nicht von gewissen Eigenschaften der Spezialisierungen profitieren kann. Spezialisierungen besitzen logischerweise mehr Informationen zu ihren Implementierungsdetails als ihre Basisklassen oder -interfaces. Betrachten wird dies für die `ArrayList<E>` als Spezialisierung des Interfaces `List<E>`.

Realisierung einer Spezialbehandlung Für die `ArrayList<E>` kann man die zuvor gezeigte allgemein gültige Sortierimplementierung für beliebige `List<E>` performanter realisieren, indem man ausnutzt, dass die `ArrayList<E>` ihre Daten bereits in Form eines Arrays hält und direkt auf der internen Datenstruktur sortiert wird. Dadurch können der Umwandschritt in ein temporäres Array und die Rückkonvertierung entfallen:

```
// ArrayList<E>
public void sort(Comparator<? super E> c)
{
    final int expectedModCount = modCount;
    Arrays.sort((E[]) elementData, 0, size, c);
    if (modCount != expectedModCount)
    {
        throw new ConcurrentModificationException();
    }
    modCount++;
}
```

Hierbei sehen wir mit der Variablen `modCount` ein Implementierungsdetail der Fail-Fast-Realisierungen aus dem Collections-Framework. Mithilfe dieses Zählers, der bei jeder Modifikation erhöht wird, können potenzielle Veränderungen erkannt und so mögliche Fehler durch nebenläufige Veränderungen entdeckt werden.

Spezialfall: Was passiert bei Konflikten?

Wenn nun die Definition von Default-Methoden möglich ist, sollte man sich auch über potenzielle Konflikte Gedanken machen. Es ist durchaus üblich, dass eine Klasse zwei oder mehr Interfaces implementiert. Nehmen wir nun an, beide Interfaces würden um eine gleichnamige Default-Methode `sameMethod(int)` wie folgt erweitert:

```
interface Interface1
{
    default int sameMethod(int x)
    {
        return 0;
    }
}

interface Interface2
{
    default int sameMethod(int x)
    {
        return 4711;
    }
}

class ErroneousCombination implements Interface1, Interface2
{
}
```

Dadurch entsteht ein Konflikt: Der Compiler kann die zu nutzende Methode nicht mehr selbstständig wählen. Das führt zu dem Kompilierfehler »Duplicate default methods named `sameMethod` with the parameters `(int)` and `(int)` are inherited from the types `Interface2` and `Interface1`«. Um diesen zu beheben, muss vom Entwickler steuernd eingegriffen werden. Zwei mögliche Abhilfen stelle ich nun kurz vor.

Lösung 1 Eine Möglichkeit, um einen solchen Konflikt zu lösen, besteht darin, eine eigene Methodenimplementierung vorzugeben:

```
public class Correction1 implements Interface1, Interface2
{
    public int sameMethod(int x)
    {
        return 7;
    }
}
```

Lösung 2 Neben einer eigenen Realisierung kann alternativ auch ein Aufruf der Funktionalität einer der Default-Methoden gewünscht sein. Das kann mit folgender spezieller Syntax mit Angabe des Namens in Kombination mit `super` erfolgen:

```
public class Correction2 implements Interface1, Interface2
{
    public int sameMethod(int x)
    {
        return Interface1.super.sameMethod(x);
    }
}
```

Vorteile und Gefahren von Default-Methoden

Default-Methoden rufen bei mir gemischte Gefühle hervor, weshalb ich nochmals kurz deren Stärken und mögliche Fallstricke rekapitulieren möchte. Default-Methoden ...

- ermöglichen **API-Erweiterungen** unter Beibehaltung von **Rückwärtskompatibilität**
- erlauben es, ein gewünschtes **Standardverhalten vorzugeben**
- kann man überschreiben und dadurch ein gewünschtes Verhalten oder eine **Spezialbehandlung** realisieren.

Durch diese Eigenschaften stellen Default-Methoden zweifellos eine wichtige Erweiterung von Java dar, ohne die eine Integration der neuen Funktionalitäten von Lambdas in das Collections-Framework und in andere Teile der JDK-Bibliotheken nur unzureichend möglich gewesen wäre. In dieser Hinsicht sind Default-Methoden für API-Designer und Framework-Entwickler ein geeignetes Konstrukt, um eine sanfte, evolutionäre Entwicklung von APIs vornehmen zu können. Dies hat die Integration der neuen Funktionalitäten in das JDK 8 bereits gezeigt.

Andererseits öffnen Default-Methoden Tür und Tor für eine laxere Programmierung mit zu wenig Fokus auf sauberes Design, da sich ja nachträglich immer noch Funktionalität in das System „hineinprökeln“ lässt. Außerdem unterstützt Java durch Default-Methoden nun auch Mehrfachvererbung: allerdings umfasst diese lediglich Verhalten und nicht Zustand. Letzteres hatte James Gosling aufgrund der damit möglichen und aus C++ bekannten Designprobleme bewusst aus Java herausgehalten.

Fallstrick mit JDK 8: Default-Methoden und Zugriff auf Attribute

Spontan könnte man auf die Idee kommen, das neue Feature der Default-Methoden dazu einzusetzen, um in einem Interface bereits Attribute und Zugriffsmethoden darauf bereitzustellen, etwa in Form folgender Implementierung:

```
public interface MisleadingDefaultMethods
{
    String name = "<Name>";    // public static final

    default void setName(final String newName)
    {
        this.name = newName;
    }

    // get analog
}
```

Zunächst sieht die Implementierung in Ordnung aus, jedoch kompiliert sie nicht. Entgegen der reinen Notation im Sourcecode, die man für die Definition eines Instanz-Attributs halten könnte, sind alle Attribute in Interfaces gemäss JLS implizit immer `public`, `static` und `final` und somit Klassenattribute. Ein Zugriff per `this.name` ist damit nicht möglich und löst Kompilierfehler aus.

Statische Methoden in Interfaces

Interfaces können mit Java 8 nicht nur Default-Methoden, sondern auch statische Methoden enthalten. Damit wird es möglich, Hilfsmethoden direkt in Interfaces bereitzustellen und dafür keine separaten Utility-Klassen mehr anbieten zu müssen. Diese Konstellation kennt man zum einen aus dem JDK und zum anderen wohl auch aus eigenen Projekten. Im JDK findet man etwa die Klasse `Paths` als Utility-Klasse zum Interface `java.nio.file.Path` im Package `java.nio.file` oder die Kombination von der Utility-Klasse `Executors` und dem Interface `Executor` aus dem Package `java.util.concurrent`.

Im JDK 8 finden sich viele Beispiele, in denen man die Hilfsmethoden statt in einer eigenen Utility-Klasse direkt im Interface selbst implementiert hat. Dies gilt etwa für das Interface `Comparator<T>`. Nachfolgendes Listing zeigt nur auszugsweise einige statische Erzeugungsmethoden für spezielle Komparatoren, hier für eine umgedrehte Sortierung, die natürliche Ordnung sowie Sortierungen, die `null`-Werte vorne bzw. hinten einsortieren:

```
@FunctionalInterface
public interface Comparator<T>
{
    // ...

    public static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
    {
        return Collections.reverseOrder();
    }
}
```

```

public static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
{
    return (Comparator<T>) Comparators.NaturalOrderComparator.INSTANCE;
}

public static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)
{
    return new Comparators.NullComparator<>(true, comparator);
}

public static <T> Comparator<T> nullsLast(Comparator<? super T> comparator)
{
    return new Comparators.NullComparator<>(false, comparator);
}

// ...
}

```

Zwar lassen sich so auf einfache Art gewisse Funktionalitäten bereitstellen, allerdings **sollte man sich aber bewusst sein, dass man damit das Konzept des Interfaces zur Definition einer Schnittstelle immer mehr verwässert**. Aber nicht nur das! In diesem Beispiel erkennen wir, dass die eigentliche Schnittstellenbeschreibung nun Abhängigkeiten von diversen speziellen Klassen und Implementierungsdetails besitzt. Für diese Realisierung im JDK ist das wohl nicht so kritisch. **Für eigene Interfaces sollte man aber sehr genau überlegen, ob man dort statische Methode anbieten möchte und welche möglichen Auswirkungen und Abhängigkeiten sich dadurch ergeben**. Deklariert man dagegen lediglich Methoden in Interfaces, nimmt also eine reine Schnittstellenbeschreibung vor, so lassen sich diese Interfaces meistens viel leichter auch in andere Projekte integrieren, ohne eine (unerwartete) Menge von weiteren Abhängigkeiten auszulösen.

2.1.3 Methodenreferenzen

Wir haben bisher gesehen, wie sich Lambdas gewinnbringend einsetzen lassen. Darüber hinaus kann auch der Einsatz der mit JDK 8 eingeführten Methodenreferenzen dazu beitragen, die Lesbarkeit des Sourcecodes zu erhöhen. Das Sprach-Feature der Methodenreferenzen besitzt die Syntax: `Klasse::Methodenname` und verweist auf ...

- eine Methode – `System.out::println`, `Person::getName`, ...
- einen Konstruktor – `ArrayList::new`, `Person[]::new`, ...

Das wirkt recht unspektakulär. Eine Methodenreferenz kann aber zur Vereinfachung der Schreibweise anstelle eines Lambdas genutzt werden:

```

public static void main(final String[] args)
{
    final List<String> names = Arrays.asList("Max", "Andy", "Michael", "Stefan");

    // Lambda
    names.forEach( it -> System.out.println(it) );
}

```

```
// Methodenreferenz
names.forEach( System.out::println );
}
```

Wie man sieht, verbessert sich die Lesbarkeit. Nachfolgend möchte ich an ein paar Beispielen zeigen, wie sich Methodenreferenzen auf Lambdas bzw. andersherum abbilden lassen. Dabei gibt es vier verschiedene Varianten, die in Tabelle ?? dargestellt sind.

Tabelle 2-1 Methodenreferenzen

Referenz auf ...	Als Methodenreferenz	Als Lambda
statische Methode	<code>String::valueOf</code>	<code>obj -> String.valueOf(obj)</code>
Instanzmethode eines Typs	<code>Object::toString</code> <code>String::compareTo</code>	<code>obj -> obj.toString()</code> <code>(str1, str2) -> str1.compareTo(str2)</code>
Instanzmethode eines Objekts	<code>person::getName()</code>	<code>() -> person.getName()</code>
Konstruktor	<code>ArrayList::new</code>	<code>() -> new ArrayList<>()</code>

Spezialfall

Wir haben eben gesehen, wie sich Methodenreferenzen und Lambdas ineinander überführen lassen. Dabei gilt: Ein Lambda ist immer dann durch eine Methodenreferenz ersetzbar, wenn neben dem Methodenaufuf keine weiteren Aktionen in dem Lambda erfolgen. Des Öfteren möchte man aber möglicherweise noch einige weitere Anweisungen ausführen, beispielsweise eine Konkatenation eines Kommas für eine komma-separierte Ausgabe. Dazu muss man einen Lambda nutzen:

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

    final List<String> names = Arrays.asList("Max", "Andy", "Michael", "Stefan");

    // names.sort((str1, str2) -> Integer.compare(str1.length(), str2.length()));
    names.sort(LambdaReturnExample::stringLengthCompare);

    // Methodenreferenz nachfolgend nicht direkt nutzbar
    names.forEach(it -> System.out.print(it.length() + ", "));
}
```

Natürlich kann man auch diesen in eine Methodenreferenz überführen: Hier definieren wir einfach eine eigene Methode `printCommaSeparatedLength(String)` und rufen diese wie folgt auf:

```
names.forEach(OwnMethodReferenceExample::printCommaSeparatedLength);
```

2.1.4 Erweiterungen im Interface `Comparator<T>`

Die Behandlung von Lambdas, Default-Methoden und Methodenreferenzen möchte ich mit einer Betrachtung der Erweiterungen im Interface `Comparator<T>` beschließen, da wird dort viele Beispiele für den sinnvollen Einsatz der JDK 8-Neuerungen finden.

Hinreichend bekannt ist, dass bei der Realisierung eines `Comparator<T>`s in Form einer Klasse einiges an Boilerplate-Code entsteht und man als Abhilfe für einfache Vergleiche sinnvollerweise Lambdas nutzen kann. Vielfach sind aber komplexere Vergleiche zu implementieren. Zur Verdeutlichung gehen wir von einer Liste von Personen aus. Naheliegend sind die Sortierungen nach Vor- bzw. Nachnamen. Diese kann man durch einen `Comparator<Person>`, dessen `compareTo(Person, Person)`-Methode aus den zwei übergebenen `Person`-Objekten den jeweiligen Namensbestandteile extrahiert und diese dann vergleicht. Dieses Prinzip ist für Komparatoren, die auf einer Vergleichbarkeit von Werten von Typ `T` und `java.lang.Comparable<T>` basieren, eigentlich immer wieder das Gleiche und der konkrete Ablauf variiert lediglich in der Extraktion der jeweiligen Werte. Im Listing ist eine typische Implementierung zum besseren Verständnis der nachfolgenden Erläuterungen gezeigt:

```
// Hinweis: Diamond Operator ist nicht für anonyme innere Klassen möglich
Comparator<Person> compareByName = new Comparator<Person>()
{
    @Override
    public int compare(final Person person1, final Person person2)
    {
        // Spezifische Extraktion
        final String value1 = person1.getName();
        final String value2 = person2.getName();

        // Vergleich basierend auf Comparable<T>
        return value1.compareTo(value2);
    }
};
```

Praktischerweise wurde das Interface `Comparator<T>` mit JDK 8 um einige nützliche Methoden ergänzt. Die folgende Aufzählung nennt die neuen Möglichkeiten und bevor wir diese in Form kleiner Beispiele genauer betrachten:

- `comparing()` – definiert einen Komparator basierend auf der Extraktion zweier Werte, die sich mithilfe des Interface `Comparable<T>` vergleichen lassen.
- `comparingInt()/Long/Double()` – definiert einen `Comparator<T>`, wobei der `int/long/double`-Wert eines speziellen Attribut des Typs `T` zum Vergleich genutzt wird.
- `naturalOrder()`, `reverseOrder()`, `reversed()` – Diese Methoden erzeugen Komparatoren gemäss der natürlichen, der dazu entgegengesetzten sowie einer zu einem bestehende Komparator inversen Sortierung.
- `thenComparing(Comparator<? super T>)`, `thenComparingInt()/-Long` und `-Double` – Hiermit wird die Hintereinanderschaltung von Komparatoren möglich.

Auf Comparable basierende Komparatoren

Der beschriebene Ablauf beim Vergleich zweier Werte vom Typ `T` wird durch einen `Comparator<T>` implementiert, den man mithilfe der statischen Methode `Comparator.comparing()` erhält – hier sieht man etwa ein nützliches Beispiel für eine statische Methode in einem Interface.² Wir können damit einen Vergleich nach Namen folgendermaßen realisieren:

```
// Einfache Komparatoren basieren auf Comparable<T> formulieren
Comparator<Person> byNameLambda = (person1, person2) ->
    person1.getName().compareTo(person2.getName());

// Varianten mit Comparator.comparing
Comparator<Person> byName1 = Comparator.comparing(person -> person.getName());
Comparator<Person> byName2 = Comparator.comparing(Person::getName);
```

Hinweis: Internationalisierung und Umlaute

Der zuvor gezeigte Vergleich von Namen basiert auf `compareTo(String)` und vergleicht zwei Strings textuell ohne auf Umlaute und regionale Besonderheiten Rücksicht zu nehmen. Als Abhilfe lässt sich jedoch ein sogenannter Collator vom Typ `java.text.Collator`, eine spezielle Form von `Comparator`, der sprachspezifische Eigenheiten beim Vergleich beachtet:

```
final Collator collator = Collator.getInstance();
persons.sort(Comparator.comparing(Person::getName, collator));
```

Hintereinanderschaltung von Komparatoren

Modifizieren wir die Anforderungen ein wenig. Da Personen durchaus den gleichen Vornamen besitzen können, wäre ein zweites Sortierkriterium wünschenswert. Wir schreiben dies wie folgt:

```
// Komparatoren für ein spezielles Attribut
Comparator<Person> byFirstname = Comparator.comparing(Person::getFirstname);
Comparator<Person> byName = Comparator.comparing(Person::getName);
Comparator<Person> byAge = Comparator.comparing(Person::getAge);

// Kombination von Komparatoren
Comparator<Person> byFirstnameAndName = byFirstname.
    thenComparing(byName);

Comparator<Person> byNameAndAge = byName.thenComparing(byAge);

persons.sort(byFirstnameAndAge);
```

²Allerdings wäre es ebenso gut möglich gewesen, diese Funktionalität durch eine separate Utility-Klasse `Comparators` bereitzustellen

Im Listing ist gezeigt, dass sich Komparatoren nun einfach hintereinander ausführen lassen, indem man die Default-Methode `thenComparing()` aufruft. Bei genauerer Betrachtung fällt auf, dass für die Alterswerte ein Auto-Boxing aus einem `int` in ein `Integer`-Objekt erfolgt, welches sich wieder über `Comparable<T>` vergleichen lässt.

Verarbeitung primitiver Typen

In seltenen Fällen mit sehr großen Datenmengen und auf kritischen Pfaden kann sich der Auto-Boxing-Automatismus jedoch ungünstig auf die Performance auswirken. Als Abhilfe finden sich im Interface `Comparator<T>` für die primitiven Typen `int`, `long` und `double` spezialisierte Varianten. Dies gilt ebenfalls für diverse in Java 8 eingeführten, weiteren Funktionalitäten. Einen Vergleich des Alters basierend auf `int`-Werten würde man folgendermaßen schreiben:

```
final Comparator<Person> byAge = Comparator.comparingInt(Person::getAge);
```

Spezielle Ordnungen

Teilweise soll ein Sortierkriterium umgedreht werden. Dazu reicht nun ein Aufruf an `reversed()`, um etwa eine Liste absteigend nach Namen sortiert werden:

```
final Comparator<Person> byNameDescending = byName2.reversed();
```

Manchmal möchte man mit einem Komparator die natürliche Ordnung abbilden. Dazu dient die Methode `naturalOrder()`. Die entgegengesetzte Sortierung erhält man mit Methode `reverseOrder()`. Wenn man diese invertiert, erhält man wieder die natürliche Ordnung.

```
public static void main(final String[] args)
{
    final Integer[] primes = { 1, 7, 3, 13, 11, 5, 17, 19 };

    // aufsteigend
    final Comparator<Integer> naturalOrder = Comparator.naturalOrder();
    // absteigend
    final Comparator<Integer> reverseOrder = Comparator.reverseOrder();
    // aufsteigend
    final Comparator<Integer> naturalOrderAgain = reverseOrder.reversed();

    sortAndPrint("naturalOrder", primes, naturalOrder);
    sortAndPrint("reverseOrder", primes, reverseOrder);
    sortAndPrint("naturalOrderAgain", primes, naturalOrderAgain);
}

private static void sortAndPrint(final String name, final Integer[] primes,
                                final Comparator<Integer> sortOrder)
{
    Arrays.sort(primes, sortOrder);
    System.out.println(name + ": " + Arrays.toString(primes));
}
```

Wenig überraschend erhält man folgende Ausgaben:

```
naturalOrder      : [1, 3, 5, 7, 11, 13, 17, 19]
reverseOrder      : [19, 17, 13, 11, 7, 5, 3, 1]
naturalOrderAgain: [1, 3, 5, 7, 11, 13, 17, 19]
```

Behandlung von null-Werten

Ab und zu sieht man sich der Herausforderung gegenüber, dass die zu sortierenden Datensätze auch `null`-Werte enthalten. Die bisher genutzten Komparatoren lösen dann eine `NullPointerException` aus. Teilweise ist dies aber nicht gewünscht, sondern `null`-Werte sollen am Anfang oder am Ende einsortiert werden. Dazu gibt es die beiden Methoden `nullsFirst()` und `nullsLast()`, die wir nachfolgend nutzen, um eine mit `null`-Werten durchsetzte Liste mit Namen zu sortieren:

```
public static void main(final String[] args)
{
    final List<String> names = Arrays.asList("A", null, "B", "C", null, "D");

    // Null-sichere Komparatoren
    final Comparator<String> naturalOrder = Comparator.naturalOrder();
    final Comparator<String> nullsFirst = Comparator.nullsFirst(naturalOrder);
    final Comparator<String> nullsLast = Comparator.nullsLast(naturalOrder);

    names.sort(nullsFirst);
    System.out.println("nullsFirst: " + names);

    names.sort(nullsLast);
    System.out.println("nullLast: " + names);
}
```

Führt man das Programm aus, so erhält man folgende Ausgaben:

```
nullsFirst: [null, null, A, B, C, D]
nullsLast: [A, B, C, D, null, null]
```

Nicht immer arbeiten die Vergleiche direkt auf den Daten einer Liste. Vielfach bestimmt ein Attribut (oder mehrere) die Sortierung. Wenn man dort einen `null`-sichere Komparator definieren möchte, muss man etwas achtsam sein (vgl. folgenden Praxistipp). Für die Klasse `Person` und ein über die Methode `getFavoriteColor()` ermittelbares optionales Attribut müssen die Komparatoren wie folgt zusammengebaut werden (zwei Varianten):

```
Comparator<Person> byFavoriteColor = Comparator.comparing(
    Person::getFavoriteColor,
    Comparator.nullsFirst(String::compareTo));

Comparator<Person> byFavoriteColorV2 = Comparator.comparing(
    Person::getFavoriteColor, nullsFirst);
```

Hier kommt eine spezielle Variante der Methode `comparing()` zum Einsatz, der als erster Parameter ein sogenannter Key-Extractor übergeben wird. Dieser bestimmt, wie

das Attribut aus den zu sortierenden Instanzen, im Beispiel `Person`-Objekten, ermittelt wird. Der zweite Parameter ist dann ein Komparator, allerdings ein solcher, mit dem Typ des extrahierten Attributs, hier also `String`.

Hinweis: Die Methoden `nullsFirst()` und `nullsLast()`

Trotz ihres durchaus sprechenden Namens sind die Methoden `nullsFirst()` und `nullsLast()` nicht ganz so leicht in der Anwendung. Intuitiv könnte man versuchen, sie um eine bestehende Komparatorimplementierung zu umwickeln, etwa folgendermaßen zur Sortierung der optionalen Angabe einer Lieblingsfarbe, die mit `getFavoriteColor()` ermittelt wird und gegebenenfalls auch den Wert `null` als Rückgabe besitzt.

