# 14 Ergänzungen in Java 9

Dieses Kapitel widmet sich einigen wichtigen Neuerungen von Java 9. Dabei liegt der Fokus auf Sprach- und API-Erweiterungen sowie auf Verbesserungen in der JVM. Das Kapitel untergliedert sich wie folgt:

- Syntaxerweiterungen (Abschnitt 14.1)
- Neues und Änderungen im JDK (Abschnitt 14.2)
- Änderungen in der JVM (Abschnitt 14.3)

Dem wichtigen Thema Modularisierung ist ein eigenes Kapitel gewidmet.

## 14.1 Syntaxerweiterungen

Bereits in JDK 7 wurden unter dem Projektnamen Coin verschiedene kleinere Syntaxerweiterungen in Java integriert. Für JDK 9 gab es ein Nachfolgeprojekt. Einige von dessen Neuerungen schauen wir uns im Anschluss an

### 14.1.1 Anonyme innere Klassen und der Diamond Operator

Bis einschließlich Java 8 kann bei der Definition anonymer innerer Klassen der Diamond Operator leider nicht genutzt werden, sondern der Typ ist explizit anzugeben. Mit JDK 9 ist es nun (endlich) erlaubt, die Typangabe auszulassen und somit den Diamond Operator zu verwenden, wie wir dies von anderen Variablendefinitionen bereits gewohnt sind. Die neue Syntax ist hier am Beispiel eines Komparators gezeigt:

```
final Comparator<String> byLength = new Comparator<>()
{
    ...
};
```

### 14.1.2 Erweiterung der `@Deprecated`-Annotation

Die `@Deprecated`-Annotation dient bekanntlich zum Markieren von obsoletem Sourcecode und besaß bislang keine Parameter. Das ändert sich mit JDK 9: Die `@Deprecated`-Annotation wurde um die zwei Parameter `since` und `forRemoval` erweitert. Das wurde nötig, weil in Zukunft geplant ist, veraltete Funktionalität aus dem

JDK zu entfernen, statt sie – wie bislang für Java üblich – aus Rückwärtskompatibilitätsgründen ewig beizubehalten. Das folgende Beispiel zeigt eine Anwendung, wie sie aus dem JDK stammen könnte:

```
@Deprecated(since = "1.5", forRemoval = true)
```

Mithilfe der neuen Parameter kann man für veralteten Sourcecode angeben, in welcher Version (`since`) dieser mit der Markierung als `@Deprecated` versehen wurde und ob der Wunsch besteht, die markierten Sourcecode-Teile in zukünftigen Versionen zu entfernen (`forRemoval`). Weil beide Parameter Defaultwerte besitzen (`since` = "" und `forRemoval` = false), können die Angaben jeweils für sich alleine stehen oder ganz entfallen.

Die Erweiterung der `@Deprecated`-Annotation lässt sich selbstverständlich auch für eigenen Sourcecode nutzen, wodurch angezeigt wird, dass gewisse Funktionalitäten für die Zukunft nicht mehr angeboten werden sollen. Darüber hinaus empfiehlt es sich, in einem Javadoc-Kommentar das `@deprecated`-Tag zu verwenden und dort den Grund der Deprecation und eine empfohlene Alternative aufzuführen. Nachfolgend ist dies exemplarisch für eine veraltete eigene Methode `someOldMethod()` gezeigt:

```
/**
 * @deprecated this method is replaced by someNewMethod()
  * ({@link #someNewMethod()}) which is more stable
 */
@Deprecated(since = "7.2", forRemoval = true)
private static void someOldMethod()
{
    // ...
}
```

### 14.1.3 Private Methoden in Interfaces

Allgemein bekannt ist, dass Interfaces der Definition von Schnittstellen dienen. Leider wurden mit JDK 8 statische Methoden und Defaultmethoden in Interfaces erlaubt, wodurch man Implementierungen in Interfaces vorgeben kann.[1] Das führt allerdings dazu, dass sich Interfaces kaum mehr von einer abstrakten Klasse unterscheiden: Abstrakte Klassen können ergänzend Zustand in Form von Attributen besitzen, was in Interfaces (noch) nicht geht.

Mit JDK 9 wurde der Unterschied zwischen Interfaces und abstrakten Klassen nochmals verringert, weil nun auch die Definition privater Methoden in Interfaces erlaubt ist. Das Argument dafür war, dass sich damit die Duplikation von Sourcecode in Defaultmethoden reduzieren ließe. Das mag richtig sein, allerdings ist es für die meisten Anwendungsprogrammierer eher fraglich, ob diese jemals Defaultmethoden selbst

[1] Dieser Schritt war designtechnisch nicht schön, aber nötig, um Rückwärtskompatibilität und doch Erweiterbarkeit zu erreichen und um vor allem die Neuerungen im Bereich der Streams nahtlos ins JDK 8 integrieren zu können.

implementieren sollten. Trotz dieser Kritik möchte ich Ihnen das Feature anhand eines Beispiels vorstellen, da es für Framework-Entwickler von Nutzen sein kann.

### Beispiel

Schauen wir uns zur Demonstration privater Methoden in Interfaces das nachfolgende Listing und vor allem die private Methode `myPrivateCalcSum(int, int)` sowie deren Aufruf aus den beiden öffentlichen Defaultmethoden an:

```
public interface PrivateMethodsExample
{
    // Tatsächliche Schnittstellendefinition - public abstract ist optional
    public abstract int method1();
    public abstract String method2();

    public default int sum(final String num1, final String num2)
    {
        final int value1 = Integer.parseInt(num1);
        final int value2 = Integer.parseInt(num2);

        return myPrivateCalcSum(value1, value2);
    }

    public default int sum(final int value1, final int value2)
    {
        return myPrivateCalcSum(value1, value2);
    }

    // Neu und unschön in JDK 9
    private int myPrivateCalcSum(final int value1, final int value2)
    {
        return value1 + value2;
    }
}
```

### Kommentar

Vielleicht fragen Sie sich, warum ich den privaten Methoden in Interfaces so ablehnend gegenüberstehe. Tatsächlich wurde die Büchse der Pandora bereits mit JDK 8 und den Defaultmethoden geöffnet. Die privaten Methoden mögen für Framework-Entwickler mitunter praktisch sein, jedoch besteht die Gefahr, dass sie für »normale« Entwickler noch attraktiver werden und von diesen somit ohne großes Hinterfragen zur Applikationsentwicklung eingesetzt werden. Das wäre aber im Hinblick auf das Design und die Klarheit von Business-Applikationen ein Schritt in die falsche Richtung.[2] Dadurch wird unter Umständen dem Schnittstellenentwurf weniger Aufmerksamkeit gewidmet, basierend auf der Annahme, dass benötigte Funktionalität immer noch nachträglich hinzugefügt werden kann.

[2] Dieser Nachteil verliert durch Nutzung einer modernen Microservice-Architektur etwas an Gewicht, da die Designsünde dann relativ isoliert existiert.

