1 Einleitung

Mit der Version 8 wurden auch in Java Lambda-Ausdrücke eingeführt. Auf den ersten Blick erscheinen diese neben den vielen anderen Sprachfeatures von Java nur als ein weiterer kleiner Zusatz. Tatsächlich bedeutet aber ihre Einführung und die damit einhergehende Unterstützung eines funktionalen Programmierstils einen revolutionären Wandel in der Art, wie man Programme in Java gestalten kann. Funktionale Programmierung verspricht eine Programmgestaltung, die in deklarativer Form, auf hohem Abstraktionsniveau, knapp und präzise und für eine parallele Ausführung geeignet ist. Funktionale Programmierung ist grundsätzlich unterschiedlich zur imperativen Programmierung, sie steht aber nicht im Gegensatz zur objektorientierten Programmierung. Wie Brian Goetz es in seinem Vorwort zu [56] ausdrückt, arbeitet Objektorientierung mit einer Abstraktion entlang der Daten, die funktionale Programmierung erlaubt aber eine Abstraktion von Verhaltensstrukturen. Damit ergänzen sich die beiden Paradigmen zu einer neuen Qualität der Programmgestaltung.

Funktionale Programmierung hat besonders in den letzten Jahren viel Aufmerksamkeit erhalten. Dabei ist funktionale Programmierung kein neues Paradigma. Tatsächlich ist die erste funktionale Programmiersprache Lisp fast so alt wie die erste imperative Programmiersprache Fortran. Die Entwicklung der theoretischen Grundlagen zur funktionalen Programmierung, der Lambda Kalkül [17], reicht sogar bis in die 1930er-Jahre zurück. Heute wird funktionale Programmierung von den meisten Programmiersprachen unterstützt.

Mehrere Entwicklungen sind wohl dafür verantwortlich, dass die funktionale Programmierung, die über so viele Jahre ein mehr oder weniger akademisches Nischendasein fristete, jetzt vom Mainstream der Programmierwelt aufgegriffen wurde:

- Mit dem Aufkommen von Prozessoren mit mehreren Kernen werden Softwarekonzepte für eine parallele Ausführung benötigt.
- Mit der zunehmenden Vernetzung, mit Systemen, die ständig online verbunden sind, mit Serverarchitekturen, die zehntausende Klienten gleichzeitig bedienen müssen, und Internet of Things-Anwendungen, die ständig auf Ände-

rungen in der Umwelt reagieren sollen, versagen bisherige Paradigmen der Programmierung.

Und gerade die funktionale Programmierung wird als Mittel gesehen, diesen Herausforderungen zu begegnen.

Eigenschaften funktionaler Programme

Funktionale Programmierung beruht auf mathematischen Funktionen. In der Mathematik ist eine Funktion eine Abbildung einer Menge von Elementen des Definitionsbereichs *D* in eine weitere Menge von Elementen des Wertebereichs *Z*:

$$f : D \rightarrow Z, x \mapsto y$$

Jedem Element des Definitionsbereichs wird ein *eindeutiger* Wert des Wertebereichs zugeordnet. Praktisch heißt dies, dass eine Funktion bei gleichem Argumentwert immer den gleichen Ergebniswert liefern muss. Eine Funktion kann damit weder ein Gedächtnis haben, noch darf sie Seiteneffekte verursachen. Der einzige Mechanismus, der durch eine Funktion zur Verfügung gestellt wird, ist die Anwendung der Funktion auf Argumente, für die die Funktion ein eindeutiges Resultat liefert. Man spricht dann von *rein-funktionaler* Programmierung.

Eine rein-funktionale Programmierung beruht daher ausschließlich auf Funktionsdefinition und Funktionsanwendung. Die so vertrauten Konzepte der veränderlichen Variablen und Wertzuweisungen gibt es nicht, tatsächlich gibt es überhaupt keine veränderlichen Daten. Funktionale Programmierung unterscheidet sich damit grundsätzlich von imperativer Programmierung, die immer mit Zuweisungen von Werten an Variablen und Verändern eines Programmzustands arbeitet.

Funktionale Programme haben im Vergleich zu Programmen mit Seiteneffekten eine Reihe von günstigen Eigenschaften. Eine Funktion ist in sich abgeschlossen, weil sie nur von den Argumenten abhängt. Sie kann daher als eine Einheit angewendet, verstanden, getestet und verifiziert werden. Eine Funktionsanwendung steht ausschließlich für den Wert, den sie berechnet, und sie kann folglich zu jedem Zeitpunkt durch seinen Wert ersetzt werden. Das macht funktionale Programme unabhängig von einem Kontrollfluss und Funktionsanwendungen können in beliebiger Reihenfolge, parallel oder nach Bedarf erfolgen.

Aber ist dieses Modell, mit Funktionen mit Rückgabewerten und mit Funktionsanwendungen zu arbeiten, nicht viel zu restriktiv? Ist funktionale Programmierung damit im Vergleich zur imperativen Programmierung weniger mächtig und flexibel? Wie im wegweisenden Artikel von John Hughes mit dem Titel »Why functional programming matters« [32] eindrucksvoll argumentiert wird, liegt die Mächtigkeit der funktionalen Programmierung in der besseren Modularität und Kombinierbarkeit von Funktionen. Dadurch, dass Funktionen in sich abgeschlossen und nicht von einer Umgebung abhängig sind, können Funktionen frei kombiniert werden.

Besonders bemerkenswert ist auch, dass gerade John Backus, der mit der Entwicklung von Fortran [7] und mit seiner Mitarbeit bei der Definition von Algol 60 [6] ganz wesentlich zur Entwicklung der imperativen Programmiersprachen beigetragen hat, zu den Schwächen der imperativen Programmierung und zur Mächtigkeit der funktionalen Programmierung sehr früh grundlegende Einsichten gab. In seiner Ansprache zur Verleihung des Turing Awards hat er sich mit sehr drastischen Worten gegen die imperative Programmierung gewandt und eine funktionale Programmierung propagiert. Aus der begleitenden Publikation mit dem vielsagenden Titel »Can Programming Be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs« [8] ist dazu folgende Aussage entnommen:

> *Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor–the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.*

Backus sieht also die Schwäche der imperativen Programmierung in der engen Anlehnung an das Verarbeitungsmodell des von Neumann-Rechners. Und er vermisst mächtige Kombinationsoperatoren zur Gestaltung von Programmen auf höherer Ebene. Der Artikel propagiert dann einen funktionalen Programmierstil mit Operatoren zur Kombination von funktionalen Bausteinen. Er schreibt dazu:

> *An alternative functional style of programming is founded on the use of combining forms for creating programs. [...] Associated with the functional style of programming is an algebra of programs whose variables range over programs and whose operations are combining forms.*

Backus hat damit früh den Weg der Entwicklung funktionaler Sprachen vorgezeichnet. Sehr anschaulich ist auch das Beispiel im Artikel, mit dem imperative Programmierung und funktionale Programmierung verglichen werden. Eine imperative Lösung des Skalarproduktes von Vektoren, folgend mit zwei Listen `a` und `b` mit Länge `n`, gestaltet sich in Java folgendermaßen:

```
int c = 0;
for (int i = 0; i < n; i++) {
  c = c + a.get(i) * b.get(i);
}
```

Die Lösung baut auf der Wertzuweisung auf. Das Ergebnis wird in der Variablen `c` akkumuliert. In jedem Schleifendurchlauf wird in einer Wertzuweisung ein

nächster Wert für `c` berechnet und gespeichert. Die Berechnung ist, wie Backus es kritisiert, *»word-at-a-time style«*.

Die im Artikel dann vorgeschlagene Lösung, übersetzt auf die heute übliche Form funktionaler Sprachen, sieht im Gegensatz dazu folgendermaßen aus:

```
map2((x, y) -> x * y).andThen(reduce((r, x) -> r + x))
```

Die Operationen beziehen sich nicht auf einzelne Elemente, sondern auf die Vektoren als Ganzes. Sie sind mit Verknüpfungsoperationen in der Form von Lambda-Ausdrücken parametrisiert. Mit `map2` werden die Elemente zweier Vektoren paarweise mit der angegebenen Verknüpfungsoperation verknüpft. Die Operation `reduce` verknüpft die Elemente eines Vektors zu einem einzelnen Wert. Mit `andThen` werden Funktionen in Serie geschaltet. Somit werden in obiger Definition zuerst die Elemente von zwei Vektoren multipliziert und schließlich alle Produkte addiert. Das entspricht exakt der mathematischen Definition des Skalarprodukts. Im Gegensatz zur obigen imperativen Lösung ist diese Definition deklarativ und auf hoher Ebene. Die Operationen beziehen sich nicht auf einzelne Elemente, sondern auf die Datenobjekte selbst. Die Operationen haben keine Seiteneffekte und sind rein-funktional.