```
Function<Person, String> keyExtractor = Person::getFavoriteColor;
Comparator<Person> byFavoriteColor = Comparator.comparing(keyExtractor);

Comparator<Person> nullSafe_V1 = Comparator.nullsFirst(byFavoriteColor);
```

Diese Variante kompiliert zwar, führt aber später zur Laufzeit beim Vergleich von `null`-Werten zu `NullPointerExceptions`. Folgende Variante kompiliert erst gar nicht:

```
Comparator<Person> nullSafe_V2 = Comparator.nullsFirst(keyExtractor);
```

Das liegt daran, dass hier statt eines `Comparator<Person>` ein `Key-Extractor` übergeben wird. Dadurch kommt es allerdings zu einer ziemlich unverständlichen Fehlermeldung:

```
method nullsFirst in interface Comparator<T#2> cannot be applied to given
types;
required: Comparator<? super T#1>
found: Person::ge[...]Color
reason: cannot infer type-variable(s) T#1
(argument mismatch; invalid method reference
method getFavoriteColor in class Person cannot be applied to given
types
required: no arguments
found: T#1,T#1
reason: actual and formal argument lists differ in length)
where T#1,T#2 are type-variables:
T#1 extends Object declared in method <T#1>nullsFirst(Comparator<? super
T#1>)
T#2 extends Object declared in interface Comparator

invalid method reference
non-static method getFavoriteColor() cannot be referenced from a static
context
```

Sofern man nicht extrem tief in der Materie steckt, wird man wohl aus solchen Fehlermeldungen nicht schlau. Man kann sich zu recht fragen, ob Java nicht in einigen Bereichen zu kompliziert wird. Ein ähnliches Negativbeispiel werden wir später kennenlernen.

Sortierung und kommaseparierte Aufbereitung mit JDK 8

Eingangs dieses Kapitels hatte ich eine Sortierung und kommaseparierte Aufbereitung von einigen Namen mithilfe einer anonymen inneren Klasse und einer `for`-Schleife gezeigt, wodurch der Sourcecode recht lang war. In den vorangegangenen Abschnitten haben wir diverse Bausteine kennengelernt, die die Applikationsentwicklung erleichtern können und eine deutlich kompaktere Schreibweise erlauben: Mithilfe von Default-Methoden und Methodenreferenzen, Lambdas sowie den beschriebenen Erweiterungen im Interface `Comparator<T>` können wir die fast 15 Zeilen nun auf folgende vier Zeilen reduzieren:

```
final List<String> names = Arrays.asList("Andy", "Michael", "Max", "Stefan");

names.sort(Comparator.comparingInt(String::length));
names.replaceAll(str -> "" + str.length());

System.out.println(String.join(", ", names));
```

Im Listing wird mit der Methode `join(CharSequence, CharSequence...)` der Klasse `String` eine weitere Neuerung aus JDK 8 genutzt, die es auf einfache Weise erlaubt, Strings miteinander zu verknüpfen. Darüber hinaus sehen wir einen Aufruf der Methode `replaceAll(UnaryOperator<T>)` aus dem Interface `java.util.List<E>`. Diese Erweiterung aus JDK 8 führt uns zur Thematik der Massenoperationen für Collections, denen wir uns im nächsten Kapitel ausführlich widmen.

2.1.5 Fazit

Mittlerweile sollten Sie einen recht guten Eindruck von Lambdas, Methodenreferenzen und Default-Methoden gewonnen haben, sodass ich ein kurzes Fazit zu diesen Neuerungen von JDK 8 ziehen möchte, bevor wir deren Einsatz in Kombination mit Collections betrachten werden.

In diesem Kapitel haben wir gelernt, dass Lambdas überall dort eingesetzt werden können, wo vor JDK 8 eine anonyme innere Klasse zur Implementierung eines SAM-Typs benötigt wurde. Als Beispiele wurden verschiedene funktionale Interfaces mithilfe von Lambdas implementiert. Dabei gelangen wir auch zu der Erkenntnis, dass man Lambdas gewinnbringend einsetzen kann, um weniger Sourcecode schreiben zu müssen. Darüber hinaus lässt sich teilweise Funktionalität einfacher beschreiben, insbesondere dann, wenn man auf Daten der umgebenden Klasse zugreifen möchte und dazu die `this`-Referenz benötigt. Mit anonymen inneren Klassen gibt es dabei – wie gesehen – syntaktische Schwierigkeiten. Aber auch Lambdas sind in ihrer Schreibweise (vor allem am Anfang für den Ungeübten) nicht immer einfach zu lesen. Hier gilt wie bei jedem Feature: Mit Bedacht und dort, wo es sinnvoll ist, sollte man es einsetzen.

Wenn Lambdas länger als etwa fünf bis zehn Zeilen werden, so sollte man sich überlegen, die Funktionalität in Form von Klassen zu realisieren, um deren potenzielle Wiederverwendbarkeit zu erhöhen. Alternativ und oftmals eleganter ist es, die Funktionalität als Methode zu realisieren und diese über eine Methodenreferenz anzusprechen.

Die Argumentation für Lambdas gilt ebenso für anonyme innere Klassen. In beiden Fällen existiert sonst einiger Sourcecode, der nicht aus anderen Kontexten (anderen Klassen, Methoden, Lambdas) zugreifbar ist und somit auch nicht wiederverwendet werden kann.³

2.2 Bulk Operations on Collections

Neben der kürzeren Schreibweise für SAM-Typen sind Lambdas insbesondere bei der Formulierung von Algorithmen und bei der Verarbeitung von Daten in Collections einsetzen. Bis zur Einführung von JDK 8 lief die Verarbeitung von Operationen auf Collections oftmals sequentiell ab. Wurde Parallelität benötigt, so musste dies explizit ausprogrammiert werden. Dazu konnte man beispielsweise auf das mit JDK 7 eingeführte Fork-Join-Framework zurückgreifen. Als Applikationsentwickler möchte man sich aber eigentlich möglichst wenig mit den zugrunde liegende Details auseinander setzen müssen, insbesondere auch nicht mit einer geeigneten Zerlegung der Aufgabe in einzelne durch Fork-Join zu verarbeitende Tasks. In der Regel besteht der Wunsch, sich auf einer höheren konzeptionellen Ebene mit dem Thema Parallelisierung beschäftigen zu können, etwa um den Standardanwendungsfall von einfach und effizient zu behandeln.

Die mit JDK 8 eingeführten Massenoperationen auf Collections (Bulk Operations on Collections) bieten sich dafür geradezu an. Sie stellen ein sehr wichtiges Sprachfeature dar und bieten eine effiziente Möglichkeit zur Parallelisierung bieten, wodurch sich die Fähigkeiten von Multicore-Maschinen besser ausnutzen lassen. Dazu wird das zuvor vorgestellte Sprachfeature der Lambdas mit einer umfangreichen Erweiterung des Collections-Frameworks kombiniert, den sogenannten Streams.

Bevor wir uns gleich genauer mit Streams beschäftigen, werfen wir als Vorbereitung einen Blick auf zwei Varianten der Iteration.

2.2.1 Externe vs. interne Iteration

Mit Iteration meint man das Durchlaufen einer Collection. Dies kann auf zweierlei Weise geschehen: als externe und als interne Iteration. Dabei ist mit externer Iteration gemeint, dass die Collection das Durchlaufen unterstützt, etwa durch indizierte Zugriffe oder aber mithilfe eines Iterators. Dabei wird der Vorgang der Iteration durch den Aufrufer bestimmt. Bei der internen Iteration wird der Vorgang des Durchlaufens durch die Collection gekapselt und dort intern realisiert. Implementierungsdetails bleiben so verborgen, allerdings sind auch Möglichkeiten zu Einflussnahme begrenzt.

Nachfolgend stelle ich kurz einige Beispiele für externe und interne Iteration und deren Bedeutung vor.

³Außer natürlich über Copy-Paste, wodurch es dann aber schnell zu einem Wartungsalbtraum kommen kann, denn: Wie findet und ändert man alle Stellen konsistent? Details dazu beschreibt BAD SMELL: UNVOLLSTÄNDIGE ÄNDERUNGEN NACH COPY-PASTE in Abschnitt 14.1.8.

Externe Iteration

Nehmen wir an, wir wollten alle Elemente einer Collection auf der Konsole ausgeben. Herkömmlicherweise könnte man dies wie folgt tun:

```
final List<String> names = Arrays.asList("Andi", "Mike", "Ralph", "Stefan" );

// Klassische Variante mit Iterator ...
final Iterator<String> it = names.iterator();
while (it.hasNext())
{
    final String name = it.next();
    System.out.println(name);
}

// ... oder alternativ indiziertem Zugriff
for (int i = 0; i < names.size(); i++)
{
    System.out.println(names.get(i));
}

// JDK 5-Schreibweise mit for-each
for (final String name : names)
{
    System.out.println(name);
}
```

An diesem Beispiel erkennt man sehr schön die iterative und sequentielle Abarbeitung sowohl für die Variante mit Iterator als auch für den danach gezeigten indizierten Zugriff. Die Variante mit der seit JDK 5 verfügbaren for-each-Schleife zeigt den sequentiellen Charakter weniger klar.⁴ In allen drei Fällen spricht man von *externer Iteration*, weil die *Traversierung im Applikationscode programmiert* wird.

Interne Iteration

Mit JDK 8 wurden die Klassen des Collections-Frameworks derart erweitert, dass sie verschiedene Verarbeitungsmethoden anbieten, die man bisher über for- oder while-Schleifen selbst programmieren musste, etwa die bereits bekannte Methode `forEach(Consumer<? super T>)`. Die interne Iteration erfordert logischerweise auch die Angabe einer auszuführenden Funktionalität. Dazu nutzt man Callback-Interfaces und seit JDK 8 zu deren Implementierung Lambdas und Default-Methoden:

```
// Interne Iteration in drei Varianten
names.forEach((String name) -> { System.out.println(name); });
names.forEach(name -> System.out.println(name) );
names.forEach(System.out::println);
```

Die im Listing gezeigt Form wird *interne Iteration* genannt. Bekanntermaßen muss die Iteration nicht vom Entwickler programmiert werden, sondern diese *im Framework realisiert wird*. Man selbst übergibt lediglich die auszuführende Aktion.

⁴Sie erleichtert lediglich die Schreibweise, stellt sogenannten syntaktischen Zucker dar, und wird beim Kompilieren in eine externe Iteration mit Iterator umgewandelt.

Externe vs. interne Iteration am Beispiel

Zwar kennen wir jetzt die Begriffe, jedoch sind die Unterschiede und Auswirkungen möglicherweise noch nicht wirklich greifbar. Daher möchte ich an einem Beispiel die Vorteile einer internen Iteration verdeutlichen.

Stellen wir uns vor, es wäre eine Hilfsmethode zu realisieren, die eine Aktion für die Objekte einer Liste ausführen soll, z. B. alle in einem Malprogramm selektierten Figuren etwas heller darstellt.

Als externe Iteration würde man dies etwa folgendermaßen realisieren:

```
public static void brightenExtern(final List<GraphicsFigure> selectedFigures)
{
    for (final GraphicsFigure figure: selectedFigures)
    {
        brighten(figure);
    }
}
```

Diese Lösung besitzt die zuvor angedeuteten Eigenschaften der vom Aufrufer kontrollierten und durchgeführten Iteration. Nachteilig ist in der Regel, dass diese sequentiell abläuft. Darüber hinaus wird die Funktionalität und die Iteration an sich miteinander gemischt – hier wiegt es nicht so schwer, da durch den Methodenaufruf recht gut für Klarheit gesorgt wird.

Schauen wir nun auf die korrespondierende interne Iteration:

```
public static void brightenIntern(final List<GraphicsFigure> selectedFigures)
{
    selectedFigures.forEach(figure ->
    {
        brighten(figure);
    });
}
```

Auf den ersten Blick sieht diese Implementierung mit Lambda kaum anders aus als die externe Variante. Dieser Eindruck täuscht, insbesondere weil ich hier bewusst eine ähnliche Formatierung des Sourcecodes gewählt habe. Rekapitulieren wir zunächst nochmals, dass hier die Iteration durch das Framework, also den Implementierer von `forEach(Consumer<? super T>)` realisiert wird. Dies bietet vor allem den Vorteil, dass theoretisch eine Parallelverarbeitung erfolgen kann, solange die Aktionen für die einzelnen Elemente voneinander unabhängig sind. Dann dürfte die Reihenfolge der Bearbeitung von der sequentiellen abweichen.

Der ganz entscheidende Unterschied ist aber, dass zur Berechnung ein Lambda genutzt wird. Durch Einsatz des SAM-Typs `Consumer<T>` kann man mit ein wenig Erfahrung die obige Methode mit minimalen Aufwand wie folgt verallgemeinern, so dass die Abarbeitung beliebiger `Consumer<T>` möglich wird:

```
public static void process(final Collection<GraphicsFigure> figures,
                          final Consumer<GraphicsFigure> consumer)
{
    figures.forEach(consumer);
}
```

```
}
```

Um die Funktionalität der Aufhellung zu realisieren, können wir der obigen Methode dann einen passenden `Consumer<GraphicsFigure>` wie folgt übergeben:

```
final Consumer<GraphicsFigures> brighten = figure -> brighten(figure);  
process(figures, brighten);
```

Man benötigt wenig Fantasie, um zu erkennen, welche Vielfalt an Möglichkeiten sich daraus ergibt, wenn man andere Realisierungen von `Consumer<GraphicsFigure>` nutzt. Hier beginnen wir zu erahnen, was funktionale Programmierung ausmacht: Man kann Funktionalität in Form von Sourcecode als Parameter übergeben (»Code as Data«) und an beliebiger Stelle bei Bedarf ausführen.

Wenn wir noch ein wenig darüber nachdenken, erkennen wir, dass die Methode `process(Collection<GraphicsFigure>, Consumer<GraphicsFigure>)` nun eigentlich überflüssig ist und man für dieses Beispiel nur noch einen Lambda und `forEach(Consumer<? super T>)` benötigt:

```
final Consumer<GraphicsFigures> brighten = figure -> brighten(figure);  
figures.forEach(brighten);
```

Auf diese Weise kann man natürlich auch beliebige andere Funktionalität ausführen.

Hinweis: Preconditions und Parameterprüfung mit interner Iteration

Nicht nur in der eigentlichen Programmlogik, sondern auch für Zustandsprüfungen kann man von interner Iteration profitieren. Als Beispiel schauen wir auf eine typische Realisierung einer Methode, die Aktionen für diejenigen Elemente einer Collection ausführt, die eine bestimmte Bedingung erfüllen:

```
public static void doAction(final List<Person> persons)
{
    // Zustandsprüfung
    Objects.requireNonNull(persons, "list of persons must not be null");
    Objects.requireNonNull(name, "name must not be null");

    final Iterator<Person> it = copy.iterator();
    while (it.hasNext())
    {
        final Person person = it.next();

        // Sicherheitsprüfung und Logik
        if (person != null && accept(person))
        {
            doAction(person);
            // ...
        }
    }
}
```

Wir konzentrieren uns hier auf die Prüfung gültiger Eingaben: Oftmals wird zwar – wie auch hier – sichergestellt, dass die Übergabeparameter ungleich `null` sind. Seit JDK 7 nutzt man dazu sinnvollerweise die Methode `Objects.requireNonNull()`. Aufwendiger gestaltet es sich häufig, die einzelnen Elemente auf Gültigkeit bzw. zumindest auf ungleich `null` zu testen. Im obigen Listing wird dies in Kombination mit der eigentlichen Anwendungslogik realisiert. Das spart zwar Schreibaufwand und ist etwas performanter, jedoch vermischt man hier Zustandsprüfung und Anwendungslogik. Das sollte man, wenn möglich, vermeiden. Eine zusätzliche externe Iteration zur Konsistenzprüfung wirkt wenig elegant:

```
// Prüflogik mit externer Iteration
for (final Person person : persons)
{
    Objects.requireNonNull(person);
}
```

Wenn wir aber eine interne Iteration mit `forEach()` mit Methodenreferenzen kombinieren, lässt sich die Zustandsprüfung wie folgt klarer realisieren und fügt sich dadurch nahtlos in die anderen Prüfungen ein:

```
// Prüflogik mit interner Iteration und Methodenreferenz
persons.forEach(Objects::requireNonNull)
```

Derart kann man mit Zustandsprüfungen für die Einhaltung von Preconditions sorgen, wodurch sich nachfolgende Programmteile auf eine korrekte Initialisierung verlassen können. Das Ganze hat allerdings einen seinen Preis: Die Iteration über die Elemente erfolgt nun zweimal, wodurch sich die Performance verschlechtert. Aus Performancegründen sollte man sich bei den Prüfungen auf die externen Schnittstellen zu anderen Systemen fokussieren.

2.2.2 Collections-Erweiterungen und interne Iteration

Neben der bereits kennengelernten internen Iteration mit `forEach()` existieren diverse weitere Beispiele für diese Art der Iteration im JDK. Einige wichtige finden wir im Collections-Framework. Bevor ich diese kurz bespreche, betrachten wir noch das funktionale Interface `java.util.function.Predicate<T>` als Basis.

Das funktionale Interface `Predicate<T>`

Das funktionale Interface `Predicate<T>` erlaubt es, sogenannte Prädikate zu formulieren. Das sind boolesche Bedingungen, die durch Aufruf der im Interface definierten Methode `boolean test(T)` ausgewertet werden. Wie schon zuvor erwähnt, wird man recht selten ein funktionales Interface mit einer Klasse realisieren, sondern nahezu immer dafür Lambdas oder auch Methodenreferenzen nutzen. Auf diese Weise lassen sich einfache Prüfungen auf den Wert `null`, einen Leerstring oder ein Alter `>= 18` wie folgt kurz und knackig formulieren:

```
public static void main(final String[] args)
{
    // Predicate formulieren
    final Predicate<String> isNull = str -> str == null;
    final Predicate<String> isEmpty = String::isEmpty;
    final Predicate<Person> isAdult = person -> person.getAge() >= 18;
    // final Predicate<Person> isAdult = Person::isAdult;

    System.out.println(isNull.test("")); // false
    System.out.println(isEmpty.test("")); // true
    System.out.println(isEmpty.test("Micha")); // false
    System.out.println(isAdult.test(new Person("Michael", 43))); // true
}
```

Das Prädikat `isAdult` kann man als Lambda schreiben oder man könnte es auch – wie oben im Kommentar angedeutet – mithilfe einer Methodenreferenz realisieren, die auf folgende Methode `isAdult()` in der Klasse `Person` verweist:

```
public class Person
{
    private int age;

    // ...

    public boolean isAdult()
    {
        return age >= 18;
    }
}
```

Komplexere Bedingungen mit Prädikaten formulieren Zwar kann man mit einfachen Prädikaten schon einige Anwendungsfälle abdecken, häufiger wird man jedoch verschiedene Bedingungen miteinander kombinieren wollen, um komplexere Ab-

fragen umsetzen zu können. Dazu bietet sich häufig der Einsatz der folgenden drei Default-Methoden an, um boolesche Verknüpfungen auszuführen:

- `negate()` – negiert die Bedingung.
- `and(Predicate<? super T>)` – verknüpft die aktuelle Bedingung mit einer anderen Bedingung mit logischem UND.
- `or(Predicate<? super T>)` – verknüpft die aktuelle Bedingung mit einer anderen Bedingung mit logischem ODER.

Mit diesem Wissen bauen wir unser Beispiel ein wenig aus und fokussieren uns nun auf die Kombination von Prädikaten.

```
public static void main(final String[] args)
{
    final List<Person> persons = createDemoData();

    // Einfache Predicate formulieren
    final Predicate<Person> isAdult = person -> person.getAge() >= 18;
    final Predicate<Person> isYoung = isAdult.negate();
    final Predicate<Person> isMale = person -> person.getGender() == Gender.MALE;

    // Negation
    final Predicate<Person> isFemale = isMale.negate();

    // Kombination von Predicates mit AND
    final Predicate<Person> maleBoys = isMale.and(isYoung);
    final Predicate<Person> femaleAdults = isFemale.and(isAdult);

    // Verschachtelte Kombination von Predicate mit OR
    final Predicate<Person> maleBoysOrFemaleAdults = maleBoys.or(femaleAdults);

    removeAll(persons, maleBoysOrFemaleAdults);
    System.out.println(persons);
}
```

Wir sehen den Aufruf einer Methode `removeAll(List<E>, Predicate<E>)`, der intuitiv verständlich ist und Elemente entfernt, die der übergebenen Bedingung entsprechen. In diesem Beispiel sind dies alle männlichen Personen unter 18 sowie alle erwachsenen Frauen.

Die Methode `Collection.removeIf()`

Wie im Beispiel schon angedeutet, lassen sich Prädikate für das Löschen von Elementen, die einer Bedingung genügen, einsetzen. Herkömmlicherweise lässt sich dies mit externer Iteration nur recht mühsam mithilfe eines Iterators in etwa wie folgt lösen:

```
// Externe Iteration
private static <E> void removeAll(final List<E> list,
                                final Predicate<? super E> condition)
{
    final Iterator<E> it = list.iterator();
    while (it.hasNext())
    {
        final E element = it.next();
```



```

        if (condition.test(element))
        {
            it.remove();
        }
    }
}

```

Praktischerweise wurden im Interface `Collection<E>` mit JDK 8 verschiedene Methoden hinzugefügt, unter anderem mit `removeIf(Predicate<T>)` ein, die funktional analog zur obigen, selbst realisierten Methode `removeAll()` arbeitet, dabei jedoch interne Iteration nutzt.

Als Beispiel sollen aus einer Liste von Namen diejenigen herausgelöscht werden, die einen Leereintrag darstellen. Das realisieren wir mit `removeIf(Predicate<T>)` und einem Lambda wie folgt:

```

public static void main(final String[] args)
{
    final List<String> names = createDemoNames();

    // Löschaktionen ausführen
    names.removeIf(String::isEmpty);

    System.out.println(names);
}

private static List<String> createDemoNames()
{
    final List<String> names = new ArrayList<>();
    names.add("Max");
    names.add(""); // Leereintrag
    names.add("Andy");
    names.add("Michael");
    names.add(" "); // potenziell auch ein "Leereintrag"
    names.add("Stefan");
    return names;
}

```

Führen wir das Programm aus, so wird die zuvor beschriebene Löschoperation ausgeführt und es kommt zu folgender Ausgabe:

```
[Max, Andy, Michael, , Stefan]
```

Wir sehen, dass der Whitespace-Eintrag (logischerweise) in der Liste verblieben ist. Man könnte nun eine komplexere Bedingung formulieren. Wir wollen jedoch dazu die mit JDK 8 im Interface `List<E>` neu eingeführte Methode `replaceAll(UnaryOperator<E>)` und das bisher unbekannte Interface `UnaryOperator<E>` kennenlernen.

Das Interface `UnaryOperator`

Das funktionale Interface `UnaryOperator<T>` definiert selbst nur die statische Methode `identity()`. Entscheidender ist, dass es indirekt über ihr Basisinterface `Function<T, R>` die Methode `apply(T)` deklariert, die ein Element vom Typ `T` wie-

derum auf ein Element vom Typ T abbildet. Beide Interfaces sind im Listing auf das für das erste Verständnis Wesentliche gekürzt:

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T>
{
    // Statische Methoden seit JDK 8 in Interfaces erlaubt
    static <T> UnaryOperator<T> identity()
    {
        return t -> t;
    }
}

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

    // ...
}
```

Das Ganze ist noch etwas abstrakt, daher schauen wir auf verschiedene Realisierungen von `UnaryOperator<String>`: Man könnte etwa alle mit `M` startenden Namen speziell markieren und groß schreiben. Ein für die Praxis eher relevantes Beispiel besteht aber in dem davor gezeigten Trimmen und der Abbildung von `null`-Werten auf gewünschte Defaultwerte. Das Ganze könnte man wie folgt lösen:

```
public static void main(final String[] args)
{
    // Mark
    final UnaryOperator<String> markTextWithM = str -> str.startsWith("M") ?
        ">>" + str.toUpperCase() + "<<" : str;

    printResult("Mark 1", markTextWithM, "is unchanged");
    printResult("Mark 2", markTextWithM, "Michael");

    // Trim
    final UnaryOperator<String> trimmer = String::trim;
    printResult("Trim", trimmer, " trim me ");

    // Map
    final UnaryOperator<String> mapNullToEmpty = str -> str == null ? "" : str;
    printResult("Map null", mapNullToEmpty, null);
    printResult("Map same", mapNullToEmpty, "stays the same");
}

private static void printResult(final String text,
                                final UnaryOperator<String> mapNullToEmpty,
                                final String value)
{
    System.out.println(text + ": '" + mapNullToEmpty.apply(value) + "'");
}
```

Startet man das Programm `UNARYOPERATOREXAMPLE`, so kommt es zu folgenden Ausgaben:

```
Mark 1: 'is unchanged'
Mark 2: '>>MICHAEL<<'
```

```
Trim: 'trim me'  
Map null: ''  
Map same: 'stays the same'
```

Die Methode `List.replaceAll()`

Auch das Interface `List<E>` wurde mit JDK 8 erweitert. Wir wollen nachfolgend auf die Methode `replaceAll(UnaryOperator<E>)`, die es ermöglicht, für alle Elemente einer Collection eine Aktion auszuführen: Jedes Element wird durch den Rückgabewert der Implementierung des funktionalen Interfaces `UnaryOperator<E>` ersetzt. Durch die Anweisungen der Realisierung wird auch die Entscheidung getroffen, welche Elemente wie bearbeitet werden sollen, sodass nicht immer alle Elemente tatsächlich auch verändert werden (müssen).⁵

Nach diesen zuvor eher theoretischen Details kommen wir auf ein konkretes Anwendungsbeispiel: im Speziellen dann, wenn man viel mit String-Operationen arbeitet, ist die `replaceAll(UnaryOperator<E>)`-Funktionalität sinnvoll und hilfreich. Nehmen wir an, wir würden entweder auf einer externen Datenquelle oder dem GUI eine Liste von Eingabewerten erhalten. Oftmals entsprechen solche Eingaben nicht den Erwartungen und verstoßen gegen Regeln, etwa sind Einträge leer, bestehen nur aus Leerzeichen oder enthalten diese am Anfang oder Ende. All dies erschwert die weitere Bearbeitung. Die Grundlagen für eine Korrekturfunktionalität, die derartige Werte umwandelt oder herausfiltert, haben wir bereits kennengelernt. Wir müssen das Ganze lediglich noch geeignet kombinieren:

```
public static void main(final String[] args)  
{  
    final List<String> names = createDemoNames();  
  
    // Spezialbehandlung von null-Werten  
    names.replaceAll(str -> str == null ? "" : str);  
  
    // Leerzeichen abschneiden  
    names.replaceAll(String::trim);  
  
    // Leereinträge herausfiltern  
    names.removeIf(String::isEmpty);  
  
    System.out.println(inputs);  
}
```

Mit dieser Erweiterung wird nicht nur der `null`-Eintrag entfernt, sondern auch derjenige, der nur Leerzeichen enthält.

