Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 141

Untersuchung der Erweiterung von Java 8 um Lambda

Timo Freiberg

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Erhard Plödereder
Betreuer/in:	Timm Felden
Beginn am:	20. Mai 2014
Beendet am:	21. November 2014
CR-Nummer:	D.1.1, D.1.5, D.3.3

Kurzfassung

Durch die Verbreitung von mehrkernigen Prozessoren wird es wichtiger, Programme zu schreiben, die parallel ausgeführt werden können. Ein funktionaler Programmierstil kann paralleles Programmieren erleichtern. Ein wichtiges Stilmittel funktionaler Sprachen sind Lambdas (anonyme Funktionen) welche z.B. in den mit Java vergleichbaren Programmiersprachen C# und Scala zur Verfügung stehen. Mit Version 8 wurden in Java Lambdas eingeführt, wodurch Java einen funktionalen Programmierstil stärker unterstützt als zuvor.

In dieser Arbeit wird untersucht, wie gut funktionale Programmiermuster in Java 8 umgesetzt werden können. Anhand von Codebeispielen werden Fälle dargestellt, in denen es sinnvoll ist, in Java einen funktionalen Programmierstil anzuwenden. Schließlich werden Faustregeln für die Benutzung von Lambdas und funktionalen Programmiermustern vorgeschlagen.

Es gibt viele Fälle, in denen ein funktionaler Programmierstil kürzer, lesbarer und weniger fehleranfällig ist als ein traditioneller imperativer Programmierstil. Außerdem kann ein funktionaler Programmierstil es stark erleichtern, parallel ausführbaren Code zu schreiben. Javas Typsystem kann den Einsatz von Lambdas jedoch erschweren. Die Änderungen in Java 8 ermöglichen einen oft besseren Programmierstil und machen die Sprache angenehmer zu benutzen. Dabei wurden die Änderungen so eingebaut, dass sie für Java-Programmierer leicht verständlich sind und eine minimale Umgewöhnung benötigen.

Inhaltsverzeichnis

1. Einleitung

Dieses Kapitel gibt eine Übersicht über Inhalt und Struktur dieses Dokuments. Außerdem wird die Zielgruppe dieser Arbeit sowie das nötige technische Vorwissen beschrieben.

1.1. Inhaltliche Zusammenfassung

In dieser Arbeit werden Lambdafunktionen sowie weitere neue Features in Java 8 vorgestellt. Außerdem wird untersucht, ob mit diesen Änderungen ein funktionaler Programmierstil möglich ist, was die Vor- und Nachteile eines solchen Programmierstiles sind und in welchen Anwendungsfällen er sinnvoll ist. Diese Punkte werden mit Codebeispielen illustriert. Danach werden Richtlinien für die Benutzung der neuen Sprachfeatures und eines funktionalen Programmierstiles in Java vorgeschlagen. Schließlich wird bewertet, inwiefern es möglich ist, in Java 8 funktional zu programmieren und wie einfach der Sprung von Java 7 auf Java 8 für einen Programmierer ist.

1.2. Zielgruppe und nötiges Vorwissen

Diese Arbeit richtet sich primär an Java-Programmierer, die wenig Erfahrung mit funktionalen Programmiersprachen haben. Erfahrung mit der Syntax von Java, einem traditionellen imperativen Programmierstil in Java sowie englischen Fachbegriffen wird vorausgesetzt. Vorwissen über Stilmittel und Aspekte funktionaler Programmierung wird nicht vorausgesetzt.

1.3. Struktur

In Kapitel 2 werden neue Features in Java 8, die einen funktionalen Programmierstil begünstigen, vorgestellt. Kapitel 3 führt typische Aspekte und Muster funktionaler Programmierung ein und stellt sie anhand von kurzen Codebeispielen vor. Ausführlichere Fallbeispiele werden in Kapitel 4 vorgestellt, in denen imperativer Code mit funktionalem Code, der die neuen Features von Java 8 nutzt, verglichen wird. In Kapitel 5 werden Richtlinien für den Einsatz der neuen Sprachfeatures sowie eines funktionalen Programmierstils vorgeschlagen. Schließlich werden in Kapitel 6 die Änderungen in Java 8 sowie der Programmierstil, der durch sie gefördert wird, zusammengefasst und bewertet.

2. Neue Features in Java 8

In diesem Kapitel werden die Änderungen in Java 8 vorgestellt, die einen funktionalen Programmierstil fördern. Syntax und typische Verwendung werden beschrieben und anhand von einfachen Beispielen dargestellt. Ausführlichere Beispiele, anhand welcher Vor- und Nachteile eines funktionalen Programmierstils in Java 8 illustriert werden, werden in Kapitel 4 vorgestellt.

2.1. Lambda-Ausdrücke

In Java gab es schon vor Java 8 eine Möglichkeit, Logik als Parameter an eine Methode zu übergeben, nämlich mittels einer anonymen inneren Klasse (aiK). Ein typischer Empfänger ist z.B. die Methode Collections.sort. Diese Methode erwartet ein Comparator-Objekt als Parameter. Solche Objekte werden häufig direkt am Verwendungsort als aiK initialisiert. Dabei ist nur die Logik in ihrer einzelnen Methode für den Programmierer von Interesse. Eine typische Benutzung wird in Abbildung 2.1 dargestellt.

Abbildung 2.1: Sortierung einer Liste von Personen nach Alter

```
1 Collections.sort(persons, new Comparator<Person>() {
2     @Override
3     public int compare(Person p1, Person p2) {
4         return p1.getAge() - p2.getAge();
5     }
6 });
```

Die Logik benötigt nur eine Zeile, die gesamte Deklaration der aiK aber fünf Zeilen. Außerdem kann jeder Teil der aiK, bis auf die Logik in Zeile 4, automatisch erkannt werden:

Die Signatur der sort-Methode bestimmt, dass ein Comparator-Objekt mit demselben Typ-Parameter `E` wie die Collection benötigt wird. Das Comparator-Interface bestimmt, dass eine compare-Methode mit zwei Parametern vom Typ `E`, die einen `int`-Wert zurückgibt, definiert werden muss. [Ora14a, Ora14e]

Lambda-Ausdrücke ermöglichen es, die selbe Methode in nur einer Zeile zu schreiben. Die Signatur des Lambdas muss identisch mit der Signatur der Methode sein, die an dieser Stelle mittels einer aiK definiert werden würde. In diesem Fall muss die Signatur der compare-Methode übereinstimmen, damit der Compiler das Lambda akzeptiert.

Abbildung 2.2: Kürzung von Abbildung 2.1 mit einem Lambda

```
Collections.sort(persons, (p1, p2) -> p1.getAge() - p2.getAge());
```

2.1.1. Syntax

Ein Lambda besteht aus einer Parametermenge (die leer sein darf), einem Pfeil (`->`) sowie einem Körper, der entweder ein Ausdruck oder ein Block sein muss.

Abbildung 2.3: Identitätsfunktion als Lambda

```
(int x) -> {return x;}
```

Abbildung 2.3 zeigt die Identitätsfunktion von `int`-Werten auf `int`-Werte. Dies ist die ausführlichste Art, diese Funktion als Lambda zu schreiben. Es ist möglich, einzelne Ausdrücke im Körper des Lambdas ohne geschweifte Klammern und ohne `return` zu schreiben, siehe Abbildung 2.4.

Abbildung 2.4: Identitätsfunktion mit minimiertem Körper

```
(int x) -> x
```

Auch die linke Seite des Pfeils lässt sich verkürzen, da die explizite Typangabe in Lambdas optional ist. Wenn nur noch ein einzelner Parameter ohne Typangabe links vom Pfeil steht, sind die Klammern um den Parameternamen optional. Abbildung 2.5 zeigt die kleinste Version der Identitätsfunktion.

Abbildung 2.5: Minimale Identitätsfunktion

```
x -> x
```

Da in Abbildung 2.5 der Typ nicht explizit festgehalten ist, könnte dieses Lambda auch an eine Methode übergeben werden, die eine Funktion benötigt, die z.B. von String nach String abbildet. Dies wird in Abbildung 2.6 dargestellt.

Die kompletten syntaktischen Regeln von Lambdas sind im offiziellen Java-Tutorial zu finden.[Ora14j]

2.1.2. Einsatz von Lambdas

In Java 8 wurden keine Funktionstypen eingeführt. Lambdas werden daher in Instanzen von Functional Interfaces umgewandelt. Diese Interfaces zeichnen sich dadurch aus, dass sie genau eine abstrakte Methode haben. Dabei werden abstrakte Methoden, die eine der Methoden von java.lang.Object überschreiben, nicht gezählt. [Ora14d]

Vor dieser Umwandlung sind Lambdas anonyme Funktionen, weshalb man das selbe Lambda in Objekte mit verschiedenem Typ umwandeln kann. Ein Beispiel dafür wird in Abbildung 2.6 dargestellt.

Abbildung 2.6: Das selbe Lambda wird in Variablen von verschiedenem Typ gespeichert

```
IntUnaryOperator f1          = x -> x + 1;
Function<Integer,Integer> f2 = x -> x + 1;

Comparator<Integer> g1 = (x,y) -> 1;
IntBinaryOperator g2   = (x,y) -> 1;

int i = 1;
Supplier<Integer> h1 = () -> i;
IntSupplier h2       = () -> i;

Function<String,String> id1   = x -> x;
Function<Integer,Integer> id2 = x -> x;
```

Damit eine Methode ein Lambda als Parameter empfangen kann, muss sie ein Functional Interface als Parameter erwarten. In Java 8 wurde das function-Paket eingeführt, das Funktionen mit keinem, einem oder zwei Parametern bereitstellt. [Ora14h]

In Abbildung 2.7 wird eine Methode vorgestellt, die ein Function-Objekt akzeptiert und die enthaltene Funktion mit dem Eingabewert `0` ausführt. Diese Methode könnte z.B. die in Abbildung 2.5 vorgestellte Identitätsfunktion empfangen. An diesem Beispiel ist außerdem zu erkennen, dass die durch das Lambda definierte Funktion durch die abstrakte Methode des Functional Interface implementiert wird. Diese Methode heisst in diesem Fall Function.apply.

Abbildung 2.7: Methode, die ein Lambda akzeptiert

```
int supplyZero(Function<Integer, Integer> f) {
    //f ist ein Objekt, das eine apply-Methode besitzt
    return f.apply(0);
}
supplyZero(x -> x); // ergibt 0
```

Da Lambdas nur die Methode eines Interfaces implementieren, können sie nicht auf eigene Klassenvariablen zugreifen, da sie keine besitzen. Genau wie aiK können sie aber auf Variablen in ihrem Umfeld zugreifen, die entweder final sind, oder niemals neu zugewiesen werden und dadurch ohne Kompilierfehler als final deklariert werden könnten. Letztere nennt man effective final Variablen, da sie wie final behandelt werden können. In diesem Zusammenhang muss man beachten, dass Lambdas immer Zugriff auf die Umgebungsvariablen von ihrem Deklarationsort haben, nicht etwa auf die Variablen am Ausführungsort. [New13]

Der einzige funktionale Unterschied zwischen Lambdas und aiK ist, dass sich `this` in einem Lambda auf die Klasse bezieht, in der das Lambda definiert ist; in einer aiK hingegen auf die aiK selbst.

2.1.3. Hintergrund

Lambdas können in Java 8 nur als Objekte evaluiert werden. Die Entscheidung gegen die Einführung von Funktionstypen wurde wie folgt begründet. Erstens würden dadurch strukturelle Typen mit nominalen Typen gemischt werden. Zweitens würde ein großer Unterschied zwischen Bibliotheken, die Funktionstypen benutzen, und solchen, die Objekte benutzen entstehen. [Goe10] Außerdem würden generische Funktionstypen durch Javas *Type Erasure* weniger nützlich sein. [Goe11a]

Die Syntax wurde gewählt, da sie der Syntax von Lambdas in C# und Scala ähnlich sieht, welche die Java am ähnlichsten Sprachen sind.[Goe11b] Es wurde speziell der Pfeil mit Bindestrich gewählt, da ein Pfeil mit Gleichheitszeichen visuell schwer von anderen Symbolen zu unterscheiden ist, die das Gleichheitszeichen beinhalten. [Ora12a, Goe11c]

2.2. Method References

Viele Lambdas bestehen nur aus einem Aufruf einer Methode. Java 8 bietet mit Method References eine alternative Schreibweise für solche Lambdas, die oft kürzer ist. Zwei äquivalente Methodenaufrufe mit einem Lambda und einer Method Reference werden in Abbildung 2.8 dargestellt.

Abbildung 2.8: Beispiel für eine Method Reference

```
run((MyClass param) -> MyClass.doSomething(param));
run(MyClass::doSomething);
```

Ähnlich wie Lambdas eine alternative Schreibweise für eine anonyme innere Klasse sind, ist eine Method Reference eine alternative Schreibweise für ein Lambda. Eine Method Reference wird also nur akzeptiert, wenn ein äquivalentes Lambda auch korrekt wäre.