In diesem Beispiel sehen wir auch bereits die wesentlichen Konzepte, die funktionale Programmierung kennzeichnen:

- *Aggregatsfunktionen*: Funktionen, die auf komplexen Datenobjekten als Ganzes operieren und diese ohne versteckte Seiteneffekte in neue Strukturen abbilden.
- *Funktionen höherer Ordnung*: Funktionen, die durch andere Funktionen parametrisiert sind.
- *Kombination von Funktionen*: Operatoren, die eine Verknüpfung von funktionalen Bausteinen zu größeren funktionalen Einheiten erlauben.

Zur Entwicklung funktionaler Programmiersprachen

Die erste funktionale Sprache Lisp wurde von John McCarthy als eine Implementierung des Lambda-Kalküls entwickelt und bereits im Jahre 1960 veröffentlicht [51]. Wenn man bedenkt, dass Fortran als erste imperative höhere Sprache von John Backus 1957 eingeführt wurde, ist funktionale Programmierung also nur rund drei Jahre jünger. Bemerkenswert ist auch, dass diese erste Version von Lisp bereits Lambda-Ausdrücke, Funktionen höherer Ordnung und Listenverarbeitung hatte, Dinge die fast 60 Jahre später wieder sehr aktuell sind.

Seit dieser frühen Zeit wurde die funktionale Programmierung stetig weiterentwickelt. Lisp hat sich über viele Versionen und Varianten in den 1980er- und Anfang der 1990er-Jahre als die Sprache der Künstlichen Intelligenz etabliert. Die Sprachvarianten CommonLisp [81] und Scheme [83] werden auch heute noch verwendet, und mit der Sprache Clojure [44] gibt es einen populären Nachfolger, der auf der Java VM läuft.

Lisp-Sprachen sind dynamisch typisiert, das heißt, Werte tragen ihren Typ, und die Typprüfung wird zur Laufzeit durchgeführt. Robin Milner hat mit der Einführung der Sprache ML den Grundstein für statisch typisierte funktionale Sprachen gelegt [53]. Bei dynamisch-typisierten Sprachen können Funktionen mit beliebigen Datentypen arbeiten. Milner erkannte, dass man bei statischer Typisierung eine vergleichbare Flexibilität mit Typparametern erreichen kann. Gleichzeitig wurde ein Typinferenz-Algorithmus zur statischen Typprüfung eingeführt. In Standard ML, einer Weiterentwicklung von ML, wurden basierend auf der Sprache HOPE [12] polymorphe algebraische Datentypen verwendet, die auch heute noch die Basis für die Typsysteme funktionaler Sprachen bilden.

Ein weiterer wichtiger Entwicklungsschritt der funktionalen Programmierung war beginnend mit den frühen 1980er-Jahren die Erforschung von Sprachen mit nicht-strikten Auswertungsverfahren [85][86]. Bei strikten Auswertungsverfahren, wie wir diese von Java und anderen Programmiersprachen kennen, werden die Ausdrücke für die Parameter vor einem Prozeduraufruf evaluiert und die Werte übergeben. Bei einer nicht-strikten Auswertung [67] können die Ausdrücke vorerst nichtevaluiert übergeben werden, um schließlich im Kontext der aufgerufenen Funktion evaluiert zu werden. Dies führte zur Einführung von nicht-strikten funktionalen Programmiersprachen, wie zum Beispiel Miranda [87], und zur Entwicklung von effizienten Ausführungskonzepten [46][65].

Wie in [31] zu lesen ist, gab es um 1987 eine Vielzahl von Sprachentwicklungen mit nicht-strikten Auswertungsverfahren, die ganz ähnliche Ziele verfolgten. Eine Gruppe von Forschern beschloss daher, als Grundlage für Forschung und Entwicklung funktionaler Programmierkonzepte einen gemeinsamen Sprachentwurf zu schaffen. Damit wurde die Sprache Haskell geboren und der Sprachentwurf Haskell 1.0 im Jahre 1990 vom Haskell Committee veröffentlicht [30]. Haskell ist heute der State of the Art für funktionale Programmierung. Die funktionalen Programmierkonzepte, die wir in Java heute sehen, haben im Wesentlichen ihr Vorbild in Haskell. Um für dieses Buch einen entsprechenden Hintergrund zu schaffen, werden wir daher im folgenden Abschnitt die elementaren Konzepte und die wichtigsten Begriffe der funktionalen Programmierung, wie sie sich in Haskell darstellen, einführen.

1.1 Elementare Konzepte und Begriffe

Die wichtigsten Konzepte der funktionalen Programmierung sind: Lambda-Ausdrücke und Funktionsobjekte, polymorphe algebraische Datentypen mit Typinferenz, Pattern Matching sowie nicht-strikte Bedarfsauswertung. Diese Konzepte werden im Folgenden kurz eingeführt. Im Laufe des Buches werden wir uns intensiv mit der Umsetzung dieser Konzepte in Java beschäftigen.

Lambda-Ausdrücke und Funktionen höherer Ordnung

Lambda-Ausdrücke repräsentieren Funktionen mit formalen Argumenten und einem definierenden Ausdruck. Die beiden Seiten sind üblicherweise mit einem Pfeilsymbol[1] getrennt. Der definierende Ausdruck legt den Funktionswert fest und hat keine Seiteneffekte. Folgendes Beispiel zeigt einen Lambda-Ausdruck mit Argument `x` und definierendem Ausdruck `x + 1`:

```
x -> x + 1
```

Lambda-Ausdrücke sind in Programmiersprachen Literale für die Erzeugung von *Funktionsobjekten*. Funktionsobjekte können wie andere Werte an Variablen zugewiesen, als Parameter übergeben und Ergebnis von Funktionen sein. Im Sprachgebrauch der funktionalen Programmierung sagt man, Funktionsobjekte sind *first class*. Mit diesem einfachen Prinzip lassen sich Funktionen mit Funktionen als Parameter bilden, sogenannte *Funktionen höherer Ordnung*[2]. Ebenso kann man Funktionen schreiben, die aus einfacheren Funktionen komplexere Funktionen erzeugen. Zum Beispiel kann man eine Funktion `andThen` definieren, die aus zwei Funktionen `f` und `g` eine Funktion erzeugt, die zuerst auf das Argument `x` die Funktion `f` und auf das Ergebnis die Funktion `g` anwendet:

```
andThen f g = x -> g (f x)
```

Bei den Variablen, die im definierenden Ausdruck eines Lambda-Ausdrucks vorkommen, unterscheidet man *gebundene* und *freie* Variablen. Gebundene Variablen sind Argumente des Lambda-Ausdrucks, freie Variablen sind nicht Argument des Lambda-Ausdrucks und müssen damit im Kontext des Lambda-Ausdrucks gebunden sein. Im folgenden Lambda-Ausdruck ist `x` gebunden und `y` frei:

```
x -> x + y
```

Freie Variablen sind für die Bildung von Funktionsobjekten kritisch und wir werden sie im Abschnitt 2.3.4 über Closures noch ausführlich behandeln.