⁵Das »All« im Namen bezieht sich also lediglich darauf, dass die übergebene Aktion für alle Elemente der Liste ausgeführt wird.

2.2.3 Streams

Wir haben bisher verschiedene spezialisierte Formen von Bulk Operations kennengelernt. Deutlich mehr Möglichkeiten bietet das in JDK 8 neu eingeführte Konzept der **Streams**. Dafür spielt das Interface `java.util.stream.Stream` eine Schlüsselrolle. Streams sind eine Abstraktion für **Folgen von Verarbeitungsschritten auf Daten**. Darüber hinaus ähneln Streams sowohl Collections als auch Iteratoren, wobei Streams keine Speicherung der Daten vornehmen und auch nur einmal traversiert werden können. Die stärkste Analogie ist wohl zu einer Abarbeitung als Pipeline oder Fließband. Dabei unterscheidet man zwischen den folgenden drei Typen von Operationen: Create (Erzeugung), Intermediate (Berechnungen) und Terminal (Ergebnisbereitstellung), die man sich wie folgt visualisieren kann:

$$\underbrace{Quelle \Rightarrow STREAM}_{Create} \Rightarrow \underbrace{OP_1 \Rightarrow OP_2 \Rightarrow \dots \Rightarrow OP_n}_{Intermediate} \Rightarrow \underbrace{Ergebnis}_{Terminal}$$

Einführendes Beispiel

Nehmen wir an, wir wollten in einer Liste von Personen prüfen, ob eine Person gewünschten Namens enthalten ist bzw. den ersten gefundenen Eintrag zurückliefern. Vor JDK 8 könnte man eine entsprechende `containsPersonWithName()` bzw. `findPersonByName()`-Methode z. B. wie folgt schreiben:⁶

```
boolean containsPersonWithName(final List<Person> persons, final String desired)
{
    return findPersonByName(person, desired) != null;
}

Person findPersonByName(final List<Person> persons, final String desired)
{
    for (final Person person : persons)
    {
        if (person.getName().equals(desired))
        {
            return person;
        }
    }

    return null;
}
```

Wenn wir Lambdas in Kombination mit den im Anschluss vorgestellten Streams nutzen, so schreiben wir:

```
// Namensfilter definieren
final Predicate<Person> nameFilter = person -> person.getName().equals(desired);
```

⁶Leider sieht man in der Realität häufiger den Fall, dass die Funktionalität bzw. genauer die Iteration zweimal implementiert wird, nämlich für `contains()` bzw. `find()`. Für `contains()` wird dann statt eines `Person`-Objekts `true` und statt `null` der Wert `false` als Ergebnis geliefert.

```
// containsPersonByName()
final boolean personFound = persons.stream().anyMatch(nameFilter);

// findPersonByName()
final Optional<Person> searchedPerson = persons.stream().filter(nameFilter).
                                                findFirst();
```

Im Listing sehen wir die Konstruktion einer Filterbedingung durch die Implementierung eines `Predicate<T>` als Lambda. Diesen `nameFilter` nutzen wir als Eingabe für einen Aufruf von `anyMatch(Predicate<T>)` zum Ermitteln, ob mindestens ein Element im Stream die übergebene Bedingung erfüllt. Außerdem wird der `nameFilter` an eine Kombination aus `filter(Predicate<T>)` und `findFirst()` übergeben, wodurch eine Filterung erfolgt und der erste Eintrag der Treffermenge ermittelt wird. Diese kann durchaus leer sein. Gewöhnlich wird so etwas durch Rückgabe von `null` oder Null-Objekten gemäss dem gleichnamigen Entwurfsmuster implementiert. Mit JDK 8 kann man die dort neu eingeführte generische Klasse `Optional<T>` nutzen, die die Darstellung optionaler Werte in Form eines Objekts ermöglicht. Existiert kein gültiger Wert, so wird dies durch die Konstante `Optional.empty` ausgedrückt. Auf die Klasse `Optional<T>` gehe ich in Abschnitt 2.5.1 genauer ein.

Neben all diesen Implementierungsneuerungen erkennt man sehr schön, dass sich Konzepte und das »Was« viel klarer erkennen lassen und nicht das »Wie« (die Implementierung der Funktionalität) im Vordergrund steht.

2.2.4 Streams — Create-Operations

Nach den ersten Beispielen zu Streams, wollen wir unsere Kenntnisse zu Streams vertiefen. In den nachfolgenden Abschnitten stelle ich Ihnen vor, auf welche vielfältigen Weisen man Streams erzeugen kann.

Streams für Arrays und Collections

Zunächst sollten wir uns fragen, wie wir an ein `Stream`-Objekt kommen. Wie bereits gesehen, ist das relativ einfach für Arrays oder Collections möglich. Für beide ist eine `stream()`-Methode definiert, die man wie folgt nutzen kann:

```
final String[] namesData = { "Karl", "Ralph", "Andi", "Andy", "Mike" };
final List<String> names = Arrays.asList(namesData);

final Stream<String> streamFromArray = Arrays.stream(namesData);
final Stream<String> streamFromList = names.stream();
```

Als Besonderheit können Collections eine sequentielle sowie eine parallele Variante eines Stream liefern:

```
final Stream<String> sequentialStream = names.stream();
final Stream<String> parallelStream = names.parallelStream();
```

Für Arrays bietet die Utility-Klasse `Arrays` nur Zugriff auf eine sequentielle Variante. Um auch hier eine Parallelverarbeitung zu erreichen, kann man die Methode `parallel()` auf dem Stream aufrufen, die das Umschalten zur Parallelverarbeitung vornimmt. Für das obige Array könnte man somit etwa Folgendes schreiben:

```
final Stream<String> parallelArrayStream = Arrays.stream(namesData).parallel();
```

Streams für vordefinierte Wertebereiche

Teilweise soll über Streams ein fixer, vordefinierter Wertebereich abgebildet und bearbeitet werden. Für diese Fälle existieren spezielle Methoden, etwa `of()`, `range()` und `chars()`:

```
final Stream<Integer> integers = Streams.of(7, 1, 7, 9); // Integer
final IntStream values = IntStream.range(0, 100); // Primitive Stream: int
final IntStream chars = "This is a test".chars(); // Primitive Stream: int
```

Hinweis: Ergänzungen zu Streams im JDK 8

Im obigen Beispiel erkennen wir, dass es neben dem generischen Interface `Stream<T>` spezielle Interfaces für primitive Datentypen, hier `java.util.stream.IntStream` gibt, die verschiedene Konstruktionsmethoden bereitstellen. Wir sehen hier die statischen Methoden `of(T...)` und `range(int, int)`. Sprach ich nicht eben von Interfaces? Wieso können denn dort statische Methoden definiert sein? Erinnern wir uns an Abschnitt 2.1.2: Im Zuge von JDK 8 und der Erweiterung um Default-Methoden können Interfaces nun auch statische Methoden bereitstellen. Insgesamt verschwimmt der Unterschied zwischen abstrakten Klassen und Interfaces immer mehr.

Im Beispiel sieht man, dass auch bisherige Klassen und Interfaces des JDKs nun Methoden anbieten, die Streams zurückliefern. Zuvor ist dies für das Interface `CharSequence` und die Methode `chars()` der Fall. Diese ist dort in Form einer Default-Methode realisiert und steht damit allen Spezialisierungen direkt zur Verfügung.

Besonderheit: Unendliche Streams

Manchmal möchte man eine unendliche Folge modellieren. Eine solche kann aufgrund Ihrer Unendlichkeit und des damit verbundenen unendlichen Platzbedarfs demnach praktisch gar nicht vollständig existieren, sondern muss Stück für Stück berechnet werden. Für primitive Typen kann man dazu die Methode `iterate(int, IntUnaryOperator)` nutzen, der man einen Startwert und danach eine Implementierung des funktionalen Interfaces `java.util.function.IntUnaryOperator` übergibt, nachfolgend in Form des Lambdas `x -> x + 1`. Das Functional Interface `UnaryOperator<T>` haben wir ja bereits kennengelernt. `IntUnaryOperator` ist eine

besondere Variante davon, die für den primitiven Typ `int` spezialisiert ist und im Gegensatz zu `UnaryOperator<Integer>` direkt auf den primitiven Typen arbeitet und somit kein Auto-Boxing benötigt.

Zur Generierung für Abfolgen von Werten beliebiger Referenz-Typen kann man die Methode `generate(Supplier<T>)` aus dem Interface `Stream` einsetzen. Das funktionale Interface `java.util.function.Supplier<T>` realisieren wir mit Methodenreferenz auf die Methode `incrementAndGet()` der Klasse `AtomicInteger` aus dem Package `java.util.concurrent.atomic` wie folgt:

```
final IntStream iteratingValues = IntStream.iterate(0, x -> x + 1);

final AtomicInteger ai = new AtomicInteger(0);
final Stream<Integer> generatedValues = Stream.generate(ai::incrementAndGet);
```

Als Wertefolgen würde folgendes produziert:

```
iteratingValues: 0,1,2,3,4,...
generatedValues: 1,2,3,4,5,...
```

Ihnen ist bestimmt aufgefallen, dass es zu abweichenden Ausgaben kommt, weil die von `generate()` erzeugte Nummernfolge mit der Ziffer 1 startet. Das dient hier nur zur Demonstration. Wollten wir mit dem Wert 0 starten, so kann dazu die Methode `getAndIncrement()` der Klasse `AtomicInteger` genutzt werden.

Meinung: Namensgebung von Lambda-Parametern

Wie Sie vielleicht bemerkt haben, verwende ich für die Parameter in Lambdas bevorzugt sprechende Namen oder aber Standards wie `it`. Meiner Meinung nach gilt auch hier, dass man so lesbar wie möglich programmieren sollte. Nur weil man funktional programmiert, heißt das nicht, dass man wieder auf Namensverkümmernungen wie `a`, `p`, `x` zurückgreifen muss. Natürlich gibt es auch Fälle, in denen Kürzel mit einem Buchstaben ihren Wert haben. Das gilt immer dann, wenn im Lambda eine beliebige Berechnung erfolgt, etwa `x -> x + 1`. Dabei trägt der Parameter keine oder nur wenig semantische Bedeutung – meistens, weil es eine »echte« mathematische Funktion ist.

2.2.5 Intermediate Operations

Nachdem wir kurz gesehen haben, wie wir ein Objekt vom Typ `Stream` entweder aus einer Collection oder aber einer anderen Quelle erhalten können, schauen wir nun darauf, was man mit diesen als Stream vorliegenden Daten anfangen kann.

Gebräuchliche Anwendungsfälle sind etwa das Filtern, das Transformieren und das Sortieren von Werten. Dazu nutzt man sogenannte **Intermediate Operations**. Man unterscheidet zwischen zustandslosen und zustandsbehafteten Varianten. Filtern ist eine zustandslose Aktion: Damit ist gemeint, dass für jedes Element des Streams unabhän-

gig von den anderen diese Aktion ausführbar ist. Dadurch lassen sich zustandslose Operationen auch hervorragend parallelisieren. Sortieren ist dagegen eine zustandsbehaftete Aktion, die die Kenntnis der anderen Elemente im Stream (oder zumindest eines Teils davon) erfordert.

Intermediate Operations beschreiben Verarbeitungsschritte, die sich einfach hintereinanderschalten lassen. Diese sind »lazy« und berechnen erst etwas, wenn dies durch eine Terminal-Operation benötigt wird. Da sie keine (oder für zustandsbehaftete Operationen nur eine Untermenge der) Daten zwischenspeichern, verbrauchen sie im Gegensatz zu Collections deutlich weniger Speicher. Die Konstruktion von Streams hat also in der Regel wenig Einfluss auf Speicherbedarf und Ausführungszeit.

2.2.6 Zustandslose Intermediate Operations

In diesem Abschnitt schauen wir auf verschiedene gebräuchliche zustandslose Intermediate Operations.

Filterung

Beginnen wir mit dem Filtern. Diese gebräuchliche Funktionalität wurde bisher leider nicht durch das JDK bereitgestellt. Glücklicherweise ist die nun sehr einfach mit JDK 8 möglich.

Schauen wir nun an einem Beispiel einer Liste von `Person`-Objekten, wie wir mithilfe von `filter(Predicate<Person>)` diejenigen ermitteln, die erwachsen sind, indem wir die Methodenreferenz `Person::isAdult` wie folgt nutzen:

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Micha", 42));
    persons.add(new Person("Barbara", 40));
    persons.add(new Person("Yannis", 5));
    persons.add(new Person("Leo", 7));

    final Stream<Person> adults = persons.stream().filter(Person::isAdult);

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

Mehrstufige Filterung In der Praxis soll oftmals eine mehrstufige Filterung nach verschiedenen Kriterien erfolgen. Mit der Pipeline- oder Fließbandanalogie im Hinterkopf kann man dazu mehrere Filter hintereinanderschalten:

```
final Stream<Person> allAdultMikes = persons.stream().
    filter(Person::isAdult).
    filter(person -> person.getName().equals("Mike"));
```


Mapping von Daten, Extraktion von Werten

Neben der Filterung ist die Konvertierung oder Extraktion von Werten eine typische Intermediate Operation. Hierbei soll eine Menge von Eingabedaten in ein anderes Format überführt werden. So könnte etwa aus einer Liste von Personen jeweils das Attribut Name, Vorname oder Alter extrahiert werden.

Theorie: Mapping von Werten In der Praxis benötigt man häufiger eine Abbildung oder ein Mapping von einem Typ auf einen anderen, etwa möchte man aus dem Typ `Person` ein Attribut herauslesen und muss auf denjenigen Typ des gewünschten Attributs, z. B. `String`, abbilden. Dazu kann man Spezialisierungen des Interface `Function<T,R>` und dort die Methode `apply(T t)` entsprechend implementieren. Dies haben wir bereits im Zusammenhang mit dem Interface `UnaryOperator<T>` kurz besprochene und die Definition wird hier zum Einstieg nochmals wiederholt:

```
interface Function<T,R>
{
    R apply(T t);
}
```

Nehmen wir an, es wäre der Name aus einem `Person`-Objekt zu extrahieren. Dies implementieren wir mithilfe eines Lambdas oder mit einer Methodenreferenz wie folgt:

```
// Lambda
Function<Person, String> nameExtractor_V1 = person -> person.getName();

// Alternativ mit Methodenreferenz
Function<Person, String> nameExtractor_V2 = Person::getName;
```

Extraktion am Beispiel Wir haben nun genug Vorwissen gesammelt und machen uns damit daran, die Extraktion des Namens bzw. des Alters für eine Liste von Personen folgendermaßen auszuprogrammieren:

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Micha", 43));
    persons.add(new Person("Barbara", 40));
    persons.add(new Person("Yannis", 5));
    persons.add(new Person("Leo", 7));

    // Mapping auf Name mit Lambda
    final Stream<Person> adults = persons.stream().filter(Person::isAdult);
    final Stream<String> namesStream = adults.map(person -> person.getName());

    // Mapping auf Alter mit Methodenreferenz
    final Stream<Integer> agesStream = persons.stream().map(Person::getAge)
                                                .filter(age -> age >= 18);

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

Führen wir das obige Programm ATTRIBUTEEXTRACTIONEXAMPLE aus, so erhalten wir folgende Ausgaben:

```
Micha
Barbara
43
40
```

Besonderheit: Inspektion von Verarbeitungsschritten

Für den Fall, dass die Filterungen bzw. ganz allgemein die Intermediate Operations komplexer werden, möchte man sich eventuell auch einmal Zwischenergebnisse ausgeben lassen und danach die Filterung fortsetzen. Ein beliebter Fallstrick dabei ist es, einen bereits mit `forEach()` ausgegebenen oder irgendwie mit einer Terminal Operation verarbeiteten Stream weiter bearbeiten zu wollen, etwa wie folgt:

```
// Hilfsvariable definieren
final Stream<Person> adults = persons.stream().filter(Person::isAdult);

// Ausgabe, um die Filterung zu überprüfen
adults.forEach(System.out::println);

// Weitere Filterung auf dem Stream vornehmen
final Stream<Person> allMikes = adults.filter(person ->
                                         person.getName().equals("Mike"));
```

Führt man diese Schritte aus, kommt es aber statt einer Weiterverarbeitung zu folgender Exception:

```
// Exception in thread "main" java.lang.IllegalStateException: stream has
// already been operated upon or closed
```

Am Text dieser Exception erkennt man die bereits erwähnte Eigenschaft von Streams, die Daten nur einmal bereitstellen zu können.

Weil die Inspektion aber eine sehr wünschenswerte Eigenschaft ist, stellt das Stream-API hierfür eine Möglichkeit bereit. Mithilfe der Intermediate Operation in Form der Methode `peek(Consumer<T>)` kann man etwa Konsolenausgaben wie mit `forEach(Consumer<T>)` durchführen, erhält aber als Rückgabe wiederum einen Stream. Obiges Beispiel kann man zur Realisierung einer Inspektion wie folgt abändern:

```
public class StreamPeekExample1
{
    public static void main(final String[] args)
    {
        final List<Person> persons = new ArrayList<>();
        persons.add(new Person("Michael", 43));
        persons.add(new Person("Micha", 42));
        persons.add(new Person("Merten", 38));
        persons.add(new Person("Max", 6));
        persons.add(new Person("Moritz", 6));
```

```
// Hilfsvariable definieren
final Stream<Person> adults = persons.stream().filter(Person::isAdult);

// Ausgabe mit peek(), um die Filterung zu überprüfen
final Stream<Person> adultsPeek = adults.peek(System.out::println);

// Weitere Filterung auf dem Stream vornehmen
final Stream<Person> allMikes = adultsPeek.filter(person ->
    person.getName().startsWith("Mi"));

// Löst die Verarbeitung aus
allMikes.forEach(System.out::println);
}
```

Nachfolgend zeige ich noch eine Alternative, die nach jedem Schritt der Pipeline eine Ausgabe vornimmt – nachfolgend jeweils fett markiert:

```
final Stream<Person> adults2 = persons.stream().filter(Person::isAdult);
final Stream<String> allMikes2 = adults2.filter(person ->
    person.getName().startsWith("Mi"))
    .peek(System.out::println)
    .map(Person::getName)
    .peek(System.out::println)
    .map(String::toUpperCase);

// Löst die Verarbeitung aus
allMikes2.forEach(System.out::println);
```

Startet man die beiden Programme nacheinander, so werden die Filtervorgänge wie folgt protokolliert:

```
Person [name = Michael / age = 43]
Person [name = Michael / age = 43]
Person [name = Micha / age = 42]
Person [name = Micha / age = 42]
Person [name = Merten / age = 38]
// -----
Person [name = Michael / age = 43]
Michael
MICHAEL
Person [name = Micha / age = 42]
Micha
MICHA
```

Man erkennt, dass die Elemente einzeln durch die Verarbeitungskette laufen. Zu diesen Ausgaben kommt es jedoch nur, weil in der letzten Zeile eine Terminal Operation ausgeführt wird. Das können Sie prüfen, wenn Sie die Zeilen mit `forEach()` auskommentieren: Dann werden Sie keine Ausgaben sehen.

Anhand der Ausgaben erkennt man auch, dass die Daten Element für Element durch die Intermediate Operations bearbeitet werden. Diese Art der Ausführung bildet die Grundlage für eine Parallelisierbarkeit. Außerdem wird der bereits erwähnte fundamentale Unterschied von Streams zu sonstigen Verarbeitungen klar, nämlich, dass Intermediate Operations wirklich erst dann abgearbeitet werden, wenn eine Terminal Operation eine Berechnung auslöst.

2.2.7 Zustandsbehaftete Intermediate Operations

Für die zustandsbehafteten Intermediate Operations lernen wir verschiedene Varianten kennen.

Ausgabe beschränken

Die zuvor gezeigte Filterung erlaubt es, den Datenbestand einzuschränken. Eine Variante davon ist es, die Ergebnismenge auf n Elemente zu beschränken. Dazu kann man einen Aufruf von `limit(long)` nutzen. In Kombination mit `skip(long)` zum Überspringen von n Datensätzen kann man sogenanntes *Paging* realisieren – eine Aufbereitung von n Ergebnissen auf einer Seite, wie man dies etwa von der Präsentation von Suchergebnissen im Internet kennt. Man kann die Ausgabe folgendermaßen auf 25 Sucheinträge ab dem 75. Eintrag beschränken:

```
searchResults.skip(75).limit(25);
```

Es gibt aber noch einen weiteren Anwendungsfall für `limit(long)` und `skip(long)` – nämlich im Zusammenhang mit unendlichen Streams, etwa einer unendlichen Folge von `int`-Werten:

```
final IntStream iteratingValues = IntStream.iterate(0, x -> x + 1);

iteratingValues.limit(10);           // => 0,1,2,3,4,5,6,7,8,9
iteratingValues.skip(50).limit(12);  // => 50,51,52,53,54,55,56,57,58,59,60,61
```

Im Beispiel sehen wir die Einschränkung eines unendlichen Streams auf bestimmte Datensätze, nämlich zunächst die 10 Einträge bzw. die folgenden 12 Einträge nach den ersten 50 Einträgen. Natürlich kann man diese Funktionalität auch für endliche Streams nutzen, etwa um für diese ein Paging zu realisieren.

Ausgabe sortieren, doppelte Elemente entfernen

Zwei weitere zustandsbehafteten Intermediate Operations sind das Sortieren und das Herausfiltern doppelter Einträge, nachfolgend für `sorted()` und `distinct()` gezeigt:

```
public static void main(final String[] args)
{
    final Stream<Integer> integers = Stream.of(7, 1, 4, 3, 7, 2, 6, 5, 7, 9, 8);

    final Stream<Integer> sortedAndDistinct = integers.sorted().distinct();

    final List<Integer> sorted = sortedAndDistinct.collect(Collectors.toList());
    System.out.println(sorted);
}
```

Führt man das Programm aus, so erhält man die erwartete Ausgabe:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Wenn Sie im Listing ganz genau hingeschaut haben, dann ist Ihnen vielleicht der Aufruf von `collect(Collectors.toList())` aufgefallen. Dabei handelt es sich wie beim zuvor mehrfach genutzten `forEach(Consumer<? super T>)` ebenfalls um eine weitere Terminal Operation, hier um eine, die die Daten aus einem Stream in eine Liste überträgt.

2.2.8 Terminal Operations

Bis hierher haben wir mithilfe von Streams verschiedene Berechnungen ausgeführt und auch schon das Öfteren mit `forEach(Consumer<T>)` eine Terminal Operation genutzt, um Konsolenausgaben zu produzieren. Betrachten wir Terminal Operations nun ein wenig allgemeiner. Erinnern wir uns zunächst nochmals daran, dass diese zur Abarbeitung der Pipeline führen und dadurch ein Ergebnis produzieren.

Verarbeitung und Konsolenausgaben mit `forEach()`

Nachfolgend ist zunächst der Vollständigkeit halber noch einmal die Aufbereitung von Konsolenausgaben mit `forEach()` gezeigt:

```
streamFromArray.forEach(System.out::println);
streamFromValues.sorted().distinct().forEach(System.out::println);
```

Bekanntermaßen kann man – nachdem die Berechnungsschritte innerhalb eines Streams abgeschlossen sind – über die Ergebnisse mithilfe von `forEach()` iterieren, z. B. um diese auszugeben. Für viele Anwendungsfälle ist es jedoch wünschenswert, die Daten aus dem Stream in eine Collection zu speichern.