2.2.1. Syntax

Eine Method Reference besteht immer aus drei Teilen: Dem Namen der Klasse bzw. des Objektes, welches die Methode beinhaltet, gefolgt von zwei Doppelpunkten, gefolgt vom Namen der Methode. [Goe12b] Es gibt vier verschiedene Typen von Method References, welche in den folgenden Codebeispielen vorgestellt werden. Alle Method References haben gemeinsam, dass ein äquivalentes Lambda aus nur einem Methodenaufruf besteht und die Parameter des Lambdas in der selben Reihenfolge an diese Methode weitergegeben werden.[Ora14g]

Typ 1 Der erste Typ dargestellt in Abbildung 2.9, ist die Referenz zu einer statischen Methode, die die Parameter des Lambdas erhält. In diesem Fall muss die Method Reference den Namen der enthaltenden Klasse sowie den Namen der statischen Methode enthalten.

Abbildung 2.9: Referenz einer statischen Methode

```
run((MyClass param) -> MyClass.staticMethod(param));
run(MyClass::staticMethod);
```

Typ 2 Der zweite Typ, dargestellt in Abbildung 2.10, ist eine Referenz zu einer Methode eines spezifischen Objektes im Scope. Diese Method Reference unterscheidet sich vom ersten Typ darin, dass der Name eines Objektes statt dem Namen einer Klasse benutzt wird.

Abbildung 2.10: Referenz einer Instanzmethode eines spezifischen Objektes

```
MyClass obj = new MyClass();
run((MyClass param) -> obj.instanceMethod(param));
run(obj::instanceMethod);

run((MyClass param) -> this.instanceMethod(param));
run(this::instanceMethod);
```

Typ 3 Der dritte Typ, dargestellt in Abbildung 2.11, spezifiziert nicht explizit den Ort der aufgerufenen Methode. Stattdessen wird eine Instanzmethode des ersten Parameters des Lambdas aufgerufen. Instanzmethoden von anderen Parametern des Lambdas (oder sonstigen Objekten) können nicht aufgerufen werden, was in der Dokumentation nicht eindeutig spezifiziert ist, und daher in Abschnitt A.5 getestet wird.[Ora14g]

Die restlichen Parameter des Lambdas, falls vorhanden, werden in gleicher Reihenfolge an die aufgerufene Methode übergeben. Da der erste Parameter des Lambdas in diesem Fall keinen Namen zugewiesen bekommt, muss diese Art von Referenz, wie der erste Typ, den Typ des Parameters spezifizieren. Dadurch ist es möglich, dass eine Methodenreferenz sowohl Typ 1 und Typ 3 auf einmal ist, also zwei verschiedene Methoden referenziert werden könnten. In diesem Fall wird ein Compilerfehler erzeugt, da Method References eindeutig sein müssen.

Diese Referenz kann nur eine Instanzmethode des ersten Parameters des Lambdas aufrufen, was nicht explizit spezifiziert wird.

Abbildung 2.11: Referenz einer Instanzmethode des ersten Parameters

```
run((MyClass param) -> param.doSomething());
run(MyClass::doSomething);
Collections.sort(list, (String a, String b) -> a.compareTo(b));
Collections.sort(list, String::compareTo);
```

Typ 4 Der letzte Typ, dargestellt in Abbildung 2.12, referenziert eine Konstruktormethode. Auch hier werden die Parameter des Lambdas in gleicher Reihenfolge an den Konstruktor weitergegeben. Eine Konstruktorreferenz besteht aus dem Klassennamen und dem Methodennamen `new`.

Abbildung 2.12: Referenz eines Konstruktors

```
run((MyClass param) -> new MyClass(param));
run(MyClass::new);
```

2.3. Streams

Streams in Java 8 sind Sequenzen von Objekten (oder Primitivwerten), die Mengenoperationen höherer Ordnung unterstützen. Streams erlauben es, Operationen auf jedes Element des Streams auszuführen, ohne je direkt einzelne Elemente zu manipulieren.

Abbildung 2.13: Eine Pipeline von Stream-Methoden

```
1 Stream.iterate(1, x -> x+1)
2     .map(x -> x*x)
3     .filter(x -> x % 2 == 0)
4     .limit(15)
5     .reduce(0, (a, b) -> a+b);
```

Abbildung 2.13 zeigt ein Beispiel für die Benutzung eines Streams. In Zeile 1 wird ein unendlicher Stream von Zahlen generiert. Das erste Element ist 1 und jedes folgende Element ist um 1 größer als das vorherige. Dieser Stream beinhaltet also alle natürlichen Zahlen. In Zeile 2 wird die durch das Lambda beschriebene Funktion auf jedes Element des ursprünglichen Streams angewandt. Aus den Ergebnissen wird ein neuer Stream generiert, der alle Quadratzahlen enthält. In Zeile 3 werden alle Elemente entfernt, die von der Lambda-Funktion auf `false` abgebildet werden. Der zurückgegebene Stream enthält also nur noch alle geraden Quadratzahlen. In Zeile 4 wird der Stream auf eine feste Zahl von Elementen begrenzt, in diesem Fall auf 15 Elemente. Dadurch kann in Zeile 5 die `reduce`-Methode benutzt werden. Diese Methode kombiniert jedes Element mit einem Akkumulatorwert (siehe Abbildung 3.10). In diesem Fall wird jedes Element nacheinander auf eine Summe aufaddiert.

Alle diese Methoden können Lambdas als Parameter empfangen, da sie die Methode, die durch das Lambda definiert wurde, auf ihre Elemente anwenden. Diese Methoden werden ausführlich in der Java-Dokumentation erklärt. [Ora14f]

Ein wichtiger Aspekt von Streams ist, dass ihre Methoden *lazy* evaluiert werden (siehe Abschnitt 3.4). Das bedeutet, dass der Aufruf in Abbildung 2.13, Zeile 2 noch kein Element verarbeitet, bis in Zeile 5 alle Elemente aufaddiert werden. Ein Vorteil davon ist, dass es möglich ist, einen unendlichen Stream zu generieren und Operationen auf die Elemente zu deklarieren, solange am Ende eine begrenzte Zahl von Elementen verarbeitet werden. [Urm14a]

2.3.1. Verwendung

Streams können mittels der neuen `Collection.stream`-Methode aus Collections generiert werden, der Stream beinhaltet dann die selben Objekte wie die Collection. Außerdem gibt es statische Methoden zur Generation von Streams, wie z.B. `Stream.iterate`, welche in Abbildung 2.13, Zeile 1 benutzt wurde.

Streams besitzen zwei Arten von Methoden, die auf ihre Elemente zugreifen. Die erste Gruppe sind die *intermediären* Methoden, welche aus einem Stream einen neuen Stream konstruieren. Da sie Instanzmethoden des Stream-Interfaces sind und einen Stream erzeugen, können sie, wie in Abbildung 2.13 sichtbar, direkt miteinander verkettet werden. Diese Methoden, unter denen sich

z.B. `map`, `filter` und `limit` befinden, werden erst ausgeführt, wenn eine *terminierende* Methode ausgeführt wird, da Streams lazy evaluiert werden. Diese zweite Gruppe von Methoden, unter denen sich neben `reduce` auch `toArray` und `count` befinden, erzeugt zwingend ein Ergebnis oder einen Nebeneffekt.

Streams können leicht parallelisiert werden, indem in der Methodenpipeline die `parallel`-Methode eingefügt wird. Es kann sogar zwischen sequentieller und paralleler Berechnung gewechselt werden, indem `sequential` und `parallel` abgewechselt werden. Parallelisierung von Stream-Operationen kann die Leistung verbessern, wenn große Datenmengen verarbeitet werden. [Ora12a, Ora14i]

2.3.2. Funktionsweise

Streams werden durch terminierende Operationen verbraucht, sie sind also nur zur einmaligen Benutzung gedacht. Da Streams lazy evaluiert werden, können alle intermediären Operationen vorgemerkt werden, und dann für die terminierende Operation in einem Durchgang durchgeführt werden.

2.3.3. Hintergrund

Streams werden lazy evaluiert, um optimierbar zu sein. Außerdem sind somit unendliche Streams möglich. [Ora14i, Urm14a] Es wurden keine Gegenstücke der Stream-Operationen für Collections eingeführt, da Collection-Methoden die eigene Instanz mutieren und *eager* statt lazy sind. Diese Eigenschaften sind für Mengenoperationen höherer Ordnung nicht sinnvoll. [Goe12c]

2.4. Optional

Optional ist eine Containerklasse, deren Zweck ist, null-Werte zu ersetzen. Da der Zugriff auf Instanzmethoden von null-Werten einen Fehler erzeugt, muss dieser Fall bei Parameterwerten oder Rückgabewerten häufig manuell überprüft werden Dies kommt besonders häufig vor, wenn null als Rückgabewert für das Fehlen eines Wertes definiert wurde. Fehler bei dieser manuellen Fallunterscheidung können nicht vom Typsystem erkannt werden. Eine konsequente Nutzung von Optional kann häufige null-Abfragen ersetzen. [Fus12]

2.4.1. Verwendung

Ein Optional-Wert kann entweder einen Wert enthalten, oder Empty sein. Optional.Empty ist ein Singleton, der anstelle von null verwendet wird. Jedes Optional-Objekt, das nicht Empty ist, enthält einen Wert, der nicht null ist. Optional bietet, ähnlich wie Stream, Methoden wie map, die Operationen auf den beinhalteten Wert ausführen, ohne diesen Wert manuell zu entpacken. Optional ist keine Spracherweiterung, daher können Optional-Werte selber null sein.

Abbildung 2.14 zeigt einige Optional-Methoden in einem trivialen Beispiel. Alle Methoden von Optional werden im Detail in der Java-Dokumentation erklärt. [Ora14c, Urm14b]

Abbildung 2.14: Beispiel für Optional-Methoden

```
1 Optional.of("test")
2     .map(s -> s.substring(1, 2))
3     .flatMap(s -> Optional.of(5))
4     .filter(i -> i < 3)
5     .orElse(0);
```

In diesem Beispiel wird zuerst der String “test” in ein Optional-Objekt gepackt. Danach wird ein Optional erzeugt, dass das Ergebnis der Anwendung des Lambdas auf den String beinhaltet, also den String “e”. Anschließend wird ein neues Lambda auf den Inhalt angewandt, das den Eingabeparameter ignoriert und ein Optional mit dem Wert 5 zurückgibt. Die darauffolgende filter-Methode gibt Optional.Empty zurück, falls das Lambda den Wert auf false abbildet. Da dies der Fall ist, ist der Wert nach Zeile 4 Empty, und die orElse-Methode gibt den Alternativwert 0 zurück.

Optional ist ein praktisches Werkzeug im Umgang mit Methoden, die null zurückgeben können. Dies wird in Beispielen in Abschnitt 4.2 illustriert.

2.5. CompletableFuture und weitere Monaden

Schon vor Java 8 gab es das Future-Interface, das asynchron berechnete Werte beinhaltet. Mit Futures ist es möglich, Werte parallel zu berechnen und erst wenn sie fertig berechnet wurden auszulesen. CompletableFuture erweitert diese Funktionalität um Methoden, die den Umgang erleichtern. Ein triviales Beispiel für ein Future, das nach einer Verzögerung einen Rückgabewert erzeugt, wird in Abbildung 2.15 dargestellt.

Abbildung 2.15: Triviales Future

```
Future<Integer> future = CompletableFuture.supplyAsync(() -> 1);
future.get(); //1
```

In diesem Fall ist der asynchron berechnete Wert eine Konstante. Für Werte deren Berechnung lange dauert oder blockieren kann, kann es sehr nützlich sein, die Berechnung an ein Future zu übergeben. Alle Methoden werden in der Java-Dokumentation im Detail erklärt. [Ora14b]

2.5.1. Gemeinsamkeiten zwischen CompletableFuture und Optional

CompletableFuture ermöglicht es, Operationen auf den beinhalteten Wert anzuwenden oder den beinhalteten Wert mit dem Wert eines weiteren CompletableFuture mit der thenCombine-Methode zu kombinieren. [Ora14b] Diese Methoden erlauben einen ähnlichen Umgang mit dem beinhalteten Wert, den auch Optional erlaubt hat. Die CompletableFuture.thenApply-Methode ist vergleichbar mit der Optional.map-Methode, die thenCompose-Methode mit der flatMap-Methode, etc.

CompletableFuture und Optional haben folgende Eigenschaften gemeinsam: Sie enthalten Werte eines spezifischen Typs. Es gibt Methoden zur Erstellung eines Objektes, das einen Wert beinhaltet, wie Optional.of und CompletableFuture.supplyAsync. Schließlich gibt es Methoden um Operationen auf den beinhalteten Wert anzuwenden, wie flatMap und thenCompose. Dadurch sind diese beiden Klassen *Monaden.* [Sau14a, RGU14]

2.5.2. Monaden

Monaden sind Container, die mit einer gewissen Logik ausgestattet sind. [She14, Iry07] Sie erlauben es, den enthaltenen Wert zu manipulieren, wobei die Monadenspezifische Logik eingehalten wird, was zu einer Kombination von Funktionalitäten führt.