Algebraische Datentypen

Das Typsystem der statisch-typisierten funktionalen Sprachen beruht auf *Algebraischen Datentypen*. Algebraische Datentypen orientieren sich an Konzepten der Mengentheorie und erscheinen in einer abstrakteren Form als die Datentypen imperativer und objektorientierter Sprachen. Sie bauen auf Mengendefinitionen durch Enumeration und dem Kartesischen Produkt auf. Durch Kombination von Enumeration und Produkttypen ergeben sich sogenannte *Variantentypen* (engl.:

1. Im Lambda-Kalkül ist einem Lambda-Ausdruck der griechische Buchstabe λ vorangestellt und statt des Pfeils wird üblicherweise ein Punkt verwendet. Daher würde man den obigen Lambda-Ausdruck im Lambda-Kalkül als $\lambda\, x\, .\, x + 1$ schreiben.
2. Wir werden Funktionen höherer Ordnung auch einfach als *höhere Funktionen* bezeichnen.

v*ariant types* oder *union types*). Folgendes Beispiel eines Datentyps `Expr` zur Repräsentation von Booleschen Ausdrücken zeigt das Prinzip. Der Typ `Expr` definiert Varianten `Lit`, `Var`, `Not`, `And` und `Or` (die Varianten sind durch Oder-Striche getrennt). Jede Variante ist ein Produkttyp und besteht aus dem Namen der Variante, dem sogenannten *Datentag*, und den Typen für die Felder:

```
data Expr =
    Lit Bool      |
    Var String    |
    Not Expr      |
    And Expr Expr |
    Or Expr Expr
```

Die obige Datendefinition ist zudem rekursiv, da die Varianten `Not`, `And` und `Or` Felder vom Typ `Expr` haben. Auf diese Weise können hierarchische Strukturen gebildet werden.

Wir werden dieses Beispiel zur Repräsentation Boolescher Ausdrücke mehrmals in den Fallbeispielen zur funktionalen Programmierung in Java wieder aufgreifen und mit obiger Definition eines Algebraischen Datentyps vergleichen.

Parametrischer Polymorphismus

Typparameter dienen zur Definition von Funktionen und Datentypen, bei denen bestimmte Typen unbestimmt sind. Man spricht in der funktionalen Programmierwelt, von wo das Konzept auch ursprünglich stammt [53][13], von einem *parametrischen Polymorphismus*. Bei Java und anderen Sprachen ist das Konzept unter dem Begriff *Generizität* bekannt. Generizität ist für eine funktionale Programmierung von besonderer Bedeutung und wir werden daher bei den sprachlichen Grundlagen zur funktionalen Programmierung im nächsten Kapitel das Thema Generizität von Java detailliert behandeln.

Typinferenz

Funktionale Programmiersprachen verwenden *Typinferenz*. Darunter versteht man, dass der Typ eines Ausdrucks einer Variablen oder die Signatur einer Funktion automatisch abgeleitet wird und nicht vom Programmierer angegeben werden muss. Bei Lambda-Ausdrücken werden üblicherweise die Typen der Argumente und des Rückgabewertes nicht angegeben, sondern durch Typinferenz abgeleitet. Zum Beispiel sind bei unserem einfachen Lambda-Ausdruck

```
x -> x + y
```

keine Typen für den Parameter `x`, die freie Variable `y` oder den Rückgabewert angegeben. Darüber hinaus ist der Operator `+` überladen und somit die Typen der Operanden nicht eindeutig. In Java könnten die Typen für Argument und Ergebnis beliebige Zahlen als auch Zeichenketten sein.

Bei imperativen Sprachen wird üblicherweise nur für Ausdrücke der Typ durch den Compiler automatisch bestimmt, aber nicht für Variablen und Funktionsdefinitionen. Bei funktionalen Sprachen ist das aber auch für Variablen und Funktionsdefinitionen möglich. Der sogenannte Hindley-Milner-Algorithmus [53][27] ist in der Lage, für einen beliebigen Ausdruck und Funktionsdefinitionen den allgemeinsten Typ abzuleiten, der alle gegebenen Einschränkungen erfüllt.

Mit der Einführung von Lambda-Ausdrücken wurde die automatische Typinferenz in Java wesentlich erweitert und wir werden diese im nächsten Kapitel bei den Themen Generizität und Lambda-Ausdrücke diskutieren.

Funktionale Datenstrukturen

Eine wesentliche Eigenschaft der Algebraischen Datentypen ist, dass sie grundsätzlich *unveränderlich* sind. Auch Datenstrukturen wie Listen, Mengen oder Maps sind unveränderlich. Man spricht von *funktionalen* oder *persistenten Datenstrukturen*[3].

Die bekannteste funktionale Datenstruktur ist die funktionale Liste. In Haskell ist diese als Algebraischer Datentyp folgendermaßen definiert:

```
data [a] = []  |  a : [a]
```

Der Typbezeichner ist `[a]` mit dem Typparameter `a`. Die Variante `[]` bezeichnet die leere Liste. Die Variante `:` ist eine Liste mit dem ersten Element vom Typ a und der Restliste vom Typ [a] (der Datentag `:` ist dabei in Infix-Notation angegeben). Listen sind also rekursive Strukturen mit einem Element und einer Restliste, wobei am Ende immer die leere Liste steht. Man kann damit durch Voranstellen von Elementen aus bestehenden Listen neue Listen erzeugen, aber bestehende Listen nicht verändern. Für weitere persistente Datenstrukturen gibt es sehr effiziente Verfahren, die zum Beispiel auch einen direkten Zugriff ermöglichen. In [61] ist eine umfassende Behandlung dieses Themas zu finden.

In diesem Buch werden wir mit einer persistenten Listenimplementierung analog zum obigen Algebraischen Datentyp arbeiten. Wir führen diese in Abschnitt 3.3 ein, erweitern sie im Laufe der folgenden Kapitel und verwenden sie bei mehreren Fallbeispielen.

Pattern Matching

Eng verbunden mit den Algebraischen Datentypen ist *Pattern Matching*. Es handelt sich dabei um eine Kontrollstruktur, mit der auf Basis von Mustern die Varianten eines Datentyps unterschieden werden. Stimmt ein gegebener Wert mit einem Muster überein, ergibt sich das Ergebnis durch den dazugehörigen Ausdruck. Muster werden analog zu den Konstruktoren der Varianten gebildet. Mit

3. Wir werden die Bezeichnungen unveränderliche, persistente und funktionale Datenstruktur synonym verwenden.

dem Datentag erfolgt zuerst die Unterscheidung der Varianten. Für die Felder können Werte oder Variablen angegeben werden. Bei Werten erfolgt ein Vergleich auf Gleichheit, bei Variablen wird der Feldwert an die Variable gebunden.

Betrachten wir dazu wieder unseren Algebraischen Datentyp `Expr` von oben. Folgende Haskell-Funktion `mkString` verwendet Pattern Matching, um eine String-Repräsentation für einen `Expr`-Wert zu erstellen:

```
mkString e =
  case (e) of
    Lit True       -> "true"
    Lit False      -> "false"
    Var name       -> name
    Not sub        -> "(! " ++ mkString(sub) ++ ")"
    And left right -> "(" ++ mkString(left) ++ " && " ++
                      mkString(right) ++ ")"
    Or left right  -> "(" ++ mkString(left) ++ " || " ++
                      mkString(right) ++ ")"
```

Mit einem `case`-Ausdruck werden für einen `Expr`-Wert `e` die möglichen Varianten unterschieden. Man sieht, dass die Muster in den Fallunterscheidungen exakt den Varianten der Datentypdefinition entsprechen. Des Weiteren entsprechen die rekursiven Aufrufe der rekursiven Definition des Datentyps. Ein großer Vorteil von Pattern Matching ist, dass mit dem Mustervergleich nicht nur die Varianten unterschieden, sondern die Feldwerte den Pattern-Variablen zugewiesen werden. Das Datenelement wird sozusagen in die Bestandteile zerlegt. So werden zum Beispiel bei den Varianten `And` und `Or` die beiden Unterausdrücke an die Variablen `left` und `right` gebunden.

In Java gibt es Pattern Matching leider nicht. Wir werden aber sehen, dass man mit Lambda-Ausdrücken Strukturen ähnlich zu Pattern Matching schaffen kann.

Nicht-strikte Auswertung

Unter einer Evaluierungsstrategie versteht man die Reihenfolge, in der Ausdrücke ausgewertet werden. Aus der imperativen und objektorientierten Programmierung kennen wir ausschließlich die *strikte* Evaluierung. Es werden dabei die Ausdrücke für die Argumente von Funktionsanwendungen zuerst von links nach rechts vollständig evaluiert und dann die Funktion mit den konkreten Ergebniswerten aufgerufen. Dies entspricht einer *call-by-value* Parameterübergabe. Eine strikte Evaluierung ist bei Programmen mit Seiteneffekten auch unvermeidlich, da hier die Reihenfolge der Evaluierung der Ausdrücke und Anweisungen Einfluss auf das Ergebnis haben kann.

