Lambda-Ausdrücke und funktionale Programmierung

„EDV-Systeme verarbeiten, womit sie gefüttert werden.
Kommt Mist rein, kommt Mist raus.“
– André Kostolany (1906–1999)

Bei der Entwicklung von Maschinensprache (bzw. Assembler) hin zur Hochsprache ist eine interessante Geschichte der Parametrisierung abzulesen. Schon die ersten Hochsprachen erlaubten eine Parametrisierung von Funktionen mit unterschiedlichen Argumenten. Die Programmiersprache Java, die im Jahr 1996 geboren wurde, bot das von Anfang an, da sie erst mehrere Jahrzehnte nach den ersten Hochsprachen entstand. Relativ spät folgten dann die Generics. Die Parametrisierung des Typs wurde erst 2004 mit der Version 5 realisiert. Bis dahin konnte eine Liste zum Beispiel Zeichenketten ebenso enthalten wie Pantoffeltierchen (als Java-Objekte). Funktionale Programmierung ermöglichte nun eine Parametrisierung des Verhaltens; eine Sortiermethode arbeitet immer gleich, aber ihr Verhalten bei den Vergleichen wird angepasst. Das ist eine ganz andere Qualität, als unterschiedliche Werte zu übergeben. Das bietet nun seit 2014 die Version Java 8 elegant und einfach mit den so genannten Lambda-Ausdrücken.

1.1 Code = Daten

Wer den Begriff „Daten“ hört, denkt zunächst einmal an Zahlen, Bytes, Zeichenketten oder auch komplexe Objekte mit ihrem Zustand. Wir wollen in diesem Kapitel diese Sicht ein wenig erweitern und auf Programmcode lenken. Java-Code, versinnbildlicht als Serie von Bytecodes, besteht auch aus Daten. Und wenn wir uns einmal auf diese Sichtweise einlassen, dass Code gleich Daten ist, dann lässt sich Code auch wie Daten übergeben und so von einem Punkt zum anderen übertragen, speichern und später referenzieren. Mit dieser Möglichkeit, Code zu übertragen, lässt sich das Verhalten von Algorithmen leicht anpassen. Beginnen wir mit ein paar Beispielen, bei denen Programmcode übergeben wird, auf den dann später zugegriffen wird:

- Ein Thread führt Programmcode im Hintergrund aus. Der Programmcode, den der Java-Thread ausführen soll, wird in ein Objekt vom Typ `Runnable` verpackt, genau genommen in eine `run()`-Methode gesetzt. Kommt der Thread zum Zuge, ruft er die `run()`-Methode auf.
- Ein `Timer` ist eine `java.util`-Klasse, die zu bestimmen Zeitpunkten Programmcode ausführen kann. Der Objektmethode `scheduleAtFixedRate(…)` wird dabei ein Objekt vom Typ `TimerTask` übergeben, das den Programmcode enthält.

- Zum Sortieren von Daten kann eine eigene Ordnung definiert werden, die dem Sortierer als `Comparator` übergeben werden kann. Der `Comparator` deklariert eine Vergleichsmethode, an die sich der Sortierer wendet, um zwei Objekte in die gewünschte Reihenfolge zu bringen.
- Aktiviert der Benutzer auf der Oberfläche eine Schaltfläche, so führt das zu einer Aktion. Der Programmcode steckt – beim UI-Framework Swing – in einem Objekt vom Typ `ActionListener` und wird an der Schaltfläche `JButton` mit `addActionListener(…)` fest gemacht. Kommt es zu einer Schaltflächenaktivierung, arbeitet das UI-System den Programmcode in der Methode `actionPerformed(…)` des gespeicherten `ActionListener` ab.

Um Programmcode von einer Stelle zur anderen zu bringen, wird in Java immer der gleiche Mechanismus eingesetzt: Eine Klasse implementiert eine (in der Regel nichtstatische) Methode, in der der auszuführende Programmcode steht. Ein Objekt dieser Klasse wird an eine andere Stelle übergeben, und der Interessent greift dann über die Methode auf den Programmcode zu. Dass ein Objekt noch mehr als diese eine Implementierung enthalten kann, etwa Variablen, Konstanten, Konstruktoren, ist dafür nicht relevant. Diesen Mechanismus schauen wir uns jetzt in verschiedenen Varianten genauer an.

Innere Klassen als Code-Transporter

Bleiben wir bei dem Beispiel mit den Vergleichen. Angenommen, wir sollen Strings so sortieren, dass Leerraum vorne und hinten bei den Vergleichen ignoriert wird, also `" Newton "` gleich `"Newton"` ist. Bei Vorgaben dieser Art muss einem Sortieralgorithmus ein Stückchen Code übergeben werden, damit er die korrekte Reihenfolge herstellen kann. Praktisch sieht das so aus:

Listing 0.1: CompareTrimmedStrings.java

```
import java.util.*;
public class CompareTrimmedStrings {
  public static void main( String[] args ) {
    class TrimmingComparator implements Comparator<String> {
      @Override public int compare( String s1, String s2 ) {
        return s1.trim().compareTo( s2.trim() );
      }
    }
    String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
    Arrays.sort( words, new TrimmingComparator() );
    System.out.println( Arrays.toString( words ) );
  }
}
```

Die Ausgabe ist:

```
[        Adele     , M,  Q,
Skyfall]
```

Der `TrimmingComparator` enthält in der `compare(…)`-Methode den Programmcode für die Vergleichslogik. Ein Exemplar vom `TrimmingComparator` wird aufgebaut und `Arrays.sort(…)` übergeben. Das geht mit weniger Code!

Innere anonyme Klassen als Code-Transporter

Klassen enthalten Programmcode, und Exemplare der Klassen werden an Methoden wie `sort(…)` übergeben, damit der Programmcode dort hinkommt, wo er gebraucht wird. Doch elegant ist das nicht. Für die Beschreibung des Programmcodes ist extra eine eigene Klasse erforderlich. Das ist viel Schreibarbeit, und über eine innere anonyme Klasse lässt sich der Programmcode schon ein wenig verkürzen:

```
String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
Arrays.sort( words, new Comparator<String>() {
  @Override public int compare( String s1, String s2 ) {
    return s1.trim().compareTo( s2.trim() );
  } } );
System.out.println( Arrays.toString( words ) );
```

Allerdings ist das immer noch aufwändig: Wir müssen eine Methode überschreiben und dann ein Objekt aufbauen. Für Programmautoren ist das lästig, und die JVM hat es mit vielen überflüssigen Klassendeklarationen zu tun. Die Frage ist: Wenn der Compiler weiß, dass bei `sort(…)` ein `Comparator` nötig ist, und wenn ein `Comparator` sowieso nur eine Methode hat, muss dann `Comparator` und `compare(…)` überhaupt genannt werden?

Abkürzende Schreibweise durch Lambda-Ausdrücke

Ab Java 8 lässt sich Programmcode leichter an eine Methode übergeben, denn es gibt eine kompakte Syntax für die Implementierung von Schnittstellen mit einer Operation. Für unser Beispiel sieht das so aus:

```
String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
Arrays.sort( words,
            (String s1, String s2) -> { return s1.trim().compareTo(s2.trim()); } );
System.out.println( Arrays.toString( words ) );
```

Der in fett gesetzte Ausdruck nennt sich *Lambda-Ausdruck*. Er ist eine kompakte Art und Weise, Schnittstellen mit genau einer Methode zu implementieren; die Schnittstelle `Comparator` hat genau eine Operation `compare(…)`.

Optisch sind sich ein Lambda-Ausdruck und eine Methodendeklaration ähnlich; was wegfällt sind Modifizierer, der Rückgabetyp, der Methodenname und (mögliche) `throws`-Klauseln.

Methodendeklaration	Lambda-Ausdruck

`public int compare`
`( String s1, String s2 )`
 	
`{ return s1.trim().compareTo( s2.t`
`() ); }`
	
	`( String s1, String s2 )`
	`->`
	`{ return s1.trim().compareTo( s2.t`
	`() ); }`

Tabelle 0.1: Vergleich der Methodendeklaration einer Schnittstelle mit dem Lambda-Ausdruck

Wenn wir uns den Lambda-Ausdruck als Implementierung dieser Schnittstelle anschauen, dann lässt sich dort nichts von `Comparator` oder `compare(…)` ablesen – ein Lambda-Ausdruck repräsentiert mehr oder weniger nur den Java-Code und lässt das, was der Compiler aus dem Kontext herleiten kann, weg.

Alle Lambda-Ausdrücke lassen sich in einer Syntax formulieren, die die folgende allgemeine Form hat:

```
( LambdaParameter ) -> { Anweisungen }
```

Lambda-Parameter sind sozusagen die Eingabewerte für die Anweisungen. Die Parameterliste wird so deklariert, wie von Methoden oder Konstruktoren bekannt, allerdings gibt es keine Varargs. Es gibt syntaktische Abkürzungen, wie wir später sehen werden, doch vorerst bleiben wir bei dieser Schreibweise.

Geschichte

Der Java-Begriff „Lambda-Ausdruck“ geht auf das Lambda-Kalkül (in der englischen Literatur *Lambda calculus* genannt, auch geschrieben als λ-calculus) aus den 1930er Jahren zurück und ist eine formale Sprache zur Untersuchung von Funktionen.

1.2 Funktionale Schnittstellen und Lambda-Ausdrücke im Detail

In unserem Beispiel haben wir den Lambda-Ausdruck als Argument von `Array.sort(…)` eingesetzt:

```
Arrays.sort( words,
             (String s1, String s2) -> { return s1.trim().compareTo(s2.trim()); } );
```

Wir hätten aber auch den Lambda-Ausdruck explizit einer lokalen Variablen zuweisen können, was deutlich macht, dass der hier eingesetzte Lambda-Ausdruck vom Typ `Comparator` ist:

```
Comparator<String> c = (String s1, String s2) -> {
                          return s1.trim().compareTo( s2.trim() ); }
Arrays.sort( words, c );
```

1.2.1 Funktionale Schnittstellen

Nicht zu jeder Schnittstelle gibt es eine Abkürzung über einen Lambda-Ausdruck, und es gibt eine zentrale Bedingung, wann ein Lambda-Ausdruck verwendet werden kann.

Definition

Schnittstellen, die nur eine Operation (abstrakte Methode) besitzen, heißen *funktionale Schnittstellen*. Ein *Funktionsdeskriptor* beschreibt diese Methode. Eine abstrakte Klasse mit genau einer abstrakten Methode zählt *nicht* als funktionale Schnittstelle.

Lambda-Ausdrücke und funktionale Schnittstellen haben eine ganz besondere Beziehung, denn ein Lambda-Ausdruck ist ein Exemplar einer solchen funktionalen Schnittstelle. Natürlich müssen Typen und Ausnahmen passen. Dass funktionale Schnittstellen genau eine abstrakte Methode vorschreiben, ist eine naheliegende Einschränkung, denn gäbe es mehrere, müsste ein Lambda-Ausdruck ja auch mehrere Implementierungen anbieten oder irgendwie eine Methode bevorzugen und andere ausblenden.

Wenn wir ein Objekt vom Typ einer funktionalen Schnittstelle aufbauen möchten, können wir folglich zwei Wege einschlagen: Es lässt sich die traditionelle Konstruktion über die Bildung von Klassen wählen, die funktionale Schnittstellen implementieren, und dann mit `new` ein Exemplar bilden, oder es lässt sich mit kompakten Lambda-Ausdrücken arbeiten. Moderne IDEs zeigen uns an, wenn kompakte Lambda-Ausdrücke zum Beispiel statt innerer anonymer Klassen genutzt werden können, und bieten uns mögliche Refactorings an. Lambda-Ausdrücke machen den Code kompakter und nach kurzer Eingewöhnung auch lesbarer.

Hinweis

Funktionale Schnittstellen müssen auf genau eine zu implementierende Methode hinauslaufen, auch wenn aus Oberschnittstellen mehrere Operationen vorgeschrieben werden, die sich aber durch den Einsatz von Generics auf eine Operation verdichten:

```
interface I<S,T extends CharSequence> {
  void len( S text );
  void len( T text );
}
interface FI extends I<String,String> { }
```

`FI` ist unsere funktionale Schnittstelle mit einer eindeutigen Operation `len(String)`.

Viele funktionale Schnittstellen in der Java-Standardbibliothek

Java bringt schon viele Schnittstellen mit, die als funktionale Schnittstellen gekennzeichnet sind. Darüber hinaus führt das Paket `java.util.function` mehr als 40 neue funktionale Schnittstellen ein. Eine kleine Auswahl:

- `interface Runnable { void run(); }`
- `interface Supplier<T> { T get(); }`
- `interface Consumer<T> { void accept(T t); }`
- `interface Comparator<T> { int compare(T o1, T o2); }`
- `interface ActionListener { void actionPerformed(ActionEvent e); }`

Ob die Schnittstelle noch andere Default-Methoden hat – also Schnittstellenmethoden mit vorgegebener Implementierung –, ist egal, wichtig ist nur, dass sie genau eine zu implementierende Operation deklariert.

1.2.2 Typ eines Lambda-Ausdrucks ergibt sich durch Zieltyp

In Java hat jeder Ausdruck einen Typ. `1` und `1*2` haben einen Typ (nämlich `int`), genauso wie `"A" + "B"` (Typ `String`) oder `String.CASE_INSENSITIVE_ORDER` (Typ `Comparator<String>`). Lambda-Ausdrücke haben auch immer einen Typ, denn ein Lambda-Ausdruck ist immer Exemplar einer funktionalen Schnittstelle. Damit steht auch der Typ fest. Allerdings ist es im Vergleich zu Ausdrücken wie `1*2` bei Lambda-Ausdrücken etwas anders gelagert, denn der Typ von Lambda-Ausdrücken ergibt sich ausschließlich aus dem Kontext. Erinnern wir uns an den Aufruf von `sort(…)`:

```
Arrays.sort( words, (String s1, String s2) -> { return … } );
```

Dort steht nichts vom Typ `Comparator`, sondern der Compiler erkennt aus dem Typ des zweiten Parameters von `sort(…)`, der ja `Comparator` ist, ob der Lambda-Ausdruck auf die Methode des `Comparator`s passt oder nicht.

Der Typ eines Lambda-Ausdrucks ist folglich abhängig davon, welche funktionale Schnittstelle er im jeweiligen Kontext gerade realisiert. Der Compiler kann ohne Kenntnis des *Zieltyps* (engl. *target type*) keinen Lambda-Ausdruck aufbauen.

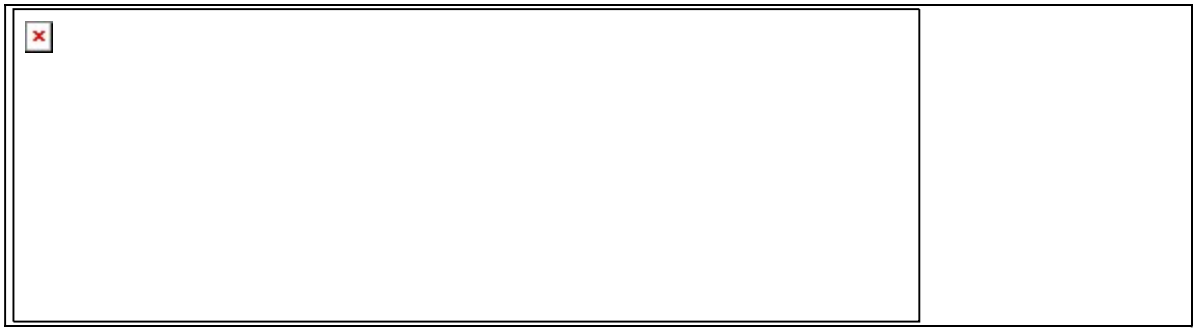

Abbildung 0.1: Typ-Inferenz vom Compiler

Beispiel

`Callable` und `Supplier` sind funktionale Schnittstellen mit Methoden, die keine Parameterlisten deklarieren und eine Referenz zurückgeben; der Code für den Lambda-Ausdruck sieht gleich aus:

```
java.util.concurrent.Callable<String> c = () -> { return "Rückgabe"; };
java.util.function.Supplier<String>   s = () -> { return "Rückgabe"; };
```

Wer bestimmt den Zieltyp?

Gerade weil an dem Lambda-Ausdruck der Typ nicht abzulesen ist, kann er nur dort verwendet werden, wo ausreichend Typinformationen vorhanden sind. Das sind unter anderem die folgenden Stellen:

- Variablendeklarationen: etwa wie bei `Supplier<String> s = () -> { return ""; }`
- Argumente an Methoden oder Konstruktoren: Der Parametertyp gibt alle Typinformationen. Ein Beispiel lieferte `Arrays.sort(…)`.
- Methodenrückgaben: Das könnte aussehen wie `Comparator<String> trimComparator() { return (s1, s2) -> { return … }; }`.
- Bedingungsoperator: Der `?:`-Operator liefert je nach Bedingung einen unterschiedlichen Lambda-Ausdruck. Beispiel: `Supplier<Double> randomNegOrPos = Math.random() > 0.5 ? () -> { return Math.random(); } : () -> { return Math.random(); };`

Parametertypen

In der Praxis ist der häufigste Fall, dass die Parametertypen von Methoden den Zieltyp vorgeben. Der Einsatz von Lambda-Ausdrücken ändert ein wenig die Sichtweise auf überladene Methoden. Unser Beispiel mit `() -> { return "Rückgabe"; }` macht das deutlich, denn es „passt“ auf den Zieltyp `Callable<String>` genauso wie auf `Supplier<String>`. Nehmen wir an, wir würden zwei überladene Methoden `run(…)` deklarieren:

- `<V> void run( Callable<V> callable ) { }`
- `<V> void run( Supplier<V> callable ) { }`

Spielen wir den Aufruf der Methoden einmal durch:

```
Callable<String> c = () -> { return "Rückgabe"; };
Supplier<String> s = () -> { return "Rückgabe"; };
run( c );
run( s );
// run( () -> { return "Rückgabe"; } ); // ☠ Compilerfehler
run( (Callable<String>) () -> { return "Rückgabe"; } );
```

Rufen wir `run(c)` bzw. `run(s)` auf, ist das kein Problem, denn `c` und `s` sind klar typisiert. Aber `run(…)` mit dem Lambda-Ausdruck aufzurufen funktioniert nicht, denn der Zieltyp (entweder `Callable` oder `Supplier`) ist mehrdeutig; der (Eclipse-)Compiler meldet: „The method run(Callable<Object>) is ambiguous for the type T“. Hier sorgt eine explizite Typumwandlung für Abhilfe.