Zum Beispiel ist Optional eine Monade, deren Logik mit null-Werten zusammenhängt. Interaktion mit dem enthaltenen Wert über map etc. bekommt den zusätzlichen Kontext, dass null-Werte dazu führen, dass der gespeicherte Wert durch Optional.Empty ersetzt wird. Stream kann mehrere Werte beinhalten, map etc. operieren daher auf allen Werten. CompletableFuture lagert die Berechnung des Wertes automatisch in einen separaten Thread aus und merkt sich sein map-Äquivalent vor bis der enthaltene Wert berechnet ist.

Eine weitere Monade, die in Abschnitt 4.3 vorgestellt wird, ist die Try-Monade. Sie evaluiert Ausdrücke, die Exceptions werfen können und speichert entweder den Ergebniswert in einer Success-Instanz oder die Exception in einer Failure-Instanz. Sie ist vergleichbar mit Optional, mit der Ausnahme, dass der Fehlerfall im Gegensatz zu Optional.Empty Informationen enthält.

Monaden sind also auf eine spezielle Funktion spezialisiert. Sie sind komponierbar und begünstigen einen funktionalen Programmierstil durch die Benutzung von map-Methoden und Funktionspipelines.

3. Funktionale Programmiermuster

In diesem Abschnitt werden einige typische Aspekte und Muster von funktionaler Programmierung erklärt und mit simplen Codebeispielen illustriert. Zusätzlich werden Vor- und Nachteile bei der Anwendung der Muster aufgezeigt, die in Java 8 umsetzbar sind.

Diese Programmiermuster wurden bei der Einarbeitung in das Thema in einem Testprojekt angewandt, das in Abschnitt A.3 dargestellt wird.

3.1. Pure Funktionen

Pure Funktionen erfüllen zwei Eigenschaften. Sie haben keine sichtbaren Nebeneffekte, verändern also keine Zustände des Programms und interagieren nicht auf sichtbare Art und Weise mit der Umgebung. Außerdem geben bei gleichen Eingabewerten immer den gleichen Wert zurück. [Mic14, Ale14]

Abbildung 3.1 stellt zwei Methoden vor, die beide eine Variable um 1 inkrementieren. Die erste Methode inkrementiert eine Klassenvariable, und hat damit einen Seiteneffekt, die zweite Methode hingegen ist eine pure Funktion.

Abbildung 3.1: Inkrementierende Methoden mit und ohne Nebeneffekte

```
class SideEffectExample {
    int counter = 1;

    void incrementWithSideEffects() {
        counter++;
    }
    int incrementPure(int x) {
        return x++;
    }
}
```

Vorteile Pure Funktionen können den Zustand des Programmes nicht verändern. Ein Verzicht auf Nebeneffekte reduziert das Risiko, Fehler auszulösen, die mit dem Zustand des Programmes zusammenhängen, welche oft schwer zu diagnostizieren sind. Da das Verhalten von puren Funktionen nicht vom Rest des Programmes abhängt, sind sie oft leicht zu verstehen.

Nachteile Nebeneffekte sind oft notwendig oder schwer zu vermeiden. Nebeneffektfrei zu programmieren kann die Laufzeit negativ beeinflussen.

3.2. Vermeidung von Mutierbarkeit

Mutierbarkeit bezeichnet die Möglichkeit, den Wert von Variablen während der Laufzeit zu ändern. Ein Objekt ist mutierbar, wenn mindestens eine der Werte dieses Objektes mutierbar ist. Eine Variable in Java ist nicht mutierbar, wenn sie als final deklariert ist und, wenn sie ein Objekt ist, alle ihre Attribute nicht mutierbar sind.

Nicht mutierbare Objekte, die einen Zustand haben, erzeugen üblicherweise ein neues Objekt mit modifiziertem Zustand, anstatt den eigenen Zustand zu ändern.

Abbildung 3.2 zeigt eine Klasse, die ihren Zustand verändern kann.

Abbildung 3.2: Beispiel für mutierbare Klassen

```
class MutableCounter {
    final int value = 0;

    void increment() {
        value++;
    }

    int getValue() {
        return value;
    }
}
```

In Abbildung 3.3 wird eine ähnliche Klasse vorgestellt, die nicht mutierbar ist. Sie erzeugt für jeden neuen Zustand ein neues Objekt.

Vorteile Ein Verzicht auf Mutierbarkeit erlaubt es, die selben Objekte an mehreren Orten im Programm zu verwenden, ohne den Zugriff auf diese Objekte zu kontrollieren oder synchronisieren. Dies gilt besonders in nebenläufigen Programmen. [Sha14]

Nachteile Bei häufiger Änderung des Zustandes werden bei der Nutzung von nicht mutierbaren Objekten häufig neue Objekte erstellt. Dies kann sich negativ auf das Laufzeitverhalten auswirken.

Abbildung 3.3: Beispiel für nicht mutierbare Klassen

```
class ImmutableCounter {
    final int value;

    ImmutableCounter() {value = 0;}

    ImmutableCounter(int value) {
        this.value = value;
    }

    ImmutableCounter increment() {
        return new ImmutableCounter(value++);
    }

    int getValue() {
        return value;
    }
}
```

3.3. Rekursion

Rekursive Methoden können sich selber aufrufen. Dadurch kann die Implementierung z.B. einer mathematischen Definition ähnlich sehen.

Zwei klassische Beispiele für rekursive Methoden werden in Abbildung 3.4 vorgestellt.

Abbildung 3.4: Beispiele für rekursive Methoden

```
class RecursivePureFunctions {
    int fibonacciNumber(int n) {
        if (n <= 1) return n;
        return fibonacciNumber(n - 1) + fibonacciNumber(n - 2);
    }

    int factorial(int n) {
        if (n == 0) return 1;
        return n*factorial(n - 1);
    }
}
```

Vorteile Rekursive Methoden können kürzer und leichter verständlich sein als äquivalente imperative Methoden.

Nachteile Java 8 führt keine Tail-Call-Optimization durch. [Sch09, Ora12b] Da die zitierten Dokumente älter sind als Java 8, wurde in Abschnitt A.4 ein Test durchgeführt, der das Fehlen dieser Optimierungstechnik nahelegt. Daher kann tiefe Rekursion zu einem StackOverflow-Fehler führen. Außerdem ist wiederholter Funktionsaufruf weniger effizient als weitere Schleifendurchläufe, wodurch Rekursion meistens zu schlechterer Leistung führt.

3.4. Lazy Evaluation

Als lazy (deutsch: faul) evaluiert bezeichnet man Ausdrücke, die erst berechnet werden, wenn ihr Wert benötigt wird. Im Gegensatz dazu wird bei eager (deutsch: eifrig) Evaluation jeder Ausdruck sofort verarbeitet.

In Abbildung 3.5 werden lazy sowie eager evaluierte Berechnungen vorgestellt.

Abbildung 3.5: Lazy und eager evaluierte Summen

```
1  class SumLazyOrEager {
2      int x = 1;
3      int y = 1;
4
5      int eagerResult = x + y;
6
7      int lazyResult() {
8          return x + y;
9      }
10 }
```

Die Summe in Zeile 5 wird bei Instanziierung des Objektes berechnet, auch wenn der Wert nie benötigt wird, dafür wird der Wert nur einmal ausgerechnet, und kann dann mehrmals ausgelesen werden. Die Methode ab Zeile 7 berechnet die Summe erst, wenn sie das erste mal aufgerufen wird. Da sie das Ergebnis nicht zwischenspeichert, wird es aber bei jeder Abfrage erneut berechnet.

Abbildung 3.6 zeigt einen lazy evaluierten Wert, der nach einmaliger Berechnung zwischengespeichert wird. Dadurch wird die lange Berechnung nur einmal ausgeführt, aber erst wenn der Wert benötigt wird.

Abbildung 3.6: Lazy evaluierter Wert wird memoisiert

```
class LazyMemoization {
    Value val;
    Value getValue() {
        if (val == null) {
            val = calculateValue();
        }
        return val;
    }
}
```

Vorteile Lazy Evaluation ermöglicht es zum Beispiel, mit unendlichen Sequenzen zu arbeiten, so lange nicht alle Elemente benötigt werden. Lazy Evaluation kann das Laufzeitverhalten verbessern, wenn dadurch Berechnungen eingespart werden können.

Nachteile Wenn eine Datenstruktur lazy evaluiert wird, aber schlussendlich alle Elemente verarbeitet werden, also keine Berechnung gespart wurde, wird die Leistung im Vergleich zu eager Evaluation verschlechtert. Der Grund dafür ist, dass lazy evaluierte Ausdrücke in *Thunks* gespeichert werden. Thunks sind Funktionen ohne Parameter, die einen Ausdruck evaluieren und zurückgeben. [Par03, Jan12] Ihre Berechnung benötigt zusätzlichen Aufwand, der die Leistung beeinträchtigt. [Has10]

3.5. Funktionen höherer Ordnung

Funktionen höherer Ordnung (HOF, Higher Order Function) zeichnen sich dadurch aus, dass sie eine oder mehrere Funktionen als Parameter erhalten und/oder eine Funktion ausgeben. Wichtige Beispiele sind HOF, die eine Funktion auf alle Elemente einer Menge anwenden.

Streams bieten solche Funktionen an und ermöglichen es somit, den Stream zu bearbeiten, ohne jemals direkten Zugriff auf die Elemente zu haben. HOF in Java 8 zeichnen sich dadurch aus, dass sie ein Functional Interface als Parameter empfangen, da es in Java 8 keine Funktionstypen gibt.

Abbildung 3.7 zeigt eine einfache HOF, die ein Lambda empfängt und es mehrfach ausführt.

Abbildung 3.7: Eine Funktion, die ein übergebenes Lambda mehrfach ausführt

```
<T> void repeat(Supplier<T> f, int times) {
    for (int i = 0; i < times; i++) {
        f.get();
    }
}
```

Diese Methode führt das in `f` beinhaltete Lambda mehrmals aus.

Abbildung 3.8 zeigt eine Implementierung der Map-Funktion für Listen.

Abbildung 3.8: Die Map-Funktion auf Listen

```
<T,U> List<U> map(List<T> list, Function<T,U> f) {
    List<U> mappedList = new ArrayList<>();
    for (T element : list) {
        mappedList.add(f.apply(element));
    }
    return mappedList;
}
```

Map empfängt eine Liste sowie eine Funktion `f`, die Elemente dieser Liste empfängt. Sie wendet `f` auf alle Elemente der Liste an und erzeugt eine Liste der Rückgabewerte.

Eine weitere wichtige Funktion für den Umgang mit Listen ist die Filter-Funktion, die in Abbildung 3.9 implementiert wird.

Abbildung 3.9: Die Filter-Funktion auf Listen

```
<T> List<T> filter(List<T> list, Predicate<T> pred) {
    List<T> filteredList = new ArrayList<>();
    for (T element : list) {
        if (pred.test(element)) {
            filteredList.add(element);
        }
    }
    return filteredList;
}
```

Filter empfängt eine Liste sowie ein Prädikat, eine Funktion, die einen Wert auf einen boolean abbildet. Sie gibt eine Liste zurück, die nur noch die Elemente enthält, für die das Prädikat true zurückgibt.

Zuletzt wird in Abbildung 3.10 die Fold-Funktion implementiert.

Abbildung 3.10: Die Fold-Funktion auf Listen

```
<T, U> U fold(List<T> list, BiFunction<U, T, U> f, U id) {
    U result = id;
    for (T element : list) {
        result = f.apply(result, element);
    }
    return result;
}
```

Fold erzeugt aus einer Liste einen Wert, der aus allen Elementen der Liste zusammengesetzt wird. Dafür wird neben der Liste eine Funktion mit zwei Parametern benötigt, die Elemente der Liste sowie Werte vom Typ des Ergebniswertes annimmt und auf den selben Typ abbildet. Außerdem wird ein Initialwert benötigt, welcher zusammen mit dem ersten Element der Liste auf das erste Zwischenergebnis abgebildet wird. Jedes Zwischenergebnis wird nacheinander mit jedem Element der Liste kombiniert und erzeugt das nächste Zwischenergebnis, bis die Kombination mit dem letzten Element der Liste den Rückgabewert erzeugt.

Vorteile HOF erlauben es, Funktionalität als Parameter zu übergeben. Ihr Einsatz kann eine Bibliothek sehr vielseitig parameterisierbar machen. Sie ermöglichen es allgemein, eine weitere Abstraktionsebene zu erschließen. [Hav14]

Nachteile Der Einsatz von HOF an sich hat keine Nachteile. Da Logik gekapselt wird, die sonst explizit ausgeschrieben werden würde, besteht das Risiko, unleserlichen Code zu schreiben.