Streams in Collections übertragen

Mithilfe sogenannter Collector-Instanzen kann man die Daten auslesen und in eine Liste übertragen. Praktischerweise existieren in der Utility-Klasse `Collectors` schon vordefinierte Typen, wie dies nachfolgend gezeigt ist:

```
final List<Integer> ages = agesStream.collect(Collectors.toList());
final List<String> names = namesStream.collect(Collectors.
                                         toCollection(ArrayList::new));
```

Im Listing sehen wir den für viele Anwendungsfälle praktischen Aufruf von `toList()`. Benötigt man mehr Kontrolle über den Typ der Ergebnisdatenstruktur, so kann man auch die Methode `toCollection()` aufrufen, der man die Referenz auf den Konstruktor der gewünschten Collection übergibt. Der benötigte Rückgabetyt wird wie schon die Typen der Eingabewert für Lambdas aus dem Kontext erschlossen. Hier entsteht also eine `ArrayList<String>`.

Berechnungen ausführen

Neben Konsolenausgaben oder der Übertragung in eine Collection stellen Berechnungen auf den Daten eine gebräuchliche Terminal Operation dar. Typisch sind etwa Berechnungen wie Minimum, Maximum, Summe oder Durchschnitt.

Nachfolgend berechnen wir die Summe aus allen Altersangaben einer Liste von Personen. Das Ergebnis ist vom Typ `int`. Wenn wir nun das Durchschnittsalter berechnen, nutzen wir dazu die Methode `average()`. Dabei gibt es zwei Dinge zu lernen. Zum einen ist das Ergebnis keine Ganzzahl mehr und zum anderen existiert möglicherweise kein Durchschnitt, nämlich dann, wenn es kein Element gibt. Um dies ausdrücken zu können, lernen wir ganz nebenbei noch die Klasse `OptionalDouble` kennen – ähnliche Klassen gibt es auch für andere primitive Typen.

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Michael", 43));
    persons.add(new Person("Lili", 34));
    persons.add(new Person("Yannis", 5));
    persons.add(new Person("Moritz", 22));

    // Summe berechnen
    final int sum = persons.stream().filter(person -> person.getName().
                                   startsWith("M")).
                       mapToInt(Person::getAge).sum();

    System.out.println("sum " + sum);

    // Durchschnitt berechnen
    final OptionalDouble avg = persons.stream().
                                   filter(person -> person.getName().
                                   startsWith("M")).
                                   mapToInt(Person::getAge).average();

    System.out.println("avg " + avg);
}
```

Hintergrundwissen: Verarbeitungsmethoden in Wrapper-Klassen

Damit verschiedene Stream-Operationen sich leichter formulieren lassen, wurden die Wrapper-Klassen `Integer`, `Long` und `Double` um verschiedene Methoden erweitert, insbesondere `min()`, `max()` und `sum()`. In der Klasse `Boolean` gibt es nun Methoden mit sprechendem Namen: `logicalAnd(boolean, boolean)`, `logicalOr(boolean, boolean)` sowie `logicalXor(boolean, boolean)`.

Besonderheit bei `parallelStream()` und `forEach()`

Wenn Sie eine Verarbeitung parallel mit `parallelStream()` ausgeführt haben, dann sind Ausgaben über `forEach()` problematisch, da die Reihenfolge der Abarbeitung eventuell nicht konform zu der Position in der Liste ist. Klingt komplizierter als es ist: Für das Sortieren kann man diesen Sachverhalt gut nachvollziehen. Schauen wir wieder auf ein einfaches Beispiel einer Liste von Namen:

```
public static void main(final String[] args)
{
    final String[] namesData = { "Stefan", "Ralph", "Andi", "Mike" };
    final List<String> names = Arrays.asList(namesData);

    names.parallelStream().sorted().forEach(System.out::println);
}
```

Das Programm produziert manchmal ungeordnete Ausgaben ähnlich zu folgender:

```
Ralph
Andi
Stefan
Mike
```

Führt man das Programm mehrmals aus, so erkennt man die zufällige Reihenfolge. Um die korrekte Reihenfolge bei Parallelverarbeitung sicherzustellen, muss man Aufrufe an die Methode `forEachOrdered()` nutzen:

```
public static void main(final String[] args)
{
    final String[] namesData = {"Stefan", "Ralph", "Andi", "Mike"};
    final List<String> names = Arrays.asList(namesData);

    names.parallelStream().sorted().forEachOrdered(System.out::println);
}
```

Allerdings sollte man dabei beachten, dass die dazu benötigte Abstimmung (Synchronisation) am Ende der Parallelverarbeitung doch einen größeren Teil des dadurch zuvor erzielten Gewinns »auffressen« kann.

Spezielle Terminal-Operations: `joining`, `groupingBy`, `partitioningBy`

Neben der Ausgabe auf der Konsole oder der Umwandlung der Daten eines Streams in eine Collection sind weitere Transformationen wünschenswert, etwa das Verknüpfen von Strings sowie die Gruppierung oder Partitionierung von Daten. Dazu bietet die Utility-Klasse `Collectors` verschiedene Hilfsmethoden. Nachfolgend wollen wir einen kurzen Blick auf folgende Methoden werfen:

- `joining()` – Fasst Einträge vom Typ `String` zusammen. Das ist nützlich, um etwa eine kommaseparierte Auflistung zu realisieren.
- `groupingBy()` – Nimmt eine Gruppierung anhand eines übergebenen Kriteriums vor.
- `counting()` – Zählt die Vorkommen in Kombination mit `groupingBy()`.
- `partitioningBy()` – Unterteilt die Eingabedaten basierend auf einer Realisierung eines `Predicate<T>` in zwei Partitionen.

Den Einsatz der obigen Methoden zeigt das folgende Listing, wobei hier zur besseren Lesbarkeit der Berechnungen statische Imports genutzt werden:

```
import static java.util.stream.Collectors.counting;
import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.Collectors.joining;
import static java.util.stream.Collectors.partitioningBy;

// ...

final List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike",
                                         "Florian", "Michael", "Sebastian");

String joined      = names.stream().sorted().collect(joining(", "));
Object grouped     = names.stream().collect(groupingBy(String::length));
Object counting    = names.stream().collect(groupingBy(String::length,
                                                         counting()));
Object partitions  = names.stream().filter(str -> str.contains("i")).collect(
    partitioningBy(str -> str.length() > 4));
```

Führen wir das Programm aus, so erhalten wir folgende Ausgaben, anhand derer sich die zuvor kurz beschriebene Arbeitsweise der Methoden erschließt:

```
joined:      Andi, Florian, Michael, Mike, Ralph, Sebastian, Stefan
grouped:     {4=[Andi, Mike], 5=[Ralph], 6=[Stefan], 7=[Florian, Michael],
              9=[Sebastian]}
counting:    {4=2, 5=1, 6=1, 7=2, 9=1}
partitions:  {false=[Andi, Mike], true=[Florian, Michael, Sebastian]}
```

2.2.9 Filter-Map-Reduce

Wir haben uns mittlerweile so viel Grundlagenwissen zu Streams erarbeitet, dass uns nun das Verständnis der mächtigen neuen Filter-Map-Reduce-Funktionalität – einer speziellen Untermenge des Stream-APIs – recht leicht fallen sollte.

Aufgabenstellung: Filtere eine Liste und extrahiere Daten

Nehmen wir an, unsere Aufgabe bestünde darin, eine Liste von Personen zu filtern, dabei alle im Juli Geborenen zu ermitteln und deren Namen kommasepariert auszugeben. Gegeben sei dazu folgende `List<Person>` als Eingabe:⁷

```
private static final List<Person> persons = Arrays.asList(
    new Person("Stefan", LocalDate.of(1971, MAY, 12)),
    new Person("Micha", LocalDate.of(1971, FEBRUARY, 7)),
    new Person("Andi Bubolz", LocalDate.of(1968, JULY, 17)),
    new Person("Andi Steffen", LocalDate.of(1970, JULY, 17)),
    new Person("Merten", LocalDate.of(1975, JUNE, 16)));
```

Die Aufgabe lässt sich in folgende drei Schritte untergliedern:

1. Filtere auf alle im Juli Geborenen

⁷Sehr aufmerksamen Leser fallen die hier genutzten Klassen bzw. Aufzählung `LocalDate` und `Month` und deren Konstanten, etwa `MAY` und `JULY`, auf. Diese sind neu im JDK 8 und werden später genauer erläutert.

2. Extrahiere ein Attribut, z.B. den Namen
3. Bereite eine kommaseparierte Liste auf

Bevor wir uns die mit JDK 8 bereitgestellte Filter-Map-Reduce-Funktionalität allerdings genau anschauen, werfen wir einen Blick darauf, wie man so etwas mit JDK 7 realisiert hätte.

Herkömmliche Realisierung

Der herkömmliche Ansatz besteht darin, die Funktionalität einzeln auszuprogrammieren. Der Übersichtlichkeit halber werden die einzelnen Schritte in Form kurzer Methoden realisiert. Zum besseren Verständnis beginnen wir mit der Implementierung einer `main()`-Methode, die folgende drei Schritte ausführt:

```
public static void main(final String[] args)
{
    // Schritt 1: Filtere
    final List<Person> bornInJuly = filterByMonth(persons, Month.JULY);

    // Schritt 2: Extrahiere
    final List<String> names = extractNameAttribute(bornInJuly);

    // Schritt 3: Bereite Ergebnis auf
    final String result = joinStrings(names, ", ");

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

Filtere auf alle im Juli Geborenen Wir konstruieren eine Ergebnisliste `bornInJuly` und fügen dort diejenigen `Person`-Objekte hinzu, die das Kriterium „Geboren im Juli“ erfüllen. Das realisieren wir folgendermaßen:

```
static List<Person> filterByMonth(final List<Person> persons, final Month month)
{
    final List<Person> filteredPersons = new ArrayList<>();
    for (final Person person : persons)
    {
        if (person.getBirthday().getMonth() == month)
        {
            filteredPersons.add(person);
        }
    }
    return filteredPersons;
}
```

Extrahiere ein Attribut Als zweiten Schritt nehmen wir eine *Extraktion* von Daten vor (man spricht auch von *Projektion*). Die Namen der Personen werden ausgelesen und als `List<String>` wie folgt bereitgestellt:

```
static List<String> extractNameAttribute(final List<Person> persons)
{
    ...
}
```

```

final List<String> names = new ArrayList<>();
for (final Person person : persons)
{
    names.add(person.getName());
}
return names;
}

```

Bereite eine kommaseparierte Liste auf Zur Aufbereitung der Ausgabe durchlaufen wir die als Parameter übergebene Liste und fügen jedes Element gefolgt von einem Komma (mit Ausnahme des letzten) in ein `StringBuilder`-Objekt per `append()` ein:

```

static String joinStrings(final List<String> names, final String delimiter)
{
    final StringBuilder sb = new StringBuilder();

    final Iterator<String> it = names.iterator();
    while (it.hasNext())
    {
        sb.append(it.next());
        if (it.hasNext())
        {
            sb.append(delimiter);
        }
    }

    return sb.toString();
}

```

Führen wir die realisierte Funktionalität – wie zuvor in `main()` gezeigt – aus, so werden die beiden im Juli geborenen Personen ausgegeben:

```

Andi Bubolz, Andi Steffen

```

Wenn man die Lösung betrachtet, so fällt negativ auf, dass diese recht lang ist. Erst bei etwas genauerem Überlegen bemerkt man einen weiteren Nachteil: Die Abarbeitung erfolgt sequentiell und die Laufzeit erhöht sich linear zu der Anzahl gespeicherter Personen. Als Alternative kann man die gesamte Funktionalität innerhalb nur einer Methode und ineinander verschränkt realisieren. Das wird zwar minimal schneller, jedoch deutlich unübersichtlicher – insbesondere, wenn die Abfragen komplexer werden. Auch kann man Erweiterungen kaum realisieren und die fehlende Kombinierbarkeit widerspricht dem Gedanken der Orthogonalität. Schauen wir jetzt auf eine mit Java 8 mögliche Variante.

Filter-Map-Reduce zur Verarbeitung von Daten mithilfe von Lambdas

Mit JDK 8 wird eine Filter-Map-Reduce-Funktionalität in Java bereitgestellt, die stark vom Einsatz von Lambdas profitiert. Dabei stehen die drei Begriffe Filter, Map und Reduce für die bereits im vorangegangenen Beispiel kennengelernten Aktionen:

- **Filter** – Aus einer Ausgangsmenge von Objekten werden diejenigen herausgefiltert, die gewünschten Anforderungen entsprechen.
- **Map** – Mit Mapping oder Projektion ist der Vorgang gemeint, der aus einem Objekt gewisse Informationen ableitet und diese in einer gewünschten Form aufbereitet. Map beschreibt eine Projektion, d. h. eine Transformation eines Elements in eine andere Repräsentation, wobei die Anzahl Elemente gleich bleibt.
- **Reduce** – Schlussendlich sollen die Berechnungsergebnisse verarbeitet werden, etwa auf der Konsole ausgegeben oder als Ergebnismenge in einer Collection aufbereitet werden. Reduce beschreibt demnach das Zusammenfassen auf ein Resultat (z. B. ein Minimum, Maximum, den Durchschnitt, die Summe oder eine Konsolenausgabe).

Diese Schritte sind in der nachfolgenden Abbildung visualisiert:

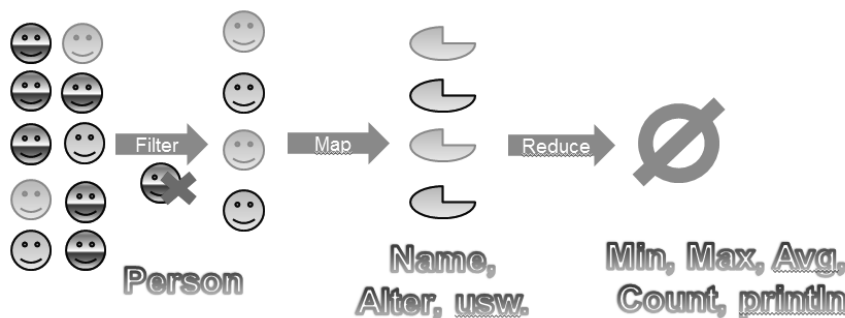


Abbildung 2-1 filterMapReduce

Filter-Map-Reduce im Einsatz Wir machen uns nun daran, die Schritte mit Filter-Map-Reduce und Lambdas zu realisieren. Die benötigten Grundlagen haben wir bereits bei der Besprechung der Intermediate Operations kennengelernt. Wir müssen dieses Wissen nur noch mit dem Aufruf von `reduce()` bzw. `collect()` kombinieren. Die gewünschte Filterung realisieren wir folgendermaßen:

```
final String bornInJuly = persons.stream().
    filter(person -> person.getBirthday()
                      .getMonth() == Month.JULY).
    map(Person::getName).
    reduce("", stringCombiner);
```

Die beiden Schritte Filter und Map sind intuitiv verständlich, beim Reduce-Schritt bedienen wir uns eines Tricks. Dort wird die nachfolgend gezeigte Realisierung namens `stringCombiner` vom Typ `BinaryOperator<T>` genutzt. Dieses funktionale Interface dient dazu, aus zwei Eingaben vom Typ `T` einen Ausgabewert vom Typ `T` zu be-

rechnen. Die Kombination zweier Strings können wir wiederum mithilfe eines Lambdas wie folgt implementieren:

```
final BinaryOperator<String> stringCombiner = (str1, str2) ->
{
    if (str1.isEmpty())
    {
        return str2;
    }
    else
    {
        return str1 + ", " + str2;
    }
};
```

Beim Betrachten dieses Lambdas erinnern Sie sich vielleicht an meinen Tipp, den ich vor einigen Seiten gegeben habe, Lambdas lediglich für solche Dinge zu nutzen, deren Implementierung nur wenige Zeilen Sourcecode umfasst. Hier ist das schon grenzwertig. Natürlich könnte man hier auch einen ternären Ausdruck nutzen, aber auch der ist schon etwas unleserlich:

```
final BinaryOperator<String> stringCombiner = (str1, str2) ->
{
    return str1.isEmpty() ? str2 : str1 + ", " + str2;
};
```

Zusammenfassen von Werten mit Collectors Weil das Zusammenfassen von Einträgen ein recht gebräuchlicher Anwendungsfall ist, bietet das JDK 8 praktischerweise eine Spezialisierung einer Reduce-Methode namens `collect()`. Vorgefertigte Implementierungen finden sich in der Utility-Klasse `Collectors`. Einige davon haben wir schon zuvor kennengelernt. Wir wählen hier `joining(String)` zur Kombination von Strings.

```
final String bornInJuly = persons.stream().
    filter(person ->
        person.birthday.getMonth() == Month.JULY)
    .map(person -> person.name)
    .collect(Collectors.joining(", "));
```

Man erkennt sehr schön, dass die gewählte Form mehr am zu lösenden Problem ausgerichtet ist und nicht ein spezieller Algorithmus zum Filtern programmiert wird. Außerdem versteckt `joining()` die Spezialbehandlung der korrekten Aufbereitung einer kommaseparierten Ausgabe.

Hinweis: Die Methode `String.join()`

Wenn tatsächlich nur textuelle Werte miteinander verknüpft werden sollen, dann kann man die in der Klasse `String` mit JDK 8 neu eingeführte Methode `join(CharSequence delimiter, CharSequence... elements)` nutzen:^a

```
final String stringConcat = String.join(" ", names);
```

Etwas unglücklich empfinde ich die Signatur, in der der Delimiter vor den zu verknüpfenden Elementen angegeben werden muss. Das liegt aber einfach daran, dass in Java Varargs nur für den letzten Parameter in einer Signatur auftreten dürfen.

^aWir haben zuvor mit `joinStrings()` eine ähnliche Methode entworfen.

Aufbereitung/Verknüpfung nicht textueller Werte Gerade haben wir gesehen, wie einfach die kommaseparierte Aufbereitung von textuellen Werten mithilfe von Lambdas und den Neuerungen im Stream-API geschrieben werden kann.

Etwas mehr Aufwand muss man betreiben, wenn man Zahlen oder Objekte auf diese Weise miteinander verknüpfen möchte. Nehmen wir an, wir wollten die Altersangaben der Personen kommasepariert aufbereiten. Dazu müssen wir die Objekte, hier vom Typ `Integer`, in Strings wandeln, bevor wir sie mit `joining()` verknüpfen können. Die Umwandlung kann explizit durch einen Aufruf von `toString()` erfolgen, oder aber implizit durch die Notation `" " + object`. Für diese Varianten schreiben wir Folgendes:

```
// Explizite Umwandlung / Mapping mit toString
final String joined1 = persons.stream().mapToInt(Person::getAge)
                                .mapToObj(Integer::toString)
                                .collect(joining(" ", " "));

// Implizite Umwandlung / Mapping durch " " + value
final String joined2 = persons.stream().map( (person) -> " " + person.getAge() )
                                .collect(joining(" ", " "));
```

Besonderheiten

Eingangs erwähnte ich, dass Streams als sequentielle oder parallele Variante erzeugt werden können. Besonders interessant ist, dass man sich zu Beginn der Verarbeitung nicht darauf festlegen muss, wie alle Schritte abgearbeitet werden sollen. Es ist vielmehr möglich, beliebig zwischen paralleler und sequentieller Abarbeitung hin und her zu schalten, wie es das nachfolgende Beispiel zeigt:

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

    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Micha", 42));
    persons.add(new Person("Barbara", 40));
    persons.add(new Person("Andy", 30));
```

```

persons.add(new Person("Yannis", 5));
persons.add(new Person("Leo", 7));

final String adults = persons.parallelStream().filter(Person::isAdult())
                             .sequential().map(Person::getName)
                             .collect(Collectors.joining(", "));

System.out.println(adults);
}

```

An diesem Beispiel erahnt man die Möglichkeiten, die sich durch die neuen Sprachfeatures ergeben. Besonders praktisch ist, dass die dahinter stehende Komplexität für den Entwickler verborgen bleibt. Man muss sich somit nicht um Details der Multithreading-Verarbeitung kümmern, sondern beschreibt die Abläufe auf einer höheren Abstraktionsebene. Allerdings muss man bei Bedarf explizit auf Parallelverarbeitung umschalten.

Anmerkung: Sinnhaftigkeit der »Umschalterei«

Bereits bei meinen Ausführungen zur Methode `forEachOrdered()` habe ich angedeutet, dass eine Umschaltung von Parallelverarbeitung auf eine anschließend sequentielle Weiterverarbeitung möglicherweise ungünstig sein kann: Die zuvor durch die Parallelverarbeitung erzielten Performance-Vorteile können so teilweise wieder zunichte gemacht werden. Ein wiederholtes Umschalten zwischen paralleler und sequentieller Abarbeitung ist daher wenig sinnvoll.

2.2.10 Fallstricke Lambdas und funktionale Programmierung

In den vorangegangenen Abschnitten haben wir gesehen, wieviel Positives durch den Einsatz von Lambdas in Kombination mit den Bulk Data Operations on Collections und der Filter-Map-Reduce-Funktionalität erzielbar ist.

Nachfolgend sollen aber zwei mögliche Probleme kurz thematisiert werden: Zum einen ist mit dem Filter-Map-Reduce-Framework die Lernkurve steiler geworden und Java ist an einigen Stellen recht komplex, da eine Vielzahl von Interfaces neu eingeführt wurde, deren Realisierung nicht immer ganz so intuitiv ist. Zum anderen stellt die funktionale Programmierung auch Ansprüche an die Fähigkeiten der Entwickler, das Paradigma richtig umzusetzen. Ungünstig ist es, wenn man versucht, Algorithmen auf herkömmliche Art und Weise auszuformulieren.

Java-Fehlermeldungen werden zu komplex

Wenn man anstelle von Lambdas versucht, die neuen Funktionalitäten in Form von Realisierungen der benötigten Interfaces zu nutzen, stösst man recht schnell auf Probleme und man erhält kaum verständliche Fehlermeldungen. Wir schauen hier zur Demonstration lediglich auf ein kurzes Beispiel und überlassen jedem Entwickler dann selbst die Meinungsbildung.

Nachfolgend betrachten wir wieder einige Personen, realisieren ein Mapping von Person auf Alter als `Function<Person, Integer>` und versuchen dann das Durchschnittsalter mit einer (versehentlich bzw. hier zu Demonstrationszwecken) nicht ganz passenden `ToIntFunction<Double>` zu berechnen:

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Micha", 42));
    // ...

    // Mapping auf Alter
    final Stream<Integer> agesStream = persons.stream().
        map(new Function<Person, Integer>()
        {
            @Override
            public Integer apply(final Person person)
            {
                return person.getAge();
            }
        });

    // Durchschnittsberechnung
    final int averageAge = agesStream.collect(Collectors.
        averagingInt(new ToIntFunction<Double>()
        {
            @Override
            public int applyAsInt(final Double value)
            {
                return value.intValue();
            }
        }));
}
```

Wenn man das Programm kompiliert (per `javac` oder in Netbeans), so bekommt man eine Fehlermeldung, die man erstmal verdauen und vor allem verstehen muss. Das gestaltet sich allein schon aufgrund der schieren Länge als eine Herausforderung:

```
no suitable method found for collect(Collector<Double,CAP#1,Double>)
  method Stream.<R#1>collect(Supplier<R#1>,BiConsumer<R#1,? super Integer>,
    BiConsumer<R#1,R#1>) is not applicable
    (cannot infer type-variable(s) R#1
      (actual and formal argument lists differ in length))
  method Stream.<R#2,A>collect(Collector<? super Integer,A,R#2>) is not
    applicable
    (inferred type does not conform to upper bound(s)
      inferred: Integer
      upper bound(s): Double,Object)
where R#1,T,R#2,A are type-variables:
  R#1 extends Object declared in method <R#1>collect(Supplier<R#1>,BiConsumer<R
    #1,? super T>,BiConsumer<R#1,R#1>)
  T extends Object declared in interface Stream
  R#2 extends Object declared in method <R#2,A>collect(Collector<? super T,A,R
    #2>)
  A extends Object declared in method <R#2,A>collect(Collector<? super T,A,R#2>)
where CAP#1 is a fresh type-variable:
  CAP#1 extends Object from capture of ?
```

Die von Eclipse produzierte Fehlermeldung ist etwas kürzer:

```
The method collect(Collector<? super Integer,A,R>) in the type Stream<Integer>
is not applicable for the arguments (Collector<Double,capture#1-of ?,Double>)
```

Wie schon bei Intermediate Operations und Berechnungen besprochen, existieren bei der Berechnung von Durchschnittswerten zwei Besonderheiten. Zum einen kann für den Fall eines leeren Streams kein Durchschnitt ermittelt werden. Zum anderen lässt sich der Durchschnitt von Integer- oder Long-Werten nur durch einen Gleitkommatyp, etwa Double, darstellen. Hier hätte man einfach einen anderen Collector sowie einen anderen Ergebnistyp wählen müssen, was aber nur Experten anhand der Fehlermeldung erraten können. Wie folgt ließe sich das Ganze korrigieren:

```
final Double averageAge = agesStream.collect(Collectors.averagingDouble(
    new ToDoubleFunction<Integer>()
{
    @Override
    public double applyAsDouble(Integer value)
    {
        return value.doubleValue();
    }
}
);
```

Leider stoßen wir hier auf eine Inkonsistenz, denn die Optionalität wird hier durch 0 ausgedrückt. Würde man dagegen für einen reinen Double-Stream den Durchschnitt berechnen, so würde ein OptionalDouble geliefert, der bei einer leeren Eingabe (keine Daten im Stream) OptionalDouble.empty als Ergebnis liefert:

```
final double[] ages = { 42, 40, 30, 5, 7};
final OptionalDouble avgAge = Arrays.stream(ages).average();
```

Fallstrick: Imperative Lösung 1 zu 1 funktional umsetzen

Jedes Programmierparadigma hat seine spezifischen Stärken und Schwächen. Wenn man aber zwei Formen miteinander mischt, so besteht auch immer die Gefahr, die Schwächen zu kombinieren. Das nachfolgend präsentierte, prägnante Beispiel stammt aus einem Blog⁸ von Johannes Weigend. Nach Rücksprache mit ihm habe ich das Ganze leicht modifiziert und freue mich, es hier präsentieren zu dürfen.