Tipp zum API-Design

Aus Sicht eines API-Designers sind überladene Methoden natürlich schön, aus Sicht des Nutzers sind Typumwandlungen aber nicht schön. Um explizite Typumwandlungen zu vermeiden, sollte auf überladene Methoden verzichtet werden, wenn diese den Parametertyp einer funktionalen Schnittstelle aufweisen. Stattdessen lassen sich die Methoden unterschiedlich benennen (was bei Konstruktoren natürlich nicht funktioniert).

Wird in unserem Fall die Methode `runCallable(…)` und `runSupplier(…)` genannt, ist keine Typumwandlung mehr nötig, und der Compiler kann den Typ herleiten.

Rückgabetypen

Typ-Inferenz spielt bei Lambda-Ausdrücken eine große Rolle – das gilt insbesondere für die Rückgabetypen, die überhaupt nicht in der Deklaration auftauchen und für die es gar keine Syntax gibt; der Compiler „inferrt“ sie. In unserem Beispiel

```
Comparator<String> c =
  (String s1, String s2) -> { return s1.trim().compareTo( s2.trim() ); };
```

ist `String` als Parametertyp der `Comparator`-Methode ausdrücklich gegeben; der Rückgabetyp `int`, den der Ausdruck `s1.trim().compareTo( s2.trim())` liefert, taucht dagegen nicht auf.

Mitunter muss dem Compiler etwas geholfen werden: Nehmen wir die funktionale Schnittstelle `Supplier<T>`, die eine Methode `T get()` deklariert, für ein Beispiel. Die Zuweisung

```
Supplier<Long> two  = () -> { return 2; }       // ☠ Compilerfehler
```

ist nicht korrekt und führt zum Compilerfehler „incompatible types: bad return type in lambda expression“. `2` ist ein Literal vom Typ `int`, und der Compiler kann es nicht an `Long` anpassen. Wir müssen schreiben

```
Supplier<Long> two  = () -> { return 2L };
```

oder

```
Supplier<Long> two  = () -> { return (long) 2 };
```

Bei Lambda-Ausdrücken gelten keine wirklich neuen Regeln im Vergleich zu Methodenrückgaben, denn auch eine Methodendeklaration wie

```
Long two() { return 2; }       // ☠ Compilerfehler
```

wird vom Compiler bemängelt. Doch weil Wrapper-Typen durch die Generics bei funktionalen Schnittstellen viel häufiger sind, treten diese Besonderheiten öfter auf als bei Methodendeklarationen.

Sind Lambda-Ausdrücke Objekte?

Ein Lambda-Ausdruck ist ein Exemplar einer funktionalen Schnittstelle und tritt als Objekt auf. Bei Objekten besteht normalerweise zu `java.lang.Object` immer eine natürliche Ist-eine-Art-von-Beziehung. Fehlt aber der Kontext, ist selbst die Ist-eine-Art-von-Beziehung zu `java.lang.Object` gestört und Folgendes nicht korrekt:

```
Object o = () -> {};           // ☠ Compilerfehler
```

Der Compilerfehler ist: „incompatible types: the target type must be a functional interface“. Nur eine explizite Typumwandlung kann den Fehler korrigieren und dem Compiler den Zieltyp vorgeben:

```
Object r = (Runnable) () -> {};
```

Lambda-Ausdrücke haben keinen eigenen Typ an sich, und für das Typsystem von Java ändert sich im Prinzip nichts. Möglicherweise ändert sich das in späteren Java-Versionen.

Hinweis

Dass Lambda-Ausdrücke Objekte sind, ist eine Eigenschaft, die nicht überstrapaziert werden sollte. So sind die üblichen `Object`-Methoden `equals(Object)`, `hashCode()`, `getClass()`, `toString()` und die zur Thread-Kontrolle ohne besondere Bedeutung. Es sollte auch nie ein Szenario geben, in dem Lambda-Ausdrücke mit `==` verglichen werden müssen, denn das Ergebnis ist laut Spezifikation undefiniert. Echte Objekte haben eine Identität, einen Identity-Hashcode, lassen sich vergleichen und mit `instanceof` testen, können mit einem synchronisierten Block abgesichert werden; all dies gilt für Lambda-Ausdrücke nicht. Im Grunde charakterisiert der Begriff „Lambda-

Ausdruck“ schon sehr gut, was wir nie vergessen sollten: Es handelt sich um einen Ausdruck, also etwas, was ausgewertet wird und ein Ergebnis produziert.

1.2.3 Annotation @FunctionalInterface

Jede Schnittstelle mit genau einer abstrakten Methode eignet sich als funktionale Schnittstelle und damit für einen Lambda-Ausdruck. Jedoch soll nicht jede Schnittstelle in der API, die im Moment nur eine abstrakte Methode deklariert, auch für Lambda-Ausdrücke verwendet werden. So kann zum Beispiel eine Weiterentwicklung der Schnittstelle mit mehreren (abstrakten) Methoden geplant sein, aber zurzeit ist nur eine abstrakte Methode vorhanden. Der Compiler kann nicht wissen, ob sich eine Schnittstelle vielleicht weiterentwickelt. Um kenntlich zu machen, dass ein `interface` als funktionale Schnittstelle gedacht ist, existiert der Annotationstyp `FunctionalInterface` im `java.lang`-Paket. Diese Annotation markiert, dass es bei genau einer abstrakten Methode und damit bei einer funktionalen Schnittstelle bleiben soll.

Beispiel

Eine eigene funktionale Schnittstelle sollte immer als `FunctionalInterface` markiert werden:

```
@FunctionalInterface
public interface MyFunctionalInterface {
  void foo();
}
```

Der Compiler prüft, ob die Schnittstelle mit einer solchen Annotation tatsächlich nur exakt eine abstrakte Methode enthält, und löst einen Fehler aus, wenn dem nicht so ist. Aus Kompatibilitätsgründen erzwingt der Compiler diese Annotation bei funktionalen Schnittstellen allerdings nicht; das ermöglicht es, dass innere Klassen, die herkömmliche Schnittstellen mit einer Methode implementieren, einfach in Lambda-Ausdrücke umgeschrieben werden können. Die Annotation ist also keine Voraussetzung für die Nutzung der Schnittstelle in einem Lambda-Ausdruck und dient bisher nur der Dokumentation. In der Java SE sind aber alle zentralen funktionalen Schnittstellen so ausgezeichnet.

Tipp

Was mit `@FunctionalInterface` ausgezeichnet ist, bekommt in der Javadoc einen Extrasatz: „Functional Interface: This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.“ Das macht funktionale Schnittstellen noch besser sichtbar.

1.2.4 Syntax für Lambda-Ausdrücke

Lambda-Ausdrücke haben wie Methoden mögliche Parameter- und Rückgabewerte. Die Java-Grammatik für die Schreibweise von Lambda-Ausdrücken sieht ein paar nützliche syntaktische Abkürzungen vor.

Ausführliche Schreibweise

Lambda-Ausdrücke lassen sich auf unterschiedliche Art und Weise schreiben, da es für diverse Konstruktionen Abkürzungen gibt. Eine Form, die jedoch immer gilt, ist:

```
( LambdaParameter ) -> { Anweisungen }
```

Der Lambda-Parameter besteht (voll ausgeschrieben) wie ein Methodenparameter aus a) dem Typ, b) dem Namen und c) optionalen Modifizierern.

Der Parametername öffnet einen neuen Gültigkeitsbereich für eine Variable, wobei der Parametername *keine* anderen Namen von lokalen Variablen überlagern darf. Hier verhält sich die Lambda-Parametervariable wie eine neue Variable aus einem inneren Block und nicht wie eine Variable aus einer inneren Klasse, wo die Sichtbarkeit anders ist.

Beispiel

Folgendes ergibt einen Compilerfehler im Lambda-Ausdruck, weil `s` schon deklariert ist; die Parametervariable vom Lambda-Ausdruck muss „frisch“ sein:

```
String s = "Make Donald Drumpf Again";
Comparator<String> c = (String s, String t) -> { … }; // ☠ Compilerfehler
```

Abkürzung 1: Typ-Inferenz

Der Java-Compiler kann viele Typen aus dem Kontext ablesen, was Typ-Inferenz genannt wird. Wir kennen so etwas vom Diamant-Operator, wenn wir etwa schreiben:

```
List<String> list = new ArrayList<>()
```

Sind für den Compiler genug Typinformationen verfügbar, dann erlaubt der Compiler bei Lambda-Ausdrücken eine Abkürzung. Bei

```
Comparator<String> c =
  (String s1, String s2) -> { return s1.trim().compareTo( s2.trim() ); };
```

ist Typ-Inferenz einfach (`Comparator<String>` sagt alles aus), daher funktioniert die folgende Abkürzung:

```
Comparator<String> c = (s1, s2) -> { return s1.trim().compareTo( s2.trim() ); };
```

Die Parameterliste enthält entweder explizit deklarierte Parametertypen oder implizite Inferred-Typen. Eine Mischung ist nicht erlaubt, der Compiler blockt so etwas wie `(String s1, s2)` oder `(s1, String s2)` mit einem Fehler ab.

Wenn der Compiler die Typen ablesen kann, sind die Parametertypen optional. Aber Typ-Inferenz ist nicht immer möglich, weshalb die Abkürzung nicht immer möglich ist. Außerdem hilft die explizite Schreibweise auch der Lesbarkeit: Kurze Ausdrücke sind nicht unbedingt die verständlichsten.

Hinweis

Der Compiler liest aus den Typen ab, ob alle Eigenschaften vorhanden sind. Die Typen sind dabei entweder explizit oder implizit gegeben.

```
Comparator<String> sc = (a, b) -
> { return Integer.compare( a.length(), b.length() ); };
Comparator<BitSet> bc = (a, b) -
> { return Integer.compare( a.length(), b.length() ); };
```

Die Klassen `String` und `BitSet` besitzen beide die Methode `length()`, daher ist der Lambda-Ausdruck korrekt. Der gleiche Lambda-Code lässt sich für zwei völlig verschiedene Klassen einsetzen, die überhaupt keine Gemeinsamkeiten haben, nur dass sie zufällig beide eine Methode namens `length()` besitzen.

Abkürzung 2: Lambda-Rumpf ist entweder einzelner Ausdruck oder Block

Besteht der Rumpf eines Lambda-Ausdrucks nur aus einem einzelnen Ausdruck, kann eine verkürzte Schreibweise die Blockklammern und das Semikolon einsparen. Statt

```
( LambdaParameter ) -> { return Ausdruck; }
```

heißt es dann

```
( LambdaParameter ) -> Ausdruck
```

Lambda-Ausdrücke mit einer `return`-Anweisung im Rumpf kommen häufig vor, da dies den typischen Funktionen entspricht. Somit ist es eine willkommene Verkürzung, wenn die abgekürzte Syntax für Lambda-Ausdrücke lediglich den Ausdruck fordert, der dann die Rückgabe bildet. Tabelle 0.2 zeigt drei Beispiel.

Lange Schreibweise	Abkürzung

`(s1, s2) ->`
`{ return s1.trim().compareTo(`
`s2.trim() ); }`
	`(s1, s2) ->`
	`s1.trim().compareTo(`
	`s2.trim() )`

`(a, b) -> { return a + b; }`	`(a, b) -> a + b`

`() -> { System.out.println(); }`	`() -> System.out.println()`

Tabelle 0.2: Ausführliche und abgekürzte Schreibweise

Ausdrücke können in Java auch zu `void` ausgewertet werden, sodass ohne Probleme ein Aufruf wie `System.out.println()` in der kompakten Schreibweise ohne Block gesetzt werden kann. Das heißt, wenn Lambda-Ausdrücke mit der kurzen Ausdruckssyntax eingesetzt werden, können diese Ausdrücke etwas zurückgeben, müssen aber nicht.

Hinweis

Die Schreibweise mit den geschweiften Klammern und den Rückgabe-Ausdrücken kann nicht gemischt werden. Entweder gibt es einen Block geschweifter Klammern und `return` oder keine Klammern und kein `return`-Schlüsselwort. Falsche Mischungen ergeben Fehler:

```
Comparator<String> c;
c = (s1, s2) -> { s1.trim().compareTo( s2.trim() ) };     // ☠ Compilerfehler 1
c = (s1, s2) -> return s1.trim().compareTo( s2.trim() ); // ☠ Compilerfehler 2
```

Würden wir in 1 ein explizites `return` nutzen, wäre alles in Ordnung, würde bei 2 das `return` wegfallen, wäre die Zeile auch compilierbar.

Ob Lambda-Ausdrücke eine Rückgabe haben, drücken zwei Begriffe aus:

- `void`-kompatibel: Der Lambda-Rumpf gibt kein Ergebnis zurück, entweder weil der Block kein `return` enthält oder ein `return` ohne Rückgabe oder weil ein `void`-Ausdruck in der verkürzten Schreibweise eingesetzt wird. Der Lambda-Ausdruck `() -> System.out.println()` ist also `void`-kompatibel, genauso wie `() -> {}`.
- Wertkompatibel: Der Rumpf beendet den Lambda-Ausdruck mit einer `return`-Anweisung, die einen Wert zurückgibt, oder besteht aus der kompakten Schreibweise mit einer Rückgabe ungleich `void`.

Eine Mischung aus `void`- und wertkompatibel ist nicht erlaubt und führt wie bei Methoden zu einem Compilerfehler.[1]

Abkürzung 3: Einzelner Identifizierer statt Parameterliste und Klammern

Besteht die Parameterliste

1. nur aus einem einzelnen Identifizierer
2. und ist der Typ durch Typ-Inferenz klar,

[1] Wohl aber gibt es wie bei `{ throw new RuntimeException(); }` Ausnahmen, bei denen Lambda-Ausdrücke beides sind.

können die runden Klammern wegfallen.

Lange Schreibweise	Typen inferred	Vollständig abgekürzt
`(String s) -> s.length()`	`(s) -> s.length()`	`s -> s.length()`
`(int i) -> Math.abs( i )`	`(i) -> Math.abs( i )`	`i -> Math.abs( i )`

Tabelle 0.3: Unterschiedlicher Grad von Abkürzungen

Kommen alle Abkürzungen zusammen, lässt sich etwa die Hälfte an Code einsparen. Aus `(int i) -> { return Math.abs( i ); }` wird einfach `i -> Math.abs( i )`.

Syntax-Hinweis

Nur bei genau einem Lambda-Parameter können die Klammern weggelassen werden, da es sonst Mehrdeutigkeiten gibt, für die es wieder komplexe Regeln zur Auflösung geben müsste. Heißt es etwa `foo( k, v -> { … } )`, ist unklar, ob `foo` zwei Parameter deklariert. Ist das zweite Argument ein Lambda-Ausdruck, oder handelt es sich um nur genau einen Parameter, wobei dann ein Lambda-Ausdruck übergeben wird, der selbst zwei Parameter deklariert? Um Problemen wie diesen aus dem Weg zu gehen, können Entwickler auf den ersten Blick sehen, dass `foo( k, v -> { … } )` eindeutig für zwei Methodenparameter steht und `foo( (k, v) -> { … } )` nur einen Parameter besitzt.

Unbenutzte Parameter in Lambda-Ausdrücken

Es kommt vor, dass ein Lambda-Ausdruck eine funktionale Schnittstelle implementiert, aber nicht jeder Parameter von Interesse ist. Als Beispiel schauen wir uns an:

```
interface Consumer<A> { void apply( A a ); }
```

Ein Konsument, der das Argument in Hochkommata ausgibt, sieht so aus:

```
Consumer<String> printQuoted = s -> System.out.printf( "'%s'", s );
printQuoted.accept( "Chris" );  // 'Chris'
```

Was ist nun, wenn ein Konsument auf das Argument gar nicht zugreifen möchte, weil zum Beispiel die aktuelle Zeit ausgegeben wird?

```
Consumer<String> printNow =
  s -> System.out.println( System.currentTimeMillis() );
```

Die Variable `s` in der Lambda-Parameterliste ist ungenutzt und wird vom Compiler auch als „unused“ bemängelt. Daher erlaubt eine spezielle Schreibweise, den Variablennamen wegzulassen und nur den Typ anzugeben:

```
Consumer<String> printNow =
  String /* bzw. (String)*/ -> System.out.println( System.currentTimeMillis() );
```

Der Typ selbst darf nicht entfallen, denn `() -> …` passt nicht auf die funktionale Schnittstelle `Consumer`, die immer einen Parameter deklariert.

Hinweis

Nur bei einem Lambda-Parameter ist diese Form der Abkürzung erlaubt. Folgendes führt zu einem Compilerfehler:

```
BiFunction<Double,Double,Double> bifunc =
(Double a, Double /* ☠ */) -> Math.signum( a );
```

1.2.5 Die Umgebung der Lambda-Ausdrücke und Variablenzugriffe

Ein Lambda-Ausdruck „sieht“ seine Umgebung genauso wie der Code, der vor oder nach dem Lambda-Ausdruck steht. Insbesondere hat ein Lambda-Ausdruck vollständigen Zugriff auf alle Eigenschaften der Klasse, genauso wie auch der einschließende äußere Block sie hat. Es gibt keinen besonderen Namensraum, sondern nur neue und vielleicht überdeckte Variablen durch die Parameter. Das ist einer der grundlegenden Unterschiede zwischen Lambda-Ausdrücken und inneren Klassen. Somit ist auch die Bedeutung von `this` und `super` bei Lambda-Ausdrücken und inneren Klassen unterschiedlich.

