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`.