3.6. Funktionspipelines

Eine Funktionspipeline wird durch Verkettung von Funktionen gebildet. In solch einer Pipeline wird der Rückgabewert jeder Funktion in die nächste Funktion in der Pipeline eingegeben.

In funktionalen Sprachen wie Haskell können Funktionen direkt hintereinander geschrieben werden, wenn die Ein- und Ausgabetypen passen: Seien die Funktionen `f: A -> B, g: B -> C` gegeben, dann ist der Ausdruck `g(f(a))` legal und erzeugt einen Wert vom Typ `C`.

In Java können Methoden auch, durch einen Punkt getrennt, aneinander gekettet werden. Allerdings sind diese Methoden nicht statisch, sondern Instanzmethoden: Seien die Methoden `B A.f(), C B.g()` gegeben, dann ist der Ausdruck `a.f().g()` legal und erzeugt auch einen Wert vom Typ `C`.

Ein Beispiel für eine simple Pipeline wird in Abbildung 3.11 vorgestellt.

Abbildung 3.11: Eine simple Methodenpipeline

```
new Incrementor(0) //0
    .increment()   //1
    .increment()   //2
    .increment();  //3
```

Hier wird ein Objekt mit dem Wert 0 erstellt, anschließend wird dieser Wert drei mal inkrementiert. Die increment-Methode ist eine Instanzmethode der Klasse Incrementor, die in Abbildung 3.12 implementiert ist.

Abbildung 3.12: Eine Klasse, die eine verkettbare Methode besitzt

```
class Incrementor {
    int value;

    Incrementor(int value) {
        this.value = value;
    }

    Incrementor increment() {
        return new Incrementor(value++);
    }
}
```

Da sie ein neues Incrementor-Objekt zurückgibt, kann die gleiche Methode auf dem neuen Objekt aufgerufen werden, etc.

Vorteile Eine Pipeline ist ein einziger Ausdruck und kann daher von vorne nach hinten gelesen werden. Im Gegensatz dazu steht z.B. bei Schleifen der auszuführende Code hinter der Deklaration der Schleife.

Nachteile Eine lange Verkettung von Methoden kann schwer nachvollziehbar sein, vor allem wenn die Pipeline zwischen vielen Typen transformiert.

3.7. Continuation Passing Style

Continuation Passing Style bezeichnet die Praxis, Funktionen eine weitere Funktion zu übergeben, welche am Ende mit dem Rückgabewert der ersten Funktion aufgerufen wird. Dieser Stil erzwingt durch das Typsystem das weitere Behandeln des Rückgabewertes einer Funktion, ähnlich wie Optional des Behandeln eines möglichen Null-Wertes erzwingt.

Wenn beispielsweise eine Methode `g` mit dem Rückgabewert einer Methode `f` aufgerufen wird, kann man im Continuation Passing Style `g` als Parameter an `f` übergeben, wie in Abbildung 3.13 dargestellt.

Abbildung 3.13: Benutzung einer Try-Monade

```
//direct style
int fDirect(int value) {
    return value * 2;
}

//continuation passing style
int fCPS(int value, Function<Integer,Integer> func) {
    return func.apply(value * 2);
}

Function<Integer,Integer> g = x -> x*x;

//...
{
    g.apply(fDirect(1)); //direct

    fCPS(1, g);          //CPS
}
```

Dieses Stilmittel ist eine Möglichkeit, Funktionen höherer Ordnung anzuwenden. Die Try-Monade, die in Abschnitt 4.3 vorgestellt wird, stellt Methoden zur Verfügung, die Continuation Passing Style anwenden.

3.8. Patternmatching

Patternmatching erlaubt es, eine Fallunterscheidung anhand von Attributen eines Wertes durchzuführen. Diese Funktionalität ähnelt der eines if-then-else-Blocks oder eines switch-Blocks. Pattern-

matching ermöglicht es zusätzlich, Attribute in der Fallunterscheidung zu binden, damit diese in dem danach auszuführenden Code benutzbar sind.

Da Patternmatching in Java nicht unterstützt wird, wird in Abbildung 3.14 die Fakultätsfunktion in der Java-ähnlichen funktionalen Programmiersprache Scala vorgestellt. Die Variable `n` wird erst in Zeile 3 im allgemeinen Fall gebunden.

Abbildung 3.14: Fakultätsfunktion in Haskell mit Patternmatching

```
1 def factorial(x:Int):Int = x match {
2     case 0 => 1
3     case n => n*factorial(n-1)
4 }
```

Patternmatching in Scala bietet noch weitere Funktionalität, die in Java nicht zur Verfügung steht. [Sar14, Sta13]

Im Vergleich dazu wird in Abbildung 3.15 die Fakultätsfunktion in Java vorgestellt, die einen switch-Block zur Fallunterscheidung benutzt. Hier muss `n` vor der Fallunterscheidung gebunden werden und es kann nur zwischen spezifischen Werten oder dem allgemeinen Fall unterschieden werden.

Abbildung 3.15: Fakultätsfunktion in Java mit einem switch-Block

```
int factorial(int n) {
    switch (n) {
        case 0 : return 1;
        default : return n*factorial(n-1);
    }
}
```

Verwendung Patternmatching ist ein übliches Stilmittel in funktionalen Sprachen, das häufig zur Fallunterscheidung benutzt wird. Es ist das einzige hier aufgeführte Feature, dass in Java 8 nicht als Sprachfeature zur Verfügung steht.

4. Funktionale Programmierbeispiele in Java 8

In diesem Kapitel wird in ausführlichen Beispielen imperatives Programmieren mit funktionalem Programmieren verglichen. Nach jedem Codebeispiel werden die verwendeten funktionalen Stilmittel beschrieben und Vor- und Nachteile der funktionalen Lösung erörtert. Die Codebeispiele sind aus dem Buch *Java 8 in Action* entnommen, und werden dort im Detail behandelt. [RGU14]

4.1. Behandlung von Collections mit Streams

In diesem Abschnitt werden beispielhafte Anfragen an eine Liste von Transactions gestellt. Diese Anfragen werden zuerst im klassischen imperativen Stil implementiert, und dann in einem funktionalen Stil mit Streams.

Beide Klassen stellen Getter-Methoden zum Zugriff auf ihre Attribute bereit. Die Attribute der Trader-Klasse werden in Abbildung 4.1 dargestellt.

Abbildung 4.1: Attribute der Trader-Klasse

```
class Trader {
    String name;
    String city;
}
```

Die Attribute der Transaction-Klasse werden in Abbildung 4.2 dargestellt.

Abbildung 4.2: Attribute der Trader-Klasse

```
class Transaction {
    Trader trader;
    int year;
    int value;
}
```

Es wird eine Liste von Transaction-Objekten namens `transactions` zur Verfügung gestellt.

4.1.1. Beispiel: Nach Jahr gefilterte und nach Wert sortierte Transaktionen

Im ersten Beispiel werden aus der Liste aller Transaktionen diese gesammelt, die im Jahr 2011 stattfanden. Diese Transaktionen werden nach Umsatz sortiert und in einer Liste zurückgegeben. Eine imperative Implementierung wird in Abbildung 4.3 dargestellt.

Abbildung 4.3: Imperativer Stil - Transaktionen von 2011, nach Umsatz sortiert

```
public List<Transaction> sortedTransactionsFrom2011() {
    List<Transaction> result = new ArrayList<>();
    for (Transaction t : transactions) {
        if (t.getYear() == 2011) {
            result.add(t);
        }
    }
    result.sort(new Comparator<Transaction>() {
        @Override
        public int compare(Transaction o1, Transaction o2) {
            return o1.getValue() - o2.getValue();
        }
    });
    return result;
}
```

Diese Methode besteht aus zwei logischen Operationen. Zuerst werden die Transaktionen ausgewählt, die dem Kriterium `t.getYear() == 2011` genügen. Diese Transaktionen werden in einer neuen Liste gespeichert. Dies ist eine Filter-Operation.

Danach werden die gefilterten Transaktionen sortiert. Dafür wird eine Comparator-Instanz erstellt, deren compare-Methode die value-Attribute der Transaktionen vergleicht. Diese anonyme innere Klasse kann in Java 8 durch ein Lambda ersetzt werden. Ein äquivalenter Aufruf der sort-Methode mit einem Lambda wird in Abbildung 4.4 dargestellt. So können 5 Zeilen Code gespart werden, die keine Logik enthalten.

Abbildung 4.4: Aufruf der sort-Methode mit einem Lambda

```
result.sort((o1, o2) -> o1.getValue() - o2.getValue());
```

Die erste logische Operation in Abbildung 4.3 benötigt jedoch 6 Zeilen, nur um eine gefilterte Liste zu erzeugen. Durch Nutzung von Streams benötigt das Erstellen einer gefilterten Kopie der Liste nur eine Methode, wie in Abbildung 4.5 dargestellt.

Abbildung 4.5: Erstellung eines gefilterten Streams aus einer Liste

```
transactions.stream().filter(t -> t.getYear() == 2011)
```

Dieser Ausdruck kann durch Einsatz von Funktionspipelines direkt mit weiteren Stream-Methoden verkettet werden, wie der Stream.sorted-Methode. Sie empfängt wie die sort-Methode von Collections

ein Comparator-Objekt, unterscheidet sich aber darin, dass sie den sortierten Stream zurückgibt anstatt den Stream, auf dem sie aufgerufen wird, zu modifizeren. Sie ist also eine pure Funktion.

In Abbildung 4.6 wird ein Ausdruck dargestellt, der sowohl die Filter- als auch die Sortieroperation enthält.

Abbildung 4.6: Filtern und sortieren eines Streams

```
transactions.stream()
    .filter(t -> t.getYear() == 2011)
    .sorted((o1, o2) -> o1.getValue() - o2.getValue())
```

Um die in diesem Stream enthaltenen Transaktionen als Liste zurückzugeben, muss eine Terminaloperation verwendet werden. Die Stream.collect-Methode erlaubt es unter anderem, die Elemente eines Streams in einer Liste zu sammeln. Somit kann die Methode aus Abbildung 4.3 auf funktionale Art und Weise implementiert werden, wie in Abbildung 4.7 dargestellt.

Abbildung 4.7: Funktionaler Stil - Transaktionen von 2011, nach Umsatz sortiert

```
public List<Transaction> sortedTransactionsFrom2011() {
    return transactions.stream()
        .filter(t -> t.getYear() == 2011)
        .sorted(Comparator.comparing(Transaction::getValue))
        .collect(Collectors.toList());
}
```

Die statische Comparator.comparing-Methode erzeugt ein Comparator-Objekt, das in diesem Fall die value-Attribute von Transaktionen vergleicht. Die Funktionalität wird durch die Namensgebung beschrieben und wird in der Java-Dokumentation weiter erklärt. [Ora14e]

Diese Methode hat, neben der Kompaktheit, klare Vorteile gegenüber der imperativen Version. Sie besteht aus einem einzigen Ausdruck, in dem jede logische Operation in einer Zeile ausgeführt wird. Die Methoden beschreiben die ausgeführten Operationen mit ihrem Namen. Die Anwendung der Operationen auf die einzelnen Elemente wird durch die Implementierung des Streams vollzogen, und nicht vom Programmierer.

Außerdem ist die zyklomatische Komplexität der Methode geringer, da keine Schleifen oder if-Blöcke nötig sind. Zyklomatische Komplexität ist eine Metrik, die zur Bewertung der Codequalität etabliert ist. [Inc12] Obwohl diese Metrik in funktionaler Programmierung durch die Nutzung von Funktionen höherer Ordnung weniger aussagekräftig ist, wird sie doch auch für die funktionale Programmiersprache Scala beachtet. [DS11, DS14]

4.1.2. Beispiel: Trader aus Cambridge ohne Duplikate, nach Namen sortiert

In diesem Beispiel wird eine Liste der Trader aus Cambridge gesucht, die nach Namen sortiert sind. Dies wird in Abbildung 4.8 imperativ gelöst.

Abbildung 4.8: Imperativer Stil - Händler aus Cambridge, nach Namen sortiert

```
public List<Trader> sortedTradersFromCambridge() {
    Set<Trader> resultSet = new TreeSet<>(new Comparator<Trader>() {
        @Override
        public int compare(Trader o1, Trader o2) {
            return o1.getName().compareTo(o2.getName());
        }});
    for (Transaction t : transactions) {
        if ("Cambridge".equals(t.getTrader().getCity())) {
            resultSet.add(t.getTrader());
        }
    }
    return new ArrayList<>(resultSet);
}
```

Hier wird ein Set als Zwischenspeicher benutzt, da keine Duplikate erwünscht sind. Die TreeSet-Klasse erlaubt es außerdem, ihre Elemente zu ordnen, was den zweiten Teil der Problemstellung erfüllt. Dafür wird ein Comparator-Objekt in den Konstruktor des TreeSets übergeben, das die Namen der Trader vergleicht. Die Schleife, in dieser Methode der letzte Block, führt schließlich die Filter-Operation aus und kopiert die Trader in das Set.