Schauen wir auf ein Programmstück, das imperativ eine Menge von Personen abläuft und für alle Personen namens »TheOne« die Methode action(Person) aufruft.

```
for (int i = 0; i < persons.size(); i++)
{
    final Person person = persons.get(i);
    if (person.getName().equals("TheOne"))
    {
        action(person);
    }
}
```

⁸<http://qaware.blogspot.ch/2013/09/lambda-and-streams-in-jdk-8.html>

Die gewünschte Funktionalität ist wie prädestiniert für den Einsatz von Filter-Map-Reduce. Wenn man nun aber, ohne funktionale Programmierung verinnerlicht zu haben, versucht, den Algorithmus mithilfe funktionaler Konstrukte imperativ nachzuprogrammieren, könnte dabei etwa Folgendes herauskommen:

```
IntStream.range(0, persons.size()).mapToObj(persons::get)
    .filter(person -> person.getName().equals("TheOne"))
    .forEach(person -> action(person));
```

Man sieht, dass hier nicht das zu lösende Problem im Mittelpunkt steht, sondern eine 1:1-Umsetzung des Algorithmus: Die `for`-Schleife wird durch `IntStream.range()` und das indizierte Auslesen durch `mapToObj()` abgebildet. Ein solches Vorgehen ist wenig sinnvoll. Schauen wir also auf die Korrektur: Mit dem bisher aufgebauten Wissen können Sie das sicher schon selbst. Wahrscheinlich würden Sie durch Nutzung eines funktionalen Stils die obige Funktionalität wie folgt realisieren:

```
persons.stream().filter(person -> person.getName().equals("TheOne"))
    .forEach(person -> action(person))
```

Fazit

Auch wenn ich zum Abschluss noch auf zwei mögliche Probleme beim Einsatz von Lambdas, dem Filter-Map-Reduce-Framework sowie der funktionalen Programmierung eingegangen bin, so überwiegen doch die positiven Seiten. Nunmehr ist es endlich möglich, Algorithmen bei Bedarf parallel auszuführen und dabei von den Multicores heutiger Prozessoren zu profitieren. Außerdem lassen sich viele Aufgabenstellungen näher am zu lösenden Problem beschreiben, als dies mit imperativer Ausformulierung eines Algorithmus möglich ist – insbesondere durch den Wegfall der anonymen Klassenrümpfe.

2.3 JSR 310: Date and Time API

In diesem Unterkapitel beschreibe ich das im Rahmen von JSR-310 erarbeitete neue Date and Time API. Zunächst stelle ich dar, wieso es notwendig wurde, einen dritten Wurf zur Datumsverarbeitung zu entwickeln, bevor ich dann auf das neue API eingehe.

2.3.1 Datumsverarbeitung vor JSR-310: Ein Blick zurück

Die Verarbeitung von Datumswerten und Zeitangaben scheint einfach, ist es aber nicht. Tatsächlich ist es sogar ziemlich kompliziert, weil verschiedene Dinge zu beachten sind, etwa der Einfluss von Zeitzonen, Schaltjahren sowie Sommer- und Winterzeit.

Beispielsweise kann man durch Aufruf der Methode `System.currentTimeMillis()` eine Zeitangabe in Form eines `long`-Werts erhalten, der die Anzahl der

seit dem 1.1.1970 vergangenen Millisekunden darstellt. Häufig benötigt man aber eine objektorientiertere Sichtweise und eine bessere Abstraktion. Das gilt etwa, wenn man Berechnungen anstellen möchte wie „Gehe einen Monat in die Vergangenheit oder Zukunft“. So etwas in Millisekunden zu berechnen, ist kompliziert, da gegebenenfalls Schaltjahre und verschiedene Längen von Monaten zu berücksichtigen sind. Mit der Klasse `java.util.Date` lassen sich derartige Berechnungen nur unzureichend abbilden, weil nur eine minimale Abstraktion eines `long` geboten wird. Zudem lauern in der Klasse `Date` einige Fallstricke, wodurch eine Vielzahl der dort definierten Methoden als `deprecated` gekennzeichnet wurden und nicht mehr verwendet werden sollten. Im JDK gibt es auch noch die Klasse `java.util.Calendar`. Diese enthält zwar weniger Fallstricke und bietet eine bessere Abstraktion (Konstanten für Monate, Addition von Zeitwerten usw.), was die Verarbeitung und vor allem Berechnungen erleichtert, allerdings ist Einiges noch immer ziemlich kompliziert, insbesondere, wenn man statt mit Datum und Uhrzeit in Kombination nur mit Zeitangaben oder Datumswerten rechnen möchte. Für diese Fälle wird dann auch `Calendar` recht unhandlich. Folgende Aufzählung nennt einige weitere Probleme der alten Datums-APIs:

- Die Klassen `Date`, `Calendar`, `SimpleDateFormat` und `DateFormatter` sind veränderlich. Im Zusammenhang mit Multithreading kann es dadurch schnell zu Inkonsistenzen kommen.
- Von der Klasse `Date` aus dem Package `java.util` sind die Klassen `Date`, `Time` und `Timestamp` aus dem Package `java.sql` abgeleitet. Diese Subklassen stellen aber semantisch keine wirklichen Subtypen dar, weil sie jeweils nur einen speziellen Aspekt eines Datumswerts beschreiben.
- Es existieren verschiedene Probleme bei der Ausgabe, etwa die fehlende Möglichkeit, `Calendar`-Objekte formatiert auszugeben.

Wie bereits an der zuvor geführten Diskussion ersichtlich und detailliert in Kapitel 9 angesprochen, ist die Verarbeitung von Datumswerten in Java recht umständlich und weder die Klasse `Date` noch das Interface `Calendar` bieten richtig gelungene APIs zur Datumsverwaltung. Zwar sind diese Probleme schon seit Längerem bekannt, doch seit JDK 1.2 gab es keine Neuerungen mehr, obwohl bereits seit 2007 in Form des JSR-310 an einer Neuentwicklung der Datumsverarbeitung gearbeitet wird. Doch erst mit Java 8 findet diese Ergänzung Einzug ins JDK. Die Zielsetzung von JSR-310 ist, alles besser und einfacher nutzbar zu machen und ein gelungenes, hilfreiches API zur Verwaltung und zur Manipulation von Datums- und Zeitwerten bereitzustellen.

Aufgrund der Unzulänglichkeiten der bisher verfügbaren Implementierungen zur Datumsverarbeitung haben viele größere Projekte vermutlich zunächst eigene kleinere Hilfsmethoden oder Utility-Klassen geschrieben. So kam es wohl auch, dass die Bibliothek `Joda-Time` entwickelt wurde. Sie hat sich mittlerweile als Defacto-Standard etabliert, weil sich durch deren Nutzung die Datumsverarbeitung stark vereinfacht. Da in `Joda-Time` alle Klassen `immutable` und damit Thread-sicher sind, ist deren Einsatz auch im Multithreading-Kontext unkritisch.

Die im Rahmen von JSR-310 entwickelten Klassen versuchen, die Probleme mit den bisherigen Datums-APIs des JDKs zu adressieren und nutzen vor allem Ideen aus der Bibliothek Joda-Time, deren Schöpfer auch eine führende Rolle bei der Entwicklung von JSR-310 inne hatte.

2.3.2 Überblick über die neu eingeführten Klassen

Das durch JSR-310 realisierte neue Date And Time API fügt dem JDK einige Funktionalität in fünf Packages unter `java.time` hinzu. Dabei unterscheidet man grob zwei Konzepte.

Zum einen ist dies die kontinuierliche oder Maschinen-Zeit bei der durch die Klasse `java.time.Instant` ein spezieller Zeitpunkt repräsentiert wird. Das ist näherungsweise mit der Intention der Klasse `Date` vergleichbar. Im Unterschied dazu wird jedoch eine Auflösung im Bereich von Nanosekunden geboten. Der Referenzzeitpunkt ist, wie bei `Date`, der 1. Januar 1970.

Zum anderen existieren Datumsklassen, die eher an menschlichen Denkweisen ausgerichtet sind: Die Klassen `LocalDate` und `LocalTime` aus dem Package `java.time` repräsentieren Datumswerte ohne Zeitzonen in Form eines Datums bzw. einer Zeit. Beide modellieren jeweils nur die durch den Klassennamen beschriebene Zeitkomponente, also Datum oder Zeit.

Wir werden nachfolgend diverse neue Klassen anhand kurzer Beispiele kennenlernen. Dabei werden zur Objektkonstruktion in der Regel folgende Varianten genutzt:

- `now()` – Datumswert basierend auf der aktuellen Zeit
- `of()`-Methoden – teilweise spezielle Methoden `ofDays()`, `ofMonths()`
- `parse()` – Parsing von textuellen Angaben

Die Klasse `Instant`

Eine Instanz vom Typ `java.time.Instant` repräsentiert einen Zeitpunkt in Nanosekunden in Bezug auf den Referenzzeitpunkt 1.1.1970 00:00:00 Uhr. Die Zeit schreitet dabei linear voran und diese Modellierung vereinfacht die Verarbeitung durch Computer, da keine Spezialfälle zu betrachten sind.

Im nachfolgenden Beispiel modellieren wir Abfahrts- und Ankunftszeiten einer Reise mit der Dauer von 5 Stunden (etwa mit der Bahn), die zum jetzigen Zeitpunkt beginnt. Diesen Startzeitpunkt ermitteln wir durch den Aufruf von `now()`. Die resultierende Ankunftszeit wird als `expectedArrivalTime` berechnet. Außerdem nehmen wir eine Verspätung von 7 Minuten an. Auf zweierlei Art wird daraus die reale Ankunftszeit wie folgt berechnet:

```
public static void main(final String[] args)
{
    // Abfahrt jetzt und Reisedauer 5 Stunden
    final Instant departureTime = Instant.now();
    final Instant expectedArrivalTime = departureTime.plus(5, ChronoUnit.HOURS);
}
```

```
// Verspätung von 7 Minuten auf zwei Arten berechnen
final Instant realArrival = expectedArrivalTime.plus(7, ChronoUnit.MINUTES);
final Instant realArrival2 = expectedArrivalTime.plus(Duration.ofMinutes(7));

System.out.println(departureTime);           // 2014-03-22T13:54:50.818Z
System.out.println(expectedArrivalTime);     // 2014-03-22T18:54:50.818Z
System.out.println(realArrival);             // 2014-03-22T19:01:50.818Z
System.out.println(realArrival2);           // 2014-03-22T19:01:50.818Z
}
```

Die Aufzählung ChronoUnit

Im vorherigen Beispiel haben wir die Aufzählung `java.time.temporal.ChronoUnit` und die Klasse `java.time.Duration` genutzt, um Zeitdauern zu spezifizieren. `ChronoUnit` ist eine Aufzählung, die all diejenigen Zeiteinheiten definiert, mit denen im Date And Time API gerechnet werden kann. In `ChronoUnit` findet man Definitionen unter anderem für Minuten, Stunden, Wochen usw. Tatsächlich sind dort Konstanten für Nanosekunden bis hin zu Jahrtausenden, sowie Äras und einer speziellen `FOREVER`-Konstante definiert.

Greifen wir das obige Beispiel auf: Wir nutzen wieder Instanzen von `ChronoUnit`, um die Zeitdauer in verschiedenen Varianten (Stunden und Minuten) darzustellen. Eine wichtige Eigenschaft ist, dass man mithilfe der Methode `between()`, die Differenz zwischen zwei Zeitpunkten in Form von `Instant`-Objekten bestimmen kann, wie wir dies nachfolgend sehen:

```
public static void main(final String[] args)
{
    // Abfahrt jetzt und Reisedauer 5 Stunden
    final Instant departureTime = Instant.now();
    final Instant arrivalTime = departureTime.plus(5, ChronoUnit.HOURS);

    System.out.println("departure now: " + departureTime);
    System.out.println("arrival now + 5h: " + arrivalTime);

    // Berechnungen durchführen: Differenz bilden
    final long inBetweenHours = ChronoUnit.HOURS.between(departureTime,
                                                         arrivalTime);
    final long inBetweenMinutes = ChronoUnit.MINUTES.between(departureTime,
                                                            arrivalTime);

    System.out.println("inBetweenHours: " + inBetweenHours);
    System.out.println("inBetweenMinutes: " + inBetweenMinutes);
}
```

Führt man die obigen Programmzeilen aus, so erhält man in etwa folgende Ausgaben auf der Konsole, die sehr schön die Berechnungen von 5 Stunden in die Zukunft sowie die Differenzbildung zwischen zwei Zeitpunkten in verschiedenen Zeiteinheiten (Stunden und Minuten) zeigen:

```
departure now:      2014-02-19T22:13:50.691Z
arrival now + 5h: 2014-02-20T03:13:50.691Z
inBetweenHours:     5
```

```
inBetweenMinutes: 300
```

Die Klasse Duration

Mit der Klasse `java.time.Duration` kann man eine Zeitdauer in Nanosekunden exakt festlegen, etwa um Differenzen zwischen zwei `Instant`-Objekten ausdrücken. Instanzen der Klasse `Duration` können durch Aufruf verschiedener Methoden konstruiert werden, wobei die Konstruktion aus übergebenen Werten verschiedener Zeiteinheiten⁹ oder aber aus der Differenz zweier `Instant`-Objekte möglich ist:

```
public static void main(final String[] args)
{
    // Erzeugung
    final Duration durationFromSecs = Duration.ofSeconds(15);
    final Duration durationFromMinutes = Duration.ofMinutes(30);
    final Duration durationFromHours = Duration.ofHours(45);
    final Duration durationFromDays = Duration.ofDays(60);

    System.out.println("From Secs:      " + durationFromSecs);
    System.out.println("From Minutes:  " + durationFromMinutes);
    System.out.println("From Hours:    " + durationFromHours);
    System.out.println("From Days:     " + durationFromDays);

    // Berechnungen
    final Instant now = Instant.now();
    final Instant silvester2013 = Instant.parse("2013-12-31T00:00:00Z");
    final Instant myBirthday2015 = Instant.parse("2015-02-07T00:00:00Z");
    final Duration duration1 = Duration.between(now, silvester2013);
    final Duration duration2 = Duration.between(now, myBirthday2015);

    System.out.println(now + " -- " + silvester2013 + ": " + duration1);
    System.out.println(now + " -- " + myBirthday2015 + ": " + duration2);
}
```

Führen wir das Programm `DURATIONEXAMPLE` aus, so kommt es zu folgenden Ausgaben, wobei insbesondere zwei Dinge von Interesse sind: Zum einen werden Zeitdifferenzen maximal in der Zeiteinheit von Stunden abgebildet und zum anderen wird ein Sprung in die Vergangenheit bzw. in die Zukunft gezeigt:

```
From Secs:      PT15S
From Minutes:   PT30M
From Hours:     PT45H
From Days:      PT1440H
2014-03-23T11:22:54.648Z -- 2013-12-31T00:00:00Z: PT-1979H-22M-54.648S
2014-03-23T11:22:54.648Z -- 2015-02-07T00:00:00Z: PT7692H37M5.352S
```

Weil das Ganze alles recht intuitiv erscheint, es aber Fallstricke gibt, wollen wir das Thema Berechnungen noch ein wenig genauer betrachten. Neben der zuvor gezeigten Differenzberechnung mit `between()` ist auch eine Addition einer durch eine `Duration` definierte Zeitspanne zu einem `Instant`-Objekt möglich. Man erhält als Ergebnis wiederum ein `Instant`-Objekt.

⁹Zeiteinheiten mit variabler Länge, wie Monate, werden nicht unterstützt.

Die Addition von Zeitspannen betrachten wir an folgendem Beispiel. Ausgehend vom 24.12.2003 soll eine Woche in die Zukunft zum Silvester-Tag 2013 gesprungen werden. Anschließend führen wir einen größeren Zeitsprung zum 18.3.2014, dem Releasedatum von JDK 8, aus. Wir schreiben dazu folgendes Programm:

```
public static void main(final String[] args)
{
    // Erzeugung
    final Instant christmas2013 = Instant.parse("2013-12-24T00:00:00Z");
    final Instant silvester2013 = Instant.parse("2013-12-31T00:00:00Z");
    final Instant jdk8Release = Instant.parse("2014-03-18T00:00:00Z");

    // Vergleichswerte errechnen
    System.out.println(Duration.between(christmas2013, silvester2013));
    System.out.println(Duration.between(silvester2013, jdk8Release));

    // Berechnungen
    final Instant calcSilvester_1 = christmas2013.plus(Duration.ofDays(7));
    final Instant calcSilvester_2 = christmas2013.plus(7, ChronoUnit.DAYS);

    System.out.println(calcSilvester_1);
    System.out.println(calcSilvester_2);

    // Problematische Berechnungen

    // kein Duration.ofWeeks(long) oder ofMonths(long)
    final Instant calcSilvester_3 = christmas2013.plus(1, ChronoUnit.WEEKS);
    final Instant calcJdk8Release = silvester2013.plus(3, ChronoUnit.MONTHS).
                                                plus(Duration.ofDays(18));

    System.out.println(calcSilvester_3);
    System.out.println(calcJdk8Release);
}
```

Während die fehlende Bereitstellung einer Methode von `ofWeeks(long)` sich noch recht gut durch eigene Berechnungen und der Nutzung von `ofDays(long)` realisieren lässt, wird dies für Monate ohne `ofMonths(long)` schwieriger. Das wirkt umständlich und man entdeckt möglicherweise die Methode `plus(long, TemporalUnit)`. Diese scheint für unsere Berechnungen sehr praktisch, um Wochen oder Monate in die Zukunft zu springen. Setzen wir diese Methode einfach einmal ein. Wenn Sie das obige Programm `DURATIONSSPECIALEXAMPLE` ausführen, werden jedoch statt der gewünschten Berechnungen Exceptions folgender Form ausgelöst:

```
Exception in thread "main" java.time.temporal.UnsupportedTemporalTypeException:
    Unsupported unit: Weeks
    at java.time.Instant.plus(Instant.java:867)
```

Zur Definition einer `Duration` können keine Zeiteinheiten genutzt werden, die sich nicht präzise durch Stunden, Minuten usw. ausdrücken lassen. Man könnte sich fragen: Wir haben doch aber eine `Duration` für die gewünschten Zeiträume basierend auf `Instant`s berechnen können. Wieso war das möglich? Die Antwort ist ganz einfach: Weil wir hier fixe Werte vorliegen haben und somit die Differenz dazwischen eindeutig zu bestimmen war. Die abstrakte Angabe von einer Woche oder einem Monat besitzt kein exaktes Äquivalent in Form einer fixen Zeitspanne. Hier steht die Modellierung

in Maschinenzeit in Konflikt mit der komplexeren Wirklichkeit. Später werden wir als Abhilfe die Klasse `Period` kennenlernen.

LocalDate, LocalTime und LocalDateTime

Die Klasse `java.time.LocalDate` repräsentiert eine Datumsangabe ohne Zeitinformationen, also eine Kombination aus Jahr, Monat und Tag, z. B. den 18. März als das Releasedatum von Java 8: 2014-18-03. Mithilfe der Klasse `java.time.LocalTime` wird eine Zeitangabe ohne Datumsangabe modelliert, z. B. 18:00:00.000. Schließlich ist die Klasse `java.time.LocalDateTime` eine Kombination aus beiden und würde wie folgt dargestellt: 2014-18-03T18:00:00.000.

```
public static void main(final String[] args)
{
    final LocalDate michasBirthday = LocalDate.of(1971, Month.FEBRUARY, 7);
    final LocalDate barbarasBirthday = michasBirthday.plusDays(17).
                                                    plusMonths(1).plusYears(2);
    final LocalDate lastDayInFebruary = michasBirthday.with(TemporalAdjusters.
                                                            lastDayOfMonth());

    System.out.println("michasBirthday:    " + michasBirthday);
    System.out.println("barbarasBirthday:  " + barbarasBirthday);
    System.out.println("lastDayInFebruary: " + lastDayInFebruary);

    final LocalTime atTen = LocalTime.of(10,00,00);
    final LocalTime tenFifteen = atTen.plusMinutes(15);
    final LocalTime breakfastTime = tenFifteen.minusHours(2);

    System.out.println("atTen:              " + atTen);
    System.out.println("tenFifteen:         " + tenFifteen);
    System.out.println("breakfastTime:      " + breakfastTime);

    final LocalDateTime jdk8Release = LocalDateTime.of(2014, 3, 18, 8, 30);
    System.out.println("jdk8Release:        " + jdk8Release);
    System.out.printf("jdk8Release:  %s.%s.%s\n", jdk8Release.getDayOfMonth(),
                                                            jdk8Release.getMonthValue(),
                                                            jdk8Release.getYear());
}
```

Im Listing sehen wir verschiedene Berechnungen mithilfe verschiedener `plusXYZ()`- sowie `minusXYZ()`-Methoden. Darüber hinaus haben wir die Utility-Klasse `TemporalAdjusters` genutzt, in der verschiedene Hilfsmethoden definiert sind. Dies ist etwa die Methode `lastDayOfMonth()` zur Berechnung des letzten Tags im Monat. Damit berechnen wir im Beispiel den letzten Tag des Monats Februar im Jahr 1971. Führt man das Programm aus, so kommt es zu folgenden Konsolenausgaben:

```
michasBirthday:    1971-02-07
barbarasBirthday:  1973-03-24
lastDayInFebruary: 1971-02-28
atTen:             10:00
tenFifteen:        10:15
breakfastTime:     08:15
jdk8Release:       2014-03-18T08:30
jdk8Release:       18.3.2014
```

Die Aufzählungen `DayOfWeek` und `Month`

Sowohl `java.time.DayOfWeek` als auch `java.time.Month` sind Aufzählungen, deren Einsatz einerseits den Sourcecode lesbarer macht und andererseits auch einfache Fehler vermeidet, weil dann typsichere Konstanten Verwendung finden. Bei Anwendung der alten APIs konnten durch nicht konsistente 0- und 1-basierte Angaben Probleme entstehen. Dass sich der Einsatz von Konstanten vorteilhaft auswirken kann, haben wir im vorangegangenen Beispiel für die Angabe des Monats Februar bereits ansatzweise kennengelernt. Im `Calendar`-API wusste man – ohne Blick in das Javadoc der API – nie so genau, ob Februar nun dem Wert 1 oder 2 entsprach.

Neben der Typsicherheit bieten die neuen Aufzählungstypen den Vorteil, dass man Berechnungen mit ihnen durchführen kann. Nachfolgend demonstriere ich dies, indem ich zu einem Sonntag 5 Tage hinzu addiere und zum Februar 13 Monate. Wie erwartet, landet man an einem Freitag bzw. im März:

```
public static void main(final String[] args)
{
    final DayOfWeek sunday = DayOfWeek.SUNDAY;
    final Month february = Month.FEBRUARY;

    System.out.println(sunday.plus(5));    // FRIDAY
    System.out.println(february.plus(13)); // MARCH
}
```

Die Klassen `YearMonth`, `MonthDay` und `Year`

Die zuvor beschriebenen `Instant`-Objekte sind zur Modellierung von wiederkehrenden Datumswerten, etwa Geburtstagen oder anderen Jahrestagen, nicht besonders gut geeignet. Das liegt daran, dass man hierzu »unvollständige Zeitangaben« benötigt, etwa Datumsangaben ohne Uhrzeit, Zeitangaben ohne Datum oder Datumsangaben ohne Jahresangaben. Derartige Angaben erscheinen zunächst unpräzise, das entspricht aber eher der menschlichen Denkweise.

Wie eingangs erwähnt, hat die Darstellung von Zeitangaben in Millisekunden, die sehr hilfreich für die Verarbeitung mit Computern ist, recht wenig mit der menschlichen Denkweise und Wahrnehmung von Zeit zu tun. Menschen denken in den Konzepten von Zeitabschnitten oder wiederkehrenden Datumsangaben, etwa 24.12. für Heiligabend, 31.12. für Silvester usw., und auch in Uhrzeiten ohne Bezug zu einem Datum, etwa 18.00 h Feierabend, oder als Kombination: Dienstags und Donnerstags 19 Uhr Karate-Training.¹⁰ Wollten wir Derartiges mithilfe der bisher existierenden APIs ausdrücken, wäre das recht schwierig geworden. Schauen wir nun auf die neuen Möglichkeiten.

Statt einer vollständigen Angabe aus Jahr, Monat und Tag, kann man sich auch Kombinationen aus Jahr und Monat, Monat und Tag sowie einfach nur Jahr vorstellen, um gewisse Datumsangaben zu modellieren.

¹⁰Insbesondere interessiert uns dabei die Zeitzone, in der die Termine stattfinden, in der Regel nicht – mit Ausnahme von Telefonterminen, die man etwa mit Geschäftspartnern in Übersee hat.


```

public static void main(final String[] args)
{
    // YearMonth: Demonstration jeweils mit und ohne Konstanten
    final YearMonth yearMonth = YearMonth.of(2014, 2);
    final YearMonth february2014 = YearMonth.of(2014, Month.FEBRUARY);

    // MonthDay: Achtung, ISO-Format mit der Reihenfolge: Monat, Tag
    final int dayOfBirthday = 7;
    final MonthDay monthDay1 = MonthDay.of(2, dayOfBirthday);
    final MonthDay monthDay2 = MonthDay.of(Month.FEBRUARY, dayOfBirthday);

    // Year
    final Year year = Year.of(2012);

    System.out.println("YearMonth: " + february2014);
    System.out.println("MonthDay:  " + monthDay2);
    System.out.println("Year:      " + year + " / isLeap? " + year.isLeap());
}

```

Führen wir das Programm aus, so erhalten wir folgende Konsolenausgaben:

```

YearMonth: 2014-02
MonthDay:  --02-07
Year:      2012 / isLeap? true

```

Anhand der Ausgaben sieht man verschiedene Notationsformen und insbesondere auch, dass man von der objektorientierten Umsetzung profitiert: Somit lässt sich etwa per Aufruf von `isLeap()` prüfen, ob ein Jahr ein Schaltjahr ist.

Die Klasse `Period`

Ähnlich wie die Klasse `Duration` modelliert die Klasse `java.time.Period` einen Zeitabschnitt. Beispiele sind etwa »2 Monate« oder »3 Tage«. Diese Art der Darstellung ist oftmals einfacher zu handhaben als eine korrespondierende Repräsentation in Nanosekunden oder Millisekunden. Konstruieren wir ein paar Instanzen von `Period` wie folgt:

```

public static void main(final String[] args)
{
    // Erzeuge ein Period-Objekt mit 1 Jahr, 6 Monaten und 3 Tagen
    final Period oneYear_sixMonths_ThreeDays = Period.ofYears(1).withMonths(6).
                                                    withDays(3);

    // Chaining von of() arbeitet anders, als man es eventuell erwartet!
    // Hier ein Period-Objekt mit 3 Tagen statt 2 Monate, 1 Woche und 3 Tagen
    final Period twoMonths_OneWeek_ThreeDays = Period.ofMonths(2).ofWeeks(1).
                                                    ofDays(3);

    final Period twoMonths_TenDays = Period.ofMonths(2).withDays(10);
    final Period sevenWeeks = Period.ofWeeks(7);
    final Period threeDays = Period.ofDays(3);

    System.out.println("1 year 6 months ...: " + oneYear_sixMonths_ThreeDays);
    System.out.println("Surprise just 3 days: " + twoMonths_OneWeek_ThreeDays);
    System.out.println("2 months 10 days:    " + twoMonths_TenDays);
    System.out.println("sevenWeeks:         " + sevenWeeks);
}

```

```

        System.out.println("threeDays: " + threeDays);
    }

```

Startet man das Programm PERIODEXAMPLE, so kommt es zu folgenden Ausgaben:

```

1 year 6 months ...: P1Y6M3D
Surprise just 3 days: P3D
2 month 10 days: P2M10D
sevenWeeks: P49D
threeDays: P3D

```

Anhand des Beispiels und dessen Ausgaben lernen wir verschiedene Besonderheiten der Klasse `Period` kennen. Zwar lassen sich Aufrufe von `of` hintereinander ausführen, es gewinnt aber der zuletzt aufgerufene. Man kann also auf diese Weise keine Zeiträume kombinieren, sondern legt einen initialen Zeitraum fest. Sollen weitere Zeitabschnitte hinzugefügt werden, so muss man dafür verschiedene `with()`-Methoden nutzen. Dabei wird ein Implementierungsdetail sichtbar. Die Klasse `Period` verwaltet drei Einzelwerte, nämlich für Jahre, Monate und Tage, aber eben nicht für Wochen. Daher gibt es keine Methode `withWeeks()`, sondern nur eine `ofWeeks()`, die intern eine Umrechnung in Tage vornimmt.

Nachdem wir nun einen ersten Eindruck gewonnen haben, schauen wir, wie einfach und lesbar Berechnungen mit dem neuen Date And Time API gestaltet werden können:

```

public class PeriodCalculationExample
{
    public static void main(final String[] args)
    {
        final LocalDateTime start = LocalDateTime.of(1971, 2, 7, 10, 11);

        final Period thirtyOneDays = Period.ofDays(31);
        System.out.println("7.2.1971 + 31 Tage: " + start.plus(thirtyOneDays));

        final Period oneMonth = Period.ofMonths(1);
        System.out.println("7.2.1971 + 1 Monat: " + start.plus(oneMonth));

        final LocalTime now = LocalTime.now();
        System.out.println("now: " + now);

        final LocalTime fiveMinutesLater = now.plus(5, ChronoUnit.MINUTES);
        System.out.println("now + 5 min: " + fiveMinutesLater);

        final LocalTime sevenHoursLater = now.plusHours(7);
        System.out.println("now + 7 hours: " + sevenHoursLater);
    }
}

```

Führt man das Programm aus, so kommt es zu folgenden Ausgaben:

```

7.2.1971 + 31 Tage: 1971-03-10T10:11
7.2.1971 + 1 Monat: 1971-03-07T10:11
now: 20:38:29.937
now + 5 min: 20:43:29.937
now + 7 hours: 03:38:29.937

```

Hintergrundwissen: Warum gibt es `Duration` und `Period`?

Zunächst verwundert die Definition von Zeitabschnitten durch zweierlei Klassen. Den Unterschied zwischen beiden Modellierungen kann man sich am besten im Zusammenhang mit Winter- und Sommerzeit klarmachen: Es gibt Tage, die 23 Stunden lang sind, und solche, die eine Dauer von 25 Stunden besitzen. Wird die Länge eines Tags jedoch fix als 24 Stunden angenommen und dieser Wert wiederum in Form einer Zeitspanne in Nanosekunden repräsentiert, so kommt es zu Berechnungsfehlern, wenn an kürzeren oder längeren Tagen ein Tag in die Zukunft oder Vergangenheit »gesprungen« werden soll: Man bewegt sich somit entweder eine Stunde zu wenig oder zu viel in die Zukunft. Nutzt man die Klasse `Period`, muss man sich um diese Details nicht kümmern, da das »Konzept Tag« und nicht dessen Pendant in Nanosekunden zum Einsatz kommt.

Die Klasse `Clock`

In einigen technischen Anwendungsfällen benötigt man Zugriff auf Millisekundenangaben. Früher hat man dazu Aufrufe von `System.currentTimeMillis()` genutzt, um Zugriff auf die aktuelle Zeit in Millisekunden seit dem 1.1.1970 zu haben. Nun nutzt man die Klasse `Clock` und schreibt etwa folgendes:

```
public static void main(final String[] args)
{
    // Basis UTC
    final Clock clock = Clock.systemUTC();
    System.out.println(clock);
    printCurrentTime(clock);

    // Basis Default Zeitzone
    final Clock clock2 = Clock.systemDefaultZone();
    System.out.println(clock2);
    printCurrentTime(clock2);
}

private static void printCurrentTime(final Clock clock)
{
    final long currentTime = clock.millis();
    System.out.println(currentTime);
}
```

Führen wir das Programm `CLOCKEXAMPLE` aus, so wird in etwa folgendes ausgegeben, wodurch wir die unterschiedlichen Zeitzonen erkennen:

```
SystemClock[Z]
1395495448297
SystemClock[Europe/Berlin]
1395495448365
```

Die Klasse `ZonedDateTime`

Neben der bereits kennengelernten Klasse `LocalDateTime` zur Repräsentation von Datum und Uhrzeit ohne Zeitzonenbezug existiert eine korrespondierende Klasse `java.time.ZonedDateTime`. Diese besitzt eine zugeordnete Zeitzone und berücksichtigt bei Berechnungen nicht nur die Zeitzone, sondern auch die Auswirkungen von Winter- und Sommerzeit. Um die aktuelle Zeit als `ZonedDateTime` zu ermitteln, kann man die Methode `now()` nutzen. Es existieren weitere Methoden, etwa um die Zeitzone und andere Werte abzufragen bzw. Instanzen von `ZonedDateTime` mit geänderter Wertebelegung zu erzeugen. Nachfolgend sind einige Beispiele gezeigt:

```
public static void main(final String[] args)
{
    // Aktuelle Zeit als ZonedDateTime-Objekt ermitteln
    final ZonedDateTime now = ZonedDateTime.now();

    // Dort die Uhrzeit ändern und in neuem Objekt speichern
    final ZonedDateTime nowButChangedTime = now.withHour(11).withMinute(44);

    // Neues Objekt mit verändertem Datum erzeugen
    final ZonedDateTime dateAndTime = nowButChangedTime.withYear(2008).
                                                         withMonth(9).
                                                         withDayOfMonth(29);

    System.out.println("now:           " + now);
    System.out.println("-> 11:44:      " + nowButChangedTime);
    System.out.println("-> 29.9.2008:  " + dateAndTime);
}
```

Interessant und etwas schade ist, dass man bei der letzten Variante `withMonth()` keine Monatskonstante verwenden kann. Führt man das Programm `ZONEDDATE-TIME-EXAMPLE` aus, so kommt es zu den nachfolgend gezeigten Ausgaben. Diese zeigen insbesondere den Einfluss von Winter- und Sommerzeit, wodurch im September 2008 die Abweichung von `+02:00` angegeben wird:

```
now:           2014-03-20T23:15:01.488+01:00[Europe/Berlin]
-> 11:44:      2014-03-20T11:44:01.488+01:00[Europe/Berlin]
-> 29.9.2008:  2008-09-29T11:44:01.488+02:00[Europe/Berlin]
```

2.3.3 Interoperabilität mit Legacy Code

Zum Abschluss unserer Entdeckungsreise des neuen Date And Time APIs wollen wir noch erkunden, wie man bestehenden Sourcecode stückweise migrieren kann. Insbesondere ist von Interesse, welche Klassen sich von der Idee her in alten JDK und dem neue Date And Time API entsprechen und wie man zwischen Instanzen der alten und neuen Klassen hin- und her konvertieren kann. Wie eingangs schon erwähnt, besteht eine Analogie zwischen den Klassen `Date` und `Instant`. Die Klasse `GregorianCalendar` kann man am ehesten mit der Klasse `ZonedDateTime` vergleichen. Folgende Aufzählung nennt passende Konvertierungsmethoden:

- `Date.from(Instant)`
- `Date.toInstant()`
- `Calendar.toInstant()`
- `GregorianCalendar.toZonedDateTime()`
- `GregorianCalendar.from(ZonedDateTime)`

Fazit

Wir haben nun einen ersten Überblick über das neue API zur Datumsverarbeitung erhalten, das in JDK 8 enthalten ist. Beim Entwurf flossen die Erfahrungen mit der Bibliothek Joda-Time ein. Viele Ideen wurden daraus entnommen und weiterentwickelt. Insgesamt macht die Arbeit mit dem neuen API durchaus Freude.

2.4 JavaFX 8

Nicht nur die Sprache Java sowie die Bibliotheken des JDKs, sondern auch JavaFX hat mit Java 8 einige Erweiterungen erfahren. Es wird in seiner Versionsnummer an die des JDKs angeglichen und bietet u. a. folgende Neuerungen:

- Unterstützung von Lambdas als `EventHandler` (implizit durch die Sprache)
- Texteffekte
- Zwei neue Controls, nämlich `DatePicker` zur Datumsauswahl und `TreeTableView` als Kombination von Tabellen- und Baumdarstellung
- 3D-Support
- Verbesserungen der Performance
- Optische Auffrischung durch ein neues Look and Feel namens Modena

Nachfolgend gehen einzelne Abschnitte auf diese Neuerungen ein, wobei die Verbesserungen der Performance nicht explizit thematisiert wird. Das gilt ebenso für das neue Look And Feel Modena, welches lediglich im Rahmen der Beschreibung der neuen Controls gezeigt wird.

2.4.1 Unterstützung von Lambdas als `EventHandler`

Wie schon bei der Vorstellung der Lambdas kurz gesehen, kann man statt anonymen innerer Klasse nun Lambdas einsetzen, um `EventHandler` zu realisieren. Das ist möglich, weil das Interface `EventHandler` ein Functional Interface ist, also genau eine abstrakte, zu realisierende Methode deklariert. Wir beschränken uns auf ein kurzes Beispiel, da die nachfolgenden Beispiele noch rege Gebrauch von Lambdas als `EventHandler` machen werden.

```
// Old Style
btn.setOnAction(new EventHandler<ActionEvent>()
```

```

{
    @Override
    public void handle(final ActionEvent event)
    {
        System.out.println("Hello World!");
    }
}

// New Style
btn.setOnAction((ActionEvent event) -> { System.out.println("Hello World!"); });

```

2.4.2 Texteffekte

Texte ließen sich bis JavaFX 2.x im Gegensatz zu anderen Nodes nicht im gleichen Umfang per CSS stylen. Mit JavaFX 8 ist dies nun möglich. Schauen wir auf ein Beispiel, welches einfarbige Füllungen, solche mit linearem Gradienten sowie unterschiedliche Strichstärken und auch eine Rotation auf verschiedene Texte anwendet.



Abbildung 2-2 RichTextExample

Das Beispiel lässt sich als Programm RICHTEXTEXAMPLE starten und basiert vollständig auf FXML und CSS. Das zugehörige FXML sieht wie folgt aus:

```

<Scene width="600" height="250" fill="white" xmlns:fx="http://javafx.com/fxml">
  <stylesheets>
    <URL value="@richtext.css" />
  </stylesheets>
  <VBox>
    <TextFlow styleClass="paragraph">
      <Text styleClass="text1">Hello </Text>
      <Text text=" " />
      <Text styleClass="text2, big">Bold</Text>
      <Text text=" " />
      <Text styleClass="text3, dropshadow">Effect</Text>
    </TextFlow>

    <TextFlow styleClass="paragraph">
      <Text styleClass="underlinefancy">
        fancy underline
        <effect>
          <DropShadow color="BLACK" offsetX="3.0" offsetY="3.0" />
        </effect>
      </Text>
    </TextFlow>
  </VBox>
</Scene>

```

```

        </Text>
    </TextFlow>

    <TextFlow styleClass="paragraph">
        <Text styleClass="rotatedThai">Thai: ?????????????????</Text>
    </TextFlow>
</VBox>
</Scene>

```

Im Listing sieht man im Tag `stylesheets`, wie man aus einer FXML-Datei direkt auf das gewünschte CSS verweist. Die korrespondierende CSS-Datei `richtext.css` zeige ich aus Platzgründen nur verkürzt:

```

.paragraph {
    -fx-font-family: Dialog;
    -fx-font-size: 60.0px;
    -fx-vgap: 30.0px;
}

.text1 {
    -fx-stroke: darkblue;
    -fx-stroke-width: 2.5;
    -fx-fill: darkgoldenrod;
}

.text2 {
    -fx-font-weight: bold;
    -fx-fill: linear-gradient(from 0.0% 0.0% to 100.0% 100.0%, repeat,
                             orange 0.0%, red 50.0%);

    -fx-stroke: black;
    -fx-stroke-width: 2.0;
}

...

.rotatedThai {
    -fx-stroke: black;
    -fx-stroke-width: 2.0;
    -fx-rotate: -5.0;
    -fx-fill: linear-gradient(from 0.0% 0.0% to 100.0% 100.0%, yellow 0.0%,
                             red 50.0%, blue 75.0%, green 100.0%);

    -fx-font-weight: bold;
}

...

```

2.4.3 Neue Controls

JavaFX bot auch vor Version 8 schon eine Menge an Bedienelementen. Allerdings fehlten Bedienelemente zur Datumsauswahl und zur Darstellung von Baum-artigen Inhalten in Tabellen, die in anderen grafischen Bibliotheken existieren. Während es derartige Bedienelemente für Swing leider nicht im Standard gab, musste die Entwicklergemeinschaft für JavaFX bis zu dessen Version 8 darauf warten. Mit JavaFX 8 gibt es nun im Package `javafx.scene.control` die zwei wichtigen und lang ersehnten Bedienelemente `DatePicker` und `TreeTableView`, die nachfolgend vorgestellt werden.

DatePicker

An einem Beispiel wollen wir kurz auf das Bedienelement `DatePicker` schauen. Dabei verwenden wir die in diesem Kapitel zuvor kennengelernten Sprachfeatures Lambdas und das neue Date And Time API. Letzteres nutzen wir dafür, die `DatePicker`-Komponente mit dem aktuellen Datum zu initialisieren sowie das gewählte Datum aus dem Bedienelement wieder auszulesen. Ein `DatePicker` bietet dazu einen Konstruktor mit einem Parameter vom Typ `LocalDate`. Zum Auslesen dient die Methode `getValue()`, die ein `LocalDate`-Objekt liefert. Dies geschieht wie folgt:

```
public class DatePickerExample extends Application
{
    @Override
    public void start(final Stage primaryStage)
    {
        // DatePicker mit Date and Time API-Funktionalität initialisieren
        final LocalDate localDate = LocalDate.now();
        final DatePicker datePicker = new DatePicker(localDate);

        datePicker.setOnAction((event) ->
        {
            // Gewähltes Datum ermitteln
            final LocalDate selectedDate = datePicker.getValue();
            System.out.println("Selected date: " + selectedDate);
        });

        final FlowPane root = new FlowPane();
        root.getChildren().add(datePicker);

        final Scene scene = new Scene(root, 300, 250);
        primaryStage.setTitle("DatePickerExample");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(final String[] args)
    {
        launch(args);
    }
}
```

Führen wir das Programm `DATEPICKEREXAMPLE` aus, so bekommen wir die Datumsauswahl in Abbildung 2-3 präsentiert.

TreeTableView

Während der `DatePicker` ein recht einfaches, aber sehr praktisches Bedienelement ist, ist die `TreeTableView` wohl eines der komplexesten Bedienelemente in JavaFX. Neben den Spalten einer Tabelle stellt es Informationen hierarchisch dar.

Die hierarchischen Daten werden über Instanzen vom Typ `TreeItem<T>` modelliert, die auch für den einfacheren `TreeView` zum Einsatz kommen. Sie legen die Baumstruktur fest, die aus Instanzen vom Typ `T` besteht. Dessen Attribute bestimmen die Spalten: Die Verbindung zu den Spalten eines `TreeTableView` wird durch Instanzen vom Typ `TreeTableColumn<T,V>` beschrieben, wobei `T` den Typ der Elemente

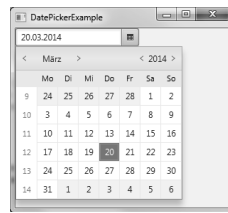


Abbildung 2-3 DatePickerExample

aus den `TreeItems` repräsentiert. Der Typ `v` entspricht dem Typ des in der Spalte darzustellenden Attributs.

Zur Demonstration nutzen wir eine Klasse `SimplePerson` mit den drei Attributen `name`, `age` und `size`. Verschiedene Personen bilden jeweils eine Gruppe. Die beiden Gruppen werden in einer Obergruppe `root` namens »All Persons« gebündelt. Im nachfolgenden Listing sieht man neben der Erstellung des Baums aus `TreeItems` durch die Methode `createTreeData()`, dass die Eigenschaft des Aufklappzustands durch `setExpanded(boolean)` gesetzt wird.

```
public class TreeTableViewExample extends Application
{
    public void start(final Stage stage)
    {
        final TreeItem<SimplePerson> root = createTreeData();

        final List<TreeTableColumn<SimplePerson,?>> columns = createColumns();

        final TreeTableView<SimplePerson> treeTableView = new TreeTableView<
            SimplePerson>(root);
        treeTableView.getColumns().addAll(columns);

        final VBox vbox = new VBox();
        vbox.getChildren().addAll(treeTableView);

        stage.setScene(new Scene(vbox, 300, 250));
        stage.setTitle("TreeTableViewExample");
        stage.show();
    }

    private TreeItem<SimplePerson> createTreeData()
    {
        final TreeItem<SimplePerson> root = new TreeItem<SimplePerson>(new
            SimplePerson("All Persons", null, -1));

        final TreeItem<SimplePerson> group1 = new TreeItem<SimplePerson>(new
            SimplePerson("Group 1", null, -1));
        final TreeItem<SimplePerson> childNode1_1 = new TreeItem<SimplePerson>(
            new SimplePerson("Micha", 43, 184));
        final TreeItem<SimplePerson> childNode1_2 = new TreeItem<SimplePerson>(
            new SimplePerson("Tom", 22, 177));
        final TreeItem<SimplePerson> childNode1_3 = new TreeItem<SimplePerson>(
            new SimplePerson("Lili", 34, 170));
    }
}
```

```

        final TreeItem<SimplePerson> group2 = new TreeItem<SimplePerson>(new
            SimplePerson("Group 2", null, -1));
        final TreeItem<SimplePerson> childNode2_1 = new TreeItem<SimplePerson>(
            new SimplePerson("Tim", 43, 181));
        final TreeItem<SimplePerson> childNode2_2 = new TreeItem<SimplePerson>(
            new SimplePerson("Jens", 47, 175));
        final TreeItem<SimplePerson> childNode2_3 = new TreeItem<SimplePerson>(
            new SimplePerson("Andy", 31, 178));

        group1.getChildren().setAll(childNode1_1, childNode1_2, childNode1_3);
        group2.getChildren().setAll(childNode2_1, childNode2_2, childNode2_3);
        root.getChildren().setAll(group1, group2);

        root.setExpanded(true);
        group1.setExpanded(true);

        return root;
    }

    private List<TreeTableColumn<SimplePerson,?>> createColumns()
    {
        final TreeTableColumn<SimplePerson, String> nameColumn = new
            TreeTableColumn<>("Name");
        nameColumn.setCellValueFactory(new TreeItemPropertyValueFactory<
            SimplePerson, String>("name"));
        nameColumn.setPrefWidth(125);

        final TreeTableColumn<SimplePerson, Integer> ageColumn = new
            TreeTableColumn<>("Age");
        ageColumn.setCellValueFactory(new TreeItemPropertyValueFactory<
            SimplePerson, Integer>("age"));
        ageColumn.setPrefWidth(50);

        final TreeTableColumn<SimplePerson, Integer> sizeColumn = new
            TreeTableColumn<SimplePerson, Integer>("Size in cm");
        sizeColumn.setCellValueFactory(new TreeItemPropertyValueFactory<
            SimplePerson, Integer>("size"));

        return Arrays.asList(nameColumn, ageColumn, sizeColumn);
    }

    public static void main(final String[] args)
    {
        launch(args);
    }
}

```

Starten wir das Programm `TREETABLEVIEWEXAMPLE`, so sehen wir eine Darstellung ähnlich wie in Abbildung 2-4.