Zugriff auf finale, lokale Variablen/Parametervariablen

Lambda-Ausdrücke können problemlos auf Objektvariablen und Klassenvariablen lesend und schreibend zugreifen. Auch auf lokale Variablen und Parameter hat ein Lambda-Ausdruck Zugriff. Doch greift ein Lambda-Ausdruck auf lokale Variablen oder Parametervariablen zu, müssen diese final sein. Liegt ein Lambda-Ausdruck zum Beispiel in einer Schleife, kann der Lambda-Ausdruck nicht auf den Schleifenzähler zugreifen, da sich dieser bei jeder Iteration ändert. (Anders sieht das bei der Variablen in der erweiterten `for`-Schleife aus, darauf kann ein Lambda-Ausdruck zugreifen.)

Dass eine Variable final ist, muss nicht extra mit einem Modifizierer geschrieben werden, aber sie muss *effektiv final* (engl. *effectively final*) sein. Effektiv final ist eine Variable, wenn sie nach der Initialisierung nicht mehr beschrieben wird.

Ein Beispiel: Der Benutzer soll über eine Eingabe die Möglichkeit bekommen, zu bestimmen, ob String-Vergleiche mit unserem trimmenden `Comparator` unabhängig von der Groß-/Kleinschreibung stattfinden sollen:

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

```
    /*final*/ boolean compareIgnoreCase = new Scanner( System.in ).nextBoolean();
    Comparator<String> c = (s1, s2) -> compareIgnoreCase ?
                                       s1.trim().compareToIgnoreCase( s2.trim() ) :
                                       s1.trim().compareTo( s2.trim() );
    String[] words = { "M", "\nSkyfall", " Q", "\t\tAdele\t" };
    Arrays.sort( words, c );
    System.out.println( Arrays.toString( words ) );
  }
}
```

Ob `compareIgnoreCase` von uns `final` gesetzt wird oder nicht, ist egal, denn die Variable wird hier effektiv final verwendet. Natürlich kann es nicht schaden, `final` als Modifizierer immer davorzusetzen, um dem Leser des Codes diese Tatsache bewusst zu machen.

Neu eingeschobene Lambda-Ausdrücke, die auf lokale Variablen oder Parametervariablen zugreifen, können also im Nachhinein zu Compilerfehlern führen. Folgendes Segment ist ohne Lambda-Ausdruck korrekt:

```
boolean compareIgnoreCase = new Scanner( System.in ).nextBoolean();  // 1
…                                                                    // 2
compareIgnoreCase = true;                                            // 3
```

Schiebt sich zwischen Zeile 1 und 3 nachträglich ein Lambda-Ausdruck, der auf `compareIgnoreCase` zugreift, gibt es anschließend einen Compilerfehler. Allerdings liegt der Fehler nicht in Zeile 3, sondern beim Lambda-Ausdruck. Denn die Variable `compareIgnoreCase` ist nach der Änderung nicht mehr effektiv final, was sie aber sein müsste, um in dem Lambda-Ausdruck verwendet zu werden.

Tipp

Lambda-Ausdrücke verhalten sich genauso wie innere anonyme Klassen, die auch nur auf finale Variablen zugreifen können. Mit Behältern wie einem Array oder den speziellen `AtomicXXX`-Klassen aus dem `java.util.concurrent.atomic`-Paket lässt sich das Problem im Prinzip lösen. Denn greift ein Lambda-Ausdruck etwa auf das Array `boolean[] compareIgnoreCase = new boolean[1];` zu, so ist die Variable `compareIgnoreCase` selbst final, aber `compareIgnoreCase[0] = true;` ist erlaubt, denn es ist ein Schreibzugriff auf das Array, nicht auf die Variable `compareIgnoreCase`. Je nach Code besteht jedoch die Gefahr, dass Lambda-Ausdrücke parallel ausgeführt werden. Wird etwa ein Lambda-Ausdruck mit Veränderung auf diesem Array-Inhalt parallel ausgeführt, so ist der Zugriff nicht synchronisiert, und das Ergebnis kann „kaputt“ sein, denn paralleler Zugriff auf Variablen muss immer koordiniert vorgenommen werden.

Namensräume

Deklariert eine innere anonyme Klasse Variablen innerhalb der Methode, so sind diese immer „neu“, das heißt, die neuen Variablen überlagern vorhandene lokale Variablen aus dem äußeren Kontext. Die Variable `compareIgnoreCase` kann im Rumpf von `compare(…)` zum Beispiel problemlos neu deklariert werden:

```
boolean compareIgnoreCase = true;
Comparator<String> c = new Comparator<String>() {
  @Override public int compare( String s1, String s2 ) {
   boolean compareIgnoreCase = false;        // völlig ok
   return …
  }
};
```

In einem Lambda-Ausdruck ist das nicht möglich, und Folgendes führt zu einer Fehlermeldung des Compilers: „variable compareIgnoreCase ist already defined“.

```
boolean compareIgnoreCase = true;
Comparator<String> c = (s1, s2) -> {
  boolean compareIgnoreCase = false;  // ☠ Compilerfehler
  return …
}
```

this-Referenz

Ein Lambda-Ausdruck unterscheidet sich von einer inneren (anonymen) Klasse auch in dem, worauf die `this`-Referenz verweist:

- Beim Lambda-Ausdruck zeigt `this` immer auf das Objekt, in dem der Lambda-Ausdruck eingebettet ist.
- Bei einer inneren Klasse referenziert `this` die innere Klasse, und die ist ein komplett neuer Typ.

Folgendes Beispiel macht das deutlich:

Listing 0.2: InnerVsLambdaThis.java

```
class InnerVsLambdaThis {
  InnerVsLambdaThis() {
    Runnable lambdaRun = () -> System.out.println( this.getClass().getName() );
    Runnable innerRun  = new Runnable() {
      @Override public void run() { System.out.println( this.getClass().getName()); }
    };
```

```
    lambdaRun.run();       // InnerVsLambdaThis
    innerRun.run();        // InnerVsLambdaThis$1
  }
  public static void main( String[] args ) {
    new InnerVsLambdaThis();
  }
}
```

Als Erstes nutzen wir `this` in einen Lambda-Ausdruck im Konstruktor der Klasse `InnerVsLambdaThis`. Damit referenziert `this` das neu gebaute `InnerVsLambdaThis`-Objekt. Bei der inneren Klasse referenziert `this` ein anderes Exemplar, und zwar vom Typ `Runnable`. Da es bei anonymen Klassen keinen Namen hat, trägt es lediglich die Kennung `InnerVsLambdaThis$1`.

Rekursive Lambda-Ausdrücke

Lambda-Ausdrücke können auf sich selbst verweisen. Da aber ein `this` zur Selbstreferenz nicht funktioniert, ist ein kleiner Umweg nötig. Erst muss eine Objekt- oder eine Klassenvariable deklariert werden, dann muss dieser Variablen ein Lambda-Ausdruck zugewiesen werden, und dann kann der Lambda-Ausdruck auf diese Variable zugreifen und einen rekursiven Aufruf starten. Für den Klassiker der Fakultät sieht das so aus:

Listing 0.3: RecursiveFactLambda.java

```
import java.util.function.IntFunction;

public class RecursiveFactLambda {

  public static IntFunction<Integer> fact =
      n -> (n == 0) ? 1 : n * RecursiveFactLambda.fact.apply( n - 1 );

  public static void main( String[] args ) {
    System.out.println( fact.apply( 5 ) );   // 120
  }
}
```

`IntFunction` ist eine funktionale Schnittstelle aus dem Paket `java.util.function` mit einer Operation `T apply(int i)`. `T` ist ein generischer Rückgabetyp, den wir hier mit `Integer` belegt haben.

`fact` hätte genauso gut als normale Methode deklariert werden können. Großartige Vorteile bietet die Schreibweise mit Lambda-Ausdrücken hier nicht. Zumal jetzt auch der Begriff *anonyme*

Methode nicht mehr so richtig passt, da der Lambda-Ausdruck ja doch einen Namen hat, nämlich `fact`.

1.2.6 Ausnahmen in Lambda-Ausdrücken

Lambda-Ausdrücke sind Implementierungen von funktionalen Schnittstellen, und bisher haben wir noch nicht die Frage betrachtet, was passiert, wenn der Codeblock vom Lambda-Ausdruck eine Ausnahme auslöst, und wer diese auffangen muss.

Ausnahmen im Codeblock eines Lambda-Ausdrucks

In `java.util.function` gibt es eine funktionale Schnittstelle `Predicate`, deren Deklaration im Kern wie folgt ist:

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

Ein `Predicate` führt einen Test durch und liefert wahr oder falsch als Ergebnis. Ein Lambda-Ausdruck kann diese Schnittstelle implementieren. Nehmen wir an, wir wollten testen, ob eine Datei die Länge 0 hat, um etwa Dateileichen zu finden. In einer ersten Idee greifen wir auf die existierende `Files`-Klasse zurück, die `size(…)` anbietet:

```
Predicate<Path> isEmptyFile = path -> Files.size( path ) == 0;  // ☠ Compilerfehler
```

Das Problem dabei ist, dass `Files.size(…)` eine `IOException` auslöst, die behandelt werden muss, und zwar *nicht* vom Block, in dem der Lambda-Ausdruck als Ganzes steht, sondern vom Code im Lambda-Ausdruck selbst. Das schreibt der Compiler so vor. Folgendes ist keine Lösung:

```
try {
  Predicate<Path> isEmptyFile = path -> Files.size( path ) == 0; // ☠
} catch ( IOException e ) { … }
```

sondern nur:

```
Predicate<Path> isEmptyFile = path -> {
  try {
    return Files.size( path ) == 0;
  } catch ( IOException e ) { return false; }
};
```

Die Eigenschaft, die Java fehlt, nennt sich *Exception-Transparenz*, und hier ist deutlich der Unterschied zwischen geprüften und ungeprüften Ausnahmen zu sehen. Bei der Exception-Transparenz wäre keine Ausnahmebehandlung im Lambda-Ausdruck nötig und an einer übergeordneten Stelle möglich. Doch da diese Möglichkeit in Java fehlt, bleibt uns nur übrig, geprüfte Ausnahmen in Lambda-Ausdrücken direkt zu behandeln.

Funktionale Schnittstellen mit throws-Klausel

Ungeprüfte Ausnahmen können immer auftreten und führen (nicht abgefangen) wie üblich zum Abbruch des Threads. Eine `throws`-Klausel an den Methoden/Konstruktoren ist dafür nicht nötig. Doch können funktionale Schnittstellen eine `throws`-Klausel mit geprüften Ausnahmen deklarieren, und die Implementierung einer funktionalen Schnittstelle kann logischerweise geprüfte Ausnahmen auslösen.

Eine Deklaration wie `Callable` aus dem Paket `java.util.concurrent` macht das deutlich (`Callable` trägt kein `@FunctionalInterface`):

```
public interface Callable<V> {
  V call() throws Exception;
}
```

Das könnte durch folgenden Lambda-Ausdruck realisiert werden:

```
Callable<Integer> randomDice = () -> (int)(Math.random() * 6) + 1;
```

Der Aufruf von `call()` auf einem `randomDice` muss mit einer Ausnahmebehandlung einhergehen, da `call()` eine `Exception` auslöst, etwa so:

```
try {
  System.out.println( randomDice.call() );
  System.out.println( randomDice.call() );
}
catch ( Exception e ) { … }
```

Dass der Aufrufer die Ausnahme behandeln muss, ist klar. Die Deklaration des Lambda-Ausdrucks enthält keinen Hinweis auf die Ausnahme, das ist ein Unterschied zum vorangegangenen Abschnitt.

Designtipp

Ausnahmen in Methoden funktionaler Schnittstellen schränken den Nutzen stark ein, und daher löst keine der funktionalen Schnittstellen aus etwa `java.util.function` eine geprüfte Ausnahme aus. Der Grund ist einfach, denn jeder Methodenaufrufer müsste sonst entweder die Ausnahme weiterleiten oder behandeln.[2]

[2] Von `Callable` gibt es zwar Nutzer, die mit Nebenläufigkeit (daher das Paket `java.util.concurrent`) in Zusammenhang stehen, aber keine weiteren Verwendungen in der Java-Bibliothek, von zwei Beispielen aus `javax.tools` abgesehen. Mit `java.util.function.Supplier` existiert eine entsprechende Alternative ohne `throws`-Klausel.

Um die Einschränkungen und Probleme mit einer `throws`-Klausel noch etwas deutlicher zu machen, stellen wir uns vor, dass die funktionale Schnittstelle `Predicate` an der Operation ein `throws Exception` (vom Sinn des Typs `Exception` an sich einmal abgesehen) enthält:

```
interface Predicate<T> { boolean test( T t ) throws Exception; } // Was wäre wenn?
```

Die Konsequenz wäre, dass jeder Aufrufer von `test(…)` nun seinerseits die `Exception` in die Hände bekäme und sie auffangen oder weiterleiten müsste. Leitet der `test(…)`-Aufrufer mit `throws Exception` die Ausnahme weiter nach oben, bekommen wir plötzlich an allen Stellen ein `throws Exception` in die Methodensignatur, was auf keinen Fall gewünscht ist. So enthält jetzt etwa `ArrayList` eine Deklaration von `removeIf(Predicate filter)`; hier müsste sich dann `removeIf(…)` – das letztendlich `filter.test(…)` aufruft – mit der Testausnahme herumärgern, und `removeIf(Predicate filter) throws Exception` ist keine gute Sache.

Von geprüft nach ungeprüft

Geprüfte Ausnahmen sind in Lambda-Ausdrücken nicht schön. Eine Lösung ist, Code, der geprüfte Ausnahmen auslöst, zu verpacken und die geprüfte Ausnahme in einer ungeprüften zu manteln. Das kann etwa so aussehen:

Listing 0.4: PredicateWithException.java

```
public class PredicateWithException {
  @FunctionalInterface
  public interface ExceptionalPredicate<T,E extends Exception> {
    boolean test( T t ) throws E;
  }
  public static <T> Predicate<T> asUncheckedPredicate(
                                  ExceptionalPredicate<T,Exception> predicate ) {
    return t -> {
      try {
        return predicate.test( t );
      }
      catch ( Exception e ) {
        throw new RuntimeException( e.getMessage(), e );
      }
    };
  }
  public static void main( String[] args )  {
    Predicate<Path> isEmptyFile =
      asUncheckedPredicate( path -> Files.size( path ) == 0 );
```

```
    System.out.println( isEmptyFile.test( Paths.get( "c:/" ) ) );
  }
}
```

Die Schnittstelle `ExceptionalPredicate` ist ein Prädikat mit optionaler Ausnahme. In der eigenen Hilfsmethode `asUncheckedPredicate(ExceptionalPredicate)` nehmen wir so ein `ExceptionalPredicate` an und packen es in ein `Predicate`, das die Methode zurückgibt. Geprüfte Ausnahmen werden in eine ungeprüfte Ausnahme vom Typ `RuntimeException` gesetzt. Somit muss `Predicate` keine geprüfte Ausnahme weiterleiten, was es ja laut Deklaration auch nicht kann.

Die Java-Bibliothek selbst bringt keine Ummantelungen dieser Art mit. Es gibt nur eine interne Methode, die etwas Vergleichbares tut:

Listing 0.5: java.nio.file.Files.java, asUncheckedRunnable(...)

```
/**
 * Convert a Closeable to a Runnable by converting checked IOException
 * to UncheckedIOException
 */
private static Runnable asUncheckedRunnable( Closeable c ) {
  return () -> {
    try {
      c.close();
    }
    catch ( IOException e ) {
      throw new UncheckedIOException( e );
    }
  };
}
```

Hier kommt die Klasse `UncheckedIOException` zum Einsatz. Diese ist eine ungeprüfte Ausnahme, die als Wrapper-Klasse für Ein-/Ausgabefehler genutzt wird. Wir finden die `UncheckedIOException` etwa bei `lines()` von `BufferedReader` bzw. `Files`, die einen `Stream<String>` mit Zeilen liefert – geprüfte Ausnahmen sind hier nur im Weg.

1.2.7 Klassen mit einer abstrakten Methode als funktionale Schnittstelle? *

Als die Entwickler der Sprache Java die Lambda-Ausdrücke diskutierten, stand auch die Frage im Raum, ob abstrakte Klassen, die nur über eine abstrakte Methode verfügen, ebenfalls für Lambda-Ausdrücke genutzt werden können.[3] Sie entschieden sich dagegen, unter anderem deswegen, weil

[3] Früher wurde hier die Abkürzung SAM (Single Abstract Method) genutzt.

bei der Implementierung von Schnittstellen die JVM weitreichende Optimierungen vornehmen kann. Und bei Klassen wird das schwierig. Das liegt auch daran, dass ein Konstruktor umfangreiche Initialisierungen mit Seiteneffekten vornimmt (die Konstruktoren aller Oberklassen nicht zu vergessen) sowie Ausnahmen auslösen könnte. Gewünscht ist aber nur die Ausführung einer Implementierung der funktionalen Schnittstelle und kein anderer Code.

Es gibt nun im JDK einige abstrakte Klassen, die genau eine abstrakte Methode vorschreiben, etwa `java.util.TimerTask`. Solche Klassen können nicht über einen Lambda-Ausdruck realisiert werden; hier müssen Entwickler weiterhin zu Klassenimplementierungen greifen, und die kürzeste Lösung ist eine innere anonyme Klasse. Eigene Hilfsklassen können natürlich den Code etwas abkürzen, aber eben nur mithilfe einer eigenen Implementierung.