Arbeiten wir aber mit reinen Funktionen, hat die Reihenfolge der Funktionsanwendungen keinen Einfluss auf das Ergebnis. In der funktionalen Programmierung kann man daher auch *nicht-strikte* Evaluierungsstrategien einsetzen. Haskell ver-

wendet grundsätzlich eine nicht-strikte Evaluierungsstrategie, die sogenannte *Bedarfsauswertung* (engl.: *lazy evaluation*, *call-by-need*) [65]. Bei der Bedarfsauswertung wird ein Ausdruck erst und nur dann evaluiert, wenn das Ergebnis gebraucht wird.

Eine Bedarfsauswertung hat zum Beispiel Vorteile, wenn unendliche oder sehr große Strukturen verarbeitet werden. Das folgende Beispiel demonstriert das Arbeiten mit unendlichen Listen. In Haskell definiert der Ausdruck `[1000..]` die unendliche Liste der Integers ab dem Wert 1000. Mit dem Ausdruck

```
head (filter isPrime [1000..])
```

wird die erste Primzahl aus der Liste ermittelt. Dazu werden aber von der unendlichen Liste nur die Zahlen 1000 bis zur ersten Primzahl 1009 benötigt und somit auch nur diese generiert.

Java verwendet wie gesagt ausschließlich die strikte Evaluierung. Nicht-strikte Evaluierung kann man aber, wie wir in Abschnitt 4.6 zeigen werden, mit Funktionsparametern nachbilden und damit auch unendliche Strukturen schaffen. Und auch die funktionalen Streams in Java, die wir in Kapitel 7 behandeln, arbeiten nach dem Prinzip der nicht-strikten Bedarfsauswertung.

1.2 Funktionale Programmierung in Java

Die Erweiterungen von Java zur funktionalen Programmierung basieren auf den elementaren Konzepten, wie wir diese im letzten Abschnitt besprochen haben. Allerdings wird zurzeit nur ein Teil der Konzepte unterstützt. So gibt es in Java weder Algebraische Datentypen noch Pattern Matching. Auf der anderen Seite stehen die Konzepte zur funktionalen Programmierung nicht isoliert, sondern sind nahtlos in die objektorientierte Welt integriert. Funktionen werden durch Java-Objekte repräsentiert und können damit mit den Mitteln der objektorientierten Programmierung behandelt werden. So kann man Funktionen in herkömmlicher Weise mit objektorientierten Methoden erzeugen, aufbauen, umformen und kombinieren.

Allerdings soll auch hier angemerkt werden, dass Java natürlich keine rein-funktionale Sprache und daher ein Programmieren völlig ohne Seiteneffekte nicht sinnvoll ist. Eine solche Programmierung, bei der dann nur rekursive Funktionen möglich wären, wird in diesem Buch auch nicht verfolgt. Im Gegenteil wird man vor allem bei der internen Implementierung von funktionalen Operatoren und Strukturen auf imperative Techniken zurückgreifen, aber möglichst die funktionalen Eigenschaften nach außen bewahren. Auch die wichtigen Java-Bibliotheken, allen voran die Streams, arbeiten gezielt mit Seiteneffekten und veränderlichen Datenstrukturen. Allerdings werden die Seiteneffekte so beschränkt, dass sich Eigenschaften wie rein-funktionale Programme ergeben und damit zum Beispiel eine parallele Ausführung möglich wird.

hende Implementierungen des Interfaces nachziehen zu müssen. Das Interface `Collection` ist dazu ein gutes Beispiel. Zur Unterstützung der Streams musste man eine Möglichkeit schaffen, für eine Collection einen Stream zu erzeugen (siehe Kapitel 7). Man hat dazu beim Interface `Collection` zwei Default-Methoden `stream` und `parallelStream` eingeführt, die zur Collection einen einfachen und einen parallelen Stream erzeugen:

```
public interface Collection<E> extends Iterable<E> {
  ...
  default Stream<E> stream() { ... }
  default Stream<E> parallelStream() { ... }
}
```

Diese Methoden haben Implementierungen, die für alle Collections gleich funktionieren. Alle Collections können daher auf diese Default-Methoden zurückgreifen. Man beachte, dass man ohne die Default-Methoden alle Implementierungen von `Collection` um diese Methoden hätte erweitern müssen.

Neben Default-Implementierungen können Interfaces seit Version 8 auch statische Methoden haben, die im Wesentlichen analog zu den statischen Methoden von Klassen sind. Wir wollen hier nicht weiter auf diese eingehen, werden diese aber in den folgenden Kapiteln in mehreren Beispielen verwenden.

2.3 Lambda-Ausdrücke

Kommen wir nun zu den Lambda-Ausdrücken, die die wesentliche Grundlage der funktionalen Programmierung in Java sind. Wie wir in der Einleitung gehört haben, repräsentiert ein Lambda-Ausdruck eine unbenannte Funktion mit den formalen Argumenten und einem Funktionsrumpf für die Bestimmung des Funktionswerts. Lambda-Ausdrücke in Java folgen ebenso diesem Prinzip. Der in der Einleitung beispielhaft angegebene Lambda-Ausdruck wird in Java analog definiert:

```
x -> x + y
```

Was bewirkt ein solcher Lambda-Ausdruck in Java? In der objektorientierten Sprache Java erzeugt ein Lambda-Ausdruck ein Objekt eines *funktionalen Interfaces*. Funktionale Interfaces sind Interfaces, die *genau eine* abstrakte Methode deklarieren[2]. Ein Lambda-Ausdruck steht damit für die Implementierung dieser einen abstrakten Methode und dem Erzeugen eines Objektes des funktionalen Interfaces. Das funktionale Interface bestimmt also den Typ des Funktionsobjekts. Die Schnittstelle der einen abstrakten Methode bestimmt den sogenannten *Funktionstyp*, also die Abbildung der Argumente in den Funktionswert.

2. Solche Interfaces werden aufgrund dieser Eigenschaft auch als *Single Abstract Method* (SAM) Typen bezeichnet. Wir werden aber hier den Begriff *Funktionales Interface* verwenden.

Eine abstrakte Methode bei funktionalen Interfaces

Die Aussage, dass ein funktionales Interface nur eine abstrakte Methode haben kann, ist nicht ganz korrekt. Erstens sind alle öffentlichen Methoden von `Object` ausgenommen, die von jedem Interface implizit abstrakt deklariert werden, aber auch von einem Interface explizit deklariert werden können. Zum Beispiel ist ein Interface, das nur die Methode `equals` abstrakt deklariert, kein funktionales Interface:

```
interface NotAFunctionalInterface {
  boolean equals(Object o);
}
```

Ebenso kann ein Interface neben einer abstrakten Methode auch noch die Methoden von `Object` abstrakt deklarieren und ist trotzdem ein funktionales Interface. Dies ist zum Beispiel beim Interface `Comparator` der Fall:

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

Eine weitere besondere Situation ergibt sich, wenn ein Interface mehrere Interfaces erweitert und diese abstrakten Methoden mit gleichen oder zumindest vom Compiler als äquivalent erkannten Schnittstellen deklarieren. Der Compiler behandelt dann diese Methoden als ident. Die Situationen, die zu Compiler-äquivalenten Methoden führen, können aber bei der Verwendung von generischen Interfaces recht komplex sein. Details dazu findet man in der Java-Sprachspezifikation in [26], Abschnitt 9.8.