Nach dem Betrachten der Darstellung und eigentlich bereits während der Modellierung fällt auf, dass wir für die Gruppenelemente keine sinnvollen Werte für die Attribute `age` und `size` angeben können. Das könnte man durch den Wert `null` ausdrücken. Für dieses Beispiel haben wir die Klasse `SimplePerson` folgendermaßen realisiert, um auf ein mögliches Problem aufmerksam zu machen:

```

public class SimplePerson
{
    private final String name;
    private final Integer age;
}

```

Name	Age	Size
▼ All Persons		-1
▼ Group 1		-1
Micha	43	184
Tom	22	177
Lili	34	170
▼ Group 2		-1
Tim	43	181
Jens	47	175
Andy	31	178

Abbildung 2-4 TreeTableViewExample

```
private final SimpleIntegerProperty size = new SimpleIntegerProperty();

public SimplePerson(final String name, final Integer age,
                    final Integer size)
{
    this.name = name;
    this.age = age;
    this.size.setValue(size);
}

public String getName()
{
    return name;
}

public Integer getAge()
{
    return age;
}

public Integer getSize()
{
    return size.getValue();
}
}
```

Dabei gibt es jedoch etwas zu beachten: Wenn ein Attribut als Property realisiert ist, dann kann man null nicht als spezielle Markierung für keinen Wert in Gruppen verwenden, da sonst NullPointerExceptions drohen.

Würde man das Attribut size mit wie folgt mit null initialisieren

```
root = new TreeItem<SimplePerson>(new SimplePerson("All Persons", null, null));
```

so würde dies Exceptions auslösen:

```
java.lang.NullPointerException
    at javafx.beans.property.IntegerProperty.setValue(IntegerProperty.java:66)
```

Als Abhilfe kann man entweder das Attribut als `Integer` definieren, oder sich folgenden Tricks bedienen: Man nutzt einen für das Alter ungültigen Wert, nämlich die zuvor schon eingesetzte `-1` und sorgt dann in der Darstellung dafür, dass dieser Wert entsprechend behandelt wird. Dazu registriert man einen eigenen Renderer, der in JavaFX aus einer Kombination eines Callbacks sowie einer speziellen `Cell<T>`, in diesem Fall einer `TreeTableCell` besteht. Hier wird gezeigt, welche Spezialbehandlung man vornehmen kann und was dabei ansonsten noch zu bedenken ist (siehe fett geschriebene Kommentare):

```
final TreeTableColumn<SimplePerson, Integer> sizeColumn = new TreeTableColumn<
    SimplePerson, Integer>("Size");
sizeColumn.setCellValueFactory(new TreeItemPropertyValueFactory<SimplePerson,
    Integer>("size"));

sizeColumn.setCellFactory(new Callback<TreeTableColumn<SimplePerson, Integer>,
    TreeTableCell<SimplePerson, Integer>>()
{
    @Override
    public TreeTableCell<SimplePerson, Integer> call(final TreeTableColumn<
        SimplePerson, Integer> p) {
        return new TreeTableCell<SimplePerson, Integer>()
        {
            @Override
            protected void updateItem(final Integer item, final boolean empty)
            {
                super.updateItem(item, empty);

                if (!empty)
                {
                    if (item.intValue() == -1)
                    {
                        // Spezialbehandlung für Kein-Wert-Indikator -1
                        setText(null);
                    }
                    else
                    {
                        // Textuelle Ergänzung
                        setText(item + " cm");
                    }
                }
                else
                {
                    // Ganz wichtig, sonst werden Texte dargestellt,
                    // obwohl Parent zugeklappt ist!
                    setText(null);
                }
            }
        };
    }
});
```

Diese Korrektur lässt sich als Programm `TREETABLEVIEWEXAMPLE2` starten und produziert eine Darstellung ähnlich wie in Abbildung 2-5.

Name	Age	Size
▼ All Persons		
▼ Group 1		
Micha	43	184 cm
Tom	22	177 cm
Lili	34	170 cm
▼ Group 2		
Tim	43	181 cm
Jens	47	175 cm
Andy	31	178 cm

Abbildung 2-5 TreeTableViewExample2

2.4.4 JavaFX 3D

Mit JavaFX 8 wurde Support für die Darstellung von dreidimensionalen Figuren integriert. Zuvor waren lediglich Pseudo-3D-Darstellungen mithilfe perspektivischer Transformationen möglich.

Nachfolgend stelle ich ein wenig Basiswissen vor, das wir zur Programmierung eines einfachen 3D-Beispiels benötigen.

Materialien und primitive 3D-Objekte

Objekte in einer dreidimensionalen Welt können durch die Anwendung verschiedener Oberflächen, Materialien und Texturen realitätsnäher gestaltet werden. Als Basis dienen Instanzen von `javafx.scene.paint.PhongMaterial`. Bei diesem lassen sich verschiedene Eigenschaften konfigurieren.

Zur Erstellung einfacher 3D-Szenarien existieren mit `Box`, `Cylinder` und `Sphere` drei vordefinierte Figuren: Diese besitzen alle den Basistyp `Node` und sind von der abstrakten Klasse `javafx.scene.shape.Shape3D` abgeleitet. Diesen Figuren können Materialien zugeordnet werden, wichtiger für die Darstellung in 3D ist aber, dass sich die Position im dreidimensionalen Raum festlegen und beliebig manipulieren lässt. Beispielsweise ist neben einer Verschiebung entlang der X-, Y- oder Z-Achse auch eine Rotation um eine Achse möglich.

```
@Override
public void start(Stage primaryStage)
{
    final PhongMaterial redMaterial = new PhongMaterial();
    redMaterial.setSpecularColor(Color.FIREBRICK);
    redMaterial.setDiffuseColor(Color.RED);

    final PhongMaterial blueMaterial = new PhongMaterial();
    blueMaterial.setSpecularColor(Color.DODGERBLUE);
    blueMaterial.setDiffuseColor(Color.BLUE);

    final Box redBox = new Box(400, 400, 200);
    redBox.setMaterial(redMaterial);
}
```

```

redBox.setTranslateX(100);
redBox.setTranslateY(150);
redBox.setTranslateZ(500);
redBox.setRotationAxis(Rotate.Y_AXIS);
redBox.setRotate(750);

final Cylinder blueCylinder = new Cylinder(200, 100);
blueCylinder.setMaterial(blueMaterial);
blueCylinder.setTranslateX(350);
blueCylinder.setTranslateY(350);
blueCylinder.setTranslateZ(150);

// Gruppieren und etwas nach hinten versetzen
final Group root = new Group(redBox, blueCylinder);
root.setTranslateZ(100);
root.setRotationAxis(Rotate.X_AXIS);
root.setRotate(25);

final Scene scene = new Scene(root, 500, 500);
final PerspectiveCamera perspectiveCamera = new PerspectiveCamera();
scene.setCamera(perspectiveCamera);

primaryStage.setTitle("Figures3DExample");
primaryStage.setScene(scene);
primaryStage.show();
}

```

Schlussendlich wird die Darstellung durch eine Instanz einer `PerspectiveCamera` bestimmt. Diese kann man beliebig im 3D-Raum positionieren und sie entspricht etwa dem eigenen Blickwinkel – etwas Ähnliches kennt man z. B. von Google Maps, wo man das Beobachter-Figürchen auch an beliebige Orte setzen kann und zudem auch Einfluss auf die Blickrichtung hat. Wenn wir das Programm `FIGURES3DEXAMPLE` starten, erhalten wir eine 3D-Darstellung wie in Abbildung 2-6.

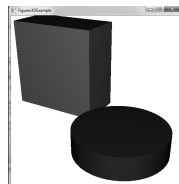


Abbildung 2-6 JavaFX8 3D BaseFigures

Light

Eine 3D-Darstellung gewinnt durch Lichtquellen und Beleuchtung an Authentizität. Jeder `Scene` können eine Menge aktiver Lichtquellen zugewiesen werden (geschieht dies nicht, so existiert immer eine Standardlichtquelle). Mit dem Typ `javafx.scene.LightBase` lassen sich momentan zwei Formen von Lichtquellen

modellieren, die vom Typ `Node` sind und damit Bestandteil des Scenegraphs sein können.

- **AmbientLight** — Unter einem `javafx.scene.AmbientLight` versteht man eine Lichtquelle, die alle Objekte gleichartig ausleuchtet. Es ist eine Art indirektes Licht, etwa Tageslicht.
- **PointLight** — Modelliert eine Lichtquelle mit einer spezifischen Position. Somit besitzt diese Lichtquelle eine Entfernung zu anderen Objekten. In der realen Welt sind `javafx.scene.PointLights` etwa Glühbirnen oder Kerzen.

Um den Beleuchtungseffekt auszuprobieren, ergänzen wir einfach folgende vier Zeilen und fügen die Lichtquelle der Gruppe folgendermaßen hinzu:

```
final PointLight pointLight = new PointLight(Color.ANTIQUEWHITE);
pointLight.setTranslateX(300);
pointLight.setTranslateY(100);
pointLight.setTranslateZ(0);

// Figuren und Lichtquelle bilden eine Gruppe
final Group root = new Group(red, blue, pointLight);
```

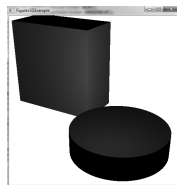


Abbildung 2-7 JavaFX8 3D BaseFigures Light

2.5 Weitere Änderungen

Nachfolgend sollen einige weitere, etwas kleinere Änderungen in Java 8 vorgestellt werden. Dieses Unterkapitel enthält ein Potpurri aus verschiedensten Änderungen in JDK 8, die man in der täglichen Arbeit wohl eher nicht regelmässig benötigen wird, davon aber zumindest einmal gehört haben sollte.

Zunächst beschreibe ich die Klasse `Optional<T>`, die die Modellierung optionaler Werte erlaubt. Danach werden einige der umfangreichen Erweiterungen im Interface `Map<K, V>` besprochen. Im Anschluss stelle ich die Funktionalität des parallelen Sortierens von Arrays vor. Dann gehe ich auf Änderungen am Speicheraufbau der JVM ein:

Die Permanent Generation wurde entfernt. Zudem erläutere ich kurz die neu hinzugekommene Unterstützung für Base64-Kodierungen. Auch schauen wir kurz auf die neue JavaScript Engine namens Nashorn und wie man damit kleine JavaScript-Programme ausführen kann. Im Bereich von NIO wurden diverse Erweiterungen vorgenommen. Ich zeige ein paar davon aus der Klasse `Files`, die die Handhabung von Dateioperationen erleichtern. Schlussendlich gehe ich noch kurz auf Erweiterungen im Bereich Concurrency, Reflection und benannte Parameter sowie auf Type Annotations ein.

2.5.1 Die Klasse `Optional<T>`

Die Modellierung optionaler oder nicht vorhandener Werte geschah bis JDK 8 häufig in Form von `null`-Werten oder mithilfe des NULL-OBJEKT-Musters.

In JDK 8 wurde die Klasse `java.util.Optional<T>` eingeführt. Dies ist insbesondere im Zusammenhang mit Streams von Interesse, wenn dort etwa Berechnungen ausgeführt werden, die nicht in jedem Fall ein Ergebnis produzieren, etwa die Berechnung eines Durchschnitts, Maximums oder Minimums, wenn keine Werte in einer Collection enthalten sind. Unpraktisch wären `NullPointerException` o. ä., wenn man mit hintereinander geschalteten Stream-Operationen arbeitet und die Verarbeitung so irgendwo mittendrin abbrechen würde.

```
public static void main(final String[] args)
{
    final Integer[] sampleValues = {1,3,5,7,11,13,17,19};
    final Integer[] noValues = {};

    final Optional<Integer> max = Arrays.stream(sampleValues).
                                         max(Comparator.naturalOrder());
    final Optional<Integer> min = Arrays.stream(noValues).
                                         min(Comparator.naturalOrder());

    // prüfe, ob es einen Wert gibt
    System.out.println(min.isPresent());

    // Zugriff auf den Wert
    final Integer maxValue = max.get();
    System.out.println(maxValue);
}
```

Führen wir das Programm aus, so kommt es zu folgender Ausgabe, die verdeutlicht, dass ein leeres Objekt als `Optional.empty` modelliert wird und ansonsten ein `Optional<T>` ein Container für ein Element vom Typ `T` ist.

```
Optional[19]
Optional.empty
false
19
```

Wenn das schon alles wäre, hätte man aber nicht sehr viel gewonnen. Der große Mehrwert besteht in verschiedenen Methoden, die Verarbeitungsschritte oder alternative Rückgabewerte erlauben. Nachfolgendes Beispiel illustriert dies und zeigt dazu die Me-

thoden `ifPresent()`, `orElse()`, `orElseGet()` und `orElseThrow()`, die jeweils Functional Interfaces als Parameter haben, die hier als Lambdas realisiert sind:

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

    final Integer[] noValues = {};

    final Optional<Integer> min = Arrays.stream(noValues).
                                      min(Comparator.naturalOrder());

    // Führe Aktion aus, wenn vorhanden
    min.ifPresent(System.out::println);

    // Alternativen Wert liefern, wenn nicht vorhanden
    System.out.println(min.orElse(-1));

    // Berechne Ersatzwert, wenn nicht vorhanden
    final Supplier<Integer> randomGenerator = () -> (int)(100 * Math.random());
    System.out.println(min.orElseGet(randomGenerator));

    // Löse eine Exception aus, wenn nicht vorhanden
    min.orElseThrow(() -> new NoSuchElementException("thres is no minimum"));

}
```

Startet man das obige Programm so kommt es zu folgenden, gekürzten Konsolenausgaben. Man sieht hieran auch, dass `ifPresent()` nicht ausgeführt wird:

```
0
7
Exception in thread "main" java.util.NoSuchElementException: thres is no minimum
```

Die gezeigten Methoden erlauben eine recht lesbare Programmierung von Alternativen, falls bei einer Berechnung einmal ein Wert nicht vorhanden sein sollte. Mit `orElse()` kann man einen alternativen Wert vorgeben, mit `orElseGet()` kann man eine Berechnung anstellen und `orElseThrow()` eine Exception auslösen.

Die gezeigten Berechnungen werden aber oftmals nicht – wie eben gezeigt – auf Wrapper-Klassen durchgeführt, sondern meistens auf Werten primitiver Typen. Da `Optional<T>` aber einer generische Klasse ist, kann man sie für diese nicht nutzen.

Verarbeitung von primitiven Werten

Weil aber die Verarbeitung primitiver Werte ein sehr gebräuchlicher Anwendungsfall ist, wurde das JDK um besondere Implementierungen von Optionals erweitert, die für die Typen `int`, `long` und `double` definiert sind, nämlich: `OptionalInt`, `OptionalLong` und `OptionalDouble`. Nachfolgendes Beispiel zeigt die zuvor durchgeführten Berechnungen zum Maximum und Minimum und darüber hinaus eine Durchschnittsbildung mithilfe der genannten Klassen:

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

    final int[] sampleValues = {1,3,5,7,11,13,17,19};

    final OptionalInt max = Arrays.stream(sampleValues).max();
    final OptionalInt min = Arrays.stream(sampleValues).min();
```

```

final OptionalDouble avg = Arrays.stream(sampleValues)
    .filter(value -> value > 10).average();

System.out.println("avg " + avg);
}

```

2.5.2 Erweiterungen im Interface Map<K, V>

Das Interface Map<K, V> hat mit Java 8 diverse Erweiterungen erfahren, etwa in Form der Methoden `getOrDefault()`, `putIfAbsent()` usw.

Immer mal wieder wünscht man sich beim Zugriff auf eine Map, dass ein Defaultwert zurückgeliefert werden kann, falls kein Eintrag zu dem Schlüssel existiert. Die Methode `getOrDefault()` realisiert diese Funktionalität und vermeidet dadurch ansonsten notwendige Spezialbehandlungen.

Insbesondere bei Multithreading vermisst man eine Funktionalität, einen Wert für einen Schlüssel in der Map zu speichern, falls es dieser dort noch nicht existiert. Herkömmlicherweise musste man entweder mit `synchronized` oder Locks arbeiten, um einen kritischen Abschnitt zu realisieren, in dem zuerst ein Lesezugriff und danach gegebenenfalls ein Schreibzugriff erfolgte. Mithilfe der Methode `putIfAbsent()` kann man sich derartige »Verrenkungen« ersparen. Weiterhin gibt es nun eine Methode `replace()` mit deren Hilfe bereits existierende Einträge ersetzt werden.

Manchmal soll nicht nur ein ganz bestimmter Wert mit `putIfAbsent()` in eine Map eingefügt werden, sondern dabei eine Berechnung für jedes nicht vorhandene Element ausgeführt werden. Das kann man beispielsweise dazu nutzen, wenn man eine MultiMap realisieren möchte, bei der für einen Schlüssel mehrere Werte gespeichert werden können.

```

// Achtung: Nur für Singelthreading korrekt
if (!map.containsKey(key))
{
    map.put(new ArrayList<>());
}

List<T> values = map.get(key);
values.add(newElement);

```

Die obige Realisierung ist nur für Singlethreading korrekt, bei Multithreading könnte mehrere Thread nahezu zeitgleich die Prüfung vornehmen und danach unterbrochen werden. Nachfolgendes Hinzufügen von Elementen wird dann möglicherweise durch frisch initialisierte `ArrayList<E>`-Instanzen wieder zunichte gemacht. Mithilfe von `computeIfAbsent()` löst man dieses Problem und zudem verbessert sich die Lesbarkeit:

```

map.computeIfAbsent(key, it -> new ArrayList<>())

List<T> values = map.get(key);
values.add(newElement);

```

2.5.3 Parallel Array Sorting

Mit JDK 8 wurde die Utility-Klasse `Arrays` um für diverse Typen überladene Methoden `parallelSort()` erweitert. Damit ist es möglich, Arrays parallel unter Ausnutzung mehrerer Threads zu sortieren. Nachfolgendes Beispiel illustriert nur einen einfachen Aufruf:

```
public class ParallelArraysExample
{
    public static void main(final String[] args)
    {
        final int[] numbers = { 1, 4, 6, 3, 1, 8, 74, 32, 1, 2 };

        System.out.println("Initial: " + Arrays.toString(numbers));
        Arrays.parallelSort(numbers);
        System.out.println("Sorted:  " + Arrays.toString(numbers));
    }
}
```

Führt man das Programm aus, so erhält man (erwartungsgemäß) folgende Ausgabe:

```
Initial: [1, 4, 6, 3, 1, 8, 74, 32, 1, 2]
Sorted:  [1, 1, 1, 2, 3, 4, 6, 8, 32, 74]
```

Vorschnell auch für kleinere Datenbestände in der Hoffnung eines Performance-Boosts eingesetzt, wird man wohl enttäuscht. Die Parallelisierung sowie die Abstimmung der Threads zum Zusammenführen der Teilergebnisse sorgen für einen gewissen Extraaufwand, der geringfügig zusätzliche Laufzeit kostet. Somit lohnt sich das Ganze erst bei größeren Datenmengen und kann sich ansonsten sogar negativ auf die Laufzeit auswirken.

2.5.4 Keine Permanent Generation mehr

Im Speicherbereich namens Permanent Generation (kurz: Perm Gen) finden sich vor allem die Metadaten zu Klassen. Mit JDK 8 wurde die Perm Gen aus den Speicherbereichen der JVM entfernt und auf eine andere Art realisiert: Metadaten der Klassen sind nun in einem grössenveränderlichen Bereich namens Metaspace abgelegt, der zudem durch nativen Speicher und nicht aus dem der JVM bereitgestellt wird. Warum wurde diese Änderung notwendig?

In umfangreichen Applikationen, die viele externe Abhängigkeiten zu anderen Bibliotheken besitzen und Klassen dynamisch erzeugen, kann es bis einschließlich JDK 7 immer mal wieder vorkommen, dass während der Programmausführung ein `OutOfMemoryError: PermGen Space` auftritt. Ursache ist, dass die Permanent Generation oftmals zu klein für die nachfolgenden Aktionen dimensioniert ist und voll läuft, was den Fehler auslöst. Das Ganze passiert insbesondere dann, wenn keine explizite (Vor-)Einstellung der Permanent Generation durch einen Aufrufparameter der JVM erfolgt. Dadurch lässt sich zwar die maximale Größe der Permanent Generation zum Programmstart festlegen, allerdings eben nur auf einen fixen Wert, der möglicher-

weise die Dynamik des Programmablaufs nicht genügend berücksichtigt: Wenn diverse Klassen dynamisch erzeugt oder nachgeladen werden, besteht weiterhin die Gefahr des `OutOfMemoryErrors`, selbst wenn zu Programmstart die Permanent Generation scheinbar ausreichend groß gewählt schien. Als potenzielle Abhilfe sieht man mitunter extrem groß dimensionierte Permanent Generations, was wiederum zu Speicherplatzverschwendung führt.

Mit der Änderung in der Speicherverwaltung der JVM trägt man diesem Umstand Rechnung und insbesondere verhindert man, dass die früher von vielen Entwicklern als Abhilfe genutzte Überdimensionierung der Perm Gen und damit auch eine Speicherverschwendung nun nicht mehr notwendig ist und sich mehr der Dynamik des Programms anpasst.

2.5.5 Base64-Kodierungen

Lange Zeit war es nur durch Nutzung externer Bibliotheken oder inoffizieller Klassen des JDKs (`BASE64Encoder`/`BASE64Decoder` aus dem package `sun.misc` möglich, Base64-Kodierungen zu verarbeiten. Diese Funktionalität findet sich nun praktischerweise im JDK.

Die Base64-Kodierung wird etwa für E-Mail-Anhänge im MIME-Format (Multi-purpose Internet Mail Extensions) verwendet. Die Art der Kodierung wurde entwickelt, weil mit dem SMTP (Simple Mail Transfer Protocol) nur Zeichen mit 7-Bit übertragen werden konnten. Das ist aber weniger, als die Kodierung von ASCII-Zeichen benötigt. Insbesondere lassen sich auch keine bytekodierten Informationen übertragen. Daher wurde eine Kodierung erdacht, die auf das oberste Bit verzichtet. Dazu werden jeweils drei Byte der Eingabe (=24 Bit) in vier 6-Bit-Blöcke aufgeteilt. Jeder dieser Blöcke kann eine Zahl zwischen 0 und 63 darstellen, was zu dem Namen Base64 geführt hat. Der Vorteil dieser Kodierung ist, dass die resultierende Zeichenfolge, nur aus wenigen, Codepage-unabhängigen ASCII-Zeichen besteht. Es werden lediglich die Zeichen A–Z, a–z, 0–9, + und / verwendet, sowie das Zeichen = als Ende-Markierung.

Mit der Base64-Kodierung wird dadurch der problemlose Transport von beliebigen Binärdaten möglich. Dies kann man auch dazu nutzen, um Binärdaten in Form eines Strings in einer Datenbank zu speichern, wenn man explizit kein **BLOB** (Binary Large Object) verwenden möchte.

Beispielsweise wird der String „Dies ist ein Test“ in Base64 zu:

```
Base 64 Encoded: RG11cyBpc3QgZWluIFRlc3Q=
```

Wie man sieht, hat man bei einer möglichen Übertragung von einem Rechner zu einem anderen ganz nebenbei den Vorteil, dass die Base64-Kodierung eine nicht lesbare Darstellung erstellt. Dies bietet zumindest einen kleinen Schutz vor unbeabsichtigten Blicken. Allerdings entsteht durch die Reduktion auf die ausgewählten Zeichen ein Overhead: Das zu übertragende Datenvolumen nimmt um rund 30 % zu.

2.5.6 Nashorn – Die neue JavaScript-Engine

Seit Version 6 enthält das JDK eine JavaScript-Engine. Mit JDK 8 wurde diese überarbeitet. Insbesondere ist sie nun deutlich performanter und besitzt eine sehr hohe Kompatibilität zum JavaScript-Standard.

Bevor wir die Ausführung von JavaScript innerhalb eines Java-Programms kurz anschauen, zeige ich zunächst, wie man auf Scripting Engines zugreift und sich dazu Informationen beschafft.