Da zwei Operationen durch die Wahl der Datenstruktur durchgeführt werden, ist diese Methode relativ direkt, vor allem, wenn die anonyme innere Klasse im TreeSet-Konstruktor durch ein Lambda ersetzt werden würde. Die äquivalente Methode mit Streams, in Abbildung 4.9 dargestellt, ist daher ähnlich lang.

Abbildung 4.9: Funktionaler Stil - Händler aus Cambridge, nach Namen sortiert

```
public List<Trader> sortedTradersFromCambridge() {
    return transactions.stream()
            .map(Transaction::getTrader)
            .filter(trader -> "Cambridge".equals(trader.getCity()))
            .distinct()
            .sorted(Comparator.comparing(Trader::getName))
            .collect(Collectors.toList());
}
```

Dieser Stil beschreibt aber deutlich ausdrücklicher, welche Operationen durchgeführt werden. Die Entfernung von Duplikaten aus dem Ergebnis wird im imperativen Beispiel nur durch die Nutzung von Set deutlich, das Sortieren nur dadurch, dass man einen Comparator im Konstruktor mitgibt. Im funktionalen Beispiel sind diese beiden Operationen durch den Aufruf von Methoden mit erklärenden Namen offensichtlich gemacht.

4.1.3. Beispiel: Sortierte Namen aller Händler in einem String

In diesem Beispiel werden die Namen aller Händler geordnet in einem String gespeichert. Die Imperative Lösung wird in Abbildung 4.10 dargestellt.

Abbildung 4.10: Imperativer Stil - Sortierte Namen aller Händler

```
public String activeTraderNames() {
    StringBuilder names = new StringBuilder();
    Set<String> nameSet = new TreeSet<>(new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.compareTo(o2);
        }});
    for (Transaction t : transactions) {
        nameSet.add(t.getTrader().getName());
    }
    for (String name : nameSet) {
        names.append(name);
    }
    return names.toString();
}
```

Das Sortieren und das Entfernen von Duplikaten wird wieder durch die Benutzung eines TreeSets als erste Hilfsvariable implizit gemacht. Das Füllen des Sets findet in der ersten Schleife statt. Danach wird der String in einer zweiten Hilfsvariable aus den Namen der Trader erstellt, wofür eine weitere Schleife notwendig ist.

Im Vergleich dazu die funktionale Lösung in Abbildung 4.11.

Abbildung 4.11: Sortierte Namen aller Händler, Funktional

```
public String fActiveTraderNames() {
    return transactions.stream()
            .map(transaction -> transaction.getTrader().getName())
            .distinct()
            .sorted()
            .collect(Collectors.joining());
}
```

Die sorted-Methode ohne Parameter kann Streams von Elementen, die eine natürliche Ordnung haben ohne einen Comparator sortieren. Die Benutzung des StringBuilders wird durch die Collectors.joining-Methode implizit gemacht, aber auch andere Datenstrukturen können durch die Stream.reduce Methoden einfach kombiniert werden.

In diesem Beispiel ist der Unterschied zwischen den imperativen und funktionalen Methoden besonders drastisch, da eine anonyme innere Klasse und zwei Schleifen eingespart wurden. Das Stream-Beispiel ist nicht komplexer als die vorigen und beschreibt nach wie vor die Vorgehensweise durch die Methodennamen.

4.1.4. Zusammenfassung

Ein großer Unterschied zwischem imperativem und funktionalem Programmierstil im Umgang mit Collections ist die Codestruktur und Leserichtung sowie die Reihenfolge der Operationen. Imperativer Code besteht aus vielen Schleifen und if-Blöcken und benutzt häufig Zwischenvariablen. Daher wird beim Lesen des Codes oft zwischen entfernten Zeilen hin und her gesprungen.

Operationen in einer einzigen Pipeline, können dagegen in Reihenfolge der Methodenaufrufe gelesen werden. Streams ermöglichen es, viele Operationen auf Collections in einer einzigen Pipeline zu komponieren.

4.2. Behandlung von null-baren Werten mit Optional

In diesem Abschnitt wird ein Beispiel vorgestellt, in dem möglicherweise fehlende Attribute der Klassen Person, Car und Insurance ausgelesen werden. Die Attribute der Klassen sind durch Getter-Methoden erreichbar. In Abbildung 4.12 werden die Attribute der Klassen dargestellt. Die Attribute, deren Wert fehlen kann, sind durch Kommentare markiert

Abbildung 4.12: Klassen, die in Abschnitt 4.2 benutzt werden

```
class Person {
    Car car; //nullbar
}
class Car {
    Insurance insurance; //nullbar
}
class Insurance {
    String name;
}
```

Eine naive imperative Implementierung einer Methode, die den Namen der Autoversicherung einer übergebenen Person zurückgibt wird in Abbildung 4.13 dargestellt.

Abbildung 4.13: Imperative Methode, die nicht auf null-Werte prüft

```
public String getCarInsuranceName(Person p) {
    return p.getCar().getInsurance().getName();
}
```

Nicht vorhandene Werte werden im Imperativen Stil durch null modelliert. Die oben gezeigte Methode könnte also eine NullPointerException auslösen. Eine korrekte Implementierung würde diese Fälle überprüfen, wie in Abbildung 4.14 dargestellt wird.

Diese Methode gibt einen alternativen Rückgabewert an, stattdessen könnte aber auch null zurückgegeben werden, falls eines der Zwischenergebnisse null ist. Diese Methode kann durch den Einsatz von Optional kürzer und mit geringerer zyklomatischer Komplexität geschrieben werden. Dies wird in Abbildung 4.15 dargestellt.

Abbildung 4.14: Imperative Behandlung von möglichen null-Werten

```
public String getCarInsuranceName(Person p) {
    String result = "unknown";
    if (p.getCar() != null) {
        Car car = p.getCar();
        if (car.getInsurance() != null) {
            result = car.getInsurance().getName();
        }
    }
    return result;
}
```

Abbildung 4.15: Einsatz von Optional, um null-Tests zu ersetzen

```
public String getCarInsuranceName(Person p) {
    return Optional.ofNullable(p.getCar())
            .map(Car::getInsurance)
            .map(Insurance::getName)
            .orElse("unknown");
}
```

Diese Methode setzt eine Pipeline ein und hat dadurch eine geringe zyklomatische Komplexität. Die Überprüfung von null-Werten wird durch Optional internalisiert. Diese Methode hat also einen klaren Vorteil in Programmierstil und Lesbarkeit.

Jedoch benutzt diese Methode Optional nur intern. Die Getter der Klassen geben immer noch null-Werte zurück, was an anderen Stellen im Programm zu Fehlern führen kann. Außerdem gibt die Methode immer noch einen alternativen Wert zurück, falls der geforderte Wert nicht existiert.

Eine bessere Alternative wäre es, sowohl den Rückgabewert der Methode, als auch die Rückgabewerte der Getter, zu Optional zu ändern. Die so geänderten Attribute der Klassen werden in Abbildung 4.16 dargestellt.

Abbildung 4.16: Modifizierte Klassen, die Optional benutzen

```
class Person {
    Optional<Car> car;
}
class Car {
    Optional<Insurance> insurance;
}
class Insurance {
    String name;
}
```

In dieser Version sind keine Kommentare notwendig, die den Benutzer darüber informieren, dass Werte null sein können, stattdessen wird dies durch das Typsystem deutlich. Eine Methode, die die Klassen aus Abbildung 4.16 verwendet und selber Optional zurückgibt, wird in Abbildung 4.17 dargestellt.

Abbildung 4.17: Alle Werte, die fehlen können, sind durch Optional ersetzt

```
public Optional<String> getCarInsuranceName(Person p) {
    return p.getCar()
            .flatMap(Car::getInsurance)
            .map(Insurance::getName);
}
```

Diese Methode erfüllt die selbe Funktionalität wie die erste imperative Methode in Abbildung 4.14, sie ist jedoch deutlich kürzer, geradliniger zu lesen und macht dem Benutzer deutlich, dass sie womöglich kein Ergebnis produzieren kann.

4.2.1. Zusammenfassung

Die Behandlung von möglichen null-Werten mit Optional erlaubt es, mit Pipelines Methoden zu komponieren und dadurch lesbareren und kürzeren Code zu schreiben. Optional als Rückgabewert macht möglicherweise fehlende Werte deutlich, was mit der Benutzung von null-Werten nur durch Kommentare mitgeteilt werden kann. Außerdem wird der Benutzer durch das Typsystem dazu angehalten, diesen Fall zu behandeln. [Urm14b]

4.3. Try-Monade

In diesem Abschnitt wird eine Monade zur Behandlung von Exceptions implementiert. Diese Implementierung ahmt die Try-Monade aus der Programmiersprache Scala nach. [Lin14]

Diese Monade umschließt Ausdrücke, deren Evaluation Exceptions auslösen kann, und versucht, den Rückgabewert zu erzeugen. Falls dies erfolgreich gelingt, wird der Wert in einer Success-Instanz gespeichert. Falls nicht, wird die Exception in einer Failure-Instanz gespeichert. Diese grundlegende Funktionalität wird in Abbildung 4.18 vorgestellt.

Abbildung 4.18: Grundlegende Funktionalität von Try

```
Try<Integer> first = Try.attempt(() -> Integer.parseInt("12"));
if (first.isSuccess()) {
    first.get(); //12
}
Try<Integer> second = Try.attempt(() -> Integer.parseInt("no int"));
if (second.isFailure()) {
    second.getException(); //NumberFormatException
}
```

Die Fabrikmethode attempt empfängt ein Supplier-Objekt, das eine Funktion ohne Parameter darstellt. So kann die Berechnung des Ausdruckes in die Fabrikmethode verlagert werden. Diese Methode gibt dann entweder ein Success-Objekt mit dem Rückgabewert, oder ein Failure-Objekt mit der Exception zurück. Die Fabrikmethode könnte also wie in Abbildung 4.19 implementiert werden.

Abbildung 4.19: Fabrikmethode der Try-Monade

```
public static <T> Try<T> attempt(TrySupplier<T> val) {
    try {
        return new Success<>(val.get());
    } catch (Exception e) {
        return new Failure<>(e);
    }
}
```

Success und Failure sind beides Kindklassen von Try, die jeweils einen Wert vom Typ `T` oder eine Exception enthalten. Diese Fabrikmethode internalisiert also den Try-Block, genau wie Optional den if-Block, der auf null überprüft, internalisiert. Die einzige Besonderheit dieser Methode ist der Parameter, da sie kein Supplier aus der function-Package ist. Der Grund dafür ist, dass in diesem Fall Exceptions geworfen werden, wofür ein eigenes Functional Interface deklariert werden muss. Die zwei eigenen Interfaces der Try-Monade werden in Abbildung 4.20 vorgestellt.

Abbildung 4.20: Functional Interfaces für Try

```
interface TryFunction<T,U> {
    public U apply(T t) throws Exception;
}
interface TrySupplier<T> {
    public T get() throws Exception;
}
```

Diese Functional Interfaces haben ähnliche Methoden wie Function und Supplier aus dem Function-Paket, werfen aber Exceptions. Dadurch, dass andere Interfaces benutzt werden, können in Variablen gespeicherte Funktionen von Try nicht mit anderen Monaden wie Optional benutzt werden. Dies verhindert außerdem ein Monaden-Interface, da dies entweder mit Optional oder Try inkompatibel wäre.

Weiterhin werden map- und flatMap-Methoden benötigt. Die Implementierungen dieser Methoden werden in Abbildung 4.21 sowie Abbildung 4.22 dargestellt.

Abbildung 4.21: Map-Methoden für Success und Failure

```
<U> Try<U> map(TryFunction<T, U> f) {
    return attempt(() -> f.apply(value));
}

public <U> Try<U> map(TryFunction<T, U> f) {
    return new Failure<>(e);
}
```

Abbildung 4.22: FlatMap-Methoden für Success und Failure

```
<U> Try<U> flatMap(TryFunction<T, Try<U>> f) {
    try {
        return f.apply(value);
    } catch (Exception e) {
        return new Failure<>(e);
    }
}
public <U> Try<U> flatMap(TryFunction<T, Try<U>> f) {
    return new Failure<>(e);
}
```

Ähnlich wie bei Optional führen (Flat)Map keine Operation auf Failures durch, sie geben einfach die gespeicherte Exception weiter.

4.3.1. Beispiel für den Einsatz der Try-Monade

IO-Operationen können einige Exceptions auslösen. Abbildung 4.23 zeigt eine Methode, die alle Zeilen einer Datei ausliest und als Stream zurückgibt. Dieser Stream ist in einem Try verpackt, da Exceptions ausgelöst werden können.