## 14.2 Neues und Änderungen im JDK

Nachdem wir verschiedene Syntaxänderungen kennengelernt haben, wollen wir uns nun einige wichtige Erweiterungen in den APIs des JDKs anschauen. Erwähnenswert sind sicher das neue Process-API sowie verschiedene Ergänzungen unter anderem im Stream-API und in den Klassen `Optional<T>` und `InputStream`. Darüber hinaus finden sich Neuerungen in den Klassen `Objects` sowie `CompletableFuture<T>`. Abschließend gehe ich auf Collection-Factory-Methoden ein. Für eine ausführlichere Behandlung der Neuerungen in Java 9 möchte ich Sie auf mein Buch »Java 9 – Die Neuerungen« [44] verweisen.

### 14.2.1 Das neue Process-API

Bis einschließlich JDK 8 sind die Möglichkeiten recht eingeschränkt, wenn es darum geht, Prozesse des Betriebssystems zu kontrollieren und zu verwalten. Ein Beispiel ist die Ermittlung der ID eines Prozesses, kurz PID genannt. Je nach Plattform muss man dies mit Java 8 unterschiedlich implementieren. Das geht etwa, indem man ein Shell-Kommando mit der Methode `exec()` aus der Klasse `java.lang.Runtime` ausführt.

#### PID mit JDK 9 ermitteln

Die Abfrage der PID mit Java 9 wird mithilfe der Klasse `java.lang.ProcessHandle` praktischerweise deutlich kürzer:

```
public static void main(final String[] args) throws InterruptedException,
                                                     IOException
{
    System.out.println("PID: " + ProcessHandle.current().pid());
}
```

***Listing 14.1*** *Ausführbar als* **'PIDEXAMPLE'**

Neben der offensichtlichen Kürze und besseren Lesbarkeit sowie Verständlichkeit bietet die Methode `pid()` einen betriebssystemunabhängigen Weg zur Ermittlung der Prozess-ID (zumindest aus Sicht des Aufrufers).

#### Das Interface `ProcessHandle`

Neben der PID kann man mithilfe von `ProcessHandle` eine ganze Reihe weiterer Informationen zu Prozessen auslesen. Dazu gibt es unter anderem folgende Methoden:

- `current()` – Ermittelt den aktuellen Prozess als `ProcessHandle`.
- `info()` – Stellt Infos zum Prozess in Form des inneren Interface `ProcessHandle.Info` bereit, etwa zu Benutzer, Kommando usw. wie dies nachfolgend aufgelistet wird.
- `info().command()` – Gibt das Kommando als `Optional<String>` aus einem `ProcessHandle.Info` zurück.

- `info().user()` – Liefert den Benutzer als `Optional<String>` aus einem `ProcessHandle.Info`.
- `info().totalCpuDuration()` – Ermittelt aus den Infos die CPU-Zeit als `Optional<Duration>`. Die Klasse `Duration` entstammt dem mit JDK 8 neu eingeführten Date and Time API.[3]

### Das Interface `ProcessHandle` im Einsatz

Zum besseren Verständnis der Arbeitsweise der genannten Methoden betrachten wir ein Beispiel:

```
public static void main(final String[] args) throws InterruptedException,
                                                     IOException
{
    final ProcessHandle current = ProcessHandle.current();
    printInfo(current);
}

private static void printInfo(final ProcessHandle current)
{
    System.out.println("PID: " + current.pid());
    System.out.println("Info: " + current.info());
    System.out.println("Command: " + current.info().command());
    System.out.println("CPU-Usage: " + current.info().totalCpuDuration ());
}
```

***Listing 14.2*** *Ausführbar als* **'PROCESSHANDLEEXAMPLE'**

Das Programm PROCESSHANDLEEXAMPLE gibt in etwa Folgendes aus (gekürzt):

```
PID: 6396
Info: [user: Optional[michaeli], cmd: /Library/Java/JavaVirtualMachines/jdk-9.
      jdk/Contents/Home/bin/java, args: [-Dfile.encoding=UTF-8, -classpath, /
      Users/min/Documents/workspaceNeon/Jdk9Examples/bin, jdk9example.processapi.
      ProcessHandleExample], startTime: Optional[2016-08-04T14:23:58.521Z],
      totalTime: Optional[PT0.321791S]]
Command: Optional[/Library/Java/JavaVirtualMachines/jdk-9.jdk/Contents/Home/bin/
      java]
CPU-Usage: Optional[PT0.469431S]
```

Neben der PID sieht man die umfangreichen Informationen aus dem `Info`-Objekt. Exemplarisch werden zudem die Werte für `command()` und `totalCpuDuration()` ausgegeben. Beide werden als `Optional<T>` zurückgeliefert. Für das mit `command()` als `Optional<String>` ermittelte Kommando erkennen wir, dass es sich um das Programm `java` aus JDK 9 handelt, das laut `totalCpuDuration()` etwa 0.47 Sekunden CPU-Zeit verbraucht hat, wie man es im `Optional<Duration>` sieht.

[3] Einen Überblick bietet Kapitel 11.

### Alle Prozesse abfragen

Neben Informationen zum aktuellen Prozess lassen sich Informationen für alle Prozesse des Benutzers sowie alle Subprozesse zu einem Prozess wie folgt ermitteln:

- `allProcesses()` – Liefert alle Prozesse als `Stream<ProcessHandle>`.
- `children()` – Ermittelt zu einem Prozess alle seine (direkten) Subprozesse als `Stream<ProcessHandle>`.

Im nachfolgenden Beispiel iterieren wir über das Ergebnis von `allProcesses()` und geben Infos zu solchen Prozessen aus, die Subprozesse besitzen. Die Anzahl an Subprozessen können wir durch Aufruf von `children().count()` erfragen:

```
public static void main(final String[] args) throws InterruptedException,
                                                     IOException
{
    System.out.println("All Processes:");
    showInfoForAllProcesses();
}

private static void showInfoForAllProcesses()
{
    ProcessHandle.allProcesses().forEach(processHandle ->
    {
        final Stream<ProcessHandle> children = processHandle.children();
        final long count = children.count();
        if (count > 0)
        {
            System.out.println("Info: " + processHandle.info() +
                               " has " + count + " children");
        }
    });
}
```

***Listing 14.3*** *Ausführbar als '***ALLPROCESSHANDLESEXAMPLE***'*

Das Programm ALLPROCESSHANDLESEXAMPLE produziert die folgenden Ausgaben (gekürzt), die eine Liste der zurückgelieferten Informationen widerspiegeln:

```
All Processes:
Info: [user: Optional[michaeli], cmd: /Applications/Adobe Acrobat Reader DC.app/
      Contents/MacOS/AdobeReader, args: [-psn_0_3822501], startTime: Optional
      [2016-08-02T21:16:30.322Z]] has 3 children
...
Info: [user: Optional[michaeli], cmd: /System/Library/CoreServices/Dock.app/
      Contents/MacOS/Dock, startTime: Optional[2016-07-24T08:17:12.938Z]] has 1
      children
Info: [user: Optional[root], startTime: Optional[2016-07-24T08:16:40.564Z]] has
      285 children
```

### Prozesse kontrollieren

Neben der Bereitstellung und Abfrage von Informationen zu Prozessen existieren auch Möglichkeiten, Prozesse zu beenden sowie auf das Ende eines Prozesses zu reagieren. Ergänzend zu den bereits aufgelisteten Methoden im Interface `ProcessHandle` findet man unter anderem folgende Methoden:

- `of(long)` – Liefert ein `Optional<ProcessHandle>` zu einer gegebenen PID.
- `destroy()` – Terminiert einen Prozess, sofern dies erlaubt ist. Ansonsten, etwa für den mit `current()` ermittelten Prozess, wird eine Exception ausgelöst:

```
Exception in thread "main" java.lang.IllegalStateException: destroy of
      current process not allowed
```

- `onExit()` – Liefert ein `CompletableFuture<ProcessHandle>` zurück, das man dazu nutzen kann, verschiedene Aktionen als Reaktion auf das Ende eines Prozesses auszuführen.