Betrachten wir zur Verwendung von Lambda-Ausdrücken ein Beispiel. Das bekannte Interface `ActionListener` definiert eine abstrakte Methode `actionPerformed`:

```
public interface ActionListener {
  public void actionPerformed(ActionEvent evt);
}
```

`ActionListener` ist also ein funktionales Interface und kann durch einen Lambda-Ausdruck implementiert werden:

```
ActionListener listener =
    evt -> System.out.println("Action performed");
```

Wie wir in der Zuweisung erkennen, muss der Lambda-Ausdruck ein Objekt vom Typ `ActionListener` erzeugen. Tatsächlich ist ein Lambda-Ausdruck ähnlich einer anonymen Klassendefinition. Obige Anweisung ist also ähnlich zu folgender Anweisung:

```
ActionListener listener = new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    System.out.println("Action performed");
  }
};
```

Das Argument `evt` im Lambda-Ausdruck stellt somit das Argument und der Rumpf im Lambda-Ausdruck den Rumpf der Methode `actionPerformed` dar. Die Methode `actionPerformed` kann nun für das erzeugte Objekt wie gewohnt aufgerufen werden:

```
listener.actionPerformed(new ActionEvent(...));
```

Damit ist vordergründig ein Lambda-Ausdruck nur eine kompakte Schreibweise für eine anonyme Klassendefinition, eingeschränkt auf funktionale Interfaces. Es gibt aber auch wesentliche Unterschiede, denen wir uns als Java-Programmierer auch bewusst sein sollten:

- Ein Lambda-Ausdruck führt keinen neuen Gültigkeitsbereich für Variablen ein, sondern verhält sich wie ein Block. Dies führt im Unterschied zu anonymen Klassen dazu, dass sich `this` und `super` nicht auf das vom Lambda-Ausdruck erzeugte Objekt beziehen, sondern auf das Objekt, in dessen Kontext die Lambda-Funktion erzeugt wurde. Des Weiteren ist es wie bei geschachtelten Blöcken in Lambda-Ausdrücken nicht zulässig, einen bereits verwendeten Variablennamen nochmals zu verwenden.
- Die Sprachspezifikation verlangt nicht, dass für jeden Lambda-Ausdruck ein neues Objekt erzeugt werden muss. Es ist im Gegensatz zulässig, dass ein Compiler aus Effizienzgründen für gleiche Lambda-Ausdrücke nur ein Objekt instanziiert.
- Lambda-Ausdrücke können keine Objektfelder deklarieren. Benötigt man Objektfelder bei der Implementierung eines funktionalen Interfaces, muss man auf innere Klassen zurückgreifen.
- Für Lambda-Ausdrücke wird kein explizites `class`-File erzeugt und der Aufruf erfolgt mit einer `invokedynamic`-Anweisung.

Aus der Sicht des Java-Programmierers können wir zusammenfassend folgende wesentliche Eigenschaften von Lambda-Ausdrücken festhalten:

- Lambda-Ausdrücke sind kompakte Implementierungen von funktionalen Interfaces.
- Lambda-Ausdrücke führen zu lesbarem Code.
- Lambda-Ausdrücke sind nahtlos in die objektorientierte Welt von Java integriert. Sie erzeugen Java-Objekte, die in gleicher Weise wie andere Objekte verwendet werden können.

2.3.1 Formen von Lambda-Ausdrücken

In Java gibt es unterschiedliche Formen, um Argumente und Funktionsrumpf von Lambda-Ausdrücken anzugeben. Man unterscheidet bei den Argumenten zwischen:

- *Explizit typisierte Argumente*: Die Typen der Argumente sind explizit angegeben.
- *Implizit typisierte Argumente*: Die Typen der Argumente sind nicht angegeben, sondern werden aus dem Kontext durch Typinferenz abgeleitet.

Beim Funktionsrumpf unterscheidet man:

- *Ausdrucksrumpf*: Der Rumpf besteht aus einem einzigen Ausdruck, der den Rückgabewert bestimmt.
- *Anweisungsrumpf*: Der Rumpf ist ein Block mit einer beliebigen Anweisungsfolge. Der Rückgabewert muss durch eine `return`-Anweisung bestimmt werden.

Zur Veranschaulichung sind im Folgenden ein Lambda-Ausdruck mit zwei Argumenten und der Berechnung der Summe der Argumente in unterschiedlichen Formen angegeben:

- Explizit typisiert, Anweisungsrumpf: `(int x, int y) -> {return x + y;}`
- Implizit typisiert, Anweisungsrumpf: `(x, y) -> {return x + y;}`
- Implizit typisiert, Ausdrucksrumpf: `(x, y) -> x + y`

Die kompakteste Schreibweise von Lambda-Ausdrücken ist damit die mit implizit typisierten Argumenten und Ausdrucksrumpf. Wenn möglich sollte man diese Form verwenden. Darüber hinaus kann man bei einem einzigen Argument auch noch die Klammern weglassen, wie wir das beim Beispiel mit dem Interface `ActionListener` gemacht haben. Natürlich muss man bei mehreren Anweisungen auf die Form Anweisungsrumpf zurückgreifen. Explizit typisierte Argumente benötigt man aber selten, zum Beispiel, wenn der Anwendungskontext nicht genügend Information enthält, dass die Typen der Argumente automatisch abgeleitet werden können. Dazu mehr im nächsten Abschnitt.

2.3.2 Typ eines Lambda-Ausdrucks

Wenn wir einen Lambda-Ausdruck wie den Ausdruck `x -> x + y` aus der Einleitung betrachten, stellt sich die Frage nach dem Typ des Ausdrucks. Wir wissen, dass der Lambda-Ausdruck ein funktionales Interface implementiert. Es ist aber durch den Lambda-Ausdruck alleine noch nicht bestimmt, welches Interface implementiert wird. Tatsächlich wird der Typ eines Lambda-Ausdrucks ausschließlich durch sein Ziel bestimmt, also durch den Typ der Variablen, des Parameters oder der Cast-Anweisung, für die der Lambda-Ausdruck verwendet wird. Man spricht in diesem Zusammenhang von *Target typing* [26].

Um diesen Prozess zu verstehen, insbesondere bei Verwendung von generischen Typen, greifen wir noch mal auf das Beispiel mit dem generischen Interface `Function` und der Methode `applyFunction` aus Abschnitt 2.1 zurück. Da `Function` ein funktionales Interface ist, können wir beim Aufruf der Methode `applyFunction` auch einen Lambda-Ausdruck verwenden:

```
applyFunction(s -> s.toString(), new Student(...));
```

Das Ziel des Lambda-Ausdrucks ist hier der erste Parameter von `applyFunction`, der den Typ `Function<? super A, ? extends B>` hat. Damit ist festgelegt, dass `Function<T, R>` mit der abstrakten Methode `R apply(T t)` das funktionale Interface darstellt, das durch den Lambda-Ausdruck implementiert wird.

Damit das Funktionsobjekt auch wirklich erzeugt werden kann, müssen noch für die generischen Typparameter die konkreten Typen bestimmt werden. Der Typ `Student` für den Typparameter `T` ergibt sich aus dem Typ des zweiten Parameters der Methode `applyFunction`. Und der Typ `String` für `R` ergibt sich aus dem Ausdruck `s.toString()` des Lambda-Ausdrucks. Somit ergibt sich als konkreter Typ des Funktionsobjekts `Function<Student, String>`.

Erst jetzt kann für den Lambda-Ausdruck ein entsprechendes Objekt erzeugt werden. Man beachte, dass alleine der Anwendungskontext den Typ bestimmt. Würde man obigen Lambda-Ausdruck in einem anderen Kontext mit einem anderen funktionalen Interface einsetzen, würde man ein ganz anderes Objekt erhalten.

2.3.3 Ausnahmen bei Lambda-Ausdrücken

Natürlich können auch in Lambda-Ausdrücken Ausnahmen auftreten. Wie diese behandelt werden müssen, ist analog zu den Methoden und damit davon abhängig, ob es sich um Systemausnahmen (*Runtime Exceptions*) oder Benutzerausnahmen (*Checked Exceptions*) handelt.

Systemausnahmen müssen auch in Lambda-Ausdrücken nicht behandelt werden. Wenn sie auftreten und nicht in einem try-catch-Block innerhalb des Lambda-Ausdrucks oder im Anwendungskontext gefangen werden, führen sie schließlich zu einem Programmabsturz. Auch bei den Benutzerausnahmen ist die Situation analog zu den Methoden: Entweder sie werden intern gefangen oder sie werden zum Rufer weitergeworfen, wobei das Werfen der Exception durch eine `throws`-Klausel bei der Methodensignatur angezeigt werden muss. Da die abstrakte Methode des Interfaces den Funktionstyp des Lambda-Ausdrucks bestimmt, muss diese eine entsprechende `throws`-Klausel definieren.

Nehmen wir dazu als Beispiel an, in einem Lambda-Ausdruck soll mit der Operation `Files.readAllLines` eine Textdatei gelesen werden:

```
path -> Files.readAllLines(path)
```

Die Operation `Files.readAllLines` kann aber eine `IOException` auslösen. Damit ist das Interface `Function` kein gültiger Typ für diesen Lambda-Ausdruck und man kann diesen zum Beispiel bei der Methode `applyFunction` nicht verwenden.