Abbildung 4.23: Methode zum Auslesen von Dateien mit Stream

```
Try<Stream<String>> getLines(String path) {
    return Try.attempt(() -> Files.lines(Paths.get(path)));
}
```

Äquivalente Methoden ohne Try werden in Abbildung 4.24 dargestellt. Diese Methoden benutzen die neue Files.lines-Methode. Es ist Konvention, Exceptions weiter zu propagieren, wie in der ersten Methode. Dadurch wird die Fehlerbehandlung allerdings nur verschoben. Try erlaubt es, die Exception zu propagieren, ermöglicht es aber, weiter mit dem Rückgabewert zu arbeiten, auch wenn er eine Exception enthält.

Abbildung 4.24: FlatMap-Methoden für Success und Failure

```
Stream<String> getLines(String path) throws IOException {
    return Files.lines(Paths.get(path));
}

Stream<String> getLines(String path) {
    try {
        return Files.lines(Paths.get(path));
    } catch (IOException e) {
        // handle e
    }
}
```

4.4. Berechnung des größten Palindrom-Produktes

In diesem Abschnitt wird ein Beispielalgorithmus implementiert, der das größte Produkt von dreistelligen Zahlen berechnet, das ein Palindrom ist. Palindrome ergeben von vorne und von hinten gelesen das selbe Ergebnis.

Dafür wird eine Methode gegeben, die rekursiv überprüft, ob ein String ein Palindrom ist. Diese Methode wird in Abbildung 4.25 dargestellt.

Abbildung 4.25: Methode, die überprüft, ob ein String ein Palindrom ist

```
boolean isPalindrome(String s) {
    if (s.length() <= 1) return true;
    return s.charAt(0) == s.charAt(s.length() - 1) &&
            isPalindrome(s.substring(1, s.length() - 1));
}
```

Diese Methode vergleicht das erste und letzte Zeichen des Strings und ruft sich dann rekursiv mit dem inneren String, ohne die davor verglichenen Zeichen auf, bis der String nur noch ein oder kein Zeichen hat. Die Berechnung des größten Palindroms auf imperative Weise wird in Abbildung 4.26 dargestellt.

Abbildung 4.26: Imperative Methode, unoptimiert

```
int result = 0;
for (int i = 999; i >= 100; i--) {
    for (int j = 999; j >= 100; j--) {
        int pal = i * j;
        if (isPalindrome(pal)) {
            if (pal > result) {
                result = pal;
            }
        }
    }
}
```

Diese Methode vergleicht durch zwei ineinander geschachtelte Schleifen alle Kombinationen von dreistelligen Zahlen. Jede Kombination wird multipliziert, durch isPalindrome gefiltert und am Ende gespeichert, wenn sie größer ist als das vorige Zwischenergebnis. Eine äquivalente Methode im funktionalen Stil, die Streams benutzt, wird in Abbildung 4.27 dargestellt.

Diese Methode führt eine komplexe Map-Operation durch, in der jede Zahl von 999 bis 100 auf das größte Produkt mit einer anderen dreistelligen Zahl abgebildet wird, falls dieses Produkt ein Palindrom ist. Alternativ werden die Elemente auf Null abgebildet. Schließlich werden die Elemente des Streams auf die größte Zahl reduziert.

In solch einem vergleichbar komplexeren Beispiel ist der funktionale Stil kein klarer Sieger in Sachen Lesbarkeit. Außerdem ist der Leistungsunterschied in diesem Beispiel signifikant, vor allem da

Abbildung 4.27: Funktionale Methode mit Streams

```
IntStream.range(999, 100)
        .map(i -> IntStream.range(999, 100)
                .map(j -> i*j)
                .filter(this::isPalindrome)
                .findFirst()
                .orElse(0))
        .reduce(0, Math::max);
```

der imperative Stil durch zwei Änderungen leicht optimiert werden kann, wie in Abbildung 4.28 dargestellt.

Abbildung 4.28: Imperative Methode, optimiert

```
int result = 0;
for (int i = 999; i >= 100; i--) {
    for (int j = 999; j >= 100; j--) {
        int pal = i * j;
        if (pal < result) {
            break; //first optimization
        }
        if (isPalindrome(pal)) {
            if (pal > result) {
                result = pal;
            }
            break; //second optimization
        }
    }
}
```

Das erste Kommentar markiert eine Zeile, die erreicht wird, wenn das Produkt kleiner ist als das aktuelle Maximum. Da die innere Schleife in jedem Durchlauf ein geringeres Produkt erzeugt, kann an dieser Stelle die innere Schleife abgebrochen werden. Aus dem gleichen Grund kann nach dem ersten Fund eines Palindroms die innere Schleife gebrochen werden, was in der Zeile des zweiten Kommentars geschieht. Diese Optimierungen verbessern die Leistung der imperativen Version um einen Faktor von 10, siehe Abschnitt A.1.

5. Richtlinien für funktionale Programmierung in Java 8

In diesem Kapitel werden Richtlinien für die Gestaltung von funktionalem Code und für den Einsatz der neuen Features von Java 8 bzw. eines funktionalen Programmierstils vorgeschlagen. Diese Regeln wurden bei der Einarbeitung in das Thema erprobt (siehe Abschnitt A.3) und halten sich an allgemeine Konventionen. [RGU14, Urm14a, Sau14a, Fus12, Urm14b]

5.1. Benutzung der neuen Features in Java 8

Lambdas anstelle von anonymen inneren Klassen Zustandslose anonyme innere Klassen in Lambdas umwandeln, da Lambdas die selbe Information enthalten und weniger Platz verbrauchen.

Lambdas mit nur einem Ausdruck bevorzugen Lambdas ohne geschweifte Klammern und ohne return-Anweisung sind platzsparender. Falls das Lambda durch die kürzere Form weniger lesbar ist, ist es zu bevorzugen, das Lambda durch Formatierung hervorzuheben.

Implizite Typen der Parameter bevorzugen Nur wenn die Parametertypen des Lambdas nicht offensichtlich oder leicht zu erkennen sind, sollten Typen deklariert werden.

Übersicht durch Leerzeichen, Einrückung und Umbrüche Visuelle Anordnung zu anderen Lambdas oder ein Zeilenumbruch vor oder nach dem Lambda bzw im Körper des Lambdas können den Code lesbarer machen.

Optionale Klammern auslassen Außer der vorige Punkt wurde schon befolgt und es trägt zur Übersicht bei.

Direkt nach dem Pfeil umbrechen Falls nicht innerhalb des Körpers umgebrochen werden kann. Weitere Umbruchregeln betreffen Methodenpipelines.

Konventionelle Regeln beachten Sowohl die Parameterliste als auch der Körper kann sonst nach üblichen Styleregeln formatiert werden.

Method References statt Lambdas bevorzugen Eine Ausnahme ist, wenn die Referenz durch den Klassen- oder Objektnamen länger wäre als ein entsprechendes Lambda. Selbst dann kann eine Method Reference lesbarer sein.

Vorhandene Functional Interfaces benutzen Wenn eine Funktion nötig ist, die nicht im Function-Paket enthalten ist, oder ein anderer Name sinnvoll ist, können eigene Interfaces eingesetzt werden.

Streams für Operationen auf gesamte Collections benutzen Operationen, die durch Iteration über die gesamte Collection durchgeführt werden, können oft mit Streams lesbarer implementiert werden.

Optional statt null zurückgeben, um fehlenden Wert zu signalisieren Dadurch wird dem Benutzer durch den Rückgabetyp angezeigt, dass ein fehlender Wert möglich ist.

Wert im Optional über map, flatMap und filter bearbeiten Anstatt den Wert manuell zu entpacken und zu modifizieren.

Wert im Optional über ifPresent und die orElse-Methoden entpacken Anstatt mit isPresent zu testen und danach mit get den Inhalt zu erreichen.

5.2. Einsatz von funktionalen Programmiermustern

Pure Funktionen verwenden Methoden, die nicht explizit für ihre Seiteneffekte benutzt werden, sollten wenn möglich pur sein.

Mutierbare Datenstrukturen vermeiden Außer es ist für die Leistung des Programms notwendig.

Rekursion mit Vorsicht einsetzen Wenn eine Methode eleganter ausgedrückt werden kann, und keine Gefahr besteht, einen StackOverflowError auszulösen.

Lazy evaluierte Strukturen ausnutzen Wenn z.B. nur Teile einer Collection bearbeitet werden müssen, kann die Nutzung von Streams die Leistung verbessern.

Wiederholte logische Muster in Funktionen höherer Ordnung extrahieren Analog mit der Extraktion von Methoden. Die Möglichkeit zur Extraktion kann aber weniger offensichtlich sein als bei Methodenextraktion.

Methodenpipelines statt Zwischenvariablen nutzen Zwischenvariablen müssen vor der Nutzung deklariert werden, oft liegen einige Zeilen zwischen Deklaration und Verwendung. Methodenpipelines können hingegen meistens direkt von vorne nach hinten gelesen werden.

Methodenpipelines direkt vor Punkten umbrechen Dadurch sind Methodenaufrufe mit zugehörigem Punkt auf einer Zeile. Es kann dadurch eine Zeile inmitten einer Pipeline auskommentiert werden, z.B. ein Aufruf der parallel-Methode in einer Stream-Pipeline. Bei längeren Pipelines sollte jede Methode auf einer separaten Zeile stehen.

Pipelines in Pipelines durch Einrückung visuell abtrennen Methoden in einer Pipeline können manchmal selber Pipelines als Parameter erhalten. Dann sollte diese innere Pipeline weiter eingerückt sein als die äußere. Weiterhin sollten innere Pipelines besonders lesbar gestaltet werden.

Logische Sprünge in Pipelines durch Leerzeilen oder kommentierte Zeilen lesbar machen Wenn eine Verkettung von Methoden schwer nachvollziehbare Operationen durchführt, kann es lesbarer sein, Leerzeilen mit Kommentaren zwischen zwei Methoden einzufügen, anstatt ein Zwischenergebnis in eine Variable zu speichern.

6. Zusammenfassung

Die neuen Features in Java 8 ermöglichen in vielen Fällen Programmierung im funktionalen Stil. Die in Kapitel 3 vorgestellten Stilmittel sind (bis auf Patternmatching) in Java gut umsetzbar. Lambdas und Method References sowie die Einführung von Monaden fördern den Einsatz von Methodenpipelines und puren Funktionen. Solch ein Programmierstil kann die Parallelisierung eines Programmes stark vereinfachen. Im Fall von Streams kann z.B. der Aufruf einer einzigen zusätzlichen Methode die Bearbeitung parallelisieren. Auch unter anderen Gesichtspunkten hat ein funktionaler Programmierstil Vorteile, z.B. in der Lesbarkeit.

Die Änderungen in Java 8 sind dabei relativ leicht verständlich geblieben, da nur Lambdas und Method References zusätzliche Syntaxregeln mit sich bringen. Die restlichen Änderungen wurden in den Bibliotheken durchgeführt. Dies macht die Umgewöhnung für Java-Programmierer einfach. Gleichzeitig ermöglichen Streams und Optional einen puren funktionalen Programmierstil, der auch in der offiziellen Dokumentation sowie Tutorials vorgestellt wird. Es wird Java-Entwicklern also vereinfacht, sich einen funktionalen Programmierstil anzueignen.

Einige Features, die für einen funktionalen Programmierstil sinnvoll sind, fehlen in Java 8 aber. Darunter befinden sich Value-Typen, die einen effizienten Umgang mit Aggregatsdatentypen ermöglichen würden. [JR14] Tupel-Typen würden es vereinfachen, in einer Methode mehrere Werte zurückzugeben, ohne selber eine Klasse dafür zu erstellen. Dies ist in einem funktionalen Programmierstil wichtiger als in einem imperativen, da pure Funktionen neben dem Rückgabewert nicht mit der Umgebung interagieren. [Sau14b]

Funktionstypen würden das Schreiben von Methoden, die Lambdas als Parameter akzeptieren, angenehmer machen. Die Verwendung von Functional Interfaces ist besonders umständlich, wenn Lambdas mit mehr als 2 Parametern benötigt werden. Dann kann entweder ein neues Functional Interface eingeführt werden oder das Function-Interface aus dem Function-Paket geschachtelt werden. Dadurch können die Parameter des Lambdas (das eigentlich aus mehreren geschachtelten Lambdas besteht), durch Pfeile getrennt, hintereinander geschrieben werden. Dies wird in Abschnitt A.3 in der Implementierung der `foldl`-Methode dargestellt. Schließlich bedeutet das Fehlen von Tail-Call-Optimization, dass Rekursion nur mit Vorsicht eingesetzt werden kann.

Schlussendlich hat Java mit der Version 1.8 einen sinnvollen Schritt in Richtung funktionaler Programmierung gemacht. Es wird ein funktionaler Programmierstil ermöglicht, der klare Vorteile gegenüber einem imperativen Programmierstil haben kann. Die Umgewöhnung für Java-Programmierer wurde gering gehalten und die Sprachentwicklung wurde klar abgegrenzt von “funktionaleren” Sprachen wie Scala.

A. Appendix

In diesem Kapitel wird Code, der bei der Einarbeitung in diese Arbeit erzeugt wurde, aufgelistet.