Verfügbare Scripting Engines auflisten

Teilweise ist es von Interesse, welche Scripting Engines verfügbar sind. Standardmäßig ist dies in JDK 8 natürlich Nashorn als JavaScript-Engine. Darüber hinaus können aber auch weitere Engines vorhanden sein, etwa für Groovy.

Den Ausgangspunkt bildet die Klasse `javax.script.ScriptEngineManager`, die eine Liste von `ScriptEngineFactory`-Instanzen liefert. Dies kann man nach verschiedenen charakteristischen Eigenschaften fragen, wie dies im folgenden Listing gezeigt ist:

```
public class ListScriptingEngines
{
    public static void main(final String args[])
    {
        final ScriptEngineManager manager = new ScriptEngineManager();
        final List<ScriptEngineFactory> factories = manager.getEngineFactories();

        for (final ScriptEngineFactory factory : factories)
        {
            System.out.println(factory.getEngineName());
            System.out.println(factory.getEngineVersion());
            System.out.println(factory.getLanguageName());
            System.out.println(factory.getLanguageVersion());
            System.out.println(factory.getExtensions());
            System.out.println(factory.getMimeTypes());
            System.out.println(factory.getNames());
            System.out.println();
        }
    }
}
```

Führen wir das Programm aus, so erhalten wir zumindest folgende Ausgabe – falls Sie weitere Scripting Engines installiert haben, etwa Groovy, so werden auch diese ausgegeben:

```
Oracle Nashorn
1.8.0
ECMAScript
ECMA - 262 Edition 5.1
[js]
[application/javascript, application/ecmascript, text/javascript, text/
  ecmascript]
[nashorn, Nashorn, js, JS, JavaScript, javascript, ECMAScript, ecmascript]
```

Mit JDK 7 kommt es dagegen zu folgenden Ausgaben:

```

Mozilla Rhino
1.7 release 3 PRERELEASE
ECMAScript
1.8
[js]
[application/javascript, application/ecmascript, text/javascript, text/
  ecmascript]
[js, rhino, JavaScript, javascript, ECMAScript, ecmascript]

```

Einfache JavaScript Anweisungen ausführen

Das Auflisten verfügbarer Scripting Engines ist recht unspektakulär. Interessanter ist es, mit deren Hilfe Scriptcode auszuführen.

Das nachfolgende Beispiel verdeutlicht, wie man JavaScript-Anweisungen ausführen kann. Ausgangspunkt ist hier wiederum die Klasse `ScriptEngineManager`. Von dieser kann man basierend auf einem Namen die dazu passende `javax.script.ScriptEngine`-Instanz abfragen. Scriptcode lässt sich dann mithilfe der Methode `eval(String)` ausführen.

Jede nicht triviale Berechnung benötigt Eingaben oder einen Ablaufkontext. Nachfolgend zeige ich insbesondere, wie man Werte mithilfe von `javax.script.Bindings` setzen kann, die dann in JavaScript-Berechnungen ausgewertet werden:

```

public class SimpleJavaScriptAndBindingDemo
{
    public static void main(String[] args) throws Exception
    {
        final ScriptEngineManager manager = new ScriptEngineManager();
        final ScriptEngine engine = manager.getEngineByName("js");

        // Executing // Ist mit JDK 7 noch erlaubt
        engine.eval("println (Hi! JavaScript executed from a Java program.'");

        // Data Binding
        engine.put("a", 2);
        engine.put("b", 7);

        final Bindings bindings = engine.getBindings(ScriptContext.ENGINE_SCOPE);

        final Object a = bindings.get("a");
        final Object b = bindings.get("b");

        System.out.println("a = " + a);
        System.out.println("b = " + b);

        // Berechnung ausführen
        final Object result = engine.eval("a + b;");
        System.out.println("a + b = " + result);

        // Berechnung, Ergebnis wird einer JavaScript-Variablen zugewiesen und
        // später aus Java gelesen
        final String script = "var ergebnis = Math.max(a, b)";
        engine.eval(script);
        final Object result2 = engine.get("ergebnis");
        System.out.println("Math.max(a, b) = " + result2);
    }
}

```

Führt man das obige Programm aus, so erhält man folgende Ausgaben:

```
a = 2
b = 7
a + b = 9
Math.max(a, b) = 7.0
```

Hinweis: Kompilierung

Neben dem Ausführen von JavaScript im Interpreter-Modus ist es auch möglich, die JavaScript-Anweisungen zu kompilieren und damit deren Abarbeitung zu beschleunigen.

```
final Compilable compEngine = (Compilable) engine;
final CompiledScript compiled = compEngine.compile("a-b;");

System.out.println(compiled.eval(bindings));
```

Dynamische Berechnungen ausführen

Meiner Meinung nach wird der große Mehrwert der Scripting Engines leider viel zu selten klar herausgestellt. Bei vielen einfachen Beispielen, die man so findet, fragt man sich nach dem Nutzen.

Ich denke, man kann die JavaScript-Engine immer dann gewinnbringend nutzen, wenn man dynamische Berechnungen ausführen möchte, etwa vom Benutzer eingegebene Funktionen in einem bestimmten Wertebereich berechnen, z. B. um eine Wertetabelle oder einen Funktionsplotter zu implementieren. Dies ist mit Java-Bordmitteln nur extrem schwierig zu realisieren, wenn die Funktion etwa frei von einem Benutzer in einem Textfeld eingegeben wird.

Weil es hier lediglich um das Prinzip der dynamischen Auswertung geht, nutze ich im folgenden der Einfachheit halber einen String, der die zu berechnende Formel enthält, die ein Benutzer hätte eingegeben können.

```
final String calculation = "7 * (x * x) + (3 - x) * (x + 3) / 10"; ;
System.out.println("f(x) = " + calculation);

for (int x = -10; x <= 10; x++)
{
    engine.put("x", x);

    final Object calculationResult = engine.eval(calculation);
    System.out.println("x = " + x + "\t / f(x) = " + calculationResult);
}
```

Führt man diese Programmzeilen aus, so wird für den Wertebereich für x von -10 bis 10 die entsprechende Formel ausgewertet und das Ergebnis per `eval()` berechnet und anschließend folgendermaßen ausgegeben:

```
x = -10 / f(x) = 690.9
x = -9 / f(x) = 559.8
...
x = -2 / f(x) = 28.5
x = -1 / f(x) = 7.8
x = 0 / f(x) = 0.9
x = 1 / f(x) = 7.8
x = 2 / f(x) = 28.5
...
x = 9 / f(x) = 559.8
x = 10 / f(x) = 690.9
```

2.5.7 Erweiterungen im NIO und der Klasse `Files`

Die Utility-Klasse wurde um verschiedene Hilfsmethoden erweitert. Insbesondere existieren unter anderem folgende Methoden:

- `list(Path)` – Liefert den Inhalt eines Verzeichnisses als `Stream<Path>`. Das Besondere dabei ist, dass der Inhalt sukzessive bei Bedarf ermittelt wird und nicht direkt von vornherein.
- `lines(Path)` – Stellt eine Datei zeilenweise in Form eines `Stream<String>` bereit.
- `write(Path, Iterable<? extends CharSequence>, OpenOption...)` – Schreibt die übergebenen Textzeilen in die durch den `Path`-Parameter referenzierte Datei.

Die Aufzählung gibt nur einen unvollständigen Überblick, da die Neuerungen deutlich umfangreicher sind. Allerdings werden oben recht nützliche Funktionalitäten dargestellt, die ich nachfolgend an einem kurzen Beispiel verdeutlichen möchte. Hierbei erzeugen und befüllen wir eine Textdatei mit ein wenig Inhalt und nutzen dazu `write()`. Die in die Datei geschriebenen Informationen ermitteln wir in Form eines `Stream<String>`. Darauf führen wir eine Filterung und eine Gruppierung aus. Abschließend inspizieren wir das Temp-Verzeichnis und filtern auf alle Dateien, die mit `txt` enden. Dabei gilt es zu beachten, dass man nicht die Methode `endsWith(String)` des `Path`-Objekts nutzen kann, da diese auf Pfadbestandteilen arbeitet und nicht auf Namen. Daher muss zuvor eine Umwandlung in einen String erfolgen:

[illegible]


```

        StandardOpenOption.APPEND);

    final Stream<String> contentAsStream = Files.lines(resultFile);

    final Map<Integer, List<String>> filteredAndGrouped = contentAsStream.
        filter(word -> word.length() > 3).
        collect(Collectors.groupingBy(
            String::length));

    System.out.println(filteredAndGrouped);

    final Stream<Path> tmpDirContent = Files.list(Paths.get("C:/tmp/"));
    // Fallstrick: endsWith arbeitet auf Path-Komponenten nicht auf Dateinamen!!
    tmpDirContent.filter(it -> it.toString().endsWith(".txt")).
        forEach(System.out::println);
}

```

Nach der zweiten Ausführung erhalten Sie in etwa folgende Ausgaben:

```

{4=[This, This], 7=[content, content]}
C:\tmp\WriteText.txt

```

2.5.8 Erweiterungen im Bereich Concurrency

Im Bereich von Multithreading und Concurrency wurden auch verschiedene Erweiterungen realisiert. Das betrifft vor allem das Package `java.util.concurrent` und seinen Subpackages mit unter anderem folgenden wichtigen Änderungen:

- **ConcurrentHashMap** – In der Klasse `ConcurrentHashMap` wurde eine Vielzahl an Methoden ergänzt. Das sind unter anderem `compute()`, `forEach()`, `forEachEntry()`, `forEachKey()`, `forEachValue()`, `merge()`, `reduce()` und `search()`. Einige davon haben wir schon bei der Betrachtung der Neuerungen im Interface `Map<K, V>` besprochen.
- **CompletableFuture<T>** – Die Klasse `CompletableFuture<T>` ist eine Spezialisierung der Interfaces `Future<T>`, die eine explizite Beendigung erlaubt, in dem Zustand als auch Ergebnis gesetzt werden können.
- **StampedLock** – Die Klasse `java.util.concurrent.locks.StampedLock` ist eine spezielle Variante eines Locks, die in ihrer Intention ähnlich einem `ReadWriteLock` ist, jedoch mit optimistischen Sperren arbeitet und damit performantere Verarbeitungen erlaubt, wenn viel Parallelität und gleichzeitige Lesezugriffe mit seltenen Schreibzugriffen erfolgen.
- **Executors** – Es gibt nun eine Variante eines Thread-Pools, der alle verfügbaren Prozessoren nutzt. Dieser wird durch `newWorkStealingPool()` erzeugt.

2.5.9 Erweiterungen im Bereich Reflection

Mit Reflection kann man Programme inspizieren und Informationen zu Klassen, Methoden, Parametern usw. ermitteln. Es ist sogar möglich, zur Laufzeit dynamisch neue Instanzen zu erzeugen, Methoden aufzurufen u. v. m.

Bis JDK 8 war es jedoch nicht möglich, die Namen der Parameter zu ermitteln, sondern diese wurden nur als `arg0`, `arg1` usw. repräsentiert. In verschiedenen Einsatzkontexten ist aber der Zugriff auf die Namen der Parameter recht wünschenswert. Im Bereich der WebServices existiert dazu eine spezielle Annotations etwa `@QueryParam` und `@PathParam`. In JavaFX 8 wurde die Annotation `@NamedArg` eingeführt, die Informationen zu dem Namen eines Arguments bereitstellt. Sie kann per Reflection abgefragt werden.

Das geht mit Java 8 generell etwas einfacher und erfordert keine Annotations und dort auch keine Wiederholung des Parameternamens. Das folgende Listing zeigt die Klasse `ReflectionParameterNamesExample`, die eine Selbstinspektion durchführt und zur Ausgabe der Informationen zu den Parametern auf die mit JDK 8 eingeführte Klasse `java.lang.reflect.Parameter` und die dort definierten Methoden zurückgreift:

```
public class ReflectionParameterNamesExample
{
    public static void main(final String[] args)
    {
        inspectClass(ReflectionParameterNamesExample.class);
    }

    public static void inspectClass(final Class<?> clazz)
    {
        System.out.println("Untersuchte Klasse: " + clazz.getCanonicalName());
        System.out.println();

        // Zugriff und Ausgabe aller öffentlichen Methoden
        final Method methods[] = clazz.getDeclaredMethods();
        System.out.println("Methoden: ");
        for (final Method method : methods)
        {
            // Neu i JDK 8
            final Parameter[] parameters = method.getParameters();

            final String asString = Modifier.toString(method.getModifiers()) +
                " " + method.getReturnType() +
                " " + method.getName() +
                buildParameterString(parameters);

            System.out.println(methodAsString);
        }
    }

    public static String buildParameterString(final Parameter[] parameters)
    {
        final StringBuilder sb = new StringBuilder("(");

        for (final Parameter param : parameters)
        {
            sb.append(Modifier.toString(param.getModifiers()));
            sb.append(" ");
            sb.append(param.getParameterizedType()); // zeigt auch Generics
            sb.append(" ");
            sb.append(param.getName());
        }

        return sb.append(")").toString();
    }
}
```

```

    }
}

```

Startet man das Programm so gibt es die Namen der Parameter wie im Sourcecode definiert an, allerdings nur sofern der Sourcecode mit einem speziellen Compiler-Flag übersetzt wurde (siehe Abbildung 2-8). Ansonsten würden diese einfach als `arg0`, `arg1` usw. ausgegeben.

Untersuchte Klasse: `chxx_jdk8.misc.ReflectionExample`

Methoden:

```

public static void main(final class [Ljava.lang.String; args)
public static class java.lang.String buildParameterString(final class
                                                             [Ljava.lang.reflect.Parameter; parameters)
public static void inspectClass(final java.lang.Class<?> clazz)

```

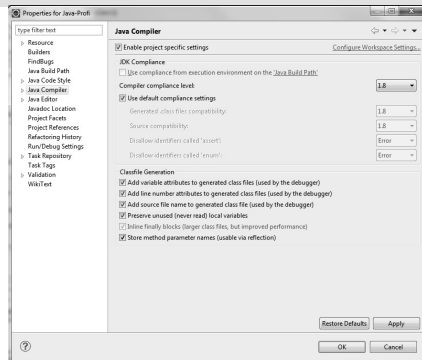


Abbildung 2-8 ParameterCompilerOption

2.5.10 Type Annotations

Während bis JDK 7 Annotations nur an ganz bestimmten Stellen im Sourcecode erlaubt waren, wurde dies mit JDK 8 deutlich gelockert. Dadurch wird es leichter, Constraints (Randbedingungen) an beliebige Elemente im Sourcecode annotieren zu können.

Beispielsweise kann man sich für Code-Prüfungen etwa folgende Annotations vorstellen:

- `@NonNull`, `@Nullable` – Attribut oder Parameter darf nicht `null` sein bzw. kann den Wert `null` enthalten.
- `@Immutable` – Das Attribut oder die Klasse ist unveränderlich.
- `@ReadOnly` – Es sollen nur Lesezugriff durch andere Objekte erlaubt sein.
- ...

Leider werden derartige Annotations nicht im JDK 8 bereitgestellt und auch nicht ausgewertet. Eine mögliche Erweiterung besteht im Checker-Framework, das frei zum Download unter <http://types.cs.washington.edu/checker-framework/> bereitsteht.

Meiner Meinung nach wäre es aber sehr wünschenswert gewesen, gewisse Prüf-Annotations bereits im JDK zu haben, ähnlich zu denen aus JavaEE, und diese auch durch den Compiler auszuwerten. Weil dies leider nicht der Fall ist, werde ich das Thema Type Annotations nicht weiter vertiefen.

2.6 Ausblick auf JDK 9: Mit JDK 8 nicht umgesetzte Features

Bereits mit JDK 7 wurden verschiedene Erweiterungen angedacht, die die Handhabung von Java spürbar vereinfacht hätten. Jeder, der schon einmal in Groovy programmiert hat, weiß, was ich meine. Dort funktionieren viele Dinge einfach so natürlich und mühelos. Beispiele sind die Integration von Collection-Zugriffen in die Sprachsyntax oder auch eine schlankere Syntax bei Vergleichen von Werten aus `enum`-Aufzählungen. In Java wird für beides doch einiges mehr an Sourcecode benötigt, wodurch zum Teil die Lesbarkeit leidet.

Integration von Collection-Zugriffen in die Sprachsyntax

Schon 2009 hat man über eine Integration einer einfacheren Schreibweise zur Erzeugung von und zum Zugriff auf Collections (sogenannte Collection-Literale) nachgedacht, wie sie z. B. in Groovy existieren.

Collection-Erzeugung Nachfolgendes Listing zeigt, wie eine mögliche Syntax für Collection-Literale, die die Elemente der Collection in geschweifte oder eckige Klammern einschließt, aussehen hätte können (es wurden verschiedene, aber recht ähnliche Vorschläge diskutiert):

```
// Mit JDK 8 leider nicht umgesetzt
final List<String> newStyleList = ["item1", "item2"];

final Map<String, String> newStyleMap = ["key1" : "value1", "key2" : "value2"];
```

Herkömmlicherweise existieren folgende zwei recht gebräuchliche Varianten, um eine Liste zu befüllen:

```
// Variante 1
final List<String> oldStyleList1 = new ArrayList<>();
oldStyleList1.add("item1");
oldStyleList1.add("item2");

// Variante 2
final List<String> oldStyleList2 = Arrays.asList("item1", "item2");
```

Beide Varianten besitzen jedoch ihre Nachteile. Die erstere ist offensichtlich viel länger als der Einsatz eines Collection-Literals. Variante 2 ist zwar ähnlich kurz, erlaubt aber

keine weiteren Modifikationen an der Zusammensetzung der Liste mehr. Collection-Literale wären also eine gute Bereicherung gewesen. Warten wir mal auf Java 9.

Tipp: Google Guava als Abhilfe

Wenn Sie nicht erst bis Java 9 warten wollen, dann empfehle ich einen Blick auf die Bibliothek Google Guava. Dort befinden sich viele hilfreiche Methoden rund um Collections und funktionale Programmierung. Google Guava steht frei unter <https://code.google.com/p/guava-libraries/> zum Download oder in Maven Central bereit (Gradle-Dependency `compile 'com.google.guava:guava:16.0.1'`). Nachfolgend nutzen wir die Methode `Lists.newArrayList(E...)`:

```
// Einsatz von Google Guava
final List<String> guavaStyleList = Lists.newArrayList("item1", "item2");
```

Datenzugriff Während man auf Arrays indiziert mit eckigen Klammern zugreift, muss man für Listen die Methode `getAt(int)` und für Maps die Methode `get(K)` nutzen. Für Wertzuweisungen muss man analog `setAt(int, Object)` bzw. `put(K, V)` aufrufen.

Wie es bereits in Groovy für indizierte Zugriffe und Wertzuweisungen auf Listen bzw. Zugriffe auf Schlüssel und Werte bei Maps möglich ist, war auch für Java eine Notation analog zu indizierten Array-Zugriffen vorgesehen, wodurch explizite Methodenaufrufe überflüssig geworden wären:

```
// Collection-Literale mit JDK 8 leider nicht umgesetzt
newStyleList[0] = "set at 0";
System.out.println("getAt(0) " + newStyleList[0]);

newStyleMap["key1"] = "put new value 1";
System.out.println("get('key2') " + newStyleMap["key2"]);
```

Vergleiche von Enums mit Operatoren

Werte von Enums lassen sich leicht miteinander vergleichen, da Enums das Interface `Comparable<T>` erfüllen und somit eine Ordnung definiert ist, die auf der Ordnungszahl basiert, die sich aus der Position des Enum-Werts im Sourcecode ableitet. Darüber hinaus kann man die Ordnungszahl zu einer Enum-Konstante mithilfe der Methode `ordinal()` ermitteln. Diese kann man mit den Operatoren `>`, `=`, `<` usw. vergleichen.

Sollen nun zwei Enum-Werte miteinander verglichen werden, so kann dies durch Einsatz der Methoden `ordinal()` bzw. `compareTo()` zwar leicht folgendermaßen realisiert werden – allerdings ist das Ganze nicht besonders schön zu lesen:

```
public enum EnumCompareExample
{
    KARO, HERZ, PIK, KREUZ;
```

```

public static void main(final String[] args)
{
    // Mögliche alte Schreibweisen
    if (KARO.compareTo(HERZ) < 0 ||           // Comparable<T>
        KARO.ordinal() < HERZ.ordinal())      // ordinal()
    {
        System.out.println("KARO < HERZ");
    }
}

```

Die Programmiersprache Groovy erlaubt es, Aufzählungswerte aus `enums` intuitiv verständlich mit den Operatoren `<`, `=` oder `>` miteinander zu vergleichen. Schon für JDK 7 gab es einen Erweiterungsvorschlag, diesen lesbaren Vergleich auch in Java zu ermöglichen. Wäre dieser umgesetzt worden, könnte man für das Beispiel der Farben von Spielkarten dann Folgendes schreiben:

```

// Mit JDK 8 leider nicht umgesetzt
public enum EnumCompareExample
{
    KARO, HERZ, PIK, KREUZ;

    public static void main(final String[] args)
    {
        if (KARO < HERZ) // Das geht leider nicht!
        {
            System.out.println("KARO < HERZ");
        }
    }
}

```

2.7 Zusammenfassung und Fazit

Durch die Lektüre dieses Kapitels sollten Sie einen recht guten Eindruck von den Neuerungen in Java 8 gewonnen haben. Dabei wurde eine Vielzahl von neuen Möglichkeiten durch Lambdas und Bulk Operations on Collections vorgestellt. Mit Java 8 gibt es nun außerdem ein gelungenes API zur Bearbeitung von Datumswerten und mit JavaFX 8 ein Framework zur Gestaltung moderner, ansprechender Benutzeroberflächen – bei Bedarf sogar in 3D.

Rekapitulieren wir kurz nochmal. Java 8 bietet mit ...

- Lambdas ein neues Programmiermodell
- Streams und Filter-Map-Reduce eine umfangreiche Erweiterung im Collections-Framework
- dem neuen Date and Time API eine deutliche Vereinfachung bei der Datumsberechnung
- JavaFX 8 verschiedene neue Bedienelemente und Unterstützung für 3D
- diversen API-Erweiterungen eine Erleichterung beim täglichen Entwickeln
- „Nashorn“ eine neue performante JavaScript-Engine

Allerdings sollte man zwei Punkte bedenken:

- Lambdas sind leider nicht ganz so eingängig wie die Closures von Groovy realisiert
- Streams erfordern einiges an Einarbeitungsaufwand, insbesondere dann, wenn teilweise die Details der verwendeten Functional Interfaces durchscheinen und für merkwürdige Fehlermeldungen sorgen.

Leider wurden diverse Funktionalitäten nicht umgesetzt, obwohl sie schon für Java 7 angekündigt waren, dann auf Java 8 verschoben wurden und hoffentlich mit Java 9 umgesetzt werden. Dies wichtigsten sind wohl:

- Modularisierung mit Jigsaw
- Integration von Collection-Literalen und Enum-Vergleichen in die Sprachsyntax

2.8 Weiterführende Literatur

Dieses Kapitel hat einen Überblick über einige Neuerungen von JDK 8 gegeben. Mit Java 8 wurden diverse bedeutende Änderungen an der Sprache selbst vorgenommen. Lambdas und die Möglichkeiten zur funktionalen Programmierung sind wegweisend, aber eigentlich auch längst überfällig, weil andere JVM-Sprachen wie Groovy oder aber auch die Microsoft Konkurrenz C# diese seit Längerem unterstützen.

Die Bandbreite an Neuerungen ist aber sehr umfangreich und konnte hier nur so weit behandelt werden, dass ein Einsatz in der Praxis erleichtert wird.

Wissenswertes zur funktionalen Programmierung und zu Java 8 finden Sie in den folgenden Büchern:

- **»Functional Programming Java: Harnessing the Power of Java 8 Lambda Expressions«** von Venkat Subramaniam [?]
Dieses Buch ist
- **»Java SE 8 for the Really Impatient «** von Cay S. Horstmann [?]
Dieses Buch

Weiterführende Informationen zum Thema Lambdas und JavaFX 8 bieten folgende Quellen:

- <http://de.slideshare.net/developmind/java8-lambda07clean>
- <https://www.java.net///community/javafx>
- JavaFX 8 Container-Terminal in 3D
<http://www.youtube.com/embed/AS26gZrYNy8?rel=0>