Wer abstrakte Methoden mit Lambda-Ausdrücken implementieren möchte, kann mit Hilfsklassen arbeiten. Denn wenn eine Hilfsklasse funktionale Schnittstellen einsetzt, so können Lambda-Ausdrücke wieder ins Spiel kommen, indem die Implementierung der abstrakten Methode an den Lambda-Ausdruck weiterleitet. Nehmen wir das Beispiel für `TimerTask` und gehen zwei unterschiedliche Strategien der Implementierung durch. Mit Delegation sieht das so aus:

Listing 0.6: TimerTaskLambda.java

```
import java.util.*;
class TimerTaskLambda {
  public static TimerTask createTimerTask( Runnable runnable ) {
    return new TimerTask() {
        @Override public void run() { runnable.run(); }
    };
  }
  public static void main( String[] args ) {
    new Timer().schedule( createTimerTask( () -> System.out.println("Hi") ), 500 );
  }
}
```

Mit Vererbung erhalten wir:

```
public class LambdaTimerTask extends TimerTask {
  private final Runnable runnable;

  public LambdaTimerTask( Runnable runnable ) {
    this.runnable = runnable;
  }
```

```
  @Override public void run() { runnable.run();  }
}
```

Der Aufruf erfolgt dann statt über `createTimerTask(…)` mit dem Konstruktor:

```
new Timer().schedule( new LambdaTimerTask( () -> System.out.println("Hi") ), 500 );
```

1.3 Methodenreferenz

Je größer Softwaresysteme werden, desto wichtiger werden Dinge wie Klarheit, Wiederverwendbarkeit und Dokumentation. Wir haben für unseren String-Comparator eine Implementierung geschrieben, anfangs über eine innere Klasse, später über einen Lambda-Ausdruck. In jedem Fall haben wir Code geschrieben. Doch was wäre, wenn eine Utility-Klasse schon eine Implementierung mitbringen würde? Dann könnte der Lambda-Ausdruck natürlich an die vorhandene Implementierung delegieren, und wir sparen Code. Schauen wir uns das mal an einem Beispiel an:

```
class StringUtils {
  public static int compareTrimmed( String s1, String s2 ) {
    return s1.trim().compareTo( s2.trim() );
  }
}
public class CompareIgnoreCase {
  public static void main( String[] args ) {
    String[] words = { "A", "B", "a" };
    Arrays.sort( words,
                 (String s1, String s2) -> StringUtils.compareTrimmed(s1, s2) );
    System.out.println( Arrays.toString( words ) );
  }
}
```

Auffällig ist hier, dass die referenzierte Methode `compareTrimmed(String,String)` von den Parametertypen und vom Rückgabetyp genau auf die `compare(…)`-Methode eines `Comparator` passt. Für genau solche Fälle gibt es eine weitere syntaktische Verkürzung, sodass im Code kein Lambda-Ausdruck, sondern nur noch ein Methodenverweis notwendig ist.

Definition

Eine *Methodenreferenz* ist ein Verweis auf eine Methode, ohne diese jedoch aufzurufen. Syntaktisch trennen zwei Doppelpunkte den Klassennamen oder die Referenz auf der linken Seite von dem Methodennamen auf der rechten.

Die Zeile

```
Arrays.sort( words, (String s1, String s2) -> StringUtils.compareTrimmed(s1, s2) );
```

lässt sich mit einer Methodenreferenz abkürzen zu:

```
Arrays.sort( words, StringUtils::compareTrimmed );
```

Die Sortiermethode erwartet vom `Comparator` eine Methode, die zwei Strings annimmt und eine Ganzzahl zurückgibt. Der Name der Klasse und der Name der Methode sind unerheblich, weshalb an dieser Stelle eine Methodenreferenz eingesetzt werden kann.

Eine Methodenreferenz ist wie ein Lambda-Ausdruck ein Exemplar einer funktionalen Schnittstelle, jedoch für eine existierende Methode einer bekannten Klasse. Wie üblich bestimmt der Kontext, von welchem Typ genau der Ausdruck ist.

Hinweis

Gleicher Code für eine Methodenreferenz kann zu komplett unterschiedlichen Typen führen – der Kontext macht den Unterschied:

```
Comparator<String>                c1 = StringUtils::compareTrimmed;
BiFunction<String,String,Integer> c2 = StringUtils::compareTrimmed;
```

1.3.1 Varianten von Methodenreferenzen

Im Beispiel ist die Methode `compareTrimmed(…)` statisch, und links vom Doppelpunkt steht der Name eines Typs. Allerdings kann beim Einsatz eines Typnamens die Methode auch nichtstatisch sein, `String::length` ist so ein Beispiel. Das wäre eine Funktion, die ein `String` auf ein `int` abbildet, in Code:

```
Function<String,Integer> len = String::length;
```

Links von den zwei Doppelpunkten kann auch eine Referenz stehen, was dann immer eine Objektmethode referenziert.

Beispiel

Während `String::length` eine Funktion ist, wäre `string::length` ein `Supplier`, unter der Annahme, dass `string` eine Referenzvariable ist:

```
String string = "Goll";
Supplier<Integer> len = string::length;
System.out.println( len.get() );     // 4
```

`System.out` ist eine Referenz, und eine Methode wie `println(…)` kann an einen `Consumer` gebunden werden. Es ist aber auch ein `Runnable`, weil es `println()` auch ohne Parameterliste gibt:

```
Consumer<String> out = System.out::println;
out.accept( "Kates kurze Kleider" );
```

```
Runnable out = System.out::println;
out.run();
```

Ist eine Hauptmethode mit `main(String... args)` deklariert, so ist das auch ein `Runnable`:

```
Runnable r = JavaApplication1::main;
```

Anders wäre das bei `main(String[])`, hier ist ein Parameter zwingend, doch ein Vararg kann auch leer sein.

Anstatt den Namen einer Referenzvariablen zu wählen, kann auch `this` das Objekt beschreiben, und auch `super` ist möglich. `this` ist praktisch, wenn die Implementierung einer funktionalen Schnittstelle auf eine Methode der eigenen Klasse delegieren möchte. Wenn zum Beispiel eine lokale Methode `compareTrimmed(…)` in der Klasse existieren würde, in der auch der Lambda-Ausdruck steht, und wenn diese Methode als `Comparator` in `Arrays.sort(…)` verwendet werden sollte, könnte es heißen: `Arrays.sort(words, this::compareTrimmed)`.

Hinweis

Es ist nicht möglich, eine spezielle Methode über die Methodenreferenz auszuwählen. Eine Angabe wie `String::valueOf` oder `Arrays::sort` ist relativ breit – bei Letzterem wählt der Compiler eine der 18 passenden überladenen Methoden aus. Da kann es passieren, dass der Compiler eine falsche Methode auswählt. In dem Fall muss ein expliziter Lambda-Ausdruck eine Mehrdeutigkeit auflösen. Bei generischen Typen kann zum Beispiel `List<String>::length` oder auch `List::length` stehen, auch hier erkennt der Compiler wieder alles selbst.

Was soll das alles?

Einem Einsteiger in die Sprache Java wird dieses Sprache-Feature wie der größte Zauber auf Erden vorkommen, und auch Java-Profis bekommen hier zittrige Finger, entweder vor Furcht oder Aufregung … In der Vergangenheit musste in Java sehr viel Code explizit geschrieben werden, aber mit diesen neuen Methodenreferenzen erkennt und macht der Compiler vieles von selbst.

Nützlich wird diese Eigenschaft mit den funktionalen Bibliotheken bei der Stream-API, die ein eigenes Kapitel im zweiten Band einnehmen. Hier nur ein kurzer Vorgeschmack:

```
Object[] words = { " ", '3', null, "2", 1, "" };
Arrays.stream( words )                    // " ", '3', null, "2", 1, ""
      .filter( Objects::nonNull )         // " ", '3', "2", 1, ""
      .map( Objects::toString )           // " ", "3", "2", "1", ""
      .map( String::trim )                // "", "3", "2", "1", ""
      .filter( s -> ! s.isEmpty() )       // "3", "2", "1"
      .map( Integer::parseInt )           // 3, 2, 1
```

```
        .sorted()                           // 1, 2, 3
        .forEach( System.out::println );    // 1 2 3
```

1.4 Konstruktorreferenz

Um ein Objekt aufzubauen, nutzen wir das Schlüsselwort `new`. Das führt zum Aufruf eines Konstruktors, dem sich optional Argumente übergeben lassen. Die Java-API deklariert aber auch Typen, von denen sich keine direkten Exemplare mit `new` aufbauen lassen. Stattdessen gibt es Erzeuger, deren Aufgabe es ist, Objekte aufzubauen. Die Erzeuger können statische oder auch nichtstatische Methoden sein:

Konstruktor …	**… erzeugt:**	**Erzeuger …**	**… baut:**
`new Integer( "1" )`	`Integer`	`Integer.valueOf( "1" )`	`Integer`
`new File( "dir" )`	`File`	`Paths.get( "dir" )`	`Path`
`new BigInteger( val )`	`BigInteger`	`BigInteger.valueOf( val )`	`BigInteger`

Tabelle 0.4: Beispiele für Konstruktoren und Erzeuger-Methoden

Beide, Konstruktoren und Erzeuger, lassen sich als spezielle Funktionen sehen, die von einem Typ in einen anderen Typ konvertieren. Damit eignen sie sich perfekt für Transformationen, und in einem Beispiel haben wir das schon eingesetzt:

```
Arrays.stream( words )
        . …
        .map( Integer::parseInt )
        . …
```

`Integer.parseInt(string)` ist eine Methode, die sich einfach mit einer Methodenreferenz fassen lässt, und zwar als `Integer::parseInt`. Aber was ist mit Konstruktoren? Auch sie transformieren! Statt `Integer.parseInt(string)` hätte ja auch `new Integer(string)` eingesetzt werden können.

Wo Methodenreferenzen statische Methoden und Objektmethoden angeben können, bieten *Konstruktorreferenzen* die Möglichkeit, Konstruktoren anzugeben, sodass diese als Erzeuger an anderer Stelle übergeben werden können. Damit lassen sich elegant Konstruktoren als Erzeuger angeben, und zwar auch von einer Klasse, die nicht über Erzeugermethoden verfügt. Wie auch bei Methodenreferenzen spielt eine funktionale Schnittstelle eine entscheidende Rolle, doch dieses Mal ist es die Methode der funktionalen Schnittstelle, die mit ihrem Aufruf zum Konstruktoraufruf

führt. Wo syntaktisch bei Methodenreferenzen rechts vom Doppelpunkt ein Methodenname steht, ist dies bei Konstruktorreferenzen ein `new`.[4] Also ergibt sich alternativ zu

```
.map( Integer::parseInt )        // Methode Integer.parseInt(String)
```

in unserem Beispiel das Ergebnis mittels:

```
.map( Integer::new )             // Konstruktor Integer(String)
```

Mit der Konstruktorreferenz gibt es vier Möglichkeiten, funktionale Schnittstellen zu implementieren; die drei verbleibenden Varianten sind Lambda-Ausdrücke, Methodenreferenzen und klassische Implementierung über eine Klasse.

Beispiel

Die funktionale Schnittstelle sei:

```
interface DateFactory { Date create(); }
```

Die folgende Konstruktorreferenz bindet den Konstruktor an die Methode `create()` der funktionalen Schnittstelle:

```
DateFactory factory = Date::new;
System.out.print( factory.create() ); // zum Beispiel Sat Dec 29 09:56:35 CET 2012
```

Beziehungsweise die letzten beiden Zeilen zusammengefasst:

```
System.out.println( ((DateFactory)Date::new).create() );
```

Soll nur der Standard-Konstruktor aufgerufen werden, muss die funktionale Schnittstelle nur eine Methode besitzen, die keinen Parameter besitzt und etwas zurückliefert. Der Rückgabetyp der Methode muss natürlich mit dem Klassentyp zusammenpassen. Das gilt für den Typ `DateFactory` aus unserem Beispiel. Doch es geht noch etwas generischer, zum Beispiel mit der vorhandenen funktionalen Schnittstelle `Supplier`, wie wir gleich sehen werden.

In der API finden sich oftmals Parameter vom Typ `Class`, die als Typangabe dazu verwendet werden, dass die Methode mit `newInstance()` Exemplare bilden kann. Der Einsatz von `Class` lässt sich durch eine funktionale Schnittstelle ersetzen, und Konstruktorreferenzen lassen sich an Stelle von `Class`-Objekten übergeben.

1.4.1 Standard- und parametrisierte Konstruktoren

Beim Standard-Konstruktor hat die Methode nur eine Rückgabe, bei einem parametrisierten Konstruktor muss die Methode der funktionalen Schnittstelle natürlich über eine kompatible Parameterliste verfügen:

[4] Da `new` ein Schlüsselwort ist, kann keine Methode so heißen; der Identifizierer ist also sicher.

Konstruktor	`Date()`	`Date(long t)`
Kompatible funktionale Schnittstelle	`interface DateFactory { Date create(); }`	`interface DateFactory { Date create(long t); }`
Konstruktorreferenz	`DateFactory factory = Date::new;`	`DateFactory factory = Date::new;`
Aufruf	`factory.create();`	`factory.create(1);`

Tabelle 0.5: Standard- und parametrisierter Konstruktor mit korrespondierenden funktionalen Schnittstellen

Hinweis

Kommt die Typ-Inferenz des Compilers an ihre Grenzen, sind zusätzliche Typinformationen gefordert. In diesem Fall werden hinter dem Doppelpunkt in eckigen Klammen weitere Angaben gemacht, etwa `Klasse::<Typ1, Typ2>new`.

1.4.2 Nützliche vordefinierte Schnittstellen für Konstruktorreferenzen

Die für einen Standard-Konstruktor passende funktionale Schnittstelle muss eine Rückgabe besitzen und keinen Parameter annehmen; die funktionale Schnittstelle für einen parametrisierten Konstruktor muss eine entsprechende Parameterliste haben. Es kommt nun häufig vor, dass der Konstruktor ein Standard-Konstruktor ist oder genau einen Parameter annimmt. Hier ist es vorteilhaft, dass für diese beiden Fälle die Java-API zwei praktische (generisch deklarierte) funktionale Schnittstellen mitbringt:

Funktionale Schnittstelle	**Funktions-Deskriptor**	**Abbildung**	**Passt auf**
`Supplier<T>`	`T get()`	() → T	Standard-Konstruktor
`Function<T,R>`	`R apply(T t)`	(T) → R	einfacher parametrisierter Konstruktor

Tabelle 0.6: Vorhandene funktionale Schnittstellen als Erzeuger

Beispiel

Die funktionale Schnittstelle `Supplier<T>` hat eine `T get()`-Methode, die wir mit dem Standard-Konstruktor von `Date` verbinden können:

```
Supplier<Date> factory = Date::new;
System.out.print( factory.get() );
```

Wir nutzen `Supplier` mit dem Typparameter `Date`, was den parametrisierten Typ `Supplier<Date>` ergibt, und `get()` liefert folglich den Typ `Date`. Der Aufruf `factory.get()` führt zum Aufruf des Konstruktors.

***Ausblick** **

Besonders interessant werden die Konstruktorreferenzen mit den neuen Bibliotheksmethoden der Stream-API. Nehmen wir eine Liste vom Typ Zeitstempel an. Der Konstruktor `Date(long)` nimmt einen solchen Zeitstempel entgegen, und mit einem `Date`-Objekt können wir Vergleiche vornehmen, etwa ob ein Datum hinter einem anderen Datum liegt. Folgendes Beispiel listet alle Datumswerte auf, die nach dem 1.1.2012 liegen:

```
Long[] timestamps = { 2432558632L, 1455872986345L };
Date thisYear = new GregorianCalendar( 2012, Calendar.JANUARY, 1 ).getTime();
Arrays.stream( timestamps )
      .map( Date::new )
      .filter( thisYear::before )
      .forEach( System.out::println );  // Fri Feb 19 10:09:46 CET 2016
```

Die Konstruktorreferenz `Date::new` hilft dabei, das `long` mit dem Zeitstempel in ein `Date`-Objekt zu konvertieren.

Denksportaufgabe

Ein Konstruktor kann als `Supplier` oder `Function` gelten. Problematisch sind mal wieder geprüfte Ausnahmen. Der Leser soll überlegen, ob der Konstruktor `URI(String str) throws URISyntaxException` über `URI::new` angesprochen werden kann.

1.5 Implementierung von Lambda-Ausdrücken *

Als die Compilerentwickler einen Prototyp für Lambda-Ausdrücke bauten, setzten sie diese technisch mit inneren Klassen um. Doch das war nur in der Testphase, denn innere Klassen sind für die JVM komplette Klassen und schwergewichtig. Das Laden und Initialisieren ist relativ teuer und würde bei den vielen kleinen Lambda-Ausdrücken einen großen Overhead darstellen. Daher nutzt die Implementierung ein *invokedynamic* genanntes Konstrukt. Das hat den großen Vorteil, dass die Laufzeitumgebung viel Gestaltungsraum in der Optimierung hat. Innere Klassen sind nur eine mögliche technische Umsetzung für Lambda-Ausdrücke, invokedynamic ist sozusagen die deklarative Variante, und innere Klassen sind die imperative. Letztendlich ist der Overhead mit invokedynamic gering, und Programmcode von inneren Klassen hin zu Lambda-Ausdrücken zu

refaktorisieren führt zu kleinen Bytecodedateien. Von der Performance her unterscheiden sich Lambda-Ausdrücke und die Implementierung funktionaler Schnittstellen und Klassen nicht, eher ist die Optimierung auf der Seite der JVM zu finden, die es mit weniger Klassendateien zu tun hat. Umgekehrt bedeutet das auch: Wenn Entwickler ihre alte vorhandene Implementierung von funktionalen Schnittstellen durch Lambda-Ausdrücke ersetzen, wird der Bytecode kompakter, da ein kleines invokedynamic viel kürzer ist als komplexe neue Klassendateien.

1.6 Funktionale Programmierung mit Java

1.6.1 Programmierparadigmen: imperativ oder deklarativ

In irgendeiner Weise muss ein Entwickler sein Problem in Programmform beschreiben, damit der Computer es letztendlich ausführen kann. Hier gibt es verschiedene Beschreibungsformen, die wir *Programmierparadigmen* nennen. Bisher haben wir uns immer mit der imperativen Programmierung beschäftigt, bei der Anweisungen im Mittelpunkt stehen. Wir haben im Deutschen den Imperativ, also die Befehlsform, die sehr gut mit dem Programmierstil vergleichbar ist, denn es handelt sich in beiden Fällen um Anweisungen der Art „tue dies, tue das“. Diese „Befehle“ mit Variablen, Fallunterscheidungen, Sprüngen beschreiben das Programm und den Lösungsweg.

Zwar ist imperative Programmierung die technisch älteste, aber nicht die einzige Form, Programme zu beschreiben; es gibt daneben die deklarative Programmierung, die nicht das Wie zur Problemlösung beschreibt, sondern das Was, also was eigentlich gefordert ist, ohne sich in genauen Abläufen zu verstricken. Auf den ersten Blick klingt das abstrakt, aber jeder, der schon einmal

- eine Selektion wie **.html* auf der Kommandozeile/im Explorer-Suchfeld getätigt,
- eine Datenbankabfrage mit SQL geschrieben,
- eine XML-Selektion mit XQuery genutzt,
- ein Build-Skript mit Ant oder make formuliert oder
- eine XML-Transformation mit XSLT beschrieben hat,

wird das Prinzip kennen.

Bleiben wir kurz bei SQL, um einen Punkt deutlich zu machen. Natürlich führt im Endeffekt die CPU die Abarbeitung der Tabellen und die Auswertungen der Ergebnisse rein imperativ aus, doch es geht um die Programmbeschreibung auf einem höheren Abstraktionsniveau. Deklarative Programme sind üblicherweise wesentlich kürzer, und damit kommen weitere Vorteile wie leichtere Erweiterbarkeit, Verständlichkeit ins Spiel. Da deklarative Programme oftmals einen mathematischen Hintergrund haben, lassen sich die Beschreibungen leichter formal in ihrer Korrektheit beweisen.

Deklarative Programmierung ist ein Programmierstil, und eine deklarative Beschreibung braucht eine Art „Ablaufumgebung“, denn SQL kann zum Beispiel keine CPU direkt ausführen. Aber anstatt nur spezielle Anwendungsfälle wie Datenbank- oder XML-Abfragen zu behandeln, können auch typische Algorithmen deklarativ formuliert werden, und zwar mit funktionaler Programmierung. Damit sind imperative Programme und funktionale Programme gleich mächtig in ihren Möglichkeiten.

1.6.2 Funktionale Programmierung und funktionale Programmiersprachen

Bei der funktionalen Programmierung stehen Funktionen im Mittelpunkt und ein im Idealfall zustandsloses Verhalten, in dem viel mit Rekursion gearbeitet wird. Ein typisches Beispiel ist die Berechnung der Fakultät. Es ist n! = 1 × 2 × 3 × … × n, und mit Schleifen und Variablen, dem imperativen Weg, sieht sie so aus:

```
public static int factorial( int n ) {
  int result = 1;
  for ( int i = 1; i <= n; i++ )
    result *= i;
  return result;
}
```

Deutlich sind die vielen Zuweisungen und die Fallunterscheidung durch die Schleife abzulesen, die typischen Indikatoren für imperative Programme. Der Schleifenzähler erhöht sich, damit kommt Zustand in das Programm, denn der aktuelle Index muss ja irgendwo im Speicher gehalten werden. Bei der rekursiven Variante ist das ganz anders, hier gibt es keine Zuweisungen im Programm, und die Schreibweise erinnert an die mathematische Definition:

```
public static int factorial( int n ) {
  return n == 0 ? 1 : n * factorial( n - 1 );
}
```

Mit der funktionalen Programmierung haben wir eine echte Alternative zur imperativen Programmierung. Die Frage ist nur: Mit welcher Programmiersprache lassen sich funktionale Programme schreiben? Im Grunde mit jeder höheren Programmiersprache! Denn funktional zu programmieren ist ja ein Programmierstil, und Java unterstützt funktionale Programmierung, wie wir am Beispiel mit der Fakultät ablesen können. Da das im Prinzip schon alles ist, stellt sich die Frage, warum funktionale Programmierung einen so schweren Stand hat und bei den Entwicklern gefürchtet ist. Das hat mehrere Gründe:

- **Lesbarkeit**: Am Anfang der funktionalen Programmiersprachen steht historisch LISP aus dem Jahr 1958, eine sehr flexible, aber ungewohnt zu lesende Programmiersprache. Unsere Fakultät sieht in LISP so aus:

```
(defun factorial (n) (if (= n 1) 1 (* n (factorial (- n 1)))))
```

Die ganzen Klammern machen die Programme nicht einfach lesbar, und die Ausdrücke stehen in der Präfix-Notation `- n 1` statt der üblichen Infix-Notation `n - 1`. Bei anderen funktionalen Programmiersprachen ist es anders, dennoch führt das zu einem gewissen Vorurteil, dass alle funktionalen Programmiersprachen schlecht lesbar seien.

- **Performance und Speicherverbrauch**: Ohne clevere Optimierungen von Seiten des Compilers und der Laufzeitumgebung führen insbesondere rekursive Aufrufe zu prall gefüllten Stacks und schlechter Laufzeit.
- **Rein funktional**: Es gibt funktionale Programmiersprachen, die als „rein“ oder „pur“ bezeichnet werden und keine Zustandsänderungen erlauben. Die Entwicklung von Ein-/Ausgabeoperationen oder simplen Zufallszahlen ist ein großer Akt, der für normale Entwickler nicht mehr nachvollziehbar ist. Die Konzepte sind kompliziert, doch zum Glück sind die meisten funktionalen Sprachen nicht so rein und erlauben Zustandsänderungen, nur Programmierer versuchen genau diese Zustandsänderungen zu vermeiden, um sich nicht die Nachteile damit einzuhandeln.
- **Funktional mit Java**: Wenn es darum geht, nur mit Funktionen zu arbeiten, kommen Entwickler schnell an einen Punkt, an dem Funktionen andere Funktionen als Argumente übergeben oder Funktionen zurückgeben. So etwas lässt sich in Java in der traditionellen Syntax nur sehr umständlich schreiben. Dies führt dazu, dass alles so unlesbar wird, dass der ganze Vorteil der kompakten deklarativen Schreibweise verloren geht.

Aus heutiger Sicht stellt sich eine Kombination aus beiden Konzepten als zukunftsweisend dar. Mit Lambda-Ausdrücken sind funktionale Programme kompakt und relativ gut lesbar, und die JVM hat gute Optimierungsmöglichkeiten. Java ermöglicht beide Programmierparadigmen, und Entwickler können den Weg wählen, der für eine Problemlösung gerade am besten ist. Diese Mehrdeutigkeit schafft natürlich auch Probleme, denn immer wenn es mehrere Lösungswege gibt, entstehen Auseinandersetzungen um die beste der Varianten – und hier kann von Entwickler zu Entwickler eine konträre Meinung herrschen. Funktionale Programmierung hat unbestrittene Vorteile, und das wollen wir uns genau anschauen.

1.6.3 Funktionale Programmierung in Java am Beispiel vom Comparator

Funktionale Programmierung hat auch daher etwas Akademisches, weil in den Köpfen der Entwickler oftmals dieses Programmierparadigma nur mit mathematischen Funktionen in Verbindung gebracht wird. Und die wenigsten werden tatsächlich Fakultät oder Fibonacci-Zahlen in Programmen benötigen und daher schnell funktionale Programmierung beiseitelegen. Doch diese Vorurteile sind unbegründet, und es ist hilfreich, funktionale Programmierung gedanklich von der Mathematik zu lösen, denn die allermeisten Programme haben nichts mit mathematischen

Funktionen im eigentlichen Sinne zu tun, wohl aber viel stärker mit formal beschriebenen Methoden.

Betrachten wir erneut unser Beispiel aus der Einleitung, die Sortierung von Strings, diesmal aus der Sicht eines funktionalen Programmierers. Ein `Comparator` ist eine einfache „Funktion“, mit zwei Parametern und einer Rückgabe. Diese „Funktion“ (realisiert als Methode) wiederum wird an die `sort(…)`-Methode übergeben. Alles das ist funktionale Programmierung, denn wir programmieren Funktionen und übergeben sie. Drei Beispiele (Generics ausgelassen):

Code	Bedeutung
`Comparator c = (c1, c2) -> …`	Implementiert eine Funktion über einen Lambda-Ausdruck.
`Arrays.sort(T[] a, Comparator c)`	Nimmt eine Funktion als Argument an.
`Collections.reverseOrder(Comparator cmp)`	Nimmt eine Funktion an und liefert auch eine zurück.

Tabelle 0.7: Beispiele für Funktionen in der Übergabe und als Rückgabe

Funktionen selbst können in Java nicht übergeben werden, also helfen sich Java-Entwickler mit der Möglichkeit, die Funktionalität in eine Methode zu setzen, sodass die Funktion zum Objekt mit einer Methode wird, was die Logik realisiert. Lambda-Ausdrücke bzw. Methoden/Konstruktorreferenzen geben eine kompakte Syntax ohne den Ballast, extra eine Klasse mit einer Methode schreiben zu müssen.

Der Typ `Comparator` ist eine funktionale Schnittstelle und steht für eine besondere Funktion mit zwei Parametern gleichen Typs und einer Ganzzahl-Rückgabe. Es gibt weitere funktionale Schnittstellen, die etwas flexibler sind als `Comparator`, in der Weise, dass etwa die Rückgabe statt `int` auch `double` oder etwas anderes sein kann.

1.6.4 Lambda-Ausdrücke als Funktionen sehen

Wir haben gesehen, dass sich Lambda-Ausdrücke in einer Syntax formulieren lassen, die folgende allgemeine Form hat:

```
( LambdaParameter ) -> { Anweisungen }
```

Der Pfeil macht gut deutlich, dass wir es bei Lambda-Ausdrücken mit Funktionen zu tun haben, die etwas abbilden. Im Fall vom `Comparator` ist es eine Abbildung von zwei Strings auf eine Ganzzahl – in eine etwas mathematischere Notation gepackt: (String, String) → int.

Beispiel

Methoden gibt es mit und ohne Rückgabe und mit und ohne Parameter. Genauso ist das mit Lambda-Ausdrücken. Ein paar Beispiele in Java-Code mit ihren Abbildungen:

Lambda-Ausdruck	Abbildung
`(int a, int b) -> a + b`	(int, int) → int
`(int a) -> Math.abs( a )`	(int) → int
`(String s) -> s.isEmpty()`	(String) → boolean
`(Collection c) -> c.size()`	(Collection) → int
`() -> Math.random()`	() → double
`(String s) -> { System.out.print( s ); }`	(String) → void
`() -> {}`	() → void

Tabelle 0.8: Lambda-Ausdrücke und was sie als Funktionen abbilden

Begriff: Funktion versus Methode

Die Java-Sprachdefinition kennt den Begriff „Funktion“ nicht, sondern spricht nur von Methoden. Methoden hängen immer an Klassen, und das heißt, dass Methoden immer an einem Kontext hängen. Das ist zentral bei der Objektorientierung, da Methoden auf Attribute lesend und schreibend zugreifen können. Lambda-Ausdrücke wiederum realisieren Funktionen, die erst einmal ihre Arbeitswerte rein aus den Parametern beziehen, sie hängen nicht an Klassen und Objekten. Der Gedanke bei funktionalen Programmiersprachen ist der, ohne Zustände auszukommen, also Funktionen so clever anzuwenden, dass sie ein Ergebnis liefern. Funktionen geben für eine spezifische Parameterkombination immer dasselbe Ergebnis zurück, unabhängig vom Zustand des umgebenden Gesamtprogramms.

1.7 Funktionale Schnittstelle aus dem java.util.function-Paket

Funktionen realisieren Abbildungen, und da es verschiedene Arten von Abbildungen geben kann, bietet die Java-Standardbibliothek im Paket `java.util.function` für die häufigsten Fälle funktionale Schnittstellen an. Ein erster Überblick:

Schnittstelle	Abbildung
`Consumer<T>`	(T) → void

`DoubleConsumer`	(double) → void
`BiConsumer<T,U>`	(T, U) → void
`Supplier<T>`	() → T
`BooleanSupplier`	() → boolean
`Predicate<T>`	(T) → boolean
`LongPredicate`	(long) → boolean
`BiPredicate<T,U>`	(T, U) → boolean
`Function<T,R>`	(T) → R
`LongToDoubleFunction`	(long) → double
`BiFunction<T,U,R>`	(T, U) → R
`UnaryOperator<T>`	(T) → T
`DoubleBinaryOperator`	(double) → boolean

Tabelle 0.9: Beispiele einiger vordefinierter funktionaler Schnittstellen

1.7.1 Blöcke mit Code und die funktionale Schnittstelle java.util.function.Consumer

Anweisungen von Code lassen sich in eine Methode eines Objekts setzen und auf diese Weise weitergeben. Das ist eine häufige Notwendigkeit, für die das Paket `java.util.function` eine einfache funktionale Schnittstelle `Consumer` vorgibt, die einen Konsumenten repräsentiert, der Daten annimmt und dann verbraucht (konsumiert).

```
interface java.util.function.Consumer<T>
```

- `void accept(T t)`
  Führt Operationen mit der Übergabe `t` durch.
- `default Consumer<T> andThen(Consumer<? super T> after)`
  Liefert einen neuen `Consumer`, der erst den aktuellen `Consumer` ausführt und danach `after`.

Die `accept(…)`-Methode bekommt ein Argument – wobei die Implementierung natürlich nicht zwingend darauf zurückgreifen muss – und liefert keine Rückgabe. Transformationen sind damit nicht möglich, denn nur über Umwege kann der Konsument die Ergebnisse speichern, und dafür ist die Schnittstelle nicht gedacht. `Consumer`-Typen sind eher gedacht als Endglied einer Kette, in der

zum Beispiel Dateien in eine Datei geschrieben werden, die vorher verarbeitet wurden. Diese Seiteneffekte sind beabsichtigt, da sie nach einer Kette von seiteneffektfreien Operationen stehen.

Immer repräsentieren Konsumenten Code, und eine API kann nun einfach einen Codeblock nach der Art `doSomethingWith(myConsumer)` annehmen, um ihn etwa in einem Hintergrund-Thread abzuarbeiten oder wiederholend auszuführen, oder kann ihn nach einer erlaubten Maximaldauer abbrechen oder die Zeit messen oder, oder, oder …

Beispiel

Implementiere einen `Consumer`-Wrapper, der die Ausführungszeit eines anderen Konsumenten loggt:

Listing 0.7: Consumers.java

```
import java.util.function.*;
import java.util.logging.Logger;
class Consumers {
  public static <T> Consumer<T> measuringConsumer( Consumer<T> block ) {
    return t -> {
      long start = System.nanoTime();
      block.accept( t );
      long duration = System.nanoTime() - start;
      Logger.getAnonymousLogger().info( "Ausführungszeit (ns): " + duration );
    };
  }
}
```

Folgender Aufruf zeigt die Nutzung:

```
Consumer<Void> wrap = measuringConsumer( Void -> System.out.println( "Test" ) );
wrap.accept( null );
```

Was wir hier implementiert haben, ist ein Beispiel für das Execute-around-Method-Muster, bei dem wir um einen Block Code noch etwas anderes legen.

Typ Consumer in der API

In der Java-API zeigt sich der Typ `Consumer` in der Regel als Argument einer Methode `forEach(Consumer)`, die Datenquellen abläuft und für jedes Element `accept(…)` aufruft. Interessant ist die Methode am Typ `Iterable`, denn die wichtigen `Collection`-Datenstrukturen wie `ArrayList` implementieren diese Schnittstelle. So lässt sich einfach über alle Daten laufen und ein Stück Code für jedes Element ausführen. Auch `Iterator` hat eine vergleichbare Methode, da heißt sie `forEachRemaining(Consumer)` – das „Remaining“ macht deutlich, dass der `Iterator`

schon ein paar `next()`-Aufrufe erlebt haben könnte und die Konsumenten daher nicht zwingend die ersten Elemente mitbekommen.

Beispiel

Gib jedes Element einer Liste auf der Konsole aus:

```
Arrays.asList( 1, 2, 3, 4 ).forEach( System.out::println );
```

Gegenüber einem normalen Durchiterieren ist die funktionale Variante ein wenig kürzer im Code, aber sonst gibt es keinen Unterschied. Auch `forEach(…)` macht auf dem `Iterable` nichts anderes, als alle Elemente über den `Iterator` zu holen.

1.7.2 Supplier

Ein *Supplier* (auch *Provider* genannt) ist eine Fabrik und sorgt für Objekte. In Java deklariert das Paket `java.util.function` die funktionale Schnittstelle `Supplier` für Objektgeber:

```
interface java.util.function.Supplier<T>
```

- `T get()`
  Führt Operationen mit der Übergabe `t` durch.

Weitere statische oder Default-Methoden deklariert `Supplier` nicht. Was `get()` nun genau liefert, ist Aufgabe der Implementierung und ein Internum. Es können neue Objekte sein, immer die gleichen Objekte (Singleton) oder Objekte aus einem Cache.

1.7.3 Prädikate und java.util.function.Predicate

Ein Prädikat ist eine Aussage über einen Gegenstand, die wahr oder falsch ist. Die Frage mit `Character.isDigit('a')`, ob das Zeichen „a“ eine Ziffer ist, wird mit falsch beantwortet – `isDigit` ist also ein Prädikat, weil es über einen Gegenstand, ein Zeichen, eine Wahrheitsaussage fällen kann.

Prädikate als Objekte sind flexibel, denn Objekte lassen sich an unterschiedliche Stellen weitergeben. Wenn etwa ein Prädikat bestimmt, was aus einer Sammlung gelöscht werden soll oder ob mindestens ein Element in einer Sammlung ist, das ein Prädikat erfüllt.

Das `java.util.function`-Paket[5] deklariert eine flexible funktionale Schnittstelle `Predicate` auf folgende Weise:

```
interface java.util.function.Predicate<T>
```

[5] Achtung, in `javax.sql.rowset` gibt es ebenfalls eine Schnittstelle `Predicate`.

- `boolean test(T t)`
  Führt einen Test auf `t` durch und liefert `true`, wenn das Kriterium erfüllt ist, sonst `false`.

Beispiel

Der Test, ob ein Zeichen eine Ziffer ist, kann durch Prädikat-Objekte nun auch anders durchgeführt werden:

```
Predicate<Character> isDigit =
   c -> Character.isDigit( c );  // kurz: Character::isDigit
System.out.println( isDigit.test('a') );  // false
```

Hätte es die Schnittstelle `Predicate` schon früher in Java 1.0 gegeben, hätte es der Methode `Character.isDigit(…)` gar nicht bedurft, es hätte auch ein `Predicate<Character>` als statische Variable in der Klasse `Character` geben können, sodass ein Test dann geschrieben würde als `Character.IS_DIGIT.test(…)` oder als Rückgabe von einer Methode `Predicate<Character> isDigit()` mit der Nutzung `Character.isDigit().test(…)`. Es ist daher gut möglich, dass sich in Zukunft die API dahingehend verändert, dass Aussagen auf Gegenständen mit Wahrheitsrückgabe nicht mehr als Methoden bei den Klassen realisiert werden, sondern als Prädikat-Objekte angeboten werden. Aber Methodenreferenzen geben zum Glück die Flexibilität, dass existierende Methoden problemlos als Lambda-Ausdrücke genutzt werden können, und so kommen wir wieder von Methoden zu Funktionen.

Typ Predicate in der API

Es gibt in der Java-API einige Stellen, an denen `Predicate`-Objekte genutzt werden:

- als Argument für Löschmethoden, um in Sammlungen Elemente zu spezifizieren, die gelöscht oder nach denen gefiltert werden soll
- bei den Default-Methoden der `Predicate`-Schnittstelle selbst, um Prädikate zu verknüpfen
- bei regulären Ausdrücken; ein `Pattern` liefert mit `asPredicate()` ein `Predicate` für Tests.
- in der Stream-API, bei der Objekte beim Durchlaufen des Stroms über ein Prädikat identifiziert werden, um sie etwa auszufiltern

Beispiel

Lösche aus einer Liste mit Zeichen alle, die Ziffern sind (es bleiben nur Zeichen übrig, etwa Buchstaben):

```
Predicate<Character> isDigit = Character::isDigit;
List<Character> list = new ArrayList<>( Arrays.asList( 'a', '1' ) );
list.removeIf( isDigit );
```

Auf diese Weise steht nicht die Schleife, sondern das Löschen im Vordergrund.

Default-Methoden von Predicate

Es gibt eine Reihe von Default-Methoden, die die funktionale Schnittstelle `Predicate` anbietet. Zusammenfassend:

```
interface java.util.function.Predicate<T>
```

- `default Predicate<T> negate()`
  Liefert vom aktuellen Prädikat eine Negation. Implementiert als `return t -> ! test(t);`.
- `default Predicate<T> and(Predicate<? super T> p)`
- `default Predicate<T> or(Predicate<? super T> p)`
  Verknüpft das aktuelle Prädikat mit einem anderen Prädikat logisch Und/Oder.
- `static <T> Predicate<T> isEqual(Object targetRef)`
  Liefert ein neues Prädikat, das einen Gleichheitstest mit `targetRef` vornimmt, im Grunde `return ref -> Objects.equals(ref, targetRef)`.

Beispiel

Lösche aus einer Liste mit Zeichen alle die, die *keine* Ziffern sind:

```
Predicate<Character> isDigit = Character::isDigit;
Predicate<Character> isNotDigit = isDigit.negate();
List<Character> list = new ArrayList<>( Arrays.asList( 'a', '1' ) );
list.removeIf( isNotDigit );
```

Prädikate aus Pattern

Die `Pattern`-Methode `asPredicate()` liefert ein `Predicate<String>`, sodass ein regulärer Ausdruck als Kriterium, zum Beispiel zum Filtern oder Löschen von Einträgen in Datenstrukturen, genutzt werden kann.

Beispiel

Lösche aus einer Liste alle Strings, die leer oder Zahlen sind:

```
List<String> list = new ArrayList<>( Arrays.asList( "Peaches", "", "25", "Geldof" )
 );
list.removeIf( Pattern.compile( "\\d+" ).asPredicate().or( String::isEmpty ) );
System.out.println( list );            // [Peaches, Geldof]
```

1.7.4 Funktionen und die allgemeine funktionale Schnittstelle java.util.function.Function

Funktionen im Sinne der funktionalen Programmierung können in verschiedenen Bauarten vorkommen: mit Parameterliste/Rückgabe oder ohne. Doch im Grunde sind es Spezialformen, und die funktionale Schnittstelle `java.util.function.Function` ist die allgemeinste, die zu einem Argument ein Ergebnis liefert.

```
interface java.util.function.Function<T,R>
```

- `R apply(T t)`
  Wendet eine Funktion an, und liefert zur Eingabe `t` eine Rückgabe.

Beispiel

Eine Funktion zur Bestimmung des Absolutwerts:

```
Function<Double,Double> abs = a -> Math.abs( a );  // alternativ Math::abs
System.out.println( abs.apply( -12. ) );           // 12.0
```

Auch bei Funktionen ergibt sich für das API-Design ein Spannungsfeld, denn im Grunde müssen „Funktionen“ nun gar nicht mehr als Methoden angeboten werden, sondern Klassen könnten sie auch als `Function`-Objekte anbieten. Doch da Methodenreferenzen problemlos die Brücke von Methodennamen zu Objekten schlagen, fahren Entwickler mit klassischen Methoden ganz gut.

Typ Function in der API

Die Stream-API ist der größte Nutznießer des `Function`-Typs. Es finden sich einige wenige Beispiele bei Objektvergleichen (`Comparator`), im Paket für Nebenläufigkeiten und bei Assoziativspeichern. Im Abschnitt über die Stream-API werden wir daher viele weitere Beispiele kennenlernen.

Beispiel

Ein Assoziativspeicher soll als Cache realisiert werden, der zu Dateinamen den Inhalt assoziiert. Ist zu dem Schlüssel (dem Dateinamen) noch kein Inhalt vorhanden, soll dieser in den Assoziativspeicher gelegt werden.

Listing 0.8: FileCache.java, FileCache

```
class FileCache {
  private Map<String,byte[]> map = new HashMap<>();
  public byte[] getContent( String filename ) {
    return map.computeIfAbsent( filename, file -> {
      try {
```

```
        return Files.readAllBytes( Paths.get( file ) );
      } catch ( IOException e ) { throw new UncheckedIOException( e ); }
    } );
  }
}
```

Auf die Methode kommt Kapitel 16, „Einführung in Datenstrukturen und Algorithmen“, noch einmal zurück, das Beispiel soll nur eine Vorstellung geben, dass Funktionen an andere Funktionen übergeben werden – hier eine `Function<String,byte[]>` an `computeIfAbsent(…)`. Sobald an den Datei-Cache der Aufruf `getContent(String)` geht, wird dieser die `Map` fragen, und wenn diese zu dem Schlüssel keinen Wert hat, wird sie den Lambda-Ausdruck auswerten, um zu dem Dateinamen den Inhalt zu liefern.

Getter-Methoden als Function über Methodenreferenzen

Methodenreferenzen gehören zu den syntaktisch knappsten Sprachmitteln von Java. In Kombination mit Gettern ist ein Muster abzulesen, das oft in Code zu sehen ist. Zunächst noch einmal zur Wiederholung von `Function` und der Nutzung bei Methodenreferenzen:

```
Function<String,String> func1a = (String s) -> s.toUpperCase();
Function<String,String> func1b = String::toUpperCase;
Function<Point,Double>  func2a = (Point p) -> p.getX();
Function<Point,Double>  func2b = Point::getX;
System.out.println( func1b.apply( "jocelyn" ) );         // JOCELYN
System.out.println( func2b.apply( new Point( 9, 0 ) ) ); // 9.0
```

Dass `Function` auf die gegebene Methodenreferenz passt, ist auf den ersten Blick unverständlich, da die Signaturen von `toUpperCase()` und `getX()` keinen Parameter deklarieren, also im üblichen Sinne keine Funktionen sind, wo etwas reinkommt und wieder rauskommt. Wir haben es hier aber mit einem speziellen Fall zu tun, denn die in der Methodenreferenz genannten Methoden sind a) nicht statisch – wie `Math::max` –, und b) ist auch keine Referenz – wie `System.out::print` – im Spiel, sondern hier wird der Compiler eine Objektmethode auf genau dem Objekt aufrufen, das als erstes Argument der funktionalen Schnittstelle übergeben wurde. (Diesen Satz bitte zweimal lesen.)

Damit ist `Function` ein praktischer Typ bei allen Szenarien, bei denen irgendwie über Getter Zustände erfragt werden, wie es etwa bei einem `Comparator` öfter vorkommt. Hier ist eine statische Methode – die Generics einmal ausgelassen – `Comparator<…> Comparator.comparing(Function<…> keyExtractor)` sehr nützlich.

Beispiel

Besorge eine Liste von Paketen, die vom Klassenlader zugänglich sind, und sortiere sie nach Namen:

```
List<Package> list = Arrays.asList( Package.getPackages() );
Collections.sort( list, Comparator.comparing( Package::getName ) );
System.out.println( list ); // [package java.io, … sun.util.locale …
```

Default-Methoden in Function

Die funktionale Schnittstelle schreibt nur eine Methode `apply(…)` vor, deklariert jedoch noch drei zusätzliche Default-Methoden:

```
interface java.util.function.Function<T,R>
```

- `static <T> Function<T,T> identity()`
  Liefert eine neue Funktion, die immer die Eingabe als Ergebnis liefert.
- `default <V> Function<T,V> andThen(Function<? super R,? extends V> after)`
  Entspricht `t -> after.apply(apply(t))`.
- `default <V> Function<V,R> compose(Function<? super V,? extends T> before)`
  Entspricht `v -> apply(before.apply(v))`.

Die Methoden `andThen(…)` und `compose(…)` unterscheiden sich darin, in welcher Reihenfolge die Funktionen aufgerufen werden. Das Gute ist, dass die Parameternamen („before“, „after“) klarmachen, was hier in welcher Reihenfolge aufgerufen wird, wenn auch „compose“ selbst wenig aussagt.

Function versus Consumer/Predicate

Im Grunde lässt sich alles als `Function` darstellen, denn

- ein `Consumer<T>` lässt sich auch als `Function<T,Void>` verstehen (es geht etwas rein, aber nichts raus),
- ein `Predicate<T>` als `Function<T,Boolean>` und
- ein `Supplier<T>` als `Function<Void,T>`.

Dennoch erfüllen diese speziellen Typen ihren Zweck, denn je genauer der Typ, desto besser.

UnaryOperator

Es gibt auch eine weitere Schnittstelle im `java.util.function`-Paket, die `Function` spezialisiert, und zwar `UnaryOperator`. Ein `UnaryOperator` ist eine spezielle Funktion, bei der die Typen für „Eingang“ und „Ausgang“ gleich sind.

```
interface java.util.function.UnaryOperator<T>
extends Function<T,T>
```

- `static <T> UnaryOperator<T> identity()`
  Liefert den Identitäts-Operator, der alle Eingaben auf die Ausgaben abbildet.

Die generischen Typen machen deutlich, dass der Typ des Methodenparameters gleich dem Ergebnistyp ist. Bis auf `identity()` gibt es keine weitere Funktionalität, die Schnittstelle dient lediglich zur Typdeklaration.

An einigen Stellen der Java-Bibliothek kommt dieser Typ auch vor, etwa bei der Methode `replaceAll(UnaryOperator)` der `List`-Typen.

Beispiel

Verdopple jeden Eintrag in der Liste:

```
List<Integer> list = Arrays.asList( 1, 2, 3 );
list.replaceAll( e -> e * 2 );
System.out.println( list );  // [2, 4, 6]
```

1.7.5 Ein bisschen Bi …

Bi ist eine bekannte lateinische Vorsilbe für „zwei“, was übertragen auf die Typen aus `java.util.function` bedeutet, das statt eines Arguments zwei übergeben werden können.

Typ	Schnittstelle	Operation
Konsument	`Consumer<T>`	`void accept(T t)`
	`BiConsumer<T,U>`	`void accept(T t, U u)`
Funktion	`Function<T,R>`	`R apply(T t)`
	`BiFunction<T,U,R>`	`R apply(T t, U u)`
Prädikat	`Predicate<T>`	`boolean test(T t)`
	`BiPredicate<T,U>`	`boolean test(T t, U u)`

Tabelle 0.10: Ein-/Zwei-Argument-Methoden im Vergleich

Die Bi-Typen haben mit den Nicht-Bi-Typen keine Typbeziehung.[6]

BiConsumer

Der `BiConsumer` deklariert die Methode `accept(T, U)` mit zwei Parametern, die jeweils unterschiedliche Typen tragen können. Haupteinsatzpunkt des Typs in der Java-Standardbibliothek sind Assoziativspeicher, die Schlüssel und Werte an `accept(…)` übergeben. So deklariert `Map` die Methode:

```
interface java.util.Map<K,V>
```

- `default void forEach(BiConsumer<? super K,? super V> action)`
  Läuft den Assoziativspeicher ab und ruft auf jedem Schlüssel-Wert-Paar die `accept(…)`-Methode vom übergebenen `BiConsumer` auf.

Beispiel

Gib die Temperaturen der Städte aus:

```
Map<String,Integer> map = new HashMap<>();
map.put( "Manila", 25 );
map.put( "Leipzig", -5 );
map.forEach( (k, v) -> System.out.printf("%s hat %d Grad%n", k, v) );
```

Ein `BiConsumer` besitzt eine Default-Methode `andThen(…)`, wie auch der `Consumer` sie zur Verkettung deklariert.

```
interface java.util.function.BiConsumer<T,U>
```

- `default BiConsumer<T,U> andThen(BiConsumer<? super T,? super U> after)`
  Verknüpft den aktuellen `BiConsumer` mit `after` zu einem neuen `BiConsumer`.

BiFunction und BinaryOperator

Eine `BiFunction` ist eine Funktion mit zwei Argumenten, während eine normale `Function` nur ein Argument annimmt.

[6] Irgendwie finden das manche Leser lustig …

Beispiel

Nutzung von `Function` und `BiFunction` mit Methodenreferenzen:

```
Function<Double,Double> sign = Math::abs;
BiFunction<Double,Double,Double> max = Math::max;
```

Die Java-Bibliothek greift viel öfter auf `Function` zurück als auf `BiFunction`. Der häufigste Einsatz findet sich in der Standardbibliothek rund um Assoziativspeicher, bei denen Schlüssel und Wert an eine `BiFunction` übergeben werden.

Beispiel

Konvertiere alle assoziierten Werte einer `HashMap` nach Großschreibung:

```
Map<Integer,String> map = new HashMap<>();
map.put( 1, "eins" ); map.put( 2, "zwEi" );
System.out.println( map );  // {1=eins, 2=zwEi}
BiFunction<Integer,String,String> func = (k, v) -> v.toUpperCase();
map.replaceAll( func );
System.out.println( map );  // {1=EINS, 2=ZWEI}
```

Ist bei einer `Function` der Typ derselbe, bietet die Java-API dafür den spezielleren Typ `UnaryOperator`. Sind bei einer `BiFunction` alle drei Typen gleich, bietet sich hier `BinaryOperator` an – zum Vergleich:

- `interface UnaryOperator<T> extends Function<T,T>`
- `interface BinaryOperator<T> extends BiFunction<T,T,T>`

Beispiel

`BiFunction` und `BinaryOperator`:

```
BiFunction<Double,Double,Double> max1 = Math::max;
BinaryOperator<Double>           max2 = Math::max;
```

`BinaryOperator` spielt bei so genannten Reduktionen eine große Rolle, wenn zum Beispiel aus zwei Werten einer wird, wie bei `Math.max(…)`; auch das beleuchtet der Abschnitt über die Stream-API später genauer.

Die Schnittstelle `BiFunction` deklariert genau eine Default-Methode:

```
interface java.util.function.BiFunction<T,U,R>
extends Function<T,T>
```

- `default <V> BiFunction<T,U,V> andThen(Function<? super R,? extends V> after)`

`BinaryOperator` dagegen wartet mit zwei statischen Methoden auf:

```
public interface java.util.function.BinaryOperator<T>
extends BiFunction<T,T,T>
```

- `static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)`
- `static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)`
  Liefert einen `BinaryOperator`, der das Maximum/Minimum bezüglich eines gegebenen `comparator` liefert.

BiPredicate

Ein `BiPredicate` testet zwei Argumente und verdichtet sie zu einem Wahrheitswert. Wie `Predicate` deklariert auch `BiPredicate` drei Default-Methoden `and(…)`, `or(…)` und `negate(…)`, wobei natürlich eine statische `isEqual(…)`-Methode wie bei `Predicate` in `BiPredicate` fehlt. Für `BiPredicate` gibt es in der Java-Standardbibliothek nur eine Verwendung bei einer Methode zum Finden von Dateien – der Gebrauch ist selten, zudem ja auch ein Prädikat immer einer Funktion mit `boolean`-Rückgabe ist, sodass es eigentlich für diese Schnittstelle keine zwingende Notwendigkeit gibt.

Beispiel

Bei `BiXXX` und zwei Argumenten hört im Übrigen die Spezialisierung auf, es gibt keine Typen `TriXXX`, `QuaddXXX` … Das ist in der Praxis auch nicht nötig, denn zum einen kann oftmals eine Reduktion stattfinden – so ist etwa `max(1, 2, 3)` gleich `max(1, max(2, 3))` –, und zum anderen kann auch der Parametertyp eine Sammlung sein, wie in `Function<List<Integer>, Integer> max`.

1.7.6 Funktionale Schnittstellen mit Primitiven

Die bisher vorgestellten funktionalen Schnittstellen sind durch die generischen Typparameter sehr flexibel, aber was fehlt, sind Signaturen mit Primitiven – Java hat das „Problem“, dass Generics nur mit Referenztypen funktionieren, nicht aber mit primitiven Typen. Aus diesem Grund gibt es von fast allen Schnittstellen aus dem `function`-Paket vier Versionen: eine generische für beliebige Referenzen sowie Versionen für den Typ `int`, `long` und `double`. Die API-Designer wollten gerne die Wrapper-Typen außen vor lassen und gewisse primitive Typen unterstützen, auch aus Performance-Gründen, um nicht immer ein Boxing durchführen zu müssen.

Tabelle 0.11 gibt einen Überblick über die funktionalen Schnittstellen, die alle keinerlei Vererbungsbeziehungen zu anderen Schnittstellen haben.

Funktionale Schnittstelle	Funktionsdeskriptor
XXXSupplier	
BooleanSupplier	boolean getAsBoolean()
IntSupplier	int getAsInt()
LongSupplier	long getAsLong()
DoubleSupplier	double getAsDouble()
XXXConsumer	
IntConsumer	void accept(int value)
LongConsumer	void accept(long value)
DoubleConsumer	void accept(double value)
ObjIntConsumer<T>	void accept(T t, int value)
ObjLongConsumer<T>	void accept(T t, long value)
ObjDoubleConsumer<T>	void accept(T t, double value)
XXXPredicate	
IntPredicate	boolean test(int value)
LongPredicate	boolean test(long value)
DoublePredicate	boolean test(double value)
XXXFunction	
DoubleToIntFunction	int applyAsInt(double value)
IntToDoubleFunction	double applyAsDouble(int value)
LongToIntFunction	int applyAsInt(long value)
IntToLongFunction	long applyAsLong(int value)
DoubleToLongFunction	long applyAsLong(double value)

`LongToDoubleFunction`	`double applyAsDouble(long value)`
`IntFunction<R>`	`R apply(int value)`
`LongFunction<R>`	`R apply(long value)`
`DoubleFunction<R>`	`R apply(double value)`
`ToIntFunction<T>`	`int applyAsInt(T t)`
`ToLongFunction<T>`	`long applyAsLong(T t)`
`ToDoubleFunction<T>`	`double applyAsDouble(T t)`
`ToIntBiFunction<T,U>`	`int applyAsInt(T t, U u)`
`ToLongBiFunction<T,U>`	`long applyAsLong(T t, U u)`
`ToDoubleBiFunction<T,U>`	`double applyAsDouble(T t, U u)`
`XXXOperator`	
`IntUnaryOperator`	`int applyAsInt(int operand)`
`LongUnaryOperator`	`long applyAsLong(long operand)`
`DoubleUnaryOperator`	`double applyAsDouble(double operand)`
`IntBinaryOperator`	`int applyAsInt(int left, int right)`
`LongBinaryOperator`	`long applyAsLong(long left, long right)`
`DoubleBinaryOperator`	`double applyAsDouble(double left, double right)`

Tabelle 0.11: Spezielle funktionale Schnittstellen für primitive Werte

Statische und Default-Methoden

Einige generisch deklarierte funktionale Schnittstellen-Typen besitzen Default-Methoden bzw. statische Methoden, und Ähnliches findet sich auch bei den primitiven funktionalen Schnittstellen wieder:

- Die `XXXConsumer`-Schnittstellen deklarieren `default XXXConsumer andThen(XXXConsumer after)`, aber nicht die `ObjXXXConsumer`-Typen, sie besitzen keine Default-Methode.
- Die `XXXPredicate`-Schnittstellen deklarieren:
  - `default XXXPredicate negate()`
  - `default XXXPredicate and(XXXPredicate other)`

  - `default XXXPredicate or(XXXPredicate other)`
- Jeder `XXXUnaryOperator` besitzt:
  - `default XXXUnaryOperator andThen(XXXUnaryOperator after)`
  - `default XXXUnaryOperator compose(XXXUnaryOperator before)`
  - `static XXXUnaryOperator identity()`
- `BinaryOperator` hat zwei statische Methoden `maxBy(…)` und `minBy(…)`, die es nicht in der primitiven Version `XXXBinaryOperator` gibt, da kein `Comparator` bei primitiven Vergleichen nötig ist.
- Die `XXXSupplier`-Schnittstellen deklarieren keine statischen Methoden oder Default-Methoden, genauso wie es auch `Supplier` nicht tut.

1.8 Optional ist keine Nullnummer

Java hat eine besondere Referenz, die Entwicklern die Haare zu Berge stehen lässt und die Grund für lange Debug-Stunden ist: die `null`-Referenz. Eigentlich sagt `null` nur aus: „nicht initialisiert“. Doch was `null` so problematisch macht, ist die `NullPointerException`, die durch referenzierte `null`-Ausdrücke ausgelöst wird.

Beispiel

Entwickler haben vergessen, das Attribut `location` mit einem Objekt zu initialisieren, sodass `setLocation(…)` fehlschlagen wird:

```
class Place {
  private Point2D location;
  public void setLocation( double longitude, double latitude ) {
    location.setLocation( longitude, latitude );  // ☠ NullPointerException
  }
}
```

Einsatz von null

Fehler dieser Art sind durch Tests relativ leicht aufzuspüren. Aber hier liegt nicht das Problem. Das eigentliche Problem ist, dass Entwickler allzu gerne die typenlose `null`[7] als magischen Sonderwert sehen, sodass sie neben „nicht initialisiert“ noch etwas anderes bedeutet:

- Erlaubt die API in Argumenten für Methoden/Konstruktoren `null`, heißt das meistens „nutze einen Default-Wert“ oder „nichts gegeben, ignorieren“.

---

[7] `null instanceof Typ` ist immer `false`.

- In Rückgaben von Methoden steht `null` oftmals für „nichts gemacht“ oder „keine Rückgabe“. Im Gegensatz dazu kodieren andere Methoden wiederum mit der Rückgabe `null`, dass eine Operation erfolgreich durchlaufen wurde, und würden sonst zum Beispiel Fehlerobjekte zurückgegeben.[8]

Beispiel 1

Die mit Javadoc dokumentiere Methode `getTask(out, fileManager, diagnosticListener, options, classes, compilationUnits)` in der Schnittstelle `JavaCompiler` ist so ein Beispiel:

- `out`: „a writer for additional output from the compiler; use system.err **if null“**
- `fileManager`: „a file manager; **if null** use the compiler’s standard filemanager“
- `diagnosticListener`: „a diagnostic listener; **if null** use the compiler’s default method for reporting diagnostics“
- `options`: „compiler options, **null** means no options“
- `classes`: „names of classes to be processed by annotation processing, **null** means no class names“
- `compilationUnits`: „the compilation units to compile, null means no compilation units“

Alle Argumente können `null` sein, `getTask(null, null, null, null, null, null)` ist ein korrekter Aufruf. Schön ist die API nicht, und besser wäre sie wohl mit einem Builder-Pattern gelöst.

Beispiel 2

Der `BufferedReader` erlaubt das zeilenweise Einlesen aus Datenquellen, und `readLine()` liefert `null`, wenn es keine Zeile mehr zu lesen gibt.

Beispiel 3

Viel Irritation gibt es mit der API vom Assoziativspeicher. Eine gewöhnliche `HashMap` kann als assoziierten Wert `null` bekommen, doch `get(key)` liefert auch dann `null`, wenn es keinen assoziierten Wert gibt. Das führt zu einer Mehrdeutigkeit, da die Rückgabe von `get(…)` nicht verrät, ob es eine Abbildung auf `null` gibt oder ob der Schlüssel nicht vorhanden ist.

```
Map<Integer,String> map = new HashMap<>();
map.put( 0, null );
System.out.println( map.containsKey( 0 ) );      // true
```

[8] Zum Glück wird `null` selten als Fehler-Identifikator genutzt, die Zeiten sind vorbei. Hier sind Ausnahmen die bessere Wahl, denn Fehler sind Ausnahmen im Programm.

```
System.out.println( map.containsValue( null ) ); // true
System.out.println( map.get( 0 ) );              // null
System.out.println( map.get( 1 ) );              // null
```

Kann die Map `null`-Werte enthalten, muss es immer ein Paar der Art `if(map.containsKey(key))`, gefolgt von `map.get(key)` geben. Am besten verzichten Entwickler auf `null` in Datenstrukturen.

Da `null` so viele Einsatzfälle hat und das Lesen der API-Dokumentation gerne übersprungen wird, sollte es zu einigen `null`-Einsätzen Alternativen geben. Manches Mal ist das einfach, etwa wenn die Rückgabe Sammlungen sind. Dann gibt es mit einer leeren Sammlung eine gute Alternative zu `null`. Das ist ein Spezialfall des so genannten Null-Object-Patterns und wird in Kapitel 16, „Einführung in Datenstrukturen und Algorithmen“, näher beschrieben.

Fehler, die aufgrund einer `NullPointerException` entstehen, ließen sich natürlich komplett vermeiden, wenn immer ordentlich auf `null`-Referenzen getestet würde. Aber gerade die `null`-Prüfungen werden von Entwicklern gerne vergessen, da ihnen nicht bewusst ist oder sie nicht erwarten, dass eine Rückgabe `null` sein kann. Gewünscht ist ein Programmkonstrukt, bei dem explizit wird, dass ein Wert nicht vorhanden sein kann, sodass nicht `null` diese Rolle übernehmen muss. Wenn im Code lesbar ist, dass ein Wert optional ist, also vorhanden sein kann oder nicht, reduziert das Fehler.

Geschichte

Tony Hoare gilt als „Erfinder“ der `null`-Referenz. Heute bereut er es und nennt die Entscheidung „my billion-dollar mistake“.[9]

1.8.1 Optional-Typ

Die Java-Bibliothek bietet eine Art Container, der ein Element enthalten kann oder nicht. Wenn der Container ein Element enthält, ist es nie `null`. Dieser Container kann befragt werden, ob er ein Element enthält oder nicht. Eine `null` als Kennung ist somit überflüssig.

[9] Er sagt dazu: „It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn’t resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.“ Unter *http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare* gibt es ein Video mit ihm und Erklärungen.

Beispiel

```
Optional<String> opt1 = Optional.of( "Aitazaz Hassan Bangash" );
System.out.println( opt1.isPresent() );   // true
System.out.println( opt1.get() );         // Aitazaz Hassan Bangash
Optional<String> opt2 = Optional.empty();
System.out.println( opt2.isPresent() );   // false
// opt2.get() -> java.util.NoSuchElementException: No value present
Optional<String> opt3 = Optional.ofNullable( "Malala" );
System.out.println( opt3.isPresent() );   // true
System.out.println( opt3.get() );         // Malala
Optional<String> opt4 = Optional.ofNullable( null );
System.out.println( opt4.isPresent() );   // false
// opt4.get() -> java.util.NoSuchElementException: No value present
```

```
final class java.util.Optional<T>
```

- `static <T> Optional<T> empty()`
  Liefert ein leeres `Optional`-Objekt.
- `boolean isPresent()`
  Liefert wahr, wenn dieses `Optional` einen Wert hat, sonst ist wie im Fall von `empty()` die Rückgabe `false`.
- `static <T> Optional<T> of(T value)`
  Baut ein neues `Optional` mit einem Wert auf, der nicht `null` sein darf; andernfalls gibt es eine `NullPointerException`. `null` in das `Optional` hineinzubekommen, geht also nicht.
- `static <T> Optional<T> ofNullable(T value)`
  Liefert ein `Optional` mit dem Wert, wenn dieser ungleich `null` ist, bei `null` ist die Rückgabe ein `Optional.empty()`.
- `T get()`
  Liefert den Wert. Enthält das `Optional` keinen Wert, weil kein Wert `isPresent()` ist, folgt eine `NoSuchElementException`.
- `T orElse(T other)`
  Ist ein Wert `isPresent()`, liefere den Wert. Ist das `Optional` leer, liefere `other`.

Des Weiteren überschreibt `Optional` die Methoden `equals(…)`, `toString()` und `hashCode()` – 0, wenn kein Wert gegeben ist, sonst Hashcode vom Element – und ein paar weitere Methoden, die wir uns später anschauen.

Hinweis

Intern `null` zu verwenden hat zum Beispiel den Vorteil, dass die Objekte serialisiert werden können. `Optional` implementiert `Serializable` *nicht*, daher sind `Optional`-Attribute nicht serialisierbar, können also etwa nicht im Fall von Remote-Aufrufen mit RMI übertragen werden. Auch die Abbildung auf XML oder auf Datenbanken ist umständlicher, wenn nicht JavaBean-Properties herangezogen werden, sondern die internen Attribute.

Ehepartner oder nicht?

`Optional` wird also dazu verwendet, im Code explizit auszudrücken, ob ein Wert vorhanden ist oder nicht. Das gilt auf beiden Seiten: Der Erzeuger muss explizit `ofXXX(…)` aufrufen und der Nutzer explizit `isPresent()` oder `get()`. Beide Seiten sind sich bewusst, dass sie es mit einem Wert zu tun haben, der optional ist, also existieren kann oder nicht. Wir wollen das in einem Beispiel nutzen, und zwar für eine Person, die einen Ehepartner haben kann:

Listing 0.9: com/tutego/insel/java/lang/Person.java, Person

```
public class Person {
  private Person spouse;
  public void setSpouse( Person spouse ) {
    this.spouse = Objects.requireNonNull( spouse );
  }
  public void removeSpouse() {
    spouse = null;
  }
  public Optional<Person> getSpouse() {
    return Optional.ofNullable( spouse );
  }
}
```

In diesem Beispiel ist `null` für die interne Referenz auf den Partner möglich; diese Kodierung soll aber nicht nach außen gelangen. Daher liefert `getSpouse()` nicht direkt die Referenz, sondern es kommt `Optional` zum Einsatz und drückt aus, ob eine Person einen Ehepartner hat oder nicht. Auch bei `setSpouse(…)` akzeptieren wir kein `null`, denn `null`-Argumente sollten so weit wie möglich vermieden werden. Ein `Optional` ist hier nicht angemessen, weil es ein Fehler ist, `null` zu übergeben. Zusätzlich sollte natürlich die Javadoc an `setSpouse(…)` dokumentieren, dass ein `null`-Argument zu einer `NullPointerException` führt. Daher passt `Optional` als Parametertyp nicht.

Listing 0.10: com/tutego/insel/java/lang/OptionalDemo.java, main()

```
Person heinz = new Person();
System.out.println( heinz.getSpouse().isPresent() ); // false
Person eva = new Person();
heinz.setSpouse( eva );
System.out.println( heinz.getSpouse().isPresent() ); // true
System.out.println( heinz.getSpouse().get() ); // com/…/Person
heinz.removeSpouse();
System.out.println( heinz.getSpouse().isPresent() ); // false
```

1.8.2 Primitive optionale Typen

Während Referenzen `null` sein können und auf diese Weise das Nichtvorhandensein anzeigen, ist das bei primitiven Datentypen nicht so einfach. Wenn eine Methode ein `boolean` zurückgibt, bleibt neben `true` und `false` nicht viel übrig, und ein „nicht zugewiesen“ wird dann doch gerne wieder über einen `Boolean` verpackt und auf `null` getestet. Gerade bei Ganzzahlen gibt es immer wieder Rückgaben wie –1.[10] Das ist bei den folgenden Beispielen der Fall:

- Wenn bei `InputStream`s `read(…)` keine Eingaben mehr kommen, wird `-1` zurückgegeben.
- `indexOf(Object)` von `List` liefert `-1`, wenn das gesuchte Objekt nicht in der Liste ist und folglich auch keine Position vorhanden ist.
- Bei einer unbekannten Bytelänge einer MIDI-Datei (Typ `MidiFileFormat`) hat `getByteLength()` als Rückgabe `-1`.

Diese magischen Werte sollten vermieden werden, und daher kann auch der optionale Typ wieder erscheinen.

Als generischer Typ kann `Optional` beliebige Typen kapseln, und primitive Werte könnten in Wrapper verpackt werden. Allerdings bietet Java für drei primitive Typen spezielle `Optional`-Typen an: `OptionalInt`, `OptionalLong`, `OptionalDouble`:

Optional<T>	OptionalInt	OptionalLong	OptionalDouble
`static <T> Optional<T> empty()`	`static OptionalInt empty()`	`static OptionalLong empty()`	`static Optional-Double empty()`
`T get()`	`int getAsInt()`	`long getAsLong()`	`double getAsDouble()`
`boolean isPresent()`			

[10] Unter *http://docs.oracle.com/javase/8/docs/api/constant-values.html* lassen sich alle Konstantendeklarationen einsehen.

`static <T> Optional<T> of(T value)`	`static OptionalInt of(int value)`	`static OptionalLong of(long value)`	`static OptionalDouble of(double value)`
`static <T> Optional<T> ofNullable(T value)`	nicht übertragbar		
`T orElse(T other)`	`int orElse(int other)`	`long orElse(long other)`	`double orElse(double other)`
`boolean equals(Object obj)`			
`int hashCode()`			
`String toString()`			

Tabelle 0.12: Methodenvergleich zwischen den vier OptionalXXX-Klassen

Die `Optional`-Methode `ofNullable(…)` fällt in den primitiven `Optional`-Klassen natürlich raus. Die optionalen Typen für die drei primitiven Typen haben insgesamt weniger Methoden, und die obere Tabelle ist nicht ganz vollständig. Wir kommen im Rahmen der funktionalen Programmierung in Java noch auf die verbleibenden Methoden wie `isPresent(…)` zurück.

Best Practice

`OptionalXXX`-Typen eignen sich hervorragend als Rückgabetyp, sind als Parametertyp denkbar, doch wenig attraktiv für interne Attribute. Intern ist `null` eine akzeptable Wahl, der „Typ“ ist schnell und speicherschonend.

1.8.3 Erstmal funktional mit Optional

Neben den vorgestellten Methoden wie `ofXXX(…)` und `isPresent()` gibt es weitere, die auf funktionale Schnittstellen zurückgreifen:

```
final class java.lang.Optional<T>
```

- `void ifPresent(Consumer<? super T> consumer)`
  Repräsentiert das `Optional` einen Wert, rufe den `Consumer` mit diesem Wert auf, andernfalls mache nichts.

- `Optional<T> filter(Predicate<? super T> predicate)`
  Enthält das `Optional` einen Wert und ist das Prädikat `predicate` auf dem Wert wahr, ist die Rückgabe das eigene `Optional` (also `this`), sonst ist die Rückgabe `Optional.empty()`.
- `<U> Optional<U> map(Function<? super T,? extends U> mapper)`
  Repräsentiert das `Optional` einen Wert, dann wende die Funktion an und verpacke das Ergebnis (wenn es ungleich `null` ist) wieder in ein `Optional`. Ist das `Optional` ohne Wert, dann ist die Rückgabe `Optional.empty()`, genauso, wenn die Funktion `null` liefert.
- `<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)`
  Wie `map(…)`, nur dass die Funktion ein `Optional` statt eines direkten Werts gibt. Liefert die Funktion `mapper` ein leeres `Optional`, so ist das Ergebnis von `flatMap(…)` auch `Optional.empty()`.
- `T orElseGet(Supplier<? extends T> other)`
  Repräsentiert das `Optional` einen Wert, so liefere ihn; ist das `Optional` leer, so beziehe den Alternativwert aus dem `Supplier`.
- `<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)`
  Repräsentiert das `Optional` einen Wert, so liefere ihn, andernfalls hole mit `Supplier` das Ausnahme-Objekt, und löse es aus.

Beispiel

Wenn das Optional keinen Wert hat, soll eine `NullPointerException` statt der `NoSuchElementException` ausgelöst werden.

```
String s = optionalString.orElseThrow( NullPointerException::new );
```

Beispiel für NullPointerException-sichere Kaskadierung von Aufrufen mit Optional

Die beiden `XXXmap(…)`-Methoden sind besonders interessant und ermöglichen einen ganz neuen Programmierstil. Warum, soll ein Beispiel zeigen.

Der folgende Zweizeiler gibt auf meinem System „MICROSOFT KERNELDEBUGGER-NETZWERKADAPTER“ aus:

```
String s = NetworkInterface.getByIndex( 2 ).getDisplayName().toUpperCase();
System.out.println( s );
```

Allerdings ist der Programmcode alles andere als gut, denn `NetworkInterface.getByIndex(int)` kann `null` zurückgeben und `getDisplayName()` auch. Um ohne eine `NullPointerException` um die Klippen zu schiffen, müssen wir schreiben:

```
NetworkInterface networkInterface = NetworkInterface.getByIndex( 2 );
if ( networkInterface != null ) {
  String displayName = networkInterface.getDisplayName();
```

```
  if ( displayName != null )
    System.out.println( displayName.toUpperCase() );
}
```

Von der Eleganz des Zweizeilers ist nicht mehr viel geblieben. Integrieren wir `Optional` (was ja eigentlich ein toller Rückgabetyp für `getByIndex()` und `getDisplayName()` wäre):

```
Optional<NetworkInterface> networkInterface = Optional.ofNullable( NetworkInterface.g
etByIndex( 2 ) );
if ( networkInterface.isPresent() ) {
  Optional<String> name = Optional.ofNullable( networkInterface.get().getDisplayName(
) );
  if ( name.isPresent() )
    System.out.println( name.get().toUpperCase() );
}
```

Mit `Optional` wird es nicht sofort besser, doch statt `if` können wir einen Lambda-Ausdruck nehmen und bei `ifPresent(…)` einsetzen:

```
Optional<NetworkInterface> networkInterface = Optional.ofNullable( NetworkInterface.g
etByIndex( 2 ) );
networkInterface.ifPresent( ni -> {
  Optional<String> displayName = Optional.ofNullable( ni.getDisplayName() );
  displayName.ifPresent( name -> {
    System.out.println( name.toUpperCase() );
  } );
} );
```

Wenn wir die lokalen Variablen `networkInterface` und `displayName` entfernen, landen wir bei:

```
Optional.ofNullable( NetworkInterface.getByIndex( 2 ) ).ifPresent( ni -> {
  Optional.ofNullable( ni.getDisplayName() ).ifPresent( name -> {
    System.out.println( name.toUpperCase() );
  } );
} );
```

Von der Struktur her ist das mit der `if`-Abfrage identisch und über die Einrückungen auch zu erkennen. Fallunterscheidungen mit `Optional` und `ifPresent(…)` umzuschreiben bringt keinen Vorteil.

In Fallunterscheidungen zu denken hilft hier nicht weiter. Was wir uns bei `NetworkInterface.getByIndex( 2 ).getDisplayName().toUpperCase()` vor Augen halten müssen, ist eine Kette von Abbildungen. `NetworkInterface.getByIndex(int)` bildet auf

`NetworkInterface` ab, `getDisplayName()` von `NetworkInterface` bildet auf `String` ab, und `toUpperCase()` bildet von einem `String` auf einen anderen `String` ab. Wir verketten drei Abbildungen und müssten ausdrücken können: Wenn eine Abbildung fehlschlägt, dann höre mit der Abbildung auf. Und genau hier kommen `Optional` und `map(…)` ins Spiel. In Code:

```
Optional<String> s = Optional.ofNullable( NetworkInterface.getByIndex( 2 ) )
                              .map( ni -> ni.getDisplayName() )
                              .map( name -> name.toUpperCase() );
s.ifPresent( System.out::println );
```

Die Klasse `Optional` hilft uns bei zwei Dingen: Erstens wird `map(…)` beim Empfangen einer `null`-Referenz auf ein `Optional.empty()` abbilden, und zweitens ist das Verketten von leeren `Optional`s kein Problem, es passiert einfach nichts – `Optional.empty().map(…)` führt nichts aus, und die Rückgabe ist einfach nur ein leeres `Optional`. Am Ende der Kette steht nicht mehr `String` (wie am Anfang des Beispiels), sondern `Optional<String>`.

Umgeschrieben mit Methodenreferenzen und weiter verkürzt ist der Code sehr gut lesbar und Null-Pointer-Exception-sicher:

```
Optional.ofNullable( NetworkInterface.getByIndex( 2 ) )
        .map( NetworkInterface::getDisplayName )
        .map( String::toUpperCase )
        .ifPresent( System.out::println );
```

Die Logik kommt ohne externe Fallunterscheidungen aus und arbeitet nur mit optionalen Abbildungen. Das ist ein schönes Beispiel für funktionale Programmierung.

Primitiv-Optionales

Die eigentliche `Optional`-Klasse ist generisch und kapselt jeden Referenztyp. Auch für die primitiven Typen `int`, `long` und `double` gibt es in drei speziellen Klassen `OptionalInt`, `OptionalLong`, `OptionalDouble` Methoden zur funktionalen Programmierung. Stellen wir die Methoden der vier `OptionalXXX`-Klassen gegenüber:

Optional<T>	OptionalInt	OptionalLong	OptionalDouble
`static <T> Optional<T> empty()`	`static OptionalInt empty()`	`static OptionalLong empty()`	`static OptionalDouble empty()`
`T get()`	`int getAsInt()`	`long getAsLong()`	`double getAsDouble()`
`boolean isPresent()`			

`static <T> Optional<T> of(T value)`	`static OptionalInt of(int value)`	`static OptionalLong of(long value)`	`static OptionalDouble of(double value)`
`static <T> Optional<T> ofNullable(T value)`	nicht übertragbar		
`T orElse(T other)`	`int orElse(int other)`	`long orElse(long other)`	`double orElse(double other)`
`boolean equals(Object obj)`			
`int hashCode()`			
`String toString()`			
`void ifPresent(Consumer<? super T> consumer)`	`void ifPresent(IntConsumer consumer)`	`void ifPresent(LongConsumer consumer)`	`void ifPresent(DoubleConsumer consumer)`
`T orElseGet(Supplier<? extends T> other)`	`int orElseGet(IntSupplier other)`	`long orElseGet(LongSupplier other)`	`double orElseGet(DoubleSupplier other)`
`<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)`	`<X extends Throwable> int orElseThrow(Supplier<? extends X> exceptionSupplier)`	`<X extends Throwable> long orElseThrow(Supplier<? extends X> exceptionSupplier)`	`<X extends Throwable> double orElseThrow(Supplier<? extends X> exceptionSupplier)`
`Optional<T> filter(Predicate<? super T> predicate)`	nicht vorhanden		
`<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)`			
`<U> Optional<U> map(Function<? super T,? extends U> mapper)`			

Tabelle 0.13: Vergleich von Optional mit den primitiven OptionalXXX-Klassen

1.9 Was ist jetzt so funktional?

Bisher hat dieser Abschnitt einen Großteil darauf verwendet, die Typen aus dem `java.util.function`-Paket vorzustellen, also die funktionalen Schnittstellen, mit denen Entwickler Abbildungen in Java ausdrücken können. Wenig war von funktionaler Programmierung und den Vorteilen die Rede, das holen wir jetzt nach.

Wiederverwertbarkeit

Zunächst einmal bieten Funktionen eine zusätzliche Ebene der Wiederverwertbarkeit von Code. Nehmen wir ein Prädikat wie

```
Predicate<Path> exists = path -> Files.exists( path );
```

Dieses `exists`-Prädikat ist relativ einfach, könnte aber natürlich komplexer sein. Der Punkt ist, dass diese Prädikate an allen möglichen Stellen wiederverwendet werden können, etwa zum Filtern in Listen oder zum Löschen von Elementen aus Listen. Das Prädikat kann als Funktion weitergereicht oder zu neuen Prädikaten verbunden werden, etwa zu:

```
Predicate<Path> exists    = path -> Files.exists( path );
Predicate<Path> directory = path -> Files.isDirectory( path );
Predicate<Path> existsAndDirectory = exists.and( directory );
```

Methoden wie `ifPresent(Predicate)` oder `removeIf(Predicate)` nehmen dann dieses Prädikat und führen Operationen durch. Diese kleinen Mini-Objekte lassen sich sehr gut testen, und das minimiert insgesamt Fehler im Code.

Während aktuelle Bibliotheken wenig davon Gebrauch machen, Typen wie `Supplier`, `Consumer`, `Function`, `Predicate` anzunehmen und zurückzugeben, wird sich dieses im Laufe der nächsten Jahre ändern.

Zustandslos, immutable

Bei der funktionalen Programmierung geht es darum, ohne externe Zustände auszukommen.

Definition

Funktionen heißen pur, wenn sie ohne einen Zustand auskommen und keine Seiteneffekte haben. `Math.max(3, 4)` ist eine pure Funktion, `System.out.println()` oder `Math.random()` sind es nicht. Einen aus puren Funktionen aufgebauten Ausdruck nennen wir *puren Ausdruck*. Er hat eine Eigenschaft, die sich in der Informatik *referenzielle Transparenz* nennt, dass nämlich das Ergebnis eines Ausdrucks an Stelle des Ausdrucks selbst gesetzt werden kann, ohne dass das Programm ein

anderes Verhalten zeigt. Statt `Math.max(3, 4)` kann jederzeit `4` gesetzt werden, das Ergebnis wäre das gleiche. Ein Compiler kann bei referenzieller Transparenz diverse Optimierungen durchführen.

Pure funktionale Programmiersprachen basieren auf puren Funktionen, und auch in Java muss nicht jede Methode einen äußeren Zustand verändern. Allerdings sind es Java-Entwickler gewohnt, in Zuständen zu denken, und daran ist an sich nichts Falsches: Ein Textdokument im Speicher ist eben ein Objektgraph genauso wie eine grafische Anwendung mit Eingabefeldern. Worauf funktionale Programmierung abzielt, sind die Operationen auf den Datenstrukturen und Berechnungen, die ohne Seiteneffekte sind.

Pure Funktionen ohne Zustand haben den Vorteil, dass sie

- beliebig oft ausgeführt werden können, ohne dass sich Systemzustände ändern,
- in beliebiger Reihenfolge ausgeführt werden können, ohne dass das Ergebnis ein anderes wird.

Diese Vorteile sind reizvoll unter dem Gesichtspunkt der Parallelisierung, denn die Prozessoren werden nicht wirklich schneller, aber wir haben mehr Prozessorkerne zur Verfügung. Pure Funktionen erlauben es Bibliotheken, Aufgaben wie Suchen und Filtern auf Kerne zu verteilen und so zu parallelisieren. Je weniger Zustand dabei im Spiel ist, desto besser, denn je weniger Zustand, desto weniger Synchronisation und Warteeffekte gibt es.

Aufpassen müssen Entwickler natürlich trotzdem, denn ein Lambda-Ausdruck muss nicht pur sein und kann Seiteneffekte haben. Daher ist es wichtig zu wissen, wann diese Lambda-Ausdrücke vielleicht nebenläufig sind und eine Synchronisation nötig ist.

Beispiel

Die Schnittstelle `Iterable` deklariert eine Methode `forEach(…)`, mit einem Parameter vom Typ einer funktionalen Schnittstelle. Hier ist ein Lambda-Ausdruck möglich. Es wäre natürlich grundlegend falsch, wenn dieser Lambda-Ausdruck selbst in die Sammlung eingreift:

```
List<Integer> ints = new ArrayList<>( Arrays.asList( 1, 99, 2 ) );
ints.forEach( v -> { System.out.println( ints + ", " + v); ints.set( v, 0 ); } );
```

Die Ausgabe ist weit von dem, was erwartet wurde, aber kein Wunder, wenn Lambda-Ausdrücke illegale Seiteneffekte hervorrufen:

```
[1, 99, 2], 1
[1, 0, 2], 0
[0, 0, 2], 2
```

Die Vermeidung von Zuständen gekoppelt an die Unveränderbarkeit von Werten (engl. *immutability*) erhöht das Verständnis des Programms, da Entwickler es schwer haben, im Kopf das System mit den ganzen Änderungen „nachzuspielen“, insbesondere wenn diese Änderungen noch nebenläufig sind. Das zeigt das vorangehende Beispiel recht gut; solche Systeme zu verstehen und

zu debuggen ist schwer. Je weniger Seiteneffekte es gibt, desto einfacher ist das Programm zu verstehen. Zustände machen ein Programm komplex, nicht nur in nebenläufigen Umgebungen. Wenn die Methode pur ist, muss ein Entwickler nichts anderes tun, als den Code der Methode zu verstehen. Wenn die Methode von Zuständen des Objekts abhängt, muss ein Entwickler den Code der gesamten Klasse verstehen. Und wenn das Objekt von Zuständen im Gesamtprogramm abhängt, ufert das Ganze aus, denn dann ist noch viel mehr vom System zu verstehen.

1.10 Zum Weiterlesen

Funktional zu programmieren ändert grundlegend das Design von Java-Programmen: weg von Methoden mit Seiteneffekten hin zu kleinen Funktionen, die Objekte mit neuen Zuständen liefern. Die Zukunft wird uns Muster und Best Practices an die Hand geben, wie in Java entwickelt wird. Auch wird sich zeigen, ob weitere Konzepte der funktionalen Programmierung in Java bzw. die JVM einfließen werden. Bisher ist zum Beispiel Immutability kein Sprachkonstrukt, sondern durch die API gewährleistet, wenn es keine Setter oder Schreibzugriffe auf Variablen gibt; Reflection kann aber auch hier einen bösen Strich durch die Rechnung machen. Doch für Java 9 sind noch keine Planungen in dieser Richtung gemacht, das Gleiche gilt für Möglichkeiten wie Pattern Matching oder algebraische Datentypen (ADT) auf der Sprachseite oder Optimierung von Endrekursion (engl. *tail call optimization*) auf Seiten der JVM, etwas, was in anderen funktionalen Programmiersprachen immer hochgehalten wird.

Entwickler, die noch tiefer in die Denkweise funktionaler Programmierung eintauchen möchten, können sich mit puren funktionalen Programmiersprachen wie Haskell beschäftigen und müssen dort ohne Seiteneffekte auskommen. Etwas einfacher für Java-Programmierer ist die Sprachfamilie ML, die auch imperative Elemente wie `while`-Schleifen bietet. Für Java-Programmierer wirkt das meist fremd, die hippe Programmiersprache Scala vereint objektorientierte und funktionale Programmierung nahezu perfekt. Java-Entwickler profitieren am meisten von den funktionalen Ansätzen bei der Stream-API, die schon kurz angerissen wurde und im zweiten Insel-Band „Java SE 8 Standard-Bibliothek“ intensiv diskutiert wird.