Die Leistungsdaten sowie die Tests zur Existenz von Tail-Call-Optimization wurden auf einem PC mit Windows 8.1 Pro 64-bit, einem AMD Phenom II X4 975 sowie 8.00GB Dual-Channel DDR3 @ 669MHz RAM durchgeführt. Als Laufzeitumgebung wurde JRE 1.8.0_05 mit Standardeinstellungen auf Eclipse 4.4.0 durchgeführt.

A.1. Daten der Leistungstests

In diesem Abschnitt werden die Leistungstests vorgestellt, in denen die Leistung von Lambdas mit der von anonymen inneren Klassen sowie imperativem Code verglichen wurde. Diese Leistungstests decken nur einfache Fälle ab und treffen daher keine allgemeingültige Aussage.

Abbildung A.1 vergleicht Lambdas mit äquivalenten Aufrufen von anonymen inneren Klassen. Die Methode, die Lambdas benutzte, brauchte ca 950ms. Die Methode, die eine innere Klasse benutzte, brauchte ca 710ms. Dies spiegelt nicht die Ergebnisse des Oracle Performance Teams wieder, welches allerdings vermutlich einen differenzierteren Leistungstest durchführte. [Goe12a]

Abbildung A.2 vergleicht Lambdas mit einer ähnlichen, imperativen Methode. Die Methode, die Lambdas benutzte, brauchte wieder ca 950ms. Die imperative Methode brauchte ca 840ms.

Abbildung A.3 vergleicht Listen mit Streams. Die Methode, die Listen benutzte, brauchte ca 180ms. Die Methode, die Streams benutzte, brauchte ca 730ms.

Abbildung A.1: Leistungsvergleich von Lambdas und anonymen inneren Klassen

```
long TIMES = 1_000_000_000L;

public int f(Function<Integer,Integer> f) {
    return f.apply(0);
}

long testLambda() {
    Timer t = new Timer();
    int val = 0;
    t.start();
    for (int i = 0; i < TIMES; i++) {
        val = f(x -> x);
    }
```

```
    return t.stop();
}

long testInnerClass() {
    Timer t = new Timer();
    int val = 0;
    t.start();
    for (int i = 0; i < TIMES; i++) {
        val = l.aiC(new Function<Integer, Integer>() {
            @Override
            public Integer apply(Integer x) {
                return x;
            }});
    }
    return t.stop();
}
```

Abbildung A.2: Leistungsvergleich von Lambdas und imperativer Programmierung

```
long TIMES = 1_000_000_000L;

public int f(Function<Integer,Integer> f) {
    return f.apply(0);
}

long testLambda() {
    Timer t = new Timer();
    int val = 0;
    t.start();
    for (int i = 0; i < TIMES; i++) {
        val = f(x -> x);
    }
    return t.stop();
}

long testImperative() {
    Timer t = new Timer();
    int val = 0;
    t.start();
    for (int i = 0; i < TIMES; i++) {
        val = imperativeId(0);
    }
    return t.stop();
}
```

Abbildung A.3: Leistungsvergleich von Streams und Listen

```
List<String> list = new ArrayList<>();
    //...
    for (int i = 0; i < 1000000; i++) {
        l.list.add(String.valueOf(i));
    }
    //...

long testList() {
    Timer t = new Timer();
    t.start();
    List<String> result = new ArrayList<>();
    for (String s : list) {
        if (s.length() % 2 == 0) {
            result.add(s.concat("done"));
        }
    }
    return t.stop();
}

long testStream() {
    Timer t = new Timer();
    t.start();
    List<String> result = list
            .stream()
            .filter(s -> s.length() % 2 == 0)
            .map(s -> s.concat("done"))
            .collect(Collectors.toList());
    return t.stop();
}
```

Abbildung A.4: Timer-Klasse zur Messung der Leistungsdaten

```
public class Timer {
    long    startTime;

    public Timer() {
        this.startTime = 0;
    }

    private void reset() {
        this.startTime = 0;
    }

    public void start() {
        if (this.startTime != 0)
            throw new IllegalStateException("Timer already started");
        this.startTime = System.nanoTime();
    }

    /**
     *
     * @return the time passed in ms
     */
    public long stop() {
        if (this.startTime == 0)
            throw new IllegalStateException("Timer not started");
        final long resultInNs = System.nanoTime() - this.startTime;
        this.reset();
        return resultInNs / 1_000_000;
    }
}
```

A.2. Try-Monade

Abbildung A.6 zeigt die Implementierung der Try-Monade, die in Abschnitt 4.3 vorgestellt wurde. Die Implementierung ahmt die Try-Monade in Scala nach. [Lin14] Die Functional Interfaces, die von dieser Klasse benutzt werden, werden in Abbildung A.5 dargestellt.

Abbildung A.5: Functional Interfaces, die von der Try-Monade benutzt werden

```
public interface TryFunction<T,U> {
    public U apply(T t) throws Exception;
}
public interface TrySupplier<T> {
    public T get() throws Exception;
}
```

Abbildung A.6: Implementierung der Try-Monade

```
public abstract class Try<T> {

    private Try() {}

    public static <T> Try<T> attempt(TrySupplier<T> val) {
        try {
            return new Success<>(val.get());
        } catch (Exception e) {
            return new Failure<>(e);
        }
    }

    public abstract T get() throws Exception;

    public abstract Exception getException();

    public abstract boolean isFailure();
    public abstract boolean isSuccess();

    public abstract <U> Try<U> map(TryFunction<T,U> f);
    public abstract <U> Try<U> flatMap(TryFunction<T,Try<U>> f);

    public abstract T orElse(T alt);
    public abstract void forEach(Consumer<T> f);

    static class Success<T> extends Try<T> {
        T value;
        private Success(T value) {
            this.value = value;
        }
```

```
@Override
public T get() {
    return value;
}
@Override
public Exception getException() {
    throw new IllegalStateException(toString() +
        " does not contain an Exception");
}
@Override
public boolean isFailure() {
    return false;
}
@Override
public boolean isSuccess() {
    return true;
}
@Override
public <U> Try<U> map(TryFunction<T, U> f) {
    return attempt(() -> f.apply(value));
}
```

```
@Override
public <U> Try<U> flatMap(TryFunction<T, Try<U>> f) {
    try {
        return f.apply(value);
    } catch (Exception e) {
        return new Failure<>(e);
    }
}

@Override
public T orElse(T alt) {
    return value;
}

@Override
public void forEach(Consumer<T> f) {
    f.accept(value);
}
```

```
    @Override
    public String toString() {
        return "Success[" + value.toString() + "]";
    }
}

static class Failure<T> extends Try<T> {
    Exception e;

    private Failure(Exception e) {
        this.e = e;
    }

    @Override
    public T get() throws Exception {
        throw e;
    }
    @Override
    public Exception getException() {
        return e;
    }
    @Override
    public boolean isFailure() {
        return true;
    }



    @Override
    public boolean isSuccess() {
        return false;
    }
    @Override
    public <U> Try<U> map(TryFunction<T, U> f) {
        return new Failure<>(e);
    }

    @Override
    public <U> Try<U> flatMap(TryFunction<T, Try<U>> f) {
        return new Failure<>(e);
    }

    @Override
    public T orElse(T alt) {
        return alt;
    }
```

```
        @Override
        public void forEach(Consumer<T> f) {}

        @Override
        public String toString() {
            return "Failure[" + e.toString() + "]";
        }
    }
}
```

A.3. Einfach verkettete, nicht mutierbare Liste

Die in Abbildung A.7 dargestellte Liste wurde zur Einarbeitung in funktionale Programmiermuster erstellt. Sie ist nicht mutierbar, komplett nebeneffektfrei und implementiert Methoden, die in Scala und Haskell für Listen bereitstehen. Anfangs wurden alle Methoden rekursiv implementiert, schließlich wurden manche Methoden imperativ implementiert während die restlichen auf diesen aufbauen, um StackOverflowErrors vorzubeugen.

Die Kindklassen `Item` und `Empty` werden in Abbildung A.8 sowie Abbildung A.9 dargestellt.

Abbildung A.7: Implementierung einer einfach verketteten Liste im funktionalen Stil

```
public abstract class HList<E> {

    public abstract boolean isEmpty();

    public abstract Optional<E> head();

    public abstract HList<E> tail();

    public static <E> HList<E> emptyList() {
        return Empty.empty();
    }

    public static <E> HList<E> create(E element) {
        return new Item<>(element, emptyList());
    }

    @SafeVarargs
    public static <E> HList<E> create(E... elements) {
        HList<E> l = emptyList();
        for (E e : elements) {
            // prepends elements because prepend is O(1), append is O(n)
            l = l.prepend(e);
        }
        // reverses list because elements are prepended, not appended
        return l.reverse();
    }

    public static <E> HList<E> create(E element, HList<E> list) {
        return new Item<>(element, list);
    }

    public HList<E> prepend(E e) {
        return create(e, this);
    }
```

```
    public HList<E> append(E e) {
        // Uses imperatively implemented foldr, does't overflow
        return foldr(el -> list -> list.prepend(el), create(e));
    }

    public HList<E> prepend(HList<E> l) {
        return l.append(this);
    }

    public HList<E> append(HList<E> l) {
        // Uses imperatively implemented foldr, does't overflow
        return foldr(el -> list -> list.prepend(el), l);
    }

    public Optional<E> get(int i) {

        /*
        //recursive implementation is prettier, but it can overflow:
        if (i == 0) return head();
        if (i < 0 || isEmpty()) return Optional.empty();
        return tail().get(i-1);
        */

        if (i < 0 || i >= size()) return Optional.empty();
        //i + 1 because get(0) returns the first element,
        //unfold(1) creates a one-element list
        return zip(unfold(i + 1), element -> index -> element)
                .last();
    }

    public int size() {
        return foldl(count -> e -> count + 1, 0);
    }

    public Optional<E> last() {
        return foldl(dummy -> element -> Optional.of(element),
            Optional.empty());
    }

    public HList<E> init() {
        return take(size() - 1);
    }

    public HList<E> take(int n) {
        //zip stops when one of the lists stops, unfold(n) has n elements.
        return zip(unfold(n), element -> dummy -> element);
    }
```

```
public HList<E> takeWhile(Predicate<E> fun) {
    //this is hard to implement without sideeffects

    HList<E> result = emptyList();
    HList<E> index = this;
    while (!index.isEmpty() && fun.test(index.head().get())) {
        //prepending new elements for performance
        result = result.prepend(index);
        index = index.tail();
    }
    return result.reverse();
}

public HList<E> reverse() {
    return foldl(l -> e -> (l.prepend(e)), emptyList());
}

public void forEach(Consumer<? super E> fun) {
    foldl(n -> e -> {fun.accept(e); return Optional.empty(); }
        , Optional.empty());
}

public <T> T foldl(Function<T, Function<E, T>> fun, T init) {
    /*
    // recursive version, can overflow
        if (isEmpty()) return init;
        return tail().foldl(fun, fun.apply(init).apply(head().get()));
    */

    T res = init;
    HList<E> l = this;
    while (!l.isEmpty()) {
        res = fun.apply(res).apply(l.head().get());
        l = l.tail();
    }
    return res;
}

public <T> T foldr(Function<E, Function<T, T>> fun, T init) {
    /*
    // recursive version, can overflow
        if (isEmpty()) return init;
        return fun.apply(head().get()).apply(tail().foldr(fun, init));
    */
    return reverse().foldl(t -> e -> fun.apply(e).apply(t), init);
}
```

```
public HList<E> filter(Predicate<E> fun) {
    return foldr(e -> l -> fun.test(e) ? l.prepend(e) : l, emptyList());
}

public <T> HList<T> map(Function<E, T> fun) {
    return foldr(e -> l -> l.prepend(fun.apply(e)), emptyList());
}

public <T, R> HList<R> zip(HList<T> l, Function<E, Function<T, R>> fun) {
    /*
    // recursive version, can overflow
        if (isEmpty() || l.isEmpty()) return emptyList();
        return create(
                fun.apply(head().get()).apply(l.head().get()),
                tail().zip(l.tail(), fun));
    */
    HList<E> l1 = this;
    HList<T> l2 = l;
    HList<R> result = emptyList();
    while (!l1.isEmpty() && !l2.isEmpty()) {
        result = result.prepend(fun.apply(l1.head().get())
            .apply(l2.head().get()));
        l1 = l1.tail();
        l2 = l2.tail();
    }
    return result.reverse();
}

public static <E> HList<E> unfold(E seed, Function<E, E> fun,
        Predicate<E> stop) {
    /*
    // recursive version, can overflow
        if (stop.test(seed)) return emptyList();
        return create(seed, unfold(fun.apply(seed), fun, stop));
    */
    E e = seed;
    HList<E> l = emptyList();
    while (!stop.test(e)) {
        //prepend is a lot cheaper than append
        l = l.prepend(e);
        e = fun.apply(e);
    }
    return l.reverse();
}
```

```
/**
 * Returns a list of consecutive integers starting at 1, ending at to.
 * Equivalent to unfold(1, to, 1);
 */
public static HList<Integer> unfold(int to) {
    if (to < 1) return emptyList();
    return unfold(1, to, 1);
}

/**
 * Returns a list of consecutive integers starting at start,
    ending at stop.
 * Equivalent to unfold(start, stop, 1) or, if start > stop,
    unfold(start, stop, -1).
 */
public static HList<Integer> unfold(int start, int stop) {
    return unfold(start, stop, start < stop ? 1 : -1);
}

/**
 * Returns a list of integers starting at start,
 * ending at stop included, with <code>(n+1) = (n) + step</code>.
 * Returns an empty list if step == 0 or step moves the sequence
    away from stop
 * (if <tt>start < stop ? step < 0 : step > 0</tt>)
 * Infinite Lists are not allowed because they are evaluated
    eagerly, leading to infinite loops.
 */
public static HList<Integer> unfold(int start, int stop, int step) {
    if (step == 0 ||
            start <= stop && step < 0 ||
            start > stop && step > 0) return emptyList();
    return unfold(start, x -> x + step, x ->
        start <= stop
            ? x > stop
            : x < stop);
}

@Override
public String toString() {
    return foldl(a -> b -> a.concat(b.toString() + ", "), "[").concat("]");
}

public String toString(String begin, String separator, String end) {
    return foldl(a -> b -> a.concat(b.toString() + separator),
            begin).concat(end);
}
```

```
    @Override
    public boolean equals(Object o) {
        if (o == null || !o.getClass().equals(this.getClass())) return false;
        @SuppressWarnings("unchecked")
        HList<E> l = (HList<E>) o;
        if (l.size() != size()) return false;
        return zip(l, a -> b -> a.equals(b))
                .foldl(a -> b -> a ? (boolean) b : false, true);
    }

    /**
     * Mirrors AbstractLists hashCode method.
     */
    @Override
    public int hashCode() {
        return foldl(n -> e -> 31*n + (e == null ? 0 : e.hashCode()), 1);
    }
}
```