Man benötigt also ein funktionales Interface, bei der die abstrakte Methode das Werfen der Exception anzeigt. Eine generische Variante könnte dazu einen eigenen Typparameter `X` mit einer Typeinschränkung `extends Exception` verwenden und wie folgt aussehen:

```
interface FunctionWithExcpt<T, R, X extends Exception> {
  R apply(T t) throws X;
}
```

Eine adaptierte Methode `applyFunctionWithExcpt` würde man dann folgendermaßen definieren:

```
<T, R, X extends Exception>
R applyFunctionWithExcpt(FunctionWithExcpt<T, R, X> fn, T obj)
    throws X {
  return fn.apply(obj);
}
```

Hier wird ein Aufruf wie oben funktionieren:

```
List<String> lines = applyFunctionWithExcpt(
    path -> Files.readAllLines(path), Paths.get("text.txt"));
```

In den meisten Fällen wird man aber ein Werfen von Exceptions durch Lambda-Ausdrücke vermeiden. Eine sauberere und funktionale Lösung ist, die Behandlung der Exception im Lambda-Ausdruck vorzunehmen und die Ausnahmesituation über den Rückgabewert anzuzeigen. Dazu eignet sich ein Optional als Rückgabewert:

```
Optional<List<String>> optionalLines =
  applyFunction(
    path -> {
      try {
        return Optional.of(Files.readAllLines(path));
      } catch (IOException e) {
        return Optional.empty();
      }
    },
    Paths.get("text.txt")
  );
```

Der Typ `Optional`, der auch mit der Version 8 eingeführt wurde, erlaubt, das Vorhandensein oder Nicht-Vorhandensein eines Wertes anzuzeigen. Er kapselt das tatsächliche Ergebnis, kann aber im Fall einer Ausnahme leer (`Optional.empty()`) sein. Bei der obigen Lösung wird somit die Exception im Lambda-Ausdruck abgefangen und über den Rückgabewert vom Typ `Optional` kenntlich gemacht. Diese Lösung ist funktional und man kann damit diesen auch in einem funktiona-

len Kontext einsetzen. Auf die funktionale Ausnahmebehandlung mit Optional werden wir in Abschnitt 3.2 und in weiteren Kapiteln in diesem Buch noch ganz besonders eingehen.

2.3.4 Closures

Ein Problem, das auch bei anonymen Klassen auftritt, ergibt sich, wenn Lambda-Ausdrücke auf lokale Variablen des Anwendungskontextes zugreifen. Erinnern wir uns an die Diskussion von Lambda-Ausdrücken aus der Einleitung: Die Variablen, die nicht durch die Argumente gebunden sind, aber im Lambda-Ausdruck verwendet werden, sind freie Variablen. Bei rein-funktionalen Sprachen machen freie Variablen auch keine Schwierigkeiten. Sind aber Seiteneffekte erlaubt, braucht man eine besondere Lösung, den sogenannten *Funktionsabschluss*, oder in Englisch die *Closure* [48].

Schauen wir uns zuerst das Problem genauer an, um dann die prinzipiellen Lösungsmöglichkeiten und die Lösungsvariante von Java zu besprechen. Gegeben sei eine Methode `shift`, die ein `Function`-Objekt von `Double` nach `Double` und einen Wert `delta` erhält und wieder ein `Function`-Objekt zurückgibt:

```
public static Function<Double, Double> shift(
           Function<Double, Double> func, double delta) {
  return x -> func.apply(x - delta);
}
```

Die Methode erzeugt und retourniert ein Funktionsobjekt, bei dem die Funktionswerte im Vergleich zur ursprünglichen Funktion um `delta` in der x-Achse verschoben sind. Man beachte, dass im Lambda-Ausdruck für den Rückgabewert die lokale Variable `delta` verwendet wird. Diese stellt also eine freie Variable im Lambda-Ausdruck dar.

Das Problem entsteht durch die dynamische Lebensdauer von lokalen Variablen. Wenden wir nämlich die Methode `shift` wie folgt an

```
Function<Double, Double> sqrShifted = shift(x -> x * x , 1.0);
```

erhalten wir ein `Function`-Objekt, das die Variable `delta` verwendet. Da aber `delta` ein lokaler Parameter von `shift` ist, wird `delta` nach dem Aufruf von `shift` vom Stack abgebaut. Bei einer Verwendung des erzeugten `Function`-Objekts wie

```
double sqr3 = sqrShifted.apply(3.0);
```

existiert die Variable `delta` somit nicht mehr.

Es gibt grundsätzlich zwei Lösungsmöglichkeiten für dieses Problem:

- *Variante 1*: Die freien lokalen Variablen in Lambda-Ausdrücken werden nicht nach dem Prinzip der dynamischen Lebensdauer behandelt, sondern bleiben über den Aufruf hinweg erhalten. Diese Variablen werden also gemeinsam mit dem Funktionsobjekt verwaltet. Hier erklärt sich auch der Begriff des

Funktionsabschlusses, der zum Ausdruck bringt, dass die Funktion gemeinsam mit ihren freien Variablen abgeschlossen wird. Diese Lösung wird zum Beispiel in C#, Scala oder Kotlin implementiert.

- *Variante 2*: Die freien Variablen werden im Lambda-Ausdruck durch ihre aktuellen Werte ersetzt. Das heißt, das erzeugte Funktionsobjekt besitzt keine freien Variablen mehr, sondern verwendet nur die bei der Erzeugung vorhandenen Werte der Variablen.

In Java wurde Variante 2 implementiert[3]. Dies hat aber zur Folge, dass man lokale Variablen, die in Lambda-Ausdrücken verwendet werden, nur einmal definieren und nicht wieder verändern darf. Schließlich müssen solche Variablen *final* sein. Allerdings kann man, im Gegensatz zur bisherigen Anforderung bei anonymen Klassen, dass man solche Variablen als `final` deklarieren muss, nun auf die Deklaration `final` verzichten. Die Variablen darf man aber nur einmal setzen und man sagt, sie müssen *effectively final* sein.

Ein Beispiel dafür ergibt sich beim Verwenden der neuen Default-Methode `forEach` beim Interface `Iterable`, mit der man über die Elemente iterieren und für jedes Element eine Lambda-Funktion aufrufen kann. Mit `forEach` könnte man zum Beispiel die Summe über eine Liste von Zahlen bilden:

```
int sum = 0;
intList.forEach(x -> sum = sum + x);   // Compilerfehler
```

Dieses Programm akzeptiert der Compiler aber nicht, weil `sum` eine lokale Variable ist und diese im Lambda-Ausdruck gesetzt wird.

Eine Lösungsmöglichkeit ist, statt des primitiven Typs `int` ein Objekt zu verwenden, das eine veränderliche Variable kapselt. Man legt ein solches Objekt an und speichert die Referenz zu diesem Objekt in einer lokalen Variablen. Man verändert damit nicht mehr die lokale Variable, sondern das Objektfeld. Dieses Vorgehen wird im Folgenden für die Summenbildung angewendet. Dabei wird eine Klasse `IntRef` mit dem int-Feld `value` verwendet, die in Listing 2–2 angegeben ist[4]:

```
IntRef sum = new IntRef(0);
intList.forEach(x ->  sum.value = sum.value + x);
```

Diese Umsetzung von Closures mit der Einschränkung auf unveränderliche lokale Variablen wird oft als Nachteil der Lambda-Ausdrücke von Java genannt. Allerdings sollte man bedenken, dass es Ziel der funktionalen Programmierung ist, möglichst ohne Seiteneffekte zu arbeiten. So ist obige Form der Summenbildung auch keine funktionale Lösung, sondern eine imperative Lösung, die eben einen Lambda-Ausdruck verwendet. In Abschnitt 4.1 werden wir sehen, wie eine funktionale Lösung einer Summenbildung aussieht, bei der dieses Problem von vornherein nicht auftritt.

3. Wobei die Werte in finalen Variablen innerhalb des Funktionsobjekts gespeichert werden.
4. Als Alternative könnte man auch ein Array mit nur einem Element verwenden. Der veränderliche Wert ist dann die eine Stelle im Array.

```
public class IntRef {
  public IntRef(int value) { this.value = value; }
  public int value;
}
```

Listing 2–2 *Klasse IntRef zur Bereitstellung eines veränderlichen Wertes am Heap. Analoge Klassen wird man für weitere Basisdatentypen und eine generische Klasse für Referenztypen bereitstellen.*

2.4 Funktionale Interfaces

Im vorherigen Abschnitt haben wir erfahren, dass jedes Interface mit nur einer abstrakten Methode mit einem Lambda-Ausdruck implementiert werden kann. Diese Eigenschaft eines Interfaces kann man auch durch die Annotation `@FunctionalInterface` festschreiben. Zum Beispiel wird man bei der Deklaration des Interfaces `Function` die Annotation voranstellen:

```
@FunctionalInterface
public interface Function<T, R> { ... }
```

Die Annotation ist zwar nicht zwingend vorgeschrieben, der Compiler garantiert aber nun, dass das Interface wirklich nur eine abstrakte Methode hat.

Im Package `java.util.function` werden eine Reihe von Interfaces definiert, die den herkömmlichen Verwendungsmustern von Lambda-Ausdrücken entsprechen und durchgehend in der Java-Bibliothek verwendet werden. So werden wir sehen, dass diese bei den Streams, aber auch bei anderen verwandten Strukturen zum Einsatz kommen.

Es gibt im Package sowohl generische Interfaces als auch eine Vielzahl von Interfaces für die Basisdatentypen. Tabelle 2–1 gibt einen Überblick über die generischen funktionalen Interfaces zusammen mit ihren abstrakten Methoden. Es werden die elementaren Interfaces mit einem oder keinem Parameter, die binären Funktionen mit zwei Parametern und die Operatoren unterschieden. Die Gestaltung dieser Interfaces sieht man am besten, wenn man die elementaren Interfaces betrachtet:

- `Function<T, R>`: Dieses Interface entspricht der Abbildung eines Wertes vom Typ `T` auf einen Wert vom Typ `R`, definiert eine Method `apply` und ist somit analog zu dem Interface `Function`, das wir oben verwendet haben.
- `Predicate<T>`: Ist eine Testfunktion, die für einen Wert eine Eigenschaft testet. Die abstrakte Methode `test` hat einen Parameter vom Typ `T` und retourniert einen Booleschen Wert.
- `Consumer<T>`: Verarbeitet ein Element. Die Methode `accept` hat einen Parameter vomTyp `T` und gibt nichts zurück.
- `Supplier<T>`: Dient dazu, Elemente zu erzeugen. Die Methode `get` hat keinen Parameter und liefert ein Element vom Typ `T`.

7 Streams

Streams sind neben den Lambda-Ausdrücken die wichtigste Neuerung in Java zur funktionalen Programmierung. Streams unterstützen das Verarbeiten einer Folge von Elementen, sowohl sequentiell als auch parallel. In diesem Kapitel werden wir uns auf die sequentielle Verarbeitung konzentrieren, um uns im nächsten Kapitel dann intensiv mit der parallelen Verarbeitung auseinanderzusetzen.

7.1 Grundlagen von Streams

Streams sind in Java Objekte, die einen Zugriff auf eine Folge von Elementen bereitstellen. Sie haben damit eine gewisse Verwandtschaft zu den Collections. Streams verhalten sich aber in wichtigen Aspekten ganz unterschiedlich zu Collections. Die besonderen Eigenschaften von Streams lassen sich gut erkennen, wenn man Streams und Collections gegenüberstellt. Tabelle 7–1 vergleicht dazu Collections und Streams:

Collection	Stream
Speichern Elemente	Liefern Elemente
Anfügen, Löschen und Zugriff	Nur Zugriff
Mehrfache Iteration	Nur einmalige Iteration
Externe Iteration	Interne Iteration
Verarbeitung strikt im eifrigen Modus	Verarbeitung nicht-strikt nach Bedarf

Tab. 7–1 *Unterschied Collections und Streams*

- Im Gegensatz zu Collections speichern Streams keine Elemente, sondern stellen nur einen Zugriff auf Elemente bereit. Die Quelle der Elemente wird vom Stream abstrahiert. Ist die Quelle der Elemente eine Collection, wird auf die Elemente der Collection zugegriffen. Die Quelle kann aber zum Beispiel auch ein externes Medium oder ein Mechanismus zum Erzeugen der Elemente sein.
- Mit Streams kann man nur auf die Elemente zugreifen. Ein Anfügen und Löschen und ganz allgemein ein Verändern der Menge der Elemente ist bei Streams nicht möglich.

- Mit Streams kann man nur einmal über die Elemente iterieren. Will man die Elemente nochmals verarbeiten, muss ein neues Stream-Objekt erzeugt werden.
- Streams arbeiten mit einer internen Iteration. Das bedeutet, dass die Iteration über die Elemente nicht im Anwenderprogramm implementiert wird, sondern intern im Stream erfolgt.
- Streams arbeiten nach dem Prinzip der Bedarfsauswertung, bei dem Elemente erst dann geliefert werden, wenn diese angefordert werden.

Die beiden letzten Eigenschaften, interne Iteration und Bedarfsauswertung, sind für Streams besonders wichtig. Wir werden daher diese Eigenschaften noch genauer analysieren.

7.1.1 Ein erstes Beispiel

Schauen wir uns aber zuerst ein erstes Beispiel an. In folgender Anweisung wird mit Stream-Operationen eine Liste von Artikeln verarbeitet. Es werden zuerst jene Artikel herausgefiltert, die einen Preis kleiner als 1000 haben, diese nach dem Preis sortiert, dann auf ihren Namen abgebildet und schließlich daraus wieder eine Liste erzeugt. Die Ergebnisliste enthält somit die Namen der Artikel mit Preisen unter 1000:

```
List<Article> articles =
    List.of(new Article("A", 1500), new Article("B", 700), ...);
List<String> cheapArticleNames =
    articles.stream()
      .filter(a -> a.price < 1000.0)
      .sorted(Comparator.comparingDouble((Article a) -> a.price))
      .map(a -> a.name)
      .collect(Collectors.toList());
```

Wir sehen, dass Streams dem Gestaltungsprinzip von Monaden folgen, wie wir sie in Abschnitt 6.3 besprochen haben. `Stream` ist dabei die monade Behälterklasse. Zuerst wird mit der Operation `stream` ein `Stream`-Objekt erzeugt. Dieses wird jeweils mit den höheren Funktionen `map` und `filter` in weitere Streams abgebildet, bis schließlich mit einer Reduktionsoperation, hier als `collect` ausgeführt, ein Ergebnis erzeugt wird.

7.1.2 Externe vs. interne Iteration

Betrachten wir nun den Unterschied zwischen einer externen Iteration bei einer imperativen Lösung und einer internen Iteration bei Streams. Wir verwenden zur Veranschaulichung das obige Beispiel, aber diesmal ohne die Sortieroperation:

```
List<String> cheapArticleNames =
    articles.stream()
      .filter(a -> a.price < 1000.0)
      .map(a -> a.name)
      .collect(Collectors.toList());
```

Eine imperative Lösung der gleichen Aufgabe würde so aussehen:

```
List<String> cheapArticleNames = new ArrayList<String>();
for (Article a : articles) {
  if (a.price < 1000.0) {
    cheapArticleNames.add(a.name);
  }
}
```

Bei der imperativen Variante ist die Iteration über die Elemente im Anwenderprogramm in der Form der for-Schleife implementiert. Die Elemente werden im Schleifenrumpf verarbeitet. Bei der funktionalen Variante ist die Schleife im Stream verborgen. Die Verarbeitungsschritte werden außerhalb der Schleife definiert.

Bei der funktionalen Variante werden die einzelnen Verarbeitungsschritte in deklarativer Weise in der Form von Aggregatsfunktionen angegeben und beziehen sich somit auf den Stream als Ganzes. Es sind damit wesentliche Vorteile verbunden:

- Das Programm enthält keine Seiteneffekte.
- Soll ein neuer Schritt eingefügt werden, kann man das einfach durch Anfügen eines weiteren Schrittes erreichen (siehe unser erstes Beispiel aus Abschnitt 7.1).
- Die Iteration ist intern und kann damit für den Stream spezifisch optimiert werden. Auch kann man, wenn grundsätzlich die Anwendung für eine parallele Verarbeitung geeignet ist, einfach von einer sequentiellen Verarbeitung zu einer parallelen Verarbeitung übergehen. Wir werden das in Kapitel 8 sehen.

Im Gegensatz dazu hat die externe Iteration den Vorteil, dass man im Anwenderprogramm die volle Kontrolle über die Iteration hat. Diese muss man naturgemäß bei der internen Iteration aufgeben.

7.1.3 Bedarfsauswertung

Eine ganz wesentliche Eigenschaft von Streams ist, dass die Zugriffe auf die Elemente nach Bedarf erfolgen. Eine Auswertung nach Bedarf haben wir bereits in Abschnitt 4.6 besprochen. Streams werden nach dem gleichen Prinzip verarbeitet.

Wir wollen im Folgenden die Verarbeitung bei Streams mit einer eifrigen Auswertung bei Listen vergleichen. In Abschnitt 4.1 haben wir höhere Funktionen für die funktionale Liste `FList` eingeführt. Mit `FList` kann man das Filtern der Artikel und die Abbildung auf den Namen somit ganz ähnlich implementieren:

```
FList<Article> articles =
    FList.of(new Article("A", 1500), new Article("B", 700), ...);
FList<String> cheapArticleNames =
    articles
        .filter(a -> a.price < 1000.0)
        .map(a -> a.name);
```

Abbildung 7–1 veranschaulicht die Verarbeitung für eine Beispielliste mit mehreren Artikeln. Mit der `filter`-Operation wird zuerst eine Liste mit den Artikeln mit einem Preis kleiner 1000 und mit `map` eine Liste mit den Namen dieser Artikel erzeugt.

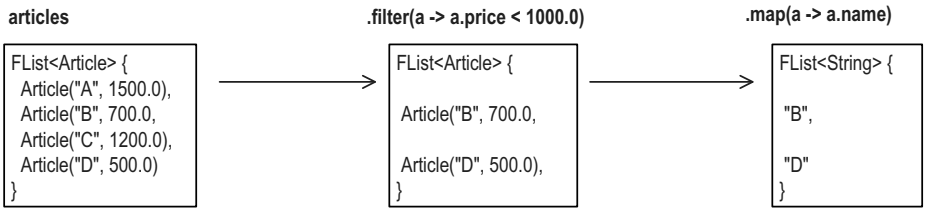

Abb. 7–1 *Strikte Verarbeitung der Liste von Artikeln bei FList*

Die Verarbeitung mit Streams funktioniert aber wie in Abbildung 7–2 gezeigt. Die Streams sind vorerst passiv und warten untätig, bis jemand auf die Elemente zugreift. Die Zugriffe gehen somit von der `collect`-Operation aus, die die Ergebnisliste aufbauen will und sich dafür die Elemente holt. Damit werden die Streams aktiv, liefern nacheinander die benötigten Elemente und wenden auf diese die Verarbeitungsschritte an. Dabei werden keine Zwischenlisten erzeugt. Bei sehr vielen Elementen kann das ein entscheidender Vorteil sein.

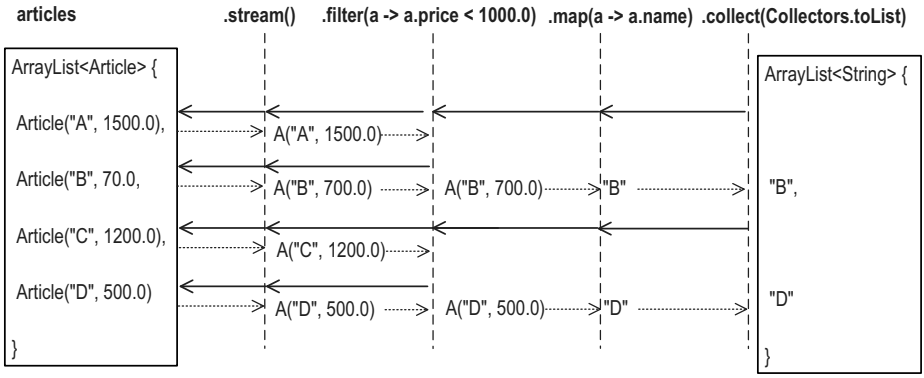

Abb. 7–2 *Faule Verarbeitung der Liste von Artikeln mit Streams*

Der wirkliche Vorteil wird aber erst deutlich, wenn wir unser Beispiel etwas ändern und nicht die Liste der günstigen Artikel bilden, sondern nur den ersten Artikel mit einem Preis kleiner 1000 benötigen. Das können wir für `FList` durch einen Zugriff auf das erste Element erreichen (dass die Liste leer sein könnte, wollen wir hier einmal ignorieren):

```
String cheapArticleName =
    articles.filter(a -> a.price < 1000.0)
            .map(a -> a.name)
            .head();
```

Es ergibt sich ein Verhalten wie in Abbildung 7–3. Es wird zuerst mit der `filter`-Operation eine Liste mit den Artikeln mit Preis kleiner 1000 gebildet, dann mit `map` eine Liste der Namen, und von dieser Liste wird das erste Element zurückgegeben. Vor allem bei langen Listen ist dies viel Aufwand, um ein einziges Element zu ermitteln.

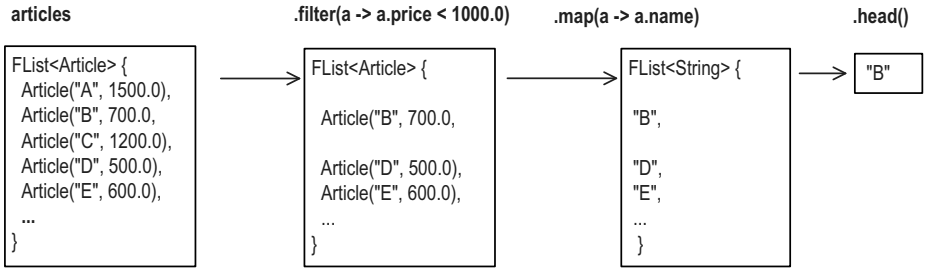

Abb. 7–3 *Zugriff auf das erste günstige Element bei* `FList`

Bei Streams gibt es eine Operation `findFirst`, die das erste Element eines Streams in einem Optional liefert. Damit ergibt sich für den Zugriff auf den ersten Artikel folgende Stream-Anweisung:

```
Optional<String> cheapArticleName =
    articles.stream()
            .filter(a -> a.price < 1000.0)
            .map(a -> a.name)
            .findFirst();
```

Abbildung 7–4 zeigt den wesentlichen Unterschied, der sich durch die Bedarfsauswertung ergibt. Die Operation wird hier von `findFirst` angestoßen, die genau ein Element benötigt. Der erste Zugriff wird durch `filter` noch weggefiltert, das zweite Element wird aber durch `filter` durchgereicht. Das von `findFirst` benötigte Element wird geliefert und die Berechnung terminiert. Statt alle Elemente zu filtern und abzubilden, ist insgesamt nur auf zwei Elemente zugegriffen worden.

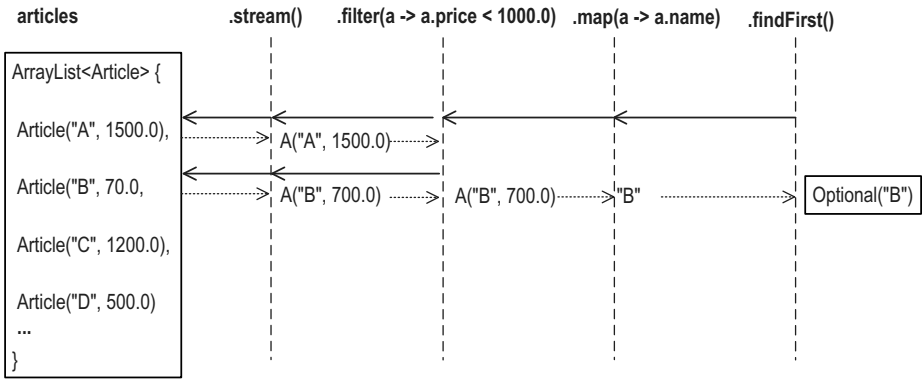

Abb. 7–4 *Faule Verarbeitung bei Streams bei Zugriff auf das erste teure Element*