Mit diesen Methoden wollen wir ein Beispiel erstellen. Es soll zunächst mit `Runtime.exec()` ein Prozess gestartet werden. Als Rückgabe erhält man ein `java.lang.Process`-Objekt. Dieses bietet seit JDK 9 ebenfalls die Methode `pid()` sowie diverse andere, die auch durch `ProcessHandle` bereitgestellt werden. Auch kann man ein `Process`-Objekt durch einen Aufruf von `toHandle()` in ein `ProcessHandle`-Objekt transformieren:

```
public static void main(final String[] args) throws InterruptedException,
                                                     IOException
{
    // Prozess erzeugen
    final String command = "sleep 60s";
    final String commandWin = "cmd timeout 60";
    final Process sleeper = Runtime.getRuntime().exec(command);
    System.out.println("Started process is " + sleeper.pid());

    // Process => ProcessHandle
    final ProcessHandle sleeperHandle = ProcessHandle.of(sleeper.pid()).
                                        orElseThrow(IllegalStateException::new);
    final ProcessHandle sleeperHandle2 = sleeper.toHandle();
    System.out.println("Same handle? " + sleeperHandle.equals(sleeperHandle2));

    // Exit Handler registrieren
    final Runnable exitHandler = () -> System.out.println("exitHandler called");
    sleeperHandle.onExit().thenRun(exitHandler);
    System.out.println("Registered exitHandler");

    // Den Prozess zerstören und ein wenig warten,
    // damit onExit() ausgeführt werden kann
    System.out.println("Destroying process " + sleeperHandle.pid());
    sleeperHandle.destroy();
    Thread.sleep(500);
}
```

***Listing 14.4*** *Ausführbar als '***CONTROLPROCESSEXAMPLE***'*

Startet man das Programm CONTROLPROCESSEXAMPLE, so können wir anhand der folgenden Ausgaben recht gut die im Listing definierten Aktionen nachvollziehen:

```
Started process is 60392
Same handle? true
Registered exitHandler
Destroying process 60392
exitHandler called
```

### Fazit

Wir haben in verschiedenen Beispielen kennengelernt, wie man mit dem neuen Process-API mit Prozessen des Betriebssystems interagieren oder zumindest Informationen darüber gewinnen kann. Die große Stärke des Process-APIs ist, dass die Aktion aus Sicht eines Java-Entwicklers betriebssystemunabhängig erfolgen kann. Damit kommt man Javas Versprechen von einer weitestgehend betriebssystemunabhängigen Programmierung wieder ein Stück näher.

## 14.2.2 Neuerungen im Stream-API

Das Stream-API mit dem Interface `Stream<T>` stellt eine der wesentlichen Neuerungen in Java 8 dar. Streams besaßen bereits von Beginn an ein recht umfangreiches API.[4] Dieses wurde mit Java 9 nochmals leicht erweitert. Zunächst schauen wir uns folgende zwei neuen Methoden an:

- `takeWhile(Predicate<T>)` – Verarbeitet Elemente des Streams, solange die als `Predicate<T>` übergebene Bedingung erfüllt ist.
- `dropWhile(Predicate<T>)` – Überspringt Elemente des Streams, solange die als `Predicate<T>` übergebene Bedingung erfüllt ist.

Diese Ergänzungen findet man analog auch in den für die primitiven Typen `int`, `long` und `double` spezialisierten Stream-Klassen `IntStream`, `LongStream` sowie `DoubleStream`. Dort ist dann jeweils das Prädikat auf den korrespondierenden Typ angepasst, etwa `takeWhile(IntPredicate)`.

### Beispiel für die Methoden `takeWhile()` und `dropWhile()`

Zur Demonstration der Methode `takeWhile()` wird ein unendlicher Stream von Ganzzahlen durch Aufruf von `iterate()` auf einem `IntStream` erzeugt, der mit der Zahl 1 beginnt. Für `dropWhile()` nutzen wir einen Stream mit dem vordefinierten Wertebereich von 7 bis 14, den wir durch Aufruf von `rangeClosed()` konstruieren. Zur Darstellung einer Besonderheit bei der Verarbeitung mit `dropWhile()` verwenden wir schließlich einen Stream mit vordefinierten Werten, der durch einen Aufruf von `of()` erzeugt wird:

[4] Einen Einstieg in die Verarbeitung von Daten mit dem Stream-API bietet Kapitel 7.

```
public static void main(final String[] args)
{
    System.out.println("takeWhile");
    final IntStream stream1 = IntStream.iterate(1, n -> n + 1);
    System.out.println(stream1.takeWhile(n -> n < 10).
                               mapToObj(Integer::toString).
                               collect(joining(", ")));

    System.out.println("\ndropWhile 1");
    final IntStream stream2 = IntStream.rangeClosed(7, 14);
    System.out.println(stream2.dropWhile(n -> n < 10).
                               mapToObj(Integer::toString).
                               collect(joining(", ")));

    System.out.println("\ndropWhile 2");
    final IntStream stream3 = IntStream.of(7,9,11,13,15,5,3,1);
    System.out.println(stream3.dropWhile(n -> n < 10).
                               mapToObj(Integer::toString).
                               collect(joining(", ")));
}
```

***Listing 14.5*** *Ausführbar als* '**STREAMSEXAMPLE**'

Für alle Streams konvertieren wir durch den Aufruf von `mapToObj(Integer::toString)`[5] die Zahlen in einen String und bereiten mit `collect(joining(", "))` eine kommaseparierte Darstellung auf – die Methode `joining()` stammt aus der Klasse `Collectors` (vgl. Abschnitt 7.1.5) und wurde zur besseren Lesbarkeit statisch importiert. Mit diesem Wissen ist leicht nachvollziehbar, dass es zu den folgenden Ausgaben kommt, wenn man das Programm STREAMSEXAMPLE startet:

```
takeWhile
1, 2, 3, 4, 5, 6, 7, 8, 9

dropWhile 1
10, 11, 12, 13, 14

dropWhile 2
11, 13, 15, 5, 3, 1
```

Das Beispiel `dropWhile 2` verdeutlicht, dass bei Aufrufen von `dropWhile()` nur zu Beginn die Einhaltung der Bedingung überprüft wird. Gilt diese einmal, so erfolgt danach keine weitere Prüfung und es werden im Anschluss möglicherweise Elemente konsumiert, die gegen die angegebene Bedingung verstoßen. Dieser Fall kann für `takeWhile()` so nicht auftreten, da dort die Verarbeitung sofort abgebrochen würde.

**Beide Methoden in Kombination** Auch in Kombination können die beiden Methoden sinnvoll eingesetzt werden. Das gilt etwa immer dann, wenn zunächst Informationen so lange aussortiert werden sollen, bis diese einem gewissen Gütekriterium

[5] Alternativ wäre natürlich auch der Einsatz des Lambda-Ausdrucks `num -> "" + num` möglich gewesen, der jedoch die Intention der Konvertierung in einen String nicht so deutlich macht, wie die Methodenreferenz `Integer::toString`.

oder Wert entsprechen, und dann im Anschluss so lange gelesen werden sollen, bis eine Abbruchbedingung erfüllt ist.

Als Beispiel werden die Informationen aus einen `Stream<String>` extrahiert, die zwischen den Markierungen `<START>` und `<END>` liegen:

```
public static void main(final String[] args)
{
    Stream<String> words = Stream.of("ab", "bla", "<START>",
                                     "Hier", "steht", "der", "Text",
                                     "<END>", "saas", "bla");

    Stream<String> content = words.dropWhile(word -> !word.equals("<START>"))
                                  .skip(1)
                                  .takeWhile(word -> !word.equals("<END>"));

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

***Listing 14.6*** *Ausführbar als* **'DROPANDTAKEWHILEEXAMPLE'**

Das `skip(1)` ist nötig, um den Begrenzer `<START>` nicht mit in die Ergebnisliste aufzunehmen. Auf ähnliche Weise könnte man übrigens auch die Header- oder Body-Informationen eines HTML-Dokuments extrahieren.

Startet man das Programm DROPANDTAKEWHILEEXAMPLE, so sieht man sehr schön die Extraktion:

```
Hier
steht
der
Text
```

### Weitere Methoden

Neben den beiden Methoden `takeWhile()` und `dropWhile()` findet man für Streams folgende Neuerungen:

- `ofNullable(T)` – Liefert einen `Stream<T>` mit einem Element, sofern das übergebene Element ungleich `null` ist. Ansonsten wird ein leerer Stream erzeugt.
- `iterate(T, Predicate<? super T>, UnaryOperator<T>)` – Es wird ein `Stream<T>` mit dem als ersten Parameter übergebenen Startwert erzeugt. Die folgenden Werte werden durch den `UnaryOperator` berechnet. Im Gegensatz zu der bereits mit JDK 8 existierenden Methode `iterate(T, UnaryOperator<T>)` wird hierbei auch noch das übergebene `Predicate<T>` geprüft und die Erzeugung gestoppt, sobald dieses nicht mehr erfüllt ist.

Für die zweite Methode möchte ich ein Beispiel präsentieren. Zuvor haben wir zur Ausgabe der Zahlen von 1 bis 9 ein `IntPredicate` und die Methode `takeWhile()` mit einer Prüfung von `n -> n < 10` vorgenommen. Stattdessen können wir diese Prüfung auch direkt im Aufruf von `iterate()` durchführen. Ausgehend vom Wert 1 erzeugen

wir mit dem Lambda `n -> n + 1` als `IntUnaryOperator` eine aufsteigende Zahlenfolge, deren Ende der Berechnung über die Bedingung `n -> n < 10` gesteuert wird.

```
public static void main(final String[] args)
{
    // iterate() unterstützt seit JDK 9 eine Bedingung
    System.out.println("iterate with predicate");
    final IntStream stream = IntStream.iterate(1, n -> n < 10, n -> n + 1);
    System.out.println(stream.mapToObj(Integer::toString).
                              collect(joining(", ")));
}
```

***Listing 14.7*** *Ausführbar als* **'STREAMSITERATEEXAMPLE'**

Das Programm STREAMSITERATEEXAMPLE gibt erwartungsgemäß Folgendes aus:

```
iterate with predicate
1, 2, 3, 4, 5, 6, 7, 8, 9
```

Die Methode `iterate()` bietet damit eine sehr ähnliche Funktionalität wie eine klassische `for`-Schleife, allerdings mit dem Unterschied, dass die Aktionen im Stream lazy, also erst durch eine Terminal Operation wie etwa das obige `collect()`, ausgeführt werden. Die drei Arten von Operationen (Create, Intermediate und Terminal) wurden bereits in Abschnitt 7.1 bei der Beschreibung des Stream-APIs vorgestellt.

#### Fazit

Das in JDK 8 eingeführte Stream-API war bereits recht umfangreich. Die mit JDK 9 ergänzten Methoden im Interface `Stream<T>` stellen eine sinnvolle Komplettierung dar und runden das Anwendungsspektrum ab.

### 14.2.3 Erweiterungen rund um die Klasse `Optional`

Die Klasse `Optional<T>` wurde mit Java 8 eingeführt und erleichtert die Behandlung und Modellierung optionaler Werte, wie dies oft für Suchen oder den Spezialfall der Berechnung auf Basis leerer Ergebnismengen der Fall ist. Im Praxiseinsatz der Klasse `Optional<T>` war jedoch bislang noch die eine oder andere Schwachstelle festzustellen. Insbesondere betrifft dies folgende Aufgabenstellungen:

1. Das Ausführen von Aktionen auch im Negativfall.
2. Die Umwandlung in einen `Stream<T>`, um Daten weiterzuverarbeiten oder eine Kompatibilität mit dem Stream-API z. B. für Frameworks, die auf Streams arbeiten, herzustellen.
3. Die Verknüpfung der Resultate mehrerer Berechnungen, die `Optional<T>` liefern.

Befassen wir uns exemplarisch mit dem ersten Punkt. Die beiden anderen Schwachstellen lassen sich zwar ebenfalls beheben, dafür schauen wir aber später direkt auf die Möglichkeiten von JDK 9.

### Einsatz der Klasse `Optional<T>` am Beispiel

Die Betrachtung von `Optional<T>` beginnen wir mit der Implementierung einer Suche in der Methode `findCustomer(String)` – vereinfachend wird dazu ein `Stream<String>` mit fixen Werten mit `anyMatch()` wie folgt durchsucht:

```
private static Optional<String> findCustomer(final String customerId)
{
    System.out.println("findCustomer(" + customerId + ")");

    final Stream<String> customers = Stream.of("Tim", "Tom", "Mike", "Andy");
    if (customers.anyMatch(name -> name.contains(customerId)))
    {
        return Optional.of(customerId);
    }
    return Optional.empty();
}
```

In der nutzenden Applikation soll zunächst nur für erfolgreiche Suchen eine Aktion erfolgen. Das lässt sich mit `ifPresent(Consumer<? super T>)` ausdrücken:

```
public static void main(final String[] args)
{
    findCustomer("Tim").ifPresent(System.out::println);
    findCustomer("UNKNOWN").ifPresent(System.out::println);
}
```

***Listing 14.8*** *Ausführbar als '*FIRSTOPTIONALEXAMPLE*'*

Exemplarisch wird zuerst nach einem Kunden mit dem Namen `Tim` gesucht, um die Ausführung im Positivfall mit `ifPresent(Consumer<? super T>)` zu zeigen. Die anschließende Suche nach `UNKNOWN` verläuft erfolglos, sodass `Optional.EMPTY` (der Rückgabewert eines Aufrufs von `Optional.empty()`) als Ergebnis zurückgeliefert wird. Deswegen wird im zweiten Fall die Aktion nicht ausgeführt. Somit gibt das Programm FIRSTOPTIONALEXAMPLE Folgendes aus:

```
findCustomer(Tim)
Tim
findCustomer(UNKNOWN)
```

Im obigen Beispiel wird deutlich, dass für die Suche nach dem nicht vorhandenen Wert `UNKNOWN` keine Ausgabe oder Warnmeldung erfolgt. Oftmals soll aber auch in dem Fall, dass eine Suche erfolglos war, eine Aktion ausgeführt werden. Das lässt sich mit JDK 8 leider nicht mehr so elegant formulieren.[6] Vielmehr muss man die Fallunterscheidung selbst programmieren und zudem im Positivfall den Ergebniswert per `get()` aus dem `Optional<T>` wie folgt ermitteln:

[6] Man kann lediglich mit der Methode `orElseThrow()` eine Exception auslösen.

```
public static void main(final String[] args)
{
    final Optional<String> optCustomer1 = findCustomer("Tim");
    if (optCustomer1.isPresent())
    {
        System.out.println("found: " + optCustomer1.get());
    }
    else
    {
        System.out.println("not found");
    }

    final Optional<String> optCustomer2 = findCustomer("UNKNOWN");
    if (optCustomer2.isPresent())
    {
        System.out.println("found: " + optCustomer2.get());
    }
    else
    {
        System.out.println("not found");
    }
}
```

***Listing 14.9*** *Ausführbar als* '**SECONDOPTIONALEXAMPLE**'

Wird das Programm SECONDOPTIONALEXAMPLE ausgeführt, so wird der Eintrag `UNKNOWN` wiederum nicht gefunden. Jedoch kommt es diesmal zu einer Warnmeldung:

```
findCustomer(Tim)
found: Tim
findCustomer(UNKNOWN)
not found
```

Die gezeigte Fallunterscheidung ist zwar nicht kompliziert, jedoch möchte man diese auch nicht jedes Mal erneut ausprogrammieren. Seit JDK 8 bietet sich folgende allgemeingültige Utility-Methode an:

```
private static <T> void ifPresentOrElse(final Optional<T> optional,
                                        final Consumer<? super T> action,
                                        final Runnable elseAction)
{
    if (optional.isPresent())
    {
        action.accept(optional.get());
    }
    else
    {
        elseAction.run();
    }
}
```

Diese würde man dann wie folgt einsetzen:

```
public static void main(final String[] args)
{
    final Optional<String> optCustomer1 = findCustomer("Tim");
    ifPresentOrElse(optCustomer1,
                    customer -> System.out.println("found: " + customer),
                    () -> System.out.println("not found"));
```

```
    final Optional<String> optCustomer2 = findCustomer("UNKNOWN");
    ifPresentOrElse(optCustomer2,
                    customer -> System.out.println("found: " + customer),
                    () -> System.out.println("not found"));
}
```

***Listing 14.10*** *Ausführbar als* **'THIRDOPTIONALEXAMPLE'**

Das Programm THIRDOPTIONALEXAMPLE führt jeweils eine Aktion für den Positivfall einer Suche nach `Tim` und für den Negativfall bei `UNKNOWN` aus. Erwartungsgemäß kommt es damit zu folgenden Ausgaben:

```
findCustomer(Tim)
found: Tim
findCustomer(UNKNOWN)
not found
```

Der Einsatz der Utility-Methode ist schon ein guter Schritt, insbesondere dann, wenn man nur JDK 8 nutzen kann. Einfacher in der Handhabung wird es jedoch durch den Einsatz von JDK 9, was wir nun betrachten wollen.

### Erweiterungen in `Optional` in JDK 9

Durch die Erweiterungen der Klasse `Optional<T>` in JDK 9 wurden alle drei zuvor aufgelisteten Schwachstellen adressiert. Dazu dienen folgende Methoden:

- `ifPresentOrElse(Consumer<? super T>, Runnable)` – Erlaubt die Ausführung einer Aktion im Positiv- oder im Negativfall.
- `stream()` – Wandelt das `Optional<T>` in einen `Stream<T>` um.
- `or(Supplier<Optional<T>> supplier)` – Ermöglicht auf elegante Weise die Verknüpfung mehrerer Berechnungen.

**Die Methode `ifPresentOrElse()`** Durch Einsatz dieser Methode lässt sich das vorherige Beispiel prägnanter schreiben – neben der Angabe zweier Aktionen profitiert man insbesondere davon, dass im Positivfall direkt der Wert zugreifbar ist:

```
public static void main(final String[] args)
{
    final Optional<String> optCustomer1 = findCustomer("Tim");
    optCustomer1.ifPresentOrElse(str -> System.out.println("found: " + str),
                                 () -> System.out.println("not found"));

    final Optional<String> optCustomer2 = findCustomer("UNKNOWN");
    optCustomer2.ifPresentOrElse(str -> System.out.println("found: " + str),
                                 () -> System.out.println("not found"));
}
```

***Listing 14.11*** *Ausführbar als* **'OPTIONALIFPRESENTORELSEEXAMPLE'**

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

```
findCustomer(Tim)
found: Tim
findCustomer(UNKNWON)
not found
```

Wir sehen dieselben Ausgaben wie zuvor, jedoch ist die durch JDK 9 bereitgestellte Funktionalität ein wenig eleganter in der Schreibweise.

**Die Methode `stream()`** Manchmal ist es praktisch, ein `Optional<T>` in einen `Stream<T>` umzuwandeln. Das wird nun durch die Methode `stream()` möglich.

Deren Nutzen kann man sich gut für einen Stream von optionalen Werten verdeutlichen, in dem nur die Einträge mit gültigen Werten verbleiben sollen. Das kann man durch die Kombination der Methoden `flatMap()` und `stream()` erreichen. Hierbei wird jedes `Optional<T>` in einen Stream überführt und durch `flatMap()` zu einem einzigen Stream mit Werten kombiniert. Das Verfahren zeige ich am Beispiel eines Streams, der aus `Optional<String>`-Elementen besteht, beispielsweise als Folge einer parallelen Suche. Am Ende sollen die Ergebnisse konsolidiert werden. Diese Anforderung realisieren wir wie folgt:

```
public static void main(final String[] args)
{
    final Stream<Optional<String>> streamOfOptionalNames = Stream.of(
                                   Optional.of("Tim"), Optional.of("Tom"),
                                   Optional.empty(), Optional.of("Mike"),
                                   Optional.empty(), Optional.of("Andy"));

    final Stream<String> streamOfNames =
                         streamOfOptionalNames.flatMap(Optional::stream);

    streamOfNames.forEach(value -> System.out.println("found: " + value));
}
```

***Listing 14.12*** *Ausführbar als* **'OPTIONALSTREAMEXAMPLE'**

Das Programm OPTIONALSTREAMEXAMPLE produziert folgende Ausgaben:

```
found: Tim
found: Tom
found: Mike
found: Andy
```

Man erkennt, dass alle leeren Elemente (`Optional.empty()`) entfernt und die Werte aus den `Optional<String>`-Instanzen extrahiert wurden. Als Ergebnis entsteht ein `Stream<String>`. Das geschieht, weil die Methode `flatMap()` ineinander verschachtelte Streams zu einem flachen Stream zusammenfasst. Wenn man `Optional::stream` nutzt, wird aus einem `Optional.empty()` ein leerer Stream. Der Aufruf von `flatMap()` entfernt diesen dann automatisch.

**Die Methode `or()`** Nach den beiden grundlegenden Erweiterungen schauen wir abschließend auf die so unscheinbar wirkende Methode `or(Supplier<? extends Optional<? extends T>>)`. Mit deren Hilfe lassen sich Methoden bzw. Aufrufketten mit Fallback-Strategien auf lesbare und verständliche Art beschreiben, wie es die Methode `multiFindCustomer(String)` eindrucksvoll zeigt:[7]

```
public static void main(final String[] args)
{
    final Optional<String> optCustomer = multiFindCustomer("Tim");
    optCustomer.ifPresentOrElse(str -> System.out.println("found: " + str),
                                () -> System.out.println("not found"));
}

private static Optional<String> multiFindCustomer(final String customerId)
{
    return findInCache(customerId)
           .or(() -> findInMemory(customerId))
           .or(() -> findInDb(customerId));
}
```

***Listing 14.13*** *Ausführbar als '*OPTIONALORBEISPIEL*'*

Zwei der aufgerufenen Suchmethoden sind bewusst sehr simpel und kurz realisiert und liefern lediglich `Optional.EMPTY` zurück. Damit ergibt sich folgende Realisierung für `findInCache(String)` und `findInDb(String)`:

```
private static Optional<String> findInCache(final String customerId)
{
    System.out.println("findInCache");
    return Optional.empty();
}

private static Optional<String> findInDb(final String customerId)
{
    System.out.println("findInDb");
    return Optional.empty();
}
```

In der Methode `multiFindCustomer(String)` werden nacheinander drei Abfragen ausgeführt, sofern nicht zuvor ein Treffer gefunden wird. Das ist hier für den zweiten Aufruf `findInMemory(String)` der Fall, weil diese Methode eine Suche in einem `Stream<String>` wie folgt ausführt:

```
private static Optional<String> findInMemory(final String customerId)
{
    System.out.println("findInMemory");
    final Stream<String> customers = Stream.of("Tim", "Tom", "Mike", "Andy");

    return customers.filter(name -> name.contains(customerId))
                    .findFirst();
}
```

[7] Bei der Beschreibung für die Methode `or()` habe ich mich von einem Beispiel des Blogs `http://blog.codefx.org/java/dev/java-9-optional/` inspirieren lassen.

Startet man das Programm OPTIONALORELSEEXAMPLE, so kommt es zu folgenden Ausgaben, die die Arbeitsweise verdeutlichen:

```
findInCache
findInMemory
Tim
```

Zunächst wird nach `Tim` im Cache durch `findInCache(String)` gesucht, jedoch ohne Erfolg. Das zurückgelieferte `Optional.EMPTY` führt automatisch dazu, dass die zweite Methode in der Aufrufkette, also hier `findInMemory(String)`, ausgeführt wird. Weil die Suche nach `Tim` dort erfolgreich ist, bricht die Verarbeitung ab und es wird das Suchergebnis als `Optional<String>` zurückgegeben.

### Fazit

Die Klasse `Optional<T>` wurde in Java 9 sinnvoll erweitert. Das gilt für die lesbare Definition von Verarbeitungsketten mit `or()` sowie für die Methode `ifPresentOrElse()`, weil damit nun endlich die Verarbeitung des Positiv- und Negativfalls direkt mithilfe einer Methode aus dem JDK möglich ist. Und schließlich gibt es für die Methode `stream()` ab und an einen Praxiseinsatz, wie es zuvor für die Kombination mehrerer Suchen mit `flatMap(Optional::stream)` angedeutet wurde.

## 14.2.4 Erweiterungen in der Klasse `InputStream`

Die Verarbeitung von `Input-` und `OutputStream`s bietet bis einschließlich JDK 8 mitunter nicht den Komfort, den man erwarten würde. Mit JDK 9 wurde die Situation leicht verbessert und die Klasse `InputStream` um folgende zwei Methoden erweitert:

- `readAllBytes()` – Liest alle Bytes aus dem Stream.
- `transferTo(OutputStream)` – Diese Methode erlaubt das Kopieren von Daten von einem `InputStream` in einen `OutputStream`.

Anhand eines Beispiels wollen wir die beiden Methoden im Einsatz erleben:

```
public static void main(final String[] args) throws IOException
{
    final byte[] buffer = { 72, 65, 76, 76, 79 };

    // Liest alle Bytes in einem Rutsch
    final byte[] result = new ByteArrayInputStream(buffer).readAllBytes();
    System.out.println(Arrays.toString(result));

    // Überträgt Daten direkt aus einem InputStream in einen OutputStream
    new ByteArrayInputStream(buffer).transferTo(System.out);
}
```

***Listing 14.14*** *Ausführbar als* **'INPUTSTREAMJDK9EXAMPLE'**

Startet man das Programm INPUTSTREAMJDK9EXAMPLE, so werden die Daten aus dem `byte[]` namens `buffer` eingelesen und später auf die Konsole transferiert. Dabei kommt es zu folgenden Ausgaben:

```
[72, 65, 76, 76, 79]
HALLO
```

Vielleicht fragen Sie sich, wieso ein `byte` als Zeichen ausgegeben wird. Das liegt daran, dass die Zeichen den ASCII-Werten der Buchstaben entsprechen.

#### Fazit

Die Erweiterung in der Klasse `InputStream` ist zwar nur klein, aber fein. Die Methode `readAllBytes()` zum Einlesen der Daten eines Streams ist viel angenehmer in der Handhabung, als alle Bytes in einer Schleife einlesen zu müssen. Auch das Kopieren von Daten ist mithilfe von `transferTo(OutputStream)` nun mit einem Einzeiler zu lösen.

### 14.2.5 Erweiterungen in der Klasse `Objects`

Die Utility-Klasse `java.util.Objects` erlaubt es, durch die Methode `requireNonNull()` eine elegante Prüfung von Preconditions bezüglich `null` durchzuführen.

Mit JDK 9 wird das Ganze noch handlicher: Es ist nun analog zu einigen Methoden aus `Optional<T>` möglich, mit `requireNonNullElse()` bzw. `requireNonNullElseGet()` einen Alternativwert im Falle eines `null`-Werts bereitzustellen bzw. durch einen `Supplier<T>` zu berechnen.

Als Beispiel dient die Methode `generateMsg()`, die zwei `String`-Parameter besitzt und daraus eine Nachricht erzeugt. Dabei erfolgt ein Null-Handling unter Einsatz der genannten Methoden.

```
private static String generateMsg(final String msg, final String param)
{
    final String message = Objects.requireNonNullElse(msg, "Default-Msg");
    final String parameter =
                Objects.requireNonNullElseGet(param, () -> "No Param");

    return message + " : " + parameter;
}
```

In der `main()`-Methode übergeben wir zwei Mal den Wert `null`:

```
public static void main(final String[] args)
{
    System.out.println(generateMsg(null, null));
}
```

***Listing 14.15*** *Ausführbar als '***OBJECTSNONNULLEXAMPLE***'*

Startet man das Programm OBJECTSNONNULLEXAMPLE, so wird die Methode `generateMsg()` mit zwei `null`-Werten aufgerufen. In der Methode selbst werden

diese mithilfe der zuvor genannten Methoden entsprechend behandelt und direkt in den String `Default-Msg` und im zweiten Fall als `Supplier<String>` in den Wert `No Param` transformiert. Somit kommt es zu der folgenden Ausgabe:

```
Default-Msg : No Param
```

### 14.2.6 Erweiterungen in der Klasse `CompletableFuture`

Die Klasse `CompletableFuture<T>` wurde in Java 8 eingeführt und bietet eine Erleichterung bei der Programmierung asynchroner Abläufe. Diese lassen sich bezüglich Lesbarkeit und Komplexität deutlich besser als Lösungen mit Threads gestalten. Eine umfangreichere Einführung zur Klasse `CompletableFuture<T>` finden Sie in Abschnitt 9.6.4.

Nachfolgend gehe ich auf die Neuerungen in JDK 9 ein, wo unter anderem folgende Methoden ergänzt wurden:

- `completeAsync(Supplier<? extends T>)` und `completeAsync(Supplier<? extends T>, Executor)` – Erfüllt das `CompletableFuture<T>` mit dem vom übergebenen `Supplier<T>` gelieferten Ergebnis. Dieses wird asynchron von einem Task berechnet, der entweder vom Default Executor oder dem übergebenen ausgeführt wird.
- `orTimeout(long, TimeUnit)` – Sofern das `CompletableFuture<T>` nicht zuvor erfolgreich ausgeführt wurde, wird es mit einer `TimeoutException` beendet, wenn die angegebene Time-out-Zeit erreicht ist.
- `completeOnTimeout(T, long, TimeUnit)` – Das `CompletableFuture<T>` wird mit dem übergebenen Wert erfüllt, falls die Berechnungen nicht innerhalb der gegebenen Time-out-Zeit zum Ergebnis führen.
- `failedFuture(Throwable)` – Gibt ein `CompletableFuture<T>` zurück, das bereits durch die übergebene Exception erfüllt wurde. Dies kann man zum Signalisieren von Fehlerzuständen während einer asynchronen Berechnung nutzen.

Für die aufgelisteten Methoden wollen wir ein Beispiel erstellen:

```
public static void main(final String[] args) throws ExecutionException
{
    new CompletableFutureJdk9Example().perform();
}

public void perform() throws ExecutionException
{
   CompletableFuture.supplyAsync(this::longRunningCreateMsg)
                    .completeAsync(() -> "COMPLETE")
                    .thenAccept(this::notifySubscribers);

   CompletableFuture.supplyAsync(this::longRunningCreateMsg)
                    .orTimeout(3, TimeUnit.SECONDS)
                    .exceptionally(ex -> "exception occurred: " + ex)
                    .thenAccept(this::notifySubscribers);
```

```
    CompletableFuture.supplyAsync(this::longRunningCreateMsg)
                     .completeOnTimeout("TIMEOUT-FALLBACK", 2, TimeUnit.SECONDS)
                     .thenAccept(this::notifySubscribers);

    CompletableFuture.failedFuture(new IllegalStateException())
                     .exceptionally(ex -> {
                            System.out.println("ALWAYS FAILING");
                            return -1;
                      });

    sleepInSeconds(10);  // Auf die Terminierung des CompletableFutures warten
}

public String longRunningCreateMsg(final int durationInSecs)
{
    System.out.println(getCurrentThread() + " >>> longRunningCreateMsg");
    sleepInSeconds(durationInSecs);
    System.out.println(getCurrentThread() + " <<< longRunningCreateMsg");

    return "longRunningCreateMsg";
}

public String getCurrentThread()
{
    return Thread.currentThread().getName();
}

public void notifySubscribers(final String msg)
{
    System.out.println(getCurrentThread() + " notifySubscribers: " + msg);
}

public String failingMsg()
{
    throw new IllegalStateException("ISE");
}

private void sleepInSeconds(final int durationInSeconds)
{
    try
    {
        TimeUnit.SECONDS.sleep(durationInSeconds);
    }
    catch (InterruptedException e)
    {  /* not possible here */ }
}
```

***Listing 14.16*** *Ausführbar als '*COMPLETABLEFUTUREJDK9EXAMPLE*'*

Zunächst wird asynchron eine rund fünf Sekunden dauernde Methode `longRunningCreateMsg()` ausgeführt. Deren Verarbeitung kann man durch `completeAsync()` vorzeitig als beendet markieren. Hier wird als Ergebnis `COMPLETE` mithilfe der Aufrufe von `thenAccept()` und `notifySubscribers()` ausgegeben. Die zweite Variante führt die lang dauernde Methode asynchron aus, diesmal kommt es aber nach drei Sekunden zu einem Time-out. Das führt zur Ausgabe von `exception occurred: java.util.concurrent.TimeoutException`. Die dritte Variante zeigt, wie man bei einem Time-out einen Wert zurückliefern kann. Dies geschieht hier nach zwei Sekunden. Verbleibt noch die Methode `failedFuture()`. Hiermit lässt sich das

`CompletableFuture<T>` mit einem Fehler komplettieren. In diesem Fall wird dadurch `ALWAYS FAILING` ausgegeben.

Startet man das obige Programm COMPLETABLEFUTUREJDK9EXAMPLE, so kommt es zu folgenden Ausgaben, wobei die Reihenfolge durch die Nebenläufigkeit abweichen kann:

```
ForkJoinPool.commonPool-worker-9 >>> longRunningCreateMsg
main notifySubscribers: COMPLETE
ForkJoinPool.commonPool-worker-2 >>> longRunningCreateMsg
ForkJoinPool.commonPool-worker-11 >>> longRunningCreateMsg
ALWAYS FAILING
CompletableFutureDelayScheduler notifySubscribers: TIMEOUT-FALLBACK
CompletableFutureDelayScheduler notifySubscribers: exception occurred: java.util
      .concurrent.TimeoutException
ForkJoinPool.commonPool-worker-9 <<< longRunningCreateMsg
ForkJoinPool.commonPool-worker-2 <<< longRunningCreateMsg
ForkJoinPool.commonPool-worker-11 <<< longRunningCreateMsg
```

Hier sieht man die Ausgabe des Ergebnisses der letzten Aktion `ALWAYS FAILING` noch vor der Protokollierung der zweiten und dritten Aktion. Insbesondere wird auch deutlich, dass alle länger laufenden Aktionen vollständig abgearbeitet werden – nur der Ergebniswert im `CompletableFuture<T>` wird durch die Aktionen anders als der Rückgabewert der Methode `longRunningCreateMsg` ausgewertet.

## 14.2.7 Collection-Factory-Methoden

Das Erzeugen von Collections für eine kleinere Menge fest definierter Werte ist mitunter etwas umständlich. Sprachen wie Groovy oder Python bieten dafür eine spezielle Syntax, sogenannte ***Collection-Literale***. Schon im Jahr 2009 hat man auch für Java über eine Integration einer solchen einfacheren Schreibweise zur Erzeugung von und zum Zugriff auf Collections nachgedacht. Allerdings wurde nichts Derartiges realisiert, obwohl es einige vielversprechende Vorschläge gab.

### Vorschlag: Collection-Literale und Collection-Erzeugung

Nachfolgendes Listing zeigt, wie eine mögliche Syntax für Collection-Literale für die Collections `List<E>`, `Set<E>` und `Map<K,V>` aussehen könnte. Dabei werden die Elemente der Collection in geschweifte oder eckige Klammern eingeschlossen:

```
// Leider weder mit JDK 8 noch JDK 9 umgesetzt
final List<String> newStyleList = ["item1", "item2"];
final Set<String> names = {"Tim", "Mike"};
final Map<String, String> newStyleMap = ["key1" : "value1", "key2" : "value2"];
```

### Realisierung mit JDK 9

Leider wurden Collection-Literale nicht in der zuvor beschriebenen Form in Java 9 realisiert. Stattdessen wurde eine Armada an Factory-Methoden mit den Namen `of()` bzw. `ofEntries()` in die Interfaces `List<E>`, `Set<E>` und `Map<K,V>` integriert, die sich wie folgt zur Erzeugung von Collections nutzen lassen:[8]

```
public static void main(final String[] args)
{
    final List<String> names = List.of("MAX", "MORITZ", "MIKE");
    names.forEach(name -> System.out.println(name)); // oder System.out::println

    final Set<Integer> numbers = Set.of(1, 2, 3);
    numbers.forEach(number -> System.out.println(number));

    final Map<Integer, String> mapping = Map.of(5, "five", 6, "six");
    final Map<Integer, String> mapping2 = Map.ofEntries(entry(5, "five"),
                                                        entry(6, "six"));

    mapping.forEach((key, value) -> System.out.println(key + ":" + value));
    mapping2.forEach((key, value) -> System.out.println(key + ":" + value));
}
```

***Listing 14.17*** *Ausführbar als '*COLLECTIONFACTORYMETHODSEXAMPLE*'*

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

```
MAX
MORITZ
MIKE
1
2
3
6:six
5:five
6:six
5:five
```

Die Reihenfolge der Elemente bei Sets und Maps ist bei der Nutzung der Collection-Factory-Methoden allerdings nicht stabil und insbesondere wird die Einfügereihenfolge nicht beibehalten. Vielmehr ist die Reihenfolge bei einer Iteration zufällig, sodass es für obiges Programm durchaus auch zu anderen Reihenfolgen kommen kann.

### Besonderheiten bei Duplikaten

Beim Einsatz der Collection-Factory-Methoden sollte man ein Detail für `Set<E>` und `Map<K,V>` kennen: Bekanntermaßen modelliert ein `Set<E>` das mathematische Konzept einer Menge und enthält somit keine Duplikate. Das gilt auch für die Schlüssel in Maps. Diese Eigenschaft wurde bei den bisherigen Collections automatisch sichergestellt, indem beim Einfügen von Elementen gegebenenfalls Duplikate aussor-

[8] Zur besseren Lesbarkeit erfolgt ein statischer Import von `java.util.Map.entry`.

tiert wurden.[9] Das war für diverse Anwendungsfälle ein recht praktisches Feature. Die Collection-Factory-Methoden weisen allerdings eine nicht überraschungsfreie Besonderheit auf: Für Sets prüfen sie beim Aufruf der Konstruktionsmethoden auf Duplikatfreiheit. Ist diese nicht gegeben, so lösen sie eine Exception aus. Gleiches gilt auch für die Schlüssel von Maps. Bei Listen findet dagegen keine Duplikatsprüfung statt. Schauen wir uns ein Beispiel an:

```
final Set<String> names = Set.of("MAX", "Moritz", "MAX");
```

Bei dieser Variablendefinition kommt es zur Laufzeit zu folgender Exception:

```
Exception in thread "main" java.lang.IllegalArgumentException:
      duplicate element: MAX at java.util.ImmutableCollections$SetN.<init>(java.
      base@9-ea/ImmutableCollections.java:329)
```

Weil die Collections direkt anhand der übergebenen Werte konstruiert werden, kann man jedoch auch ein Grund für dieses Verhalten finden, nämlich die Vermeidung von Inkonsistenzen durch Flüchtigkeitsfehler in Form einer Mehrfachangabe.

#### Fazit

Die Collection-Factory-Methoden können mich nicht vollständig überzeugen. Für die Definition kleiner Wertbestände gefallen sie mir schon, auch wenn sie nicht ganz so elegant in der Schreibweise sind, wie es Collection-Literale wären. Allerdings ist die Duplikatbehandlung für `Set<E>` zumindest nicht überraschungsfrei – die ausgelöste Exception ist meiner Meinung nach sogar kontraintuitiv. Insgesamt lässt sich festhalten, dass die Negativpunkte im Programmieralltag kaum ins Gewicht fallen.

## 14.3 Änderungen in der JVM

In diesem Unterkapitel beschäftigen wir uns mit ein paar Änderungen in der JVM von Oracle, die mit JDK 9 eingeführt werden.

### 14.3.1 Garbage Collection

Java befreit den Entwickler weitestgehend von der Aufgabe, sich über die Verwaltung und Freigabe von Speicher Gedanken zu machen. Dazu ist ein Mechanismus namens Garbage Collection in die JVM integriert, der nicht mehr verwendete Objekte erkennen und deren Speicherplatz freigeben kann.

Bei der Garbage Collection finden sich für die JVM zwei Neuerungen: zum einen beim Standard-Garbage-Collector und zum anderen in Form der Entfernung veralteter Kombinationen von Garbage Collectors.

[9] Das erfordert, dass die steuernden Methoden wie `equals()`, `hashCode()` usw. korrekt implementiert sind. Details finden Sie in Abschnitt 6.1.9.