Abbildung A.8: Item-Kindklasse

```
class Item<E> extends HList<E> {
    private  E          head;
    private  HList<E>   tail;

    Item(E head, HList<E> tail) {
        this.tail = tail;
        this.head = Objects.requireNonNull(head);
    }

    @Override
    public boolean isEmpty() {
        return false;
    }

    @Override
    public Optional<E> head() {
        return Optional.of(head);
    }

    @Override
    public HList<E> tail() {
        return tail;
    }
}
```

Abbildung A.9: Empty-Kindklasse

```
final class Empty<E> extends HList<E> {

    private final static HList<?> EMPTY = new Empty<>();

    @SuppressWarnings("unchecked")
    public static <E> Empty<E> empty() {
        return (Empty<E>) EMPTY;
    }

    private Empty() {}

    @Override
    public boolean isEmpty() {
        return true;
    }

    @Override
    public Optional<E> head() {
        return Optional.empty();
    }

    @Override
    public HList<E> tail() {
        return empty();
    }
}
```

A.4. Tail-Call-Optimization-Test

In Abbildung A.10 werden zwei Methoden vorgeführt, die die selbe Funktionalität iterativ sowie rekursiv implementieren, sowie eine Methode, die diese Methoden ausführt. Die rekursive Methode löst konsistent einen `StackOverflowError` aus.

Abbildung A.10: Test zur Existenz von Tail-Call-Optimization

```
public class TailRec {
    static int times = 100000;

    public static void main(String[] args) {
        System.out.println(new TailRec().iterate(0, times));
        System.out.println(new TailRec().recurse(0, times));
    }

    int iterate(int val, int times) {
        int result = val;
        for (int i = 0; i < times; i++) {
            result++;
        }
        return result;
    }

    int recurse(int val, int times) {
        if (times == 0) return val;
        return recurse(++val, --times);
    }
}
```

A.5. Method Referenz Typ 3 Test

In diesem Abschnitt wird versucht, eine Typ 3 Method Reference zu benutzen, die eine Instanzmethode des zweiten Parameters aufruft. Dies erzeugt einen Fehler, im Gegensatz zu einem Aufruf einer Instanzmethode des ersten Parameters.

Abbildung A.11: Test der Anwendung von Typ 3 Method References

```
static void testMethodRef() {
    System.out.println(testMethodRefOrder((a, b) -> a.m(b)));
    System.out.println(testMethodRefOrder(A::m));

    System.out.println(testMethodRefOrder((a, b) -> b.m(a)));

    System.out.println(testMethodRefOrder(B::m));
    /*
    The method testMethRefOrder(BiFunction<Test.A,Test.B,Integer>)
    in the type Test is not applicable for the arguments (B::m)
    */
}

static A a = new A();
static B b = new B();

static class A {
    public int m(B b) {
        return 1;
    }
}

static class B {
    public int m(A a) {
        return 2;
    }
}

static Integer testMethodRefOrder(BiFunction<A, B, Integer> f) {
    return f.apply(a, b);
}
```

Literaturverzeichnis

[Ale14] A. Alexander. Scala idiom - Methods should not have side effects. http://alvinalexander.com/scala/scala-idiom-methods-functions-no-side-effects, 2014. (Zitiert auf Seite 19)

[DS11] J. I. Daniel Spiewak. java - Cyclomatic Complexity of Java - stackoverflow. http://stackoverflow.com/a/6331574, 2011. (Zitiert auf Seite 31)

[DS14] J. I. Daniel Spiewak. Scalastyle: Implemented Rules. http://www.scalastyle.org/rules-0.5.0.html, 2014. (Zitiert auf Seite 31)

[Fus12] M. Fusco. No more excuses to use null references in Java 8. http://java.dzone.com/articles/no-more-excuses-use-null, 2012. (Zitiert auf den Seiten 15 und 41)

[Goe10] B. Goetz. State of the Lambda Version 2. http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-2.html, 2010. (Zitiert auf Seite 12)

[Goe11a] B. Goetz. A peek past lambda. http://mail.openjdk.java.net/pipermail/lambda-dev/2011-August/003877.html, 2011. (Zitiert auf Seite 12)

[Goe11b] B. Goetz. Syntax decision. http://mail.openjdk.java.net/pipermail/lambda-dev/2011-September/003936.html, 2011. (Zitiert auf Seite 12)

[Goe11c] B. Goetz. Syntax decision. http://mail.openjdk.java.net/pipermail/lambda-dev/2011-September/004021.html, 2011. (Zitiert auf Seite 12)

[Goe12a] B. Goetz. Lambda: A Peek Under The Hood. http://de.slideshare.net/jaxlondon2012/lambda-a-peek-under-the-hood-brian-goetz, 2012. (Zitiert auf Seite 47)

[Goe12b] B. Goetz. Method reference double-colon syntax. http://mail.openjdk.java.net/pipermail/lambda-dev/2012-May/004979.html, 2012. (Zitiert auf Seite 12)

[Goe12c] B. Goetz. State of the Lambda: Libraries Edition. http://cr.openjdk.java.net/~briangoetz/lambda/collections-overview.html, 2012. (Zitiert auf Seite 15)

[Has10] Haskell.org. Thunk. https://www.haskell.org/haskellwiki/Thunk, 2010. (Zitiert auf Seite 23)

[Hav14] M. Haverbeke. Higher-Order Functions :: Eloquent JavaScript. http://eloquentjavascript.net/05_higher_order.html, 2014. (Zitiert auf Seite 24)

[Inc12] K. Inc. McCabe Cyclomatic Complexity. http://docs.klocwork.com/Insight-10.0/McCabe_Cyclomatic_Complexity, 2012. (Zitiert auf Seite 31)

[Iry07] J. Iry. Monads are Elephants Part 1. http://james-iry.blogspot.it/2007/09/monads-are-elephants-part-1.html, 2007. (Zitiert auf Seite 17)

[Jan12] C. Janssen. Thunk - Definition from Techopedia. http://www.techopedia.com/definition/2818/thunk, 2012. (Zitiert auf Seite 23)

[JR14] G. S. John Rose, Brian Goetz. State of the Values. http://cr.openjdk.java.net/~jrose/values/values-0.html, 2014. (Zitiert auf Seite 45)

[Lin14] M. Linhares. Scala's Either, Try and the M word. http://mauricio.github.io/2014/02/17/scala-either-try-and-the-m-word.html, 2014. (Zitiert auf den Seiten 36 und 51)

[Mic14] Microsoft. Refactoring Into Pure Functions. http://msdn.microsoft.com/en-us/library/bb669139.aspx, 2014. (Zitiert auf Seite 19)

[New13] T. Neward. Java 8: Lambdas, Part 1. *Java Magazine*, 2013. http://www.oracle.com/technetwork/articles/java/architect-lambdas-part1-2080972.html. (Zitiert auf Seite 11)

[Ora12a] Oracle. JSR 335: Lambda Expressions for the Java™ Programming Language. http://cr.openjdk.java.net/~dlsmith/jsr335-0.6.1/, 2012. (Zitiert auf den Seiten 12 und 15)

[Ora12b] Oracle. Performance techniques used in the Hotspot JVM. https://wikis.oracle.com/display/HotSpotInternals/PerformanceTechniques, 2012. (Zitiert auf Seite 22)

[Ora14a] Oracle. Class Collections (Java Documentation). https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html, 2014. (Zitiert auf Seite 9)

[Ora14b] Oracle. Class CompletableFuture<T> (Java Documentation). http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html, 2014. (Zitiert auf Seite 16)

[Ora14c] Oracle. Class Optional<T> (Java Documentation). http://docs.oracle.com/javase/8/docs/api/java/util/Optional.html, 2014. (Zitiert auf Seite 16)

[Ora14d] Oracle. FunctionalInterface Definition. http://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html, 2014. (Zitiert auf Seite 10)

[Ora14e] Oracle. Interface Comparator<T> (Java Documentation). http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html, 2014. (Zitiert auf den Seiten 9 und 31)

[Ora14f] Oracle. Interface Stream<T> (Java Documentation). http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html, 2014. (Zitiert auf Seite 14)

[Ora14g] Oracle. Method References. https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html, 2014. (Zitiert auf den Seiten 12 und 13)

[Ora14h] Oracle. Package java.util.function (Java Documentation). http://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html, 2014. (Zitiert auf Seite 11)

[Ora14i] Oracle. Package java.util.stream (Java Documentation). https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html, 2014. (Zitiert auf Seite 15)

[Ora14j] Oracle. Syntax of Lambda Expressions. http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax, 2014. (Zitiert auf Seite 10)

[Par03] S. P. Parker. *McGraw-Hill dictionary of scientific and technical terms*. McGraw-Hill, 2003. http://encyclopedia2.thefreedictionary.com/Thunk+(data). (Zitiert auf Seite 23)

[RGU14] A. M. Raoul-Gabriel Urma, Mario Fusco. *Java 8 in Action: Lambdas, Streams and Functional-style Programming v9*. Manning Publications, 2014. (Zitiert auf den Seiten 17, 29 und 41)

[Sar14] F. Sarradin. Playing with Scala's pattern matching. http://kerflyn.wordpress.com/2011/02/14/playing-with-scalas-pattern-matching/, 2014. (Zitiert auf Seite 27)

[Sau14a] P. yves Saumont. What's Wrong in Java 8, Part IV: Monads. http://java.dzone.com/articles/whats-wrong-java-8-part-iv, 2014. (Zitiert auf den Seiten 17 und 41)

[Sau14b] P. yves Saumont. What's Wrong in Java 8, Part V: Tuples. http://java.dzone.com/articles/whats-wrong-java-8-part-v, 2014. (Zitiert auf Seite 45)

[Sch09] A. Schwaighofer. Tail Call Optimization in the Java HotSpot™ VM. http://www.ssw.uni-linz.ac.at/Research/Papers/Schwaighofer09Master/schwaighofer09master.pdf, 2009. (Zitiert auf Seite 22)

[Sha14] A. Sharma. 6 Benefits of Programming with Immutable Objects in Java. https://www.linkedin.com/today/post/article/20140528113353-16837833-6-benefits-of-programming-with-immutable-objects-in-java, 2014. (Zitiert auf Seite 20)

[She14] O. Shelajev. Monadic futures in Java 8. http://zeroturnaround.com/rebellabs/monadic-futures-in-java8/, 2014. (Zitiert auf Seite 17)

[Sta13] A. Staveley. Scala Pattern Matching: A Case for New Thinking? http://java.dzone.com/articles/scala-pattern-matching-case, 2013. (Zitiert auf Seite 27)

[Urm14a] R.-G. Urma. Processing Data with Java SE 8 Streams, Part 1. *Java Magazine*, 2014. http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html. (Zitiert auf den Seiten 14, 15 und 41)

[Urm14b] R.-G. Urma. Tired of Null Pointer Exceptions? Consider Using Java SE 8's Optional! http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html, 2014. (Zitiert auf den Seiten 16, 36 und 41)

Alle URLs wurden zuletzt am 21. November 